Welcome to the RapidMiner Operator Reference, the final result of a long working process. When we first started to plan this reference, we had an extensive discussion about the purpose of this book. Would anybody want to read the hole book? Starting with Ada Boost and reading the description of every single operator to the X-Validation? Or would it only serve for looking up particular operators, although that is also possible in the program interface itself?

We decided for the latter and with growing progress in the making of this book, we realized how futile this entire discussion has been. It was not long until the book reached the 600 pages limit and now we nearly hit the 1000 pages, what is far beyond anybody would like to read entirely. Even if there would be a great structure, explaining the usage of single groups of operators as guiding transitions between the explanations of single operators, nobody could comprehend all that. The reader would have long forgotten about the Loop Clusters operator until he get’s to know about cross validation. So we didn’t dump any effort in that and hence the book has become a pure reference. For getting to know RapidMiner itself, this is not a suitable document. Therefore we would rather recommend to read the manual as a starting point. There are other documents available for particular scenarios, like using RapidMiner as a researcher or when you want to extend it’s functionality. Please take a look at our website rapidminer.com to get an overview, which documentations are available.

From that fact, we can draw some suggestions about how to read this book: Whenever you want to know about a particular operator, just open the index at the end of this book, and directly jump to the operator. The order of the
operators in this book is determined by the group structure in the operator tree, as you will immediately see, when taking a look at the contents. As operators for similar tasks are grouped together in RapidMiner, these operators are also near to each other in this book. So if you are interested in broadening your perspective of RapidMiner beyond an already known operator, you can continue reading a few pages before and after the operator you picked from the index.

Once you read the description of an operator, you can jump to the tutorial process, that will explain a possible use case. Often the functionality of an operator can be understood easier with a context of a complete process. All these processes are also available in RapidMiner. You simply need to open the description of this operator in the help view and scroll down. After pressing on the respective link, the process will be opened and you can inspect the details, execute it and analyse the results from break points. Apart from that, the explanation of the parameters will give you a good insight of what the operator is capable of and what it can be configured for.

I think there’s nothing left to say except wishing you a lot of illustrative encounters with the various operators. And if you really read it from start to end, please tell us, as we have bets running on that. Of course we will verify that by checking if you found all the easter eggs . . .

Sebastian Land
# Contents

## 1 Process Control

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remember</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>7</td>
</tr>
<tr>
<td>Join Paths</td>
<td>10</td>
</tr>
<tr>
<td>Handle Exception</td>
<td>12</td>
</tr>
<tr>
<td>Throw Exception</td>
<td>15</td>
</tr>
<tr>
<td>1.1 Parameter</td>
<td>17</td>
</tr>
<tr>
<td>Set Parameters</td>
<td>17</td>
</tr>
<tr>
<td>Optimize Parameters (Grid)</td>
<td>21</td>
</tr>
<tr>
<td>Optimize Parameters (Evolutionary)</td>
<td>26</td>
</tr>
<tr>
<td>1.2 Loop</td>
<td>32</td>
</tr>
<tr>
<td>Loop</td>
<td>32</td>
</tr>
<tr>
<td>Loop Attributes</td>
<td>36</td>
</tr>
<tr>
<td>Loop Values</td>
<td>42</td>
</tr>
<tr>
<td>Loop Examples</td>
<td>46</td>
</tr>
<tr>
<td>Loop Clusters</td>
<td>49</td>
</tr>
<tr>
<td>Loop Data Sets</td>
<td>51</td>
</tr>
<tr>
<td>Loop and Average</td>
<td>54</td>
</tr>
<tr>
<td>Loop Parameters</td>
<td>57</td>
</tr>
<tr>
<td>Loop Files</td>
<td>62</td>
</tr>
<tr>
<td>X-Prediction</td>
<td>65</td>
</tr>
<tr>
<td>1.3 Branch</td>
<td>71</td>
</tr>
<tr>
<td>Branch</td>
<td>71</td>
</tr>
</tbody>
</table>
## Contents

Select Subprocess .............................................. 75

1.4 Collections ................................................ 79
  Collect ....................................................... 79
  Select ......................................................... 82
  Loop Collection .............................................. 85

2 Utility .............................................................. 89
  Subprocess .................................................... 89

2.1 Macros .......................................................... 93
  Set Macro ....................................................... 93
  Set Macros ..................................................... 98
  Generate Macro ............................................... 102
  Extract Macro ................................................ 110

2.2 Logging ......................................................... 118
  Log .............................................................. 118
  Provide Macro as Log Value .................................. 125
  Log to Data .................................................... 128

2.3 Execution ....................................................... 132
  Execute Process ............................................... 132
  Execute Script ............................................... 135
  Execute SQL .................................................... 141
  Execute Program .............................................. 145

2.4 Files .......................................................... 148
  Write as Text .................................................. 149
  Copy File ....................................................... 151
  Rename File .................................................... 153
  Delete File ..................................................... 156
  Move File ....................................................... 158
  Create Directory .............................................. 161

2.5 Data Generation ............................................... 163
  Generate Data ................................................ 163
  Generate Nominal Data ....................................... 165
  Generate Direct Mailing Data ................................. 167
  Generate Sales Data .......................................... 169
| Contents |
|---------------------|-------|
| Add Noise           | 171   |
| 2.6 Miscellaneous   | 177   |
| Materialize Data    | 177   |
| Free Memory         | 179   |
| 3 Repository Access | 183   |
| Retrieve            | 183   |
| Store               | 185   |
| 4 Import            | 189   |
| 4.1 Data            | 189   |
| Read csv            | 189   |
| Read Excel          | 194   |
| Read SAS            | 197   |
| Read Access         | 199   |
| Read AML            | 201   |
| Read ARFF           | 206   |
| Read Database       | 210   |
| Stream Database     | 215   |
| Read SPSS           | 219   |
| 4.2 Models          | 222   |
| Read Model          | 222   |
| 4.3 Attributes      | 224   |
| Read Weights        | 224   |
| 5 Export            | 227   |
| Write               | 227   |
| 5.1 Data            | 229   |
| Write AML           | 229   |
| Write Arff          | 232   |
| Write Database      | 236   |
| Update Database     | 240   |
| Write Special Format| 244   |
| 5.2 Models          | 248   |
| Write Model         | 248   |
# Contents

Write Clustering ........................................ 250

5.3 Attributes ............................................. 252
Write Weights ............................................. 252
Write Constructions ..................................... 255

5.4 Results ................................................ 257
Write Performance ..................................... 257

5.5 Other .................................................. 259
Write Parameters ..................................... 259
Write Threshold ....................................... 261

6 Data Transformation .................................. 265

6.1 Name and Role Modification ...................... 265
  Rename .................................................. 265
  Rename by Replacing ................................ 267
  Set Role ............................................... 273
  Exchange Roles ...................................... 277

6.2 Type Conversion ..................................... 279
  Numerical to Binominal .............................. 279
  Numerical to Polynominal ......................... 285
  Numerical to Real .................................. 290
  Real to Integer ..................................... 295
  Nominal to Binominal ............................... 300
  Nominal to Text ..................................... 306
  Nominal to Numerical ............................... 310
  Nominal to Date ..................................... 318
  Text to Nominal ..................................... 327
  Date to Numerical .................................. 331
  Date to Nominal ..................................... 336
  Parse Numbers ....................................... 343
  Format Numbers ..................................... 349
  Guess Types ......................................... 356

6.2.1 Discretization ..................................... 361
  Discretize by Size .................................. 361
  Discretize by Binning ............................... 368
## Contents

- Discretize by Frequency ........................................ 375
- Discretize by User Specification ................................. 382
- Discretize by Entropy ........................................... 389

### 6.3 Attribute Set Reduction and Transformation ............... 395

#### 6.3.1 Generation ............................................. 395
- Generate ID .................................................. 396
- Generate Empty Attribute .................................... 397
- Generate Copy ............................................. 400
- Generate Attributes ......................................... 402
- Generate Concatenation ...................................... 417
- Generate Aggregation ....................................... 419
- Generate Function Set ....................................... 425
- Optimization ............................................... 429
- Optimize by Generation (YAGGA) ........................... 429

#### 6.3.2 Transformation .......................................... 434
- Principal Component Analysis ................................. 434
- Principal Component Analysis (Kernel) ....................... 437
- Independent Component Analysis ............................ 442
- Generalized Hebbian Algorithm ............................... 445
- Singular Value Decomposition ................................ 448
- Self-Organizing Map ........................................ 452

#### 6.3.3 Selection ................................................ 455
- Select Attributes ........................................... 455
- Select by Weights ........................................... 461
- Remove Attribute Range ..................................... 465
- Remove Useless Attributes .................................. 467
- Remove Correlated Attributes ................................. 472
- Work on Subset ............................................. 476
- Optimization ............................................... 483
- Forward Selection ......................................... 483
- Backward Elimination ....................................... 487
- Optimize Selection ......................................... 492
- Optimize Selection (Evolutionary) ......................... 498

XI
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Value Modification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set Data</td>
<td>504</td>
</tr>
<tr>
<td></td>
<td>Declare Missing Value</td>
<td>507</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Numerical Value Modification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normalize</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>Scale by Weights</td>
<td>518</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Nominal Value Modification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Map</td>
<td>521</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>527</td>
</tr>
<tr>
<td></td>
<td>Cut</td>
<td>532</td>
</tr>
<tr>
<td></td>
<td>Split</td>
<td>537</td>
</tr>
<tr>
<td></td>
<td>Merge</td>
<td>544</td>
</tr>
<tr>
<td></td>
<td>Remap Binominals</td>
<td>546</td>
</tr>
<tr>
<td>6.5</td>
<td>Data Cleansing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Replace Missing Values</td>
<td>552</td>
</tr>
<tr>
<td></td>
<td>Fill Data Gaps</td>
<td>558</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Outlier Detection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Detect Outlier (Distances)</td>
<td>560</td>
</tr>
<tr>
<td></td>
<td>Detect Outlier (Densities)</td>
<td>563</td>
</tr>
<tr>
<td></td>
<td>Detect Outlier (LOF)</td>
<td>566</td>
</tr>
<tr>
<td></td>
<td>Detect Outlier (COF)</td>
<td>569</td>
</tr>
<tr>
<td>6.6</td>
<td>Filtering</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Filter Examples</td>
<td>575</td>
</tr>
<tr>
<td></td>
<td>Remove Duplicates</td>
<td>580</td>
</tr>
<tr>
<td></td>
<td>Filter Example Range</td>
<td>585</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Sampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sample</td>
<td>588</td>
</tr>
<tr>
<td></td>
<td>Sample (Stratified)</td>
<td>591</td>
</tr>
<tr>
<td></td>
<td>Sample (Bootstrapping)</td>
<td>595</td>
</tr>
<tr>
<td></td>
<td>Split Data</td>
<td>598</td>
</tr>
<tr>
<td>6.7</td>
<td>Sorting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sort</td>
<td>601</td>
</tr>
<tr>
<td>6.8</td>
<td>Rotation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pivot</td>
<td>604</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>6.9</td>
<td>Aggregation</td>
<td>615</td>
</tr>
<tr>
<td>6.10</td>
<td>Set Operations</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td>Append</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td>Join</td>
<td>624</td>
</tr>
<tr>
<td></td>
<td>Set Minus</td>
<td>630</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>633</td>
</tr>
<tr>
<td></td>
<td>Superset</td>
<td>637</td>
</tr>
<tr>
<td>7</td>
<td>Modeling</td>
<td>641</td>
</tr>
<tr>
<td>7.1</td>
<td>Classification and Regression</td>
<td>641</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Lazy Modeling</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>Default Model</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>K-NN</td>
<td>643</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Bayesian Modeling</td>
<td>651</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>651</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes (Kernel)</td>
<td>655</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Tree Induction</td>
<td>659</td>
</tr>
<tr>
<td></td>
<td>Decision Tree</td>
<td>659</td>
</tr>
<tr>
<td></td>
<td>ID3</td>
<td>665</td>
</tr>
<tr>
<td></td>
<td>CHAID</td>
<td>669</td>
</tr>
<tr>
<td></td>
<td>Decision Stump</td>
<td>673</td>
</tr>
<tr>
<td></td>
<td>Random Tree</td>
<td>675</td>
</tr>
<tr>
<td></td>
<td>Random Forest</td>
<td>680</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Rule Induction</td>
<td>684</td>
</tr>
<tr>
<td></td>
<td>Rule Induction</td>
<td>684</td>
</tr>
<tr>
<td></td>
<td>Subgroup Discovery</td>
<td>687</td>
</tr>
<tr>
<td></td>
<td>Tree to Rules</td>
<td>692</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Neural Net Training</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td>Perceptron</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td>Neural Net</td>
<td>696</td>
</tr>
</tbody>
</table>
Contents

7.1.6 Function Fitting ............................................. 701
  Linear Regression ............................................. 701
  Polynomial Regression ......................................... 705
7.1.7 Logistic Regression ......................................... 709
  Logistic Regression ........................................... 709
7.1.8 Support Vector Modeling ................................... 714
  Support Vector Machine ........................................ 714
  Support Vector Machine (LibSVM) .............................. 720
  Support Vector Machine (Evolutionary) ......................... 726
  Support Vector Machine (PSO) ................................ 733
7.1.9 Discriminant Analysis ....................................... 738
  Linear Discriminant Analysis ................................ 738
7.1.10 Meta Modeling ............................................. 742
  Vote .......................................................... 742
  Polynomial by Binomial Classification ......................... 744
  Classification by Regression .................................. 747
  Bayesian Boosting ............................................. 749
  AdaBoost ..................................................... 754
  Bagging ....................................................... 757
  Stacking ...................................................... 761
  MetaCost ...................................................... 765
7.2 Attribute Weighting ........................................... 769
  Weight by Information Gain .................................. 769
  Weight by Information Gain Ratio .............................. 771
  Weight by Correlation ......................................... 774
  Weight by Chi Squared Statistic ............................... 777
  Weight by Relief .............................................. 780
  Weight by SVM ................................................ 782
  Weight by PCA ............................................... 785
  Weight by Component Model ................................... 788
  Data to Weights ................................................ 791
  Weights to Data ............................................... 793
7.2.1 Optimization ................................................ 795
  Optimize Weights (Evolutionary) .............................. 795
7.3 Clustering and Segmentation ........................................ 800
   K-Means ............................................................... 800
   K-Means (Kernel) .................................................... 805
   K-Medoids ............................................................ 810
   DBSCAN ............................................................... 817
   Expectation Maximization Clustering ............................... 823
   Support Vector Clustering .......................................... 828
   Agglomerative Clustering ........................................... 832
   Top Down Clustering ................................................ 838
   Extract Cluster Prototypes ........................................ 841
7.4 Association and Item Set Mining .................................. 844
   FP-Growth ........................................................... 844
   Create Association Rules ......................................... 848
   Generalized Sequential Patterns .................................. 852
7.5 Correlation and Dependency Computation ....................... 856
   Correlation Matrix .................................................. 856
7.6 Similarity Computation .............................................. 859
   Data to Similarity .................................................... 859
   Cross Distances ...................................................... 864
7.7 Model Application .................................................... 870
   Apply Model .......................................................... 870
   Group Models ........................................................ 873
   7.7.1 Thresholds ...................................................... 876
       Find Threshold ..................................................... 876
       Create Threshold ................................................ 880
       Apply Threshold ................................................ 883
   7.7.2 Confidences ..................................................... 886
       Drop Uncertain Predictions ....................................... 886
8 Evaluation .................................................................. 891
  8.1 Validation ............................................................ 891
       Split Validation ..................................................... 891
       X-Validation ........................................................ 896
       Bootstrapping Validation ........................................ 903
## 8.2 Performance Measurement
- Performance ........................................... 908
- Extract Performance ................................. 912
- Combine Performances .............................. 915

### 8.2.1 Classification and Regression
- Performance (Classification) ....................... 918
- Performance (Binominal Classification) .......... 924
- Performance (Regression) .......................... 932
- Performance (Costs) ................................ 937

### 8.2.2 Clustering
- Cluster Distance Performance .................... 941
- Cluster Density Performance ..................... 944
- Item Distribution Performance ................... 947

## 8.3 Significance
- T-Test .................................................. 950
- ANOVA ............................................... 954

## 8.4 Visual Evaluation
- Create Lift Chart ................................. 958
- Compare ROCs .................................... 962
1 Process Control

Remember

This operator stores the given object in the object store of the process. The stored object can be retrieved from the store by using the Recall operator.

Description

The Remember operator can be used to store the input object into the object store of the process under the specified name. The name of the object is specified through the name parameter. The io object parameter specifies the class of the object. The stored object can later be restored by the Recall operator by using the same name and class (i.e. the name and class that was used to store it using the Remember operator). There is no scoping mechanism in RapidMiner processes therefore objects can be stored (using Remember operator) and retrieved (using Recall operator) at any nesting level. But care should be taken that the execution order of operators is such that the Remember operator for an object always executes before the Recall operator for that object. The combination of these two operators can be used to build complex processes where an input object is used in completely different parts or loops of the processes.
1. Process Control

Differentiation

Recall The Remember operator is always used in combination with the Recall operator. The Remember operators stores the required object into the object store and the Recall operator retrieves the stored object when required. See page 4 for details.

Input Ports

store (sto) Any object can be provided here. This object will be stored in the object store of the process. It should be made sure that the class of this object is selected in the io object parameter.

Output Ports

stored (sto) The object that was given as input is passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the object will be stored even if this port is left without connections.

Parameters

name (string) The name under which the input object is stored is specified through this parameter. The same name will be used for retrieving this object through the Recall operator.

io object (selection) The class of the input object is selected through this parameter.
Related Documents

Recall (4)

Tutorial Processes

Introduction to Remember and Recall operators

This process uses the combination of the Remember and Recall operators to display the testing data set of the Split Validation operator. The testing data set is present in the testing subprocess of the Split Validation operator but it is not available outside the Split Validation operator.

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it. The test set size parameter is set to 5 and the training set size parameter is set to -1. Thus the test set in the testing subprocess will be composed of 5 examples. The Default Model operator is used in the training subprocess to train a model. The testing data set is available at the test port of the testing subprocess. The Remember operator is used to store the testing data set into the object store of the process. The name and io object parameters are set to 'Testset' and 'ExampleSet' respectively. The Apply Model and Performance operator are applied in the testing subprocess later. In the main process, the Recall operator is used to retrieve the testing data set. The name and io object parameters of the Recall operator are set to 'Testset' and 'ExampleSet' respectively to retrieve the object that was stored by the Remember operator. The output of the Recall operator is connected to the result port of the process. Therefore the testing data set can be seen in the Results Workspace.
1. Process Control

Recall

This operator retrieves the specified object from the object store of the process. The objects can be stored in the object store by using the Remember operator.

Description

The Recall operator can be used for retrieving the specified object from the object store of the process. The name of the object is specified through the *name* parameter. The *io object* parameter specifies the class of the required object. The Recall operator is always used in combination with the operators like the Remember operator. For Recall operator to retrieve an object, first it is necessary that the object should be stored in the object store by using operators like the Remember operator. The name and class of the object are specified when the object is stored using the Remember operator. The same name (in *name* parameter) and class (in *io object* parameter) should be specified in the Recall operator to retrieve that object. The same stored object can be retrieved multiple number of times if the *remove from store* parameter of the Recall operator is not set to true. There is no scoping mechanism in RapidMiner processes therefore objects can be stored (using Remember operator) and retrieved (using Recall operator) at any nesting level. But care should be taken that the execution...
order of operators is such that the Remember operator for an object always executes before the Recall operator for that object. The combination of these two operators can be used to build complex processes where an input object is used in completely different parts or loops of the processes.

**Differentiation**

**Remember** The Recall operator is always used in combination with the Remember operator. The Remember operators stores the required object into the object store and the Recall operator retrieves the stored object when required. See page 1 for details.

**Output Ports**

**result (res)** The specified object is retrieved from the object store of the process and is delivered through this output port.

**Parameters**

**name (string)** The name of the required object is specified through this parameter. This name should be the same name that was used while storing the object in an earlier part of the process.

**io object (selection)** The class of the required object is selected through this parameter. This class should be the same class that was used while storing the object in an earlier part of the process.

**remove from store (boolean)** If this parameter is set to true, the specified object is removed from the object store after it has been retrieved. In such a case the object can be retrieved just once. If this parameter is set to false, the object remains in the object store even after retrieval. Thus the object can be retrieved multiple number of times (by using the Recall operator multiple number of times).
1. Process Control

Related Documents

Remember (1)

Tutorial Processes

Introduction to Remember and Recall operators

This process uses the combination of the Remember and Recall operators to display the testing data set of the Split Validation operator. The testing data set is present in the testing subprocess of the Split Validation operator but it is not available outside the Split Validation operator.

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it. The test set size parameter is set to 5 and the training set size parameter is set to -1. Thus the test set in the testing subprocess will be composed of 5 examples. The Default Model operator is used in the training subprocess to train a model. The testing data set is available at the tes port of the testing subprocess. The Remember operator is used to store the testing data set into the object store of the process. The Apply Model and Performance operator are applied in the testing subprocess later. In the main process, the Recall operator is used to retrieve the testing data set. The name and io object parameters of the Recall operator are set to 'Testset' and 'ExampleSet' respectively to retrieve the object that was stored by the Remember operator. The output of the Recall operator is connect to the result port of the process. Therefore the testing data set can be seen in the Results Workspace.
Multiply

This operator copies its input object to all connected output ports. It does not modify the input object.

Description

The Multiply operator copies the objects at its input port to the output ports multiple number of times. As more ports are connected, more copies are generated. The input object is copied by reference; hence the underlying data of the ExampleSet is never copied (unless the Materialize Data operator is used). As copy-by-reference is usually lighter than copy-by-value, copying objects is cheap through this operator. When copying ExampleSets only the references to attributes are copied. It is very important to note here that when attributes are changed or added in one copy of the ExampleSet, this change has no effect on other copies. However, if data is modified in one copy, it is also modified in the other copies generated by the Multiply operator.
1. Process Control

Input Ports

**input** *(inp)* It can take various kinds of objects as input e.g. an ExampleSet or even a model.

Output Ports

**output** *(out)* There can be many output ports. As one output port is connected, another output port is created for further connections. All ports deliver unchanged copies of the input object.

Tutorial Processes

Multiplying data sets

In this Example Process the Retrieve operator is used to load the Labor-Negotiations data set. A *breakpoint* is inserted after this operator so that the data can be viewed before applying the Multiply operator. You can see that this data set has many missing values. Press the green-colored Run button to continue the process.

4 copies of the data set are generated using the Multiply operator. The Replace Missing Values operator is applied on the first copy. The Select Attributes operator is applied on the second copy. The Generate ID operator is applied on the third copy and the forth copy is connected directly to the *results* port.

The Replace Missing Values operator replaces all the missing values with the average value of that attribute. As this is a change in data instead of a change in attributes, this change is made in all the copies.

The Select Attributes operator selects one attribute i.e. the duration attribute.
Please note that special attributes are also selected even if they are not explicitly specified. Thus the special attribute of the Labor-Negotiations data set (i.e. the Class attribute) is automatically selected. Results from this operator show only two attributes: the Duration attribute and a special attribute (the Class attribute). As this is a change in attributes instead of a change in data, it is only applied to this copy and all other copies are not affected. Similarly the Generate ID operator adds a new attribute (the id attribute) to the data set. As this is not a change in data (it is a change in attributes), it is relevant only for this copy and other copies are not affected.

The Last copy generated by the Multiply operator is connected directly to the results port without applying any operator. This copy is not the same as the input ExampleSet. This copy has no missing values. This is because of the Replace Missing Values operator, it made a change in data and changes in data are reflected in all copies.

Multiplying models
1. Process Control

In this Example Process the Retrieve operator is used to load the Golf dataset. The k-NN operator is applied on it to learn a classification model. This model is given as input to the Multiply operator. 2 copies are generated by the Multiply operator. One copy of model is used to apply this model on the Golf-Testset data set and the other copy is used to apply the model on the Golf data set. This simple Example Process was added to show that the Multiply operator can multiply different objects e.g. models.

Join Paths

This operator delivers the first non-null input to its output.

Description

The Join Paths operator can have multiple inputs but it has only one output. This operator returns the first non-null input that it receives. This operator can
be useful when some parts of the process are susceptible of producing null results which can halt the entire process. In such a scenario the Join Paths operator can be used to filter out this possibility.

Input Ports

input \((inp)\) This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). Multiple inputs can be provided but only the first non-null object will be returned by this operator.

Output Ports

output \((out)\) The first non-null object that this operator receives is returned through this port.

Tutorial Processes

Returning the first non-null object

This Example Process starts with the Subprocess operator. Two outputs of the Subprocess operator are attached to the first two input ports of the Join Paths operator. But both these inputs are null because the Subprocess operator has no inner operators. The 'Golf' and 'Polynomial' data sets are loaded using the Retrieve operator. The Join Paths operator has four inputs but it returns only the 'Golf' data set because it is the first non-null input that it received.
1. Process Control

**Handle Exception**

This is a nested operator that is used for exception handling. This operator executes the operators in its *Try* subprocess and returns its results if there is no error. If there is an error in the *Try* subprocess, the *Catch* subprocess is executed instead and its results are returned.

**Description**

The Handle Exception operator is a nested operator i.e. it has two subprocesses: *Try* and *Catch*. This operator first tries to execute the *Try* subprocess. If there is no error i.e. the execution is successful, then this operator returns the results of the *Try* subprocess. In case there is any error the process is not stopped. Instead, the *Catch* subprocess is executed and its results are delivered by this operator. The error message can be saved using the *exception macro* parameter. This Try/Catch concept is like the exception handling construct used in many programming languages. You need to have a basic understanding of macros if you want to save the error message using the *exception macro*. Please study the docu-
mentation of the Extract Macro operator for basic understanding of macros. For more information regarding subprocesses please study the Subprocess operator. Please use this operator with care since it will also cover unexpected errors.

Input Ports

**in (in )** This operator can have multiple inputs. When one input is connected, another in port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first in port of this operator is available at the first in port of the nested chain (inside the subprocess). Do not forget to connect all inputs in the correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

**out (out)** This operator can have multiple out ports. When one output is connected, another out port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first out port of the subprocess is delivered at the first out port of the outer process. Don't forget to connect all outputs in the correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Parameters

**exception macro (string)** This parameter specifies the name of the macro which will store the error message (if any). This macro can be accessed by other operators by using `%{macro_name}` syntax.
1. Process Control

Tutorial Processes

Using Try/Catch for exception handling

The goal of this Example Process is to deliver the 'Iris' data set after renaming its attributes. In case there is an error, the original 'Iris' data set should be delivered along with the error message.

The 'Iris' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. You can see that there are four regular attributes a1, a2, a3 and a4. The Handle Exception operator is applied next. Have a look at the subprocesses of this operator. The Rename operator is applied in the *Try* subprocess for renaming the attributes. The *old name* and *new name* parameters of the Rename operator are deliberately left empty which is an error because these are mandatory parameters. The *Catch* subprocess takes the original ExampleSet and applies the Log operator and delivers the ExampleSet without modifications to the output. The Log operator is used for recording the error message in case there is an error.

Execute the process and switch to the Results Workspace. You can see that the attributes of the ExampleSet have not been renamed. This is because there was an error in the *Try* subprocess, therefore it was not executed. The error message can be seen in the log which says that values for mandatory parameters of the Rename operator were not provided. The original ExampleSet can be seen in the Results Workspace because when the Handle Exception operator encountered an error in the *Try* subprocess, it stopped its execution and executed the *Catch* subprocess instead and delivered its results.

Now set the *old name* and *new name* parameters of the Rename operator to 'label' and 'new_label' respectively and run the process again. This time you can see that the attributes have been renamed and there is no error message. This is so because the Handle Exception operator first tried to execute the *Try* subprocess. Since there was no error in it, its results were delivered and the *Catch* subprocess was not executed.
Throw Exception

This operator throws an exception every time it is executed.

Description

The Throw Exception operator will throw an exception with an user-defined message as soon as it is executed. This will cause the process to fail. It can be useful if e. g. a certain result is equal to a failure.

Input Ports

through (thr) Data delivered to this port will be passed to the output port without any modification. Whenever an input port gets occupied, two new input and output ports become available. The order remains the same. Data delivered at the first input port is available at the first output port. It’s not necessary to connect this port, the exception will be thrown anyway.
1. Process Control

Output Ports

through (thr) Provides the data which was delivered at the corresponding input port without any changes. Whenever an input port gets occupied, two new input and output ports become available. The order remains the same. Data delivered at the first input port is available at the first output port. It's not necessary to connect this port, the exception will be thrown anyway.

Parameters

message (string) The error message that should be shown/logged is specified through this parameter.

Tutorial Processes

Throw exeption if no examples passed

The 'Iris' data set is loaded with the Retrieve operator. The Filter Examples operator is applied on the data and filters examples of the attribute a1 that have a value greater than 10.

The ExampleSet is passed on to a Branch operator. If there is at least one example, the data is passed on without changing. If there are zero examples, the Throw Exception operator makes the process fail with the entered message. Because the filter condition applies on no example of the data set, the process will fail i.e. the Throw Exception operator will be executed.
1.1. Parameter

Set Parameters

This operator applies a set of parameters to the specified operators. This operator is mostly used for applying an optimal set of parameters of one operator to another similar operator.

Description

The Set Parameters operator takes a set of parameters as input. Operators like Optimize Parameters (Grid) or Read Parameters can be used as a source of parameter set. The Set Parameters operator takes this parameter set and assigns these parameter values to the parameters of the specified operator which can be specified through the name map parameter. This menu has two columns: 'set operator name' and 'operator name'. The 'set operator name' column is used for specifying the name of the operator (or the name of the parameter set) that generated this parameter set and the 'operator name' column specifies the name of the target operator i.e. the operator that will use this parameter set.

This operator can be very useful, e.g. in the following scenario. If one wants to find the best parameters for a certain learning scheme, one is usually also interested in the model generated with these optimal parameters. Finding the best parameter set is easily possible by operators like the Optimize Parameters (Grid) operator. But generating a model with these optimal parameters is not possi-
1. Process Control

ble because such parameter optimization operators do not return the IOObjects produced within, but only a parameter set and a performance vector. This is, because the parameter optimization operators know nothing about models, but only about the performance vectors produced within and producing performance vectors does not necessarily require a model. To solve this problem, one can use the Set Parameters operator. Usually, a process with the Set Parameters operator contains at least two operators of the same type, typically a learner. One learner may be an inner operator of a parameter optimization scheme and may be named 'Learner', whereas a second learner of the same type named 'OptimalLearner' comes after the parameter optimization scheme and should use the optimal parameter set returned by the optimization scheme. The Set Parameters operator takes the parameter set returned by the optimization scheme as input. The name 'Learner', is specified in the 'set operator name' column and the name 'OptimalLearner' is specified in the 'operator name' column of the name map parameter. Each parameter in this list maps the name of an operator that was used during the optimization (in our case this is 'Learner') to an operator that should now use these parameters (in our case this is 'OptimalLearner'). An important thing to note here is the sequence of operators. The 'Learner', operator should be first in sequence, followed by the Set Parameters operator and finally the 'OptimalLearner' operator.

Differentiation

Clone Parameters The Clone Parameters operator is a very generic form of the Set Parameters operator. It differs from the Set Parameters operator because it does not require a ParameterSet as input. It simply reads a parameter value from a source and uses it to set the parameter value of a target parameter. This operator is more generic than the Set Parameters operator and could completely replace it. See page ?? for details.
Input Ports

**parameter set** (*par*) This input port expects a parameter set. It is the output of the Optimize Parameters (Grid) operator in the attached Example Process. The output of other operators like the Read Parameters operator can also be used as a source of a parameter set.

Parameters

**name map** (*list*) The Set Parameters operator takes a parameter set and assigns these parameter values to the parameters of the specified operator which can be specified through the *name map* parameter. This menu has two columns i.e. 'set operator name' and 'operator name'. The 'set operator name' column is used for specifying the operator that generated this parameter set (or the name of the parameter set) and the 'operator name' column specifies the name of the target operator i.e. the operator that will use this parameter.

Related Documents

Clone Parameters (??)

Tutorial Processes

Building a model using an optimal parameter set

The 'Polynomial' data set is loaded using the Retrieve operator. The Optimize Parameters (Grid) operator is applied on it to produce an optimal set of parameters. The X-Validation operator is applied in the subprocess of the Optimize
1. Process Control

Parameters (Grid) operator. The X-Validation operator uses an SVM (LibSVM) operator for training a model. This SVM (LibSVM) operator is named 'Learner'. After execution of the Optimize Parameters (Grid) operator, it returns a parameter set with optimal values of the $C$ and $degree$ parameters of the SVM (LibSVM) operator. This parameter set is provided to the Set Parameters operator. The Set Parameters operator provides these optimal parameters to the SVM (LibSVM) operator in the main process (outside the Optimize Parameter (Grid) operator). This SVM (LibSVM) operator is named 'OptimalLearner' and it comes after the Set Parameters operator in the execution sequence. Have a look at the name map parameter of the Set Parameters operator. The name 'Learner', is specified in the 'set operator name' column and the name 'OptimalLearner' is specified in the 'operator name' column of the name map parameter. Each parameter in this list maps the name of an operator that was used during the optimization (in our case this is 'Learner') to an operator that should now use these parameters (in our case this is 'OptimalLearner'). The 'OptimalLearner' uses the optimal parameters of 'Learner' and applies the model on the 'Polynomial' data set. Have a look at the parameters of the 'OptimalLearner' before the execution of the process. The $degree$ and $C$ parameters are set to 1 and 0.0 respectively. These values are changed to 3 and 250 after execution of the process because these are the optimal values for these parameters generated by the Optimize Parameters (Grid) operator and then these parameters were used by 'OptimalLearner'. These parameters can also be seen during the process execution (at the breakpoint after the Optimize Parameters (Grid) operator).
1.1. Parameter

Optimize Parameters (Grid)

This operator finds the optimal values of the selected parameters of the operators in its subprocess.

Description

The Optimize Parameters (Grid) operator has a subprocess in it. It executes the subprocess for all combinations of selected values of the parameters and then delivers the optimal parameter values through the parameter port. The performance vector for optimal values of parameters is delivered through the performance port. Any additional results of the subprocess are delivered through the result ports.

The entire configuration of this operator is done through the edit parameter settings parameter. Complete description of this parameter is described in the parameters section.
1. Process Control

Please note that selecting a large number of parameters and/or large number of steps (or possible values of parameters) results in a huge number of combinations. For example, if you select 3 parameters and 25 steps for each parameter then the total number of combinations would be above 390625 (i.e. $25 \times 25 \times 25$). The subprocess is executed for all possible combinations. Running a subprocess for such a huge number of iterations will take a lot of time. So always carefully limit the parameters and their steps.

This operator returns an optimal parameter set which can also be written to a file with the Write Parameters operator. This parameter set can be read in another process using the Read Parameters operator.

Other parameter optimization schemes are also available. The Optimize Parameters (Evolutionary) operator might be useful if the best ranges and dependencies are not known at all. Another operator which works similar to this parameter optimization operator is the Loop Parameters operator. In contrast to the optimization operators, this operator simply iterates through all parameter combinations. This might be especially useful for plotting purposes.

Differentiation

Optimize Parameters (Evolutionary) The Optimize Parameters (Evolutionary) operator finds the optimal values for a set of parameters using an evolutionary approach which is often more appropriate than a grid search (as in the Optimize Parameters (Grid) operator) or a greedy search (as in the Optimize Parameters (Quadratic) operator) and leads to better results. The Optimize Parameters (Evolutionary) operator might be useful if the best ranges and dependencies are not known at all. See page 26 for details.
1.1. Parameter

Input Ports

input (inp) This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first input port of this operator is available at the first input port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

performance (per) This port delivers the Performance Vector for the optimal values of the selected parameters. A Performance Vector is a list of performance criteria values.

parameters (par) This port delivers the optimal values of the selected parameters. This optimal parameter set can also be written to a file with the Write Parameters operator. The written parameter set can be read in another process using the Read Parameters operator.

result (res) Any additional results of the subprocess are delivered through the result ports. This operator can have multiple outputs. When one result port is connected, another result port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first result port of the subprocess is delivered at the first result port of the operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

Parameters

edit parameter settings The parameters are selected through the edit parameter settings menu. You can select the parameters and their possible values
1. Process Control

through this menu. This menu has an Operators window which lists all the operators in the subprocess of this operator. When you click on any operator in the Operators window, all parameters of that operator are listed in the Parameters window. You can select any parameter through the arrow keys of the menu. The selected parameters are listed in the Selected Parameters window. Only those parameters should be selected for which you want to find optimal values. This operator finds optimal values of the parameters in the specified range. The range of every selected parameter should be specified. When you click on any selected parameter (parameter in Selected Parameters window) the Grid/Range and Value List option is enabled. These options allow you to specify the range of values of the selected parameters. The Min and Max fields are for specifying the lower and upper bounds of the range respectively. As all values within this range cannot be checked, the steps field allows you to specify the number of values to be checked from the specified range. Finally the scale option allows you to select the pattern of these values. You can also specify the values in form of a list.

Related Documents

Optimize Parameters (Evolutionary) (26)

Tutorial Processes

Finding optimal values of parameters of the SVM operator

The 'Weighting' data set is loaded using the Retrieve operator. The Optimize Parameters (Grid) operator is applied on it. Have a look at the Edit Parameter Settings parameter of the Optimize Parameters (Grid) operator. You can see in the Selected Parameters window that the C and gamma parameters of the SVM operator are selected. Click on the SVM.C parameter in the Selected Parameters window, you will see that the range of the C parameter is set from 0.001 to
100000. 11 values are selected (in 10 steps) logarithmically. Now, click on the SVM.\(\text{gamma}\) parameter in the Selected Parameters window, you will see that the range of the \(\text{gamma}\) parameter is set from 0.001 to 1.5. 11 values are selected (in 10 steps) logarithmically. There are 11 possible values of 2 parameters, thus there are 121 (i.e. 11 x 11) combinations. The subprocess will be executed for all combinations of these values, thus it will iterate 121 times. In every iteration, the value of the \(C\) and/or \(\text{gamma}\) parameters of the SVM(LibSVM) operator is changed. The value of the \(C\) parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration. Similarly, the value of the \(\text{gamma}\) parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 1.5 in the last iteration.

Have a look at the subprocess of the Optimize Parameters (Grid) operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

The \textit{log} parameter of the Log operator stores five things.

1. The iterations of the Optimize Parameters (Grid) operator are counted by apply-count of the SVM operator. This is stored in a column named 'Count'.

2. The value of the \textit{classification error} parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'.

3. The value of the \textit{classification error} parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'.

4. The value of the \(C\) parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'.
1. Process Control

5. The value of the *gamma* parameter of the SVM (LibSVM) operator is stored in a column named 'SVM gamma'.

Also note that the stored information will be written into a file as specified in the *filename* parameter.

At the end of the process, the Write Parameters operator is used for writing the optimal parameter set in a file. This file can be read using the Read Parameters operator to use these parameter values in another process.

Run the process and turn to the Results Workspace. You can see that the optimal parameter set has the following values: SVM.C = 100000.0 and SVM.gamma = 0.0010. Now have a look at the values saved by the Log operator to verify these values. Switch to Table View to see the stored values in tabular form. You can see that the minimum Testing Error is 0.008 (in 11th iteration). The values of the *C* and *gamma* parameters for this iteration are the same as given in the optimal parameter set.

Optimize Parameters (Evolutionary)

This operator finds the optimal values of the selected parameters of the operators in its subprocess. It uses an evolutionary computation approach.
1.1. Parameter

Description

This operator finds the optimal values for a set of parameters using an evolutionary approach which is often more appropriate than a grid search (as in the Optimize Parameters (Grid) operator) or a greedy search (as in the Optimize Parameters (Quadratic) operator) and leads to better results. This is a nested operator i.e. it has a subprocess. It executes its subprocess for a multiple number of times to find optimal values for the specified parameters.

This operator delivers the optimal parameter values through the parameter port which can also be written into a file with the Write Parameters operator. This parameter set can be read in another process using the Read Parameters operator. The performance vector for optimal values of parameters is delivered through the performance port. Any additional results of the subprocess are delivered through the result ports.

Other parameter optimization schemes are also available in RapidMiner. The Optimize Parameters (Evolutionary) operator might be useful if the best ranges and dependencies are not known at all. Another operator which works similar to this parameter optimization operator is the Loop Parameters operator. In contrast to the optimization operators, this operator simply iterates through all parameter combinations. This might be especially useful for plotting purposes.

Differentiation

Optimize Parameters (Grid) The Optimize Parameters (Grid) operator executes its subprocess for all combinations of the selected values of the parameters and then delivers the optimal parameter values. See page 21 for details.
1. Process Control

Input Ports

**input** *(inp)* This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of this operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

**performance** *(per)* This port delivers the Performance Vector for the optimal values of the selected parameters. A Performance Vector is a list of performance criteria values.

**parameter** *(par)* This port delivers the optimal values of the selected parameters. This optimal parameter set can be written into a file with the Write Parameters operator. The written parameter set can be read in another process using the Read Parameters operator.

**result** *(res)* Any additional results of the subprocess are delivered through the *result* ports. This operator can have multiple outputs. When one *result* port is connected, another *result* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *result* port of the subprocess is delivered at the first *result* port of the operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

Parameters

**edit parameter settings** *(menu)* The parameters are selected through the *edit parameter settings* menu. You can select the parameters and their possible values.
through this menu. This menu has an **Operators** window which lists all the operators in the subprocess of this operator. When you click on any operator in the **Operators** window, all parameters of that operator are listed in the **Parameters** window. You can select any parameter through the arrow keys of the menu. The selected parameters are listed in the **Selected Parameters** window. Only those parameters should be selected for which you want to find optimal values. This operator finds optimal values of the parameters in the specified range. The range of every selected parameter should be specified. When you click on any selected parameter (parameter in the **Selected Parameters** window) the **Grid/Range** option is enabled. This option allows you to specify the range of values of the selected parameters. The **Min** and **Max** fields are for specifying the lower and upper bounds of the range respectively. The **steps** and **scale** options are disabled for this operator.

**max generations (integer)** This parameter specifies the number of generations after which the algorithm should be terminated.

**use early stopping (boolean)** This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improvement (integer)** This parameter is only available when the **use early stopping** parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after \(n\) generations without improvement in the performance. \(n\) is specified by this parameter.

**specify population size (boolean)** This parameter specifies the size of the population. If it is not set to true, one individual per example of the given ExampleSet is used.

**population size (integer)** This parameter is only available when the **specify population size** parameter is set to true. This parameter specifies the population size i.e. the number of individuals per generation.

**keep best (boolean)** This parameter specifies if the best individual should survive. This is also called elitist selection. Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection.

**mutation type (selection)** This parameter specifies the type of the mutation operator.

**selection type (selection)** This parameter specifies the selection scheme of this evolutionary algorithms.
1. Process Control

tournament fraction (real) This parameter is only available when the selection type parameter is set to 'tournament'. It specifies the fraction of the current population which should be used as tournament members.
crossover prob (real) The probability for an individual to be selected for crossover is specified by this parameter.
use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.
local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.
show convergence plot (boolean) This parameter indicates if a dialog with a convergence plot should be drawn.

Related Documents

Optimize Parameters (Grid) (21)

Tutorial Processes

Finding optimal values of parameters of the SVM operator through the Optimize Parameters (Evolutionary) operator

The 'Weighting' data set is loaded using the Retrieve operator. The Optimize Parameters (Evolutionary) operator is applied on it. Have a look at the Edit Parameter Settings parameter of the Optimize Parameters (Evolutionary) operator. You can see in the Selected Parameters window that the C and gamma parameters of the SVM operator are selected. Click on the SVM.C parameter in the Selected Parameters window, you will see that the range of the C parameter is set from 0.001 to 100000. Now, click on the SVM.gamma parameter in the Se-
1.1. Parameter

In the selected Parameters window, you will see that the range of the \textit{gamma} parameter is set from 0.001 to 1.5. In every iteration of the subprocess, the value of the \textit{C} and/or \textit{gamma} parameters of the SVM (LibSVM) operator is changed in search of optimal values.

Have a look at the subprocess of the Optimize Parameters (Evolutionary) operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

The log parameter of the Log operator stores five things.

1. The iterations of the Optimize Parameters (Evolutionary) operator are counted by the apply-count of the SVM operator. This is stored in a column named 'Count'.

2. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'.

3. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'.

4. The value of the \textit{C} parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'.

5. The value of the \textit{gamma} parameter of the SVM (LibSVM) operator is stored in a column named 'SVM gamma'.

Also note that the stored information will be written into a file as specified in the filename parameter.

At the end of the process, the Write Parameters operator is used for writing the optimal parameter set in a file. This file can be read using the Read Parameters operator.
1. Process Control

operator to use these parameter values in another process.

Run the process and turn to the Results Workspace. You can see that the optimal parameter set has the following values: SVM.C = 56462 and SVM.gamma = 0.115 approximately. Now have a look at the values saved by the Log operator to verify these values. Switch to Table View to see the stored values in tabular form. You can see that the minimum Testing Error is 0.064 (in 20th iteration). The values of the $C$ and $\gamma$ parameters for this iteration are the same as given in the optimal parameter set.

Loop

This operator iterates over its subprocess for a specified number of times. The subprocess can use a macro that increments after every iteration.

Description

The subprocess of the Loop operator executes $n$ number of times where $n$ is the value of the iterations parameter. Optionally, a macro can be generated for the loop that increments after every iteration. The set iteration macro parameter should be set to true to define the iteration macro. The name and the start
value of the macro can be specified by the *macro name* and *macro start value* parameters respectively. To avoid the loop from executing for a very long period of time the *limit time* parameter can be used to define a timeout interval. The loop stops the execution as soon as the timeout interval is completed (if the process is not already finished). You need to have basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

The results of the subprocess are collected and returned as a collection of objects. You can access the elements of this collection either with an operator that has a flexible number of ports like the Append Operator or with specialized operators for accessing collections. They can be found at 'Process Control / Collections' in the Operators Window.

This operator can be considered to be one of the most general looping operators. Most looping operators loop on a certain criteria and they are optimized for that purpose. For example the Loop Examples and the Loop Attributes operators loop on examples and selected attributes respectively. The Loop operator should be used if your required functionality is not present in another more specific looping operator. For example for looping on all examples of an ExampleSet the Loop Examples operator should be used instead of the Loop operator. Though the Loop operator can also be used for iterating on all examples but the Loop Examples operator facilitates in this task. The attached Example Process describes how the Loop operator can be used for looping over all examples.

**Input Ports**

*input* (*inp*) This operator can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *input* port of this operator is available at the first *input* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order.
1. Process Control

Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

output *(out)* The Loop operator can have multiple output ports. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first output port of the subprocess is delivered at the first output of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Parameters

set iteration macro *(boolean)* This parameter specifies if a macro should be defined for the loop. The macro value will increment after every iteration. The name and start value of the macro can be specified by the macro name and macro start value parameters respectively.

macro name *(string)* This parameter is only available when the set iteration macro parameter is set to true. This parameter specifies the name of the macro.

macro start value *(integer)* This parameter is only available when the set iteration macro parameter is set to true. This parameter specifies the starting value of the macro. The value of the macro increments after every iteration of the loop.

iterations *(integer)* This parameter specifies the required number of iterations of the subprocess of the Loop operator.

limit time *(boolean)* This parameter gives the option of a timeout. If checked, the loop will be aborted after a specified interval of time (if it has not finished before this interval).

timeout *(integer)* This parameter is only available when the limit time is set to true. This parameter specifies the timeout in minutes.
Tutorial Processes

Subtracting the average of an attribute from all examples

This Example Process describes how the Loop operator can be used for looping over all examples. The Loop Examples operator should be preferred for this task. This process does exactly what the Example Process of the Loop Examples operator does. The only difference is that this process uses the Loop operator instead of the Loop Examples operator. It can be easily seen that using the Loop Examples operator is much easier because it has been designed for looping examples.

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Extract Macro operator is applied on it. The Extract Macro operator generates a macro named 'avg_temp' which stores the average value of the Temperature attribute of the 'Golf' data set. The Extract Macro operator is applied again to store the number of examples in the 'Golf' data set in a macro named 'count'. The Loop operator is applied next. The Loop operator iterates over all the examples of the 'Golf' data set and sets the value of the Temperature attribute in each example to the difference of the average value (stored in the 'avg_temp' macro) from the existing value of the Temperature attribute. The set iteration macro parameter is set to true for defining a macro that can be accessed in the subprocess and that increments after every iteration. The macro name parameter of the Loop operator is set to 'example' and the macro start value parameter of the Loop operator is set to 1. The iterations parameter is set to '%{count}', thus the subprocess executes $n$ times where $n$ is the value of the 'count' macro which is equal to the number of examples in the ExampleSet.

Have a look at the subprocess of the Loop operator. The Filter Example Range operator is applied first to filter the required example from the ExampleSet. The first example and last example parameters are set to '%{example}1' to filter one example of the ExampleSet. The resultant ExampleSet is composed of just a
single example i.e. the example of the current iteration. The Extract Macro operator is applied next to store the value of the Temperature attribute of the example of the current iteration in a macro named 'temp'. Then the Generate Macro operator is applied to generate a new macro from the 'temp' macro. The name of the new macro is 'new_temp' and it stores the difference of the current temperature value (stored in the 'temp' macro) and the average temperature value (stored in the 'avg_temp' macro). Finally the Set Data operator is applied. The value parameter is set to '%{new_temp}' to store the value of the 'new_temp' macro in the current example.

This subprocess is executed once for each example and in every iteration it replaces the value of the Temperature attribute of the example of that iteration with the value of the 'new_temp' macro. As a result of this operator a collection of ExampleSets is generated. Each ExampleSet is composed of a single example. The Append operator is used in the main process to unite all these ExampleSets into a single ExampleSet. The resultant ExampleSet can be seen in the Results Workspace. You can see that all values of the Temperature attribute have been replaced with new values.

Loop Attributes

This operator selects a subset (one or more attributes) of the input
ExampleSet and iterates over its subprocess for all the selected attributes. The subprocess can access the attribute of current iteration by a macro.

**Description**

The Loop Attributes operator has a number of parameters that allow you to select the required attributes of the input ExampleSet. Once the attributes are selected, the Loop Attributes operator applies its subprocess for each attribute i.e. the subprocess executes $n$ number of times where $n$ is the number of selected attributes. In all iterations the attribute of the current iteration can be accessed using the macro specified in the *iteration macro* parameter. You need to have basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

**Input Ports**

*example set (exa)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

*example set (exa)* The resultant ExampleSet is delivered through this port.
1. Process Control

Parameters

**attribute filter type (selection)** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet, no attributes are removed. This is the default option.

- **single** This option allows the selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows the selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for the attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example `real` and `integer` types both belong to the `numeric` type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the `value_type` option. This option allows the selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example `value_series_start` and `value_series_end` block types both belong to the `value_series` block type. When this option is selected some other parame-
1.2. Loop

ters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the attribute parameter if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes that will make it to the output port; all other attributes will be removed.

**regular expression** *(string)* The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** *(boolean)* If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in regular expression parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from
1. Process Control

a drop down list. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. the value type parameter's value. One of the following types can be selected here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' >', ' = ' and ' < ' e.g. ' < 5' will not work, so use ' < 5' instead.

**include special attributes (boolean)** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are delivered to the output port ir-
1.2. Loop

respective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are removed and previously removed attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

iteration macro (string) This parameter specifies the name of the macro which holds the name of the current attribute in each iteration.

Tutorial Processes

Generating new attributes in the Loop Attributes operator

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Loop Attributes operator. Have a look at the parameters of the Loop Attributes operator. The attribute filter type parameter is set to 'value type' and the value type parameter is set to 'numeric' and the include special attributes parameter is set to true. Thus all numeric attributes are selected from the 'Golf' data set i.e. the Temperature and Humidity attributes are selected. Therefore the subprocess of the Loop Attributes operator will iterate twice. In each iteration the current attribute can be accessed by the 'loop_attribute' macro defined by the iteration macro parameter. Now have a look at the subprocess of the Loop Attributes operator. The Extract Macro operator is applied first. The parameters of the Extract Macro operator are adjusted such that the 'avg' macro holds the average or mean of the attribute of the current iteration. Please note how the 'loop_attribute' macro is used in parameters of the Extract Macro operator. Next, the Generate Attributes operator is applied. It generates a new attribute from the attribute of the current iteration. The new attribute holds the devia-
1. Process Control

ation of examples from the mean of that attribute. The mean was stored in the 'avg' macro. Please note carefully the use of macros in the function descriptions parameter of the Generate Attributes operator.

Thus the subprocess of the Loop Attributes operator executes twice, once for each value of selected attributes. In the first iteration a new attribute is created with the name 'Deviation(Temperature)' which holds the deviations of the Temperature values from the mean of the Temperature attribute. In the second iteration a new attribute is created with the name 'Deviation(Humidity)' which holds the deviations of the Humidity values from the mean of the Humidity attribute.

Loop Values

This operator iterates over its subprocess for all the possible values of the selected attribute. The subprocess can access the attribute value of the current iteration by a macro.

Description

The Loop Values operator has a parameter named attribute that allows you to select the required attribute of the input ExampleSet. Once the attribute is selected, the Loop Values operator applies its subprocess for each possible value of the selected attribute i.e. the subprocess executes $n$ number of times where
1.2. Loop

$n$ is the number of possible values of the selected attribute. In all iterations the attribute value of the current iteration can be accessed using the macro specified in the *iteration macro* parameter. You need to have basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

It is important to note that the subprocess of the Loop Values operator executes for all possible values of the selected attribute. Suppose the selected attribute has three possible values and the ExampleSet has 100 examples. The Loop Values operator will iterate only three times (not 100 times); once for each possible value of the selected attribute. This operator is usually applied on nominal attributes.

**Input Ports**

example set *(exa)* This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

output *(out)* The Loop Values operator can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at first output port of subprocess is delivered at first output of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.
1. Process Control

Parameters

attribute *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *attribute* parameter if the meta data is known.

iteration macro *(string)* This parameter specifies the name of the macro which holds the current value of the selected attribute in each iteration.

Tutorial Processes

The use of the Loop Values operator in complex preprocessing

This Example Process is also part of the RapidMiner tutorial. It is included here to show the usage of the Loop Values operator in complex preprocessing. This process will cover a number of concepts of macros including redefining macros, the macro of the Loop Values operator and the use of the Extract Macro operator. This process starts with a subprocess which is used to generate data. What is happening inside this subprocess is not relevant to the use of macros, so it is not discussed here. A *breakpoint* is inserted after this subprocess so that you can view the ExampleSet. You can see that the ExampleSet has 12 examples and 2 attributes: 'att1' and 'att2'. 'att1' is nominal and has 3 possible values: 'range1', 'range2' and 'range3'. 'att2' has real values.

The Loop Values operator is applied on the ExampleSet. The *attribute* parameter is set to 'att1' therefore the Loop Values operator iterates over the values of the specified attribute (i.e. att1) and applies the inner operators on the given ExampleSet while the current value can be accessed via the macro defined by the *iteration macro* parameter which is set to 'loop_value', thus the current value can be accessed by specifying *%{loop_value}* in the parameter values. As att1 has 3 possible values, Loop Values will iterate 3 times, once for each possible value of
att1.

Here is an explanation of what happens inside the Loop Values operator. It is provided with an ExampleSet as input. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter' and the parameter string is set to 'att1 = %{loop_value}'. Note the use of the loop_value macro here. Only those examples are selected where the value of att1 is equal to the value of the loop_value macro. A breakpoint is inserted here so that you can view the selected examples. Then the Aggregation operator is applied on the selected examples. This average value is stored in a new ExampleSet in the attribute named 'average(att2)'. A breakpoint is inserted here so that you can see the average of the att2 values of the selected examples. The Extract Macro operator applied on this new ExampleSet to store this average value in a macro named 'current_average'. The originally selected examples are passed to the Generate Attributes operator that generates a new attribute named 'att2_abs_avg' which is defined by the expression 'abs(att2 - %{current_average})'. Note the use of the current_average macro here. Its value is subtracted from all values of att2 and stored in a new attribute named 'att2_abs_avg'. The Resultant ExampleSet is delivered at the output of the Loop Values operator. A breakpoint is inserted here so that you can see the ExampleSet with the 'att2_abs_avg' attribute. This output is fed to the Append operator in the main process. It merges the results of all the iterations into a single ExampleSet which is visible at the end of this process in the Results Workspace.

Here is what you see when you run the process.

- ExampleSet generated by the first Subprocess operator. Then the process enters the Loop Value operator and iterates 3 times.

Iteration 1:

- ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range1'
- Average of 'att2' values for the selected examples. The average is -1.161.
1. Process Control

- ExampleSet with 'att2_abs_avg' attribute for iteration 1.

Iteration 2:

- ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range2'

- Average of 'att2' values for the selected examples. The average is -1.656.

- ExampleSet with 'att2_abs_avg' attribute for iteration 2.

Iteration 3:

- ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range3'

- Average of 'att2' values for the selected examples. The average is 1.340.

- ExampleSet with 'att2_abs_avg attribute' for iteration 3.

Now the process comes out of the Loop Values operator and the Append operator merges the final ExampleSets of all three iterations into a single ExampleSet that you can see in the Results Workspace.

Loop Examples

This operator iterates over its subprocess for all the examples of the given ExampleSet. The subprocess can access the index of the example of the current iteration by a macro.
Description

The subprocess of the Loop Examples operator executes \( n \) number of times where \( n \) is the total number of examples in the given ExampleSet. In all iterations, the index of the example of the current iteration can be accessed using the macro specified in the *iteration macro* parameter. You need to have a basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for a basic understanding of macros. The Extract Macro operator is also used in the attached Example Process. For more information regarding subprocesses please study the Subprocess operator.

One important thing to note about this operator is the behavior of the *example set* output port of its subprocess. The subprocess is given the ExampleSet provided at the outer *example set* input port in the first iteration. If the *example set* output port of the subprocess is connected the ExampleSet delivered here in the last iteration will be used as input for the following iteration. If it is not connected the original ExampleSet will be delivered in all iterations.

It is important to note that the subprocess of the Loop Examples operator is executed for all examples of the given ExampleSet. If you want to iterate for possible values of a particular attribute please use the Loop Values operator. The subprocess of the Loop Values operator is executed for all possible values of the selected attribute. Suppose the selected attribute has three possible values and the ExampleSet has 100 examples. The Loop Values operator will iterate only three times (not 100 times); once for each possible value of the selected attribute. The Loop Examples operator, on the other hand, will iterate 100 times on this ExampleSet.

Input Ports

**example set** *(exa)* This input port expects an ExampleSet. It is the output of the Extract Macro operator in the attached Example Process. The output of other operators can also be used as input.
1. Process Control

Output Ports

d example set (exa) The ExampleSet that is connected at the example set output port of the inner subprocess is delivered through this port. If no ExampleSet is connected then the original ExampleSet is delivered.

d output (out) The Loop Examples operator can have multiple output ports. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first output port of the subprocess is delivered at the first output of the outer process. Don’t forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Parameters

d iteration macro (string) This parameter specifies the name of the macro which holds the index of the example of the current iteration in each iteration.

Tutorial Processes

Subtracting the average of an attribute from all examples

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Extract Macro operator is applied on it. The Extract Macro operator generates a macro named 'avg_temp' which stores the average value of the Temperature attribute of the 'Golf' data set. The Loop Examples operator is applied next. The Loop Examples operator iterates over all the examples of the 'Golf' data set and sets the value of the Temperature attribute in each example to the difference of the average value (stored in 'avg_temp' macro) from the existing value of the Temperature
1.2. Loop

attribute. The *iteration macro* parameter of the Loop Examples operator is set to 'example'. Thus in each iteration of the subprocess the index of the current example can be accessed by the 'example' macro.

Have a look at the subprocess of the Loop Examples operator. The Extract Macro operator is applied first to store the value of the Temperature attribute of the example of the current iteration in a macro named 'temp'. Then the Generate Macro operator is applied to generate a new macro from the 'temp' macro. The name of the new macro is 'new_temp' and it stores the difference of the current temperature value (stored in 'temp' macro) and the average temperature value (stored in 'avg_temp' macro). Finally the Set Data operator is applied. The *example index* parameter is set to '%{example}' to access the example of the current iteration. The *value* parameter is set to '%{new_temp}' to store the value of the 'new_temp' macro in the current example.

This subprocess is executed once for each example and in every iteration it replaces value of the Temperature attribute of the example of that iteration with the value of 'new_temp' macro. The resultant ExampleSet can be seen in the Results Workspace. You can see that all values of the Temperature attribute have been replaced with new values.
1. Process Control

Loop Clusters

This operator iterates over its subprocess for each cluster in the input ExampleSet. In each iteration the subprocess receives examples belonging to the cluster of that iteration.

Description

The Loop Clusters operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop Clusters operator executes \( n \) number of times, where \( n \) is the number of clusters in the given ExampleSet. It is compulsory that the given ExampleSet should have a cluster attribute. Numerous clustering operators are available in RapidMiner that generate a cluster attribute e.g. the K-Means operator. The subprocess executes on examples of one cluster in an iteration, on examples of the next cluster in next iteration and so on. Please study the attached Example Process for better understanding.

Input Ports

**example set** *(exa)* This input port expects an ExampleSet. It is compulsory that the ExampleSet should have an attribute with cluster role. It is output of the K-Means operator in the attached Example Process.

**in** *(in)* This operator can have multiple *in* input ports. When one input is connected, another *in* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object delivered at first *in* port of the operator is available at first *in* port of the subprocess. Don't forget to connect all outputs in correct order. Make sure that you have connected right number of ports at all levels of chain.
Output Ports

out (out) This operator can have multiple out output ports. When one output is connected, another out output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at first out port of subprocess is delivered at first out output port of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected right number of ports at all levels of chain.

Tutorial Processes

Introduction to the Loop Clusters operator

The 'Ripley-Set' data set is loaded using the Retrieve operator. The K-Means operator is applied on it for generating a cluster attribute. A breakpoint is inserted here so that you can have a look at the clustered ExampleSet. You can see that there is an attribute with cluster role. It has two possible values i.e. cluster_0 and cluster_1. This means that there are two clusters in the ExampleSet. The Loop Clusters operator is applied next. The subprocess of the Loop Clusters operator executes twice; once for each cluster. Have a look at the subprocess of the Loop Clusters operator. The Log operator is applied in the subprocess. A breakpoint is inserted before the Log operator so that you can see the examples of each iteration. In first iteration you will see that all examples belong to cluster_1 while in the second iteration all examples belong to cluster_0.
1. Process Control

Loop Data Sets

This operator iterates over its subprocess for every ExampleSet given at its input ports.

Description

The subprocess of the Loop Data Sets operator executes $n$ number of times where $n$ is the number of ExampleSets provided as input to this operator. You must have basic understanding of Subprocesses in order to understand this operator. For more information regarding subprocesses please study the Subprocess operator. For each input ExampleSet the Loop Data Sets operator executes the inner operators of the subprocess like an operator chain. This operator can be used to conduct a process consecutively on a number of different data sets. If the only best parameter is set to true then only the results generated during the iteration with best performance are delivered as output. For this option it is compulsory to attach a performance vector to the performance port in the subprocess of this operator. The Loop Data Sets operator uses this performance vector to select the iteration with best performance.

Input Ports

example set (exa) This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another ExampleSet (if any). The order of inputs remains the same. The ExampleSet supplied at the first input port of this operator is available at the first input port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.
Output Ports

output (out) The Loop Data Sets operator can have multiple output ports. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first output port of the subprocess is delivered at the first output of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Parameters

only best (boolean) If the only best parameter is set to true then only the results generated during the iteration with best performance are delivered as output. For this option it is compulsory to attach a performance vector to the performance port in the subprocess of this operator. The Loop Data Sets operator uses this performance vector to select the iteration with best performance.

Tutorial Processes

Selecting the ExampleSet with best performance

This Example Process explains the usage of the only best parameter of the Loop Data Sets operator. 'Golf', 'Golf-Testset' and 'Iris' data sets are loaded using the Retrieve operator. All these ExampleSets are provided as input to the Loop Data Sets operator. Have a look at the subprocess of the Loop Data Sets operator. The Split Validation operator is used for training and testing a K-NN model on the given ExampleSet. The Split Validation operator returns the performance vector of the model. This performance vector is used by the Loop Data Sets operator for finding the iteration with the best performance. The results of the
1. Process Control

iteration with best performance are delivered because the only best parameter is set to true.

When this process is executed, the 'Iris' data set is delivered as result. This is because the iteration with the 'Iris' data set had the best performance vector. If you insert a breakpoint after the Split Validation operator and run the process again, you can see that the 'Golf', 'Golf-Testset' and 'Iris' data sets have 25%, 50% and 93.33% accuracy respectively. As the iteration with 'Iris' data set had the best performance therefore its results are returned by this operator (remember only best parameter is set to true). This operator can also return other objects like model etc.

Loop and Average

This operator iterates over its subprocess the specified number of times and delivers the average of the inner results.
1.2. Loop

Description

The Loop and Average operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop and Average operator executes \( n \) number of times, where \( n \) is the value of the \textit{iterations} parameter specified by the user. The subprocess of this operator must always return a performance vector. These performance vectors are averaged and returned as result of this operator. For more information regarding subprocesses please study the Subprocess operator.

Differentiation

\textbf{Loop and Deliver Best} This operator iterates over its subprocess the specified number of times and delivers the results of the iteration that has the best performance. See page ?? for details.

Input Ports

\textit{in} (\textit{in } ) This operator can have multiple inputs. When one input is connected, another \textit{in} port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first \textit{in} port of this operator is available at the first \textit{in} port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

\textbf{averagable} (\textit{ave}) This operator can have multiple \textbf{averagable} output ports. When one output is connected, another \textbf{averagable} output port becomes available which is ready to deliver another output (if any). The order of outputs remains
1. Process Control

the same. The Average Vector delivered at the first *averagable* port of the subprocess is delivered at the first *averagable* output port of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

**Parameters**

*iterations* (*integer*) This parameter specifies the number of iterations of the subprocess of this operator.

*average performances only* (*boolean*) This parameter indicates if only performance vectors or all types of averagable result vectors should be averaged.

**Related Documents**

Loop and Deliver Best (??)

**Tutorial Processes**

Taking average of performance vectors

The 'Golf' data set is loaded using the Retrieve operator. The Loop And Average operator is applied on it. The *iterations* parameter is set to 3; thus the subprocess of the Loop And Average operator will be executed three times. A performance vector will be generated in every iteration and the average of these performance vectors will be delivered as the result of this operator.

Have a look at the subprocess of the Loop And Average operator. The Split Validation operator is used for training and testing a Naive Bayes model. A *breakpoint* is inserted after the Split Validation operator so that the performance
vector can be seen in each iteration. Run the process. You will see the performance vector of the first iteration. It has 25% accuracy. Keep continuing the process; you will see the performance vectors of the second and third iterations (with 75% and 100% accuracy respectively). The Loop and Average operator takes the average of these three results and delivers it through its output port. The average of these three results is 66.67% (i.e. \( \frac{25\% + 75\% + 100\%}{3} \)). The resultant average vector can be seen in the Results Workspace.

Loop Parameters

This operator iterates over its subprocess for all the defined parameter combinations. The parameter combinations can be set by the wizard provided in parameters.

Description

The Loop Parameters operator has a subprocess in it. It executes the subprocess for all combinations of selected values of the parameters. This can be very useful for plotting or logging purposes and sometimes for simply configuring the parameters for the inner operators as a sort of meta step (e.g. learning curve generation). Any results of the subprocess are delivered through the result ports.
1. Process Control

The entire configuration of this operator is done through the *edit parameter settings* parameter. Complete description of this parameter is described in the parameters section.

Please note that this operator has two modes: synchronized and non-synchronized which depend on the setting of the *synchronize* parameter. In the latter, all parameter combinations are generated and the subprocess is executed for each combination. In the synchronized mode, no combinations are generated but the set of all pairs of the increasing number of parameters are used. For the iteration over a single parameter there is no difference between both modes. Please note that the number of parameter possibilities must be the same for all parameters in the synchronized mode.

If the *synchronize* parameter is not set to true, selecting a large number of parameters and/or large number of steps (or possible values of parameters) results in a huge number of combinations. For example, if you select 3 parameters and 25 steps for each parameter then the total number of combinations would be above 390625 (i.e. 25 x 25 x 25). The subprocess is executed for all possible combinations. Running a subprocess for such a huge number of iterations will take a lot of time. So always carefully limit the parameters and their steps.

Differentiation

**Optimize Parameters (Grid)** The Optimize Parameters (Grid) operator executes the subprocess for all combinations of selected values of the parameters and then delivers the optimal parameter values. The Loop Parameters operator, in contrast to the optimization operators, simply iterates through all parameter combinations. This might be especially useful for plotting purposes. See page 21 for details.
1.2. Loop

Input Ports

**input** (*inp*) This operator can have multiple inputs. When one input is connected, another **input** port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first **input** port of this operator is available at the first **input** port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

**result** (*res*) Any results of the subprocess are delivered through the **result** ports. This operator can have multiple outputs. When one **result** port is connected, another **result** port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first **result** port of the subprocess is delivered at the first **result** port of the operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

Parameters

**edit parameter settings** (*menu*) The parameters are selected through the **edit parameter settings** menu. You can select the parameters and their possible values through this menu. This menu has an **Operators** window which lists all the operators in the subprocess of this operator. When you click on any operator in the **Operators** window, all parameters of that operator are listed in the **Parameters** window. You can select any parameter through the arrow keys of the menu. The selected parameters are listed in the **Selected Parameters** window. Only those parameters should be selected for which you want to iterate the subprocess. This operator iterates through parameter values in the specified range. The range of
1. Process Control

every selected parameter should be specified. When you click on any selected pa-
parameter (parameter in Selected Parameters window) the Grid/Range and Value
List option is enabled. These options allow you to specify the range of values of
the selected parameters. The Min and Max fields are for specifying the lower and
upper bounds of the range respectively. As all values within this range cannot be
checked, the steps field allows you to specify the number of values to be checked
from the specified range. Finally the scale option allows you to select the pattern
of these values. You can also specify the values in form of a list.

synchronize (boolean) This operator has two modes: synchronized and non-
synchronized which depend on the setting of the synchronize parameter. If the
synchronize parameter is set to false, all parameter combinations are generated
and the inner operators are applied for each combination. If the synchronize pa-
parameter is set to true, no combinations are generated but the set of all pairs of the
increasing number of parameters are used. For the iteration over a single param-
eter there is no difference between both modes. Please note that the number of
parameter possibilities must be the same for all parameters in the synchronized
mode.

Related Documents

Optimize Parameters (Grid) (21)

Tutorial Processes

Iterating through the parameters of the SVM operator

The 'Weighting' data set is loaded using the Retrieve operator. The Loop Pa-
rameters operator is applied on it. Have a look at the Edit Parameter Settings
parameter of the Loop Parameters operator. You can see in the Selected Parame-
ters window that the C and gamma parameters of the SVM operator are selected.
Click on the \textit{SVM.C} parameter in the \textit{Selected Parameters} window, you will see that the range of the \textit{C} parameter is set from 0.001 to 100000. 11 values are selected (in 10 steps) logarithmically. Now, click on the \textit{SVM.gamma} parameter in the \textit{Selected Parameters} window, you will see that the range of the \textit{gamma} parameter is set from 0.001 to 1.5. 11 values are selected (in 10 steps) logarithmically. There are 11 possible values of 2 parameters, thus there are 121 (i.e. 11 x 11) combinations. The subprocess will be executed for all combinations of these values because the \textit{synchronize} parameter is set to false, thus it will iterate 121 times. In every iteration, the value of the \textit{C} and/or \textit{gamma} parameters of the SVM(LibSVM) operator is changed. The value of the \textit{C} parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration. Similarly, the value of the \textit{gamma} parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 1.5 in the last iteration.

Have a look at the subprocess of the Loop Parameters operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

The \textit{log} parameter of the Log operator stores five things.

1. The iterations of the Loop Parameters operator are counted by apply-count of the SVM operator. This is stored in a column named 'Count'.

2. The value of the \textit{classification error} parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'.

3. The value of the \textit{classification error} parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'.

4. The value of the \textit{C} parameter of the SVM (LibSVM) operator is stored in
1. Process Control

a column named 'SVM C'.

5. The value of the gamma parameter of the SVM (LibSVM) operator is stored in a column named 'SVM gamma'.

Also note that the stored information will be written into a file as specified in the filename parameter.

Run the process and turn to the Results Workspace. Now have a look at the values saved by the Log operator. Switch to Table View to see the stored values in tabular form.

Loop Files

This operator iterates over its subprocess for all the files in the specified directory and delivers file names, paths and paths of their parent as macro to the inner operators.

Description

The Loop Files operator is a nested operator i.e. it has a subprocess. The subprocess of the Loop Files operator executes for all the files in the specified directory. The directory is specified through the directory parameter. The file name, path and the path of the file's parent is available to the operators in the subprocess through the corresponding macros. If you want the subprocess operator to iterate on some specific files, say files with a particular extension,
you can use the filter parameter for specifying the desired condition in form of a regular expression. This operator can iterate through the subdirectories of the specified directory if the iterate over subdirs parameter is set to true. This operator is even capable of iterating recursively through all the subdirectories (and subdirectories of subdirectories and so on) if the recursive parameter is set to true. You need to have basic understanding of macros in order to apply this operator. Please study the documentation of the Extract Macro operator for basic understanding of macros. For more information regarding subprocesses please study the Subprocess operator.

Input Ports

input (inp) This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of the inputs remains the same. The Object supplied at the first input port of this operator is available at the first input port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

output (out) The Loop Files operator can have multiple output ports. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of the outputs remains the same. The Object delivered at the first output port of the subprocess is delivered at the first output of the outer process. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.
1. Process Control

Parameters

directory (filename) This parameter specifies the directory to iterate over.
filter (string) This parameter specifies a regular expression which is used as a filter for the file and directory names, e.g. 'a.*b' for all files starting with 'a' and ending with 'b'.
filtered string (selection) This parameter specifies which part of the file name and path is matched against the filter expression specified by the filter parameter.
file name macro (string) This parameter specifies the name of the macro which delivers the name (without path) of the current file. This macro can be accessed by the operators of the subprocess by using '%{macro_name}' syntax.
file path macro (string) This parameter specifies the name of the macro which delivers the absolute path and name of the current file. This macro can be accessed by the operators of the subprocess by using '%{macro_name}' syntax.
parent path macro (string) This parameter specifies the name of the macro which delivers the absolute path of the current file's directory. This macro can be accessed by the operators of the subprocess by using '%{macro_name}' syntax.
recursive (boolean) This parameter indicates if this operator should recursively iterate over all the files/directories of the subdirectories.
iterate over files (boolean) If this parameter is set to true, this operator will iterate over files in the specified directory and set their path and name macros.
iterate over subdirs (boolean) If this parameter is set to true, this operator will iterate over subdirectories in the given directory and set their path and name macros.

Tutorial Processes

Generating an ExampleSet with names of all files in a directory

This Example Process shows how the Loop Files operator can be used for it-
1.2. Loop

Iterating over files in a directory. You need to have at least a basic understanding of macros and logs in order to understand this process completely. The goal of this process is to simply provide the list of all files in the specified directory in form of an ExampleSet. This process starts with the Loop Files operator. All parameters are used with default values. The Log operator is used in the subprocess of the Loop Files operator to store the name of the files in the log table in every iteration. The Provide Macro As Log Value operator is used before the Log operator to make the file name macro of the Loop Files operator available to the Log operator. The name of the file name macro (specified through the file name macro parameter) is 'file_name', therefore the macro name parameter of the Provide Macro As Log Value operator is set to 'file_name'. After the execution of the Loop Files operator, names of all the files in the specified directory are stored in the log table. To convert this data into an ExampleSet, the Log to Data operator is applied. The resultant ExampleSet is connected to the result port of the process and it can be viewed in the Results Workspace. As the path of the RapidMiner repository was specified in the directory parameter of the Loop Files parameter, the ExampleSet has the names of all the files in your RapidMiner repository. You can see names of all the files including files with '.properties' and '.rmp' extensions. If you want only the file names with '.rmp' extension, you can set the filter parameter to '*.rmp'.
1. Process Control

**X-Prediction**

This operator works like the X-Validation operator but it returns a labeled ExampleSet instead of a performance vector.

**Description**

The X-Prediction operator is a nested operator. It has two subprocesses: a training subprocess and a testing subprocess. The training subprocess is used for training a model. The trained model is then applied in the testing subprocess. The testing subprocess returns a labeled ExampleSet. It is compulsory for the training subprocess to deliver a model and it is compulsory for the testing subprocess to deliver a labeled ExampleSet.

The input ExampleSet is partitioned into $k$ subsets of equal size. Of the $k$ subsets, a single subset is retained as the testing data set (i.e. input of the testing subprocess), and the remaining $k - 1$ subsets are used as training data set (i.e. input of the training subprocess). The model is trained on the training data set in the training subprocess and then it is applied on the testing data set in the testing subprocess. This process is repeated $k$ times, with each of the $k$ subsets used exactly once as the testing data. The value $k$ can be adjusted using the *number of validations* parameter.

**Differentiation**

**X-Validation** The X-Validation and X-Prediction operators work in the same way. The major difference is the objects returned by these operators. The X-Validation operator returns a performance vector whereas the X-Prediction operator returns a labeled ExampleSet. See page 896 for details.
1.2. Loop

Input Ports

**example set (exa)** This input port expects an ExampleSet for training a model. The same ExampleSet will be used during the testing subprocess for testing the model.

Output Ports

**labelled data (lab)** The labeled ExampleSet is delivered through this port. This is the major difference between the X-Prediction and X-Validation operators because the latter returns a performance vector.

Parameters

**leave one out (boolean)** As the name suggests, the *leave one out* parameter involves using a single example from the original ExampleSet as the testing data (in the testing subprocess), and the remaining examples as the training data (in the training subprocess). This is repeated so each example in the ExampleSet is used once as the testing data. Thus, it is repeated 'n' number of times, where 'n' is the total number of examples in the ExampleSet. This is the same as applying the X-Prediction operator with the *number of validations* parameter set equal to the number of examples in the original ExampleSet. This is usually very expensive for large ExampleSets from a computational point of view because the training process is repeated a large number of times (number of examples time). If this parameter is set to true, the *number of validations* parameter is ignored.

**number of validations (integer)** This parameter specifies the number of subsets the ExampleSet should be divided into (each subset has an equal number of examples). Also the same number of iterations will take place. Each iteration involves training a model and testing that model. If this parameter is set equal to the total number of examples in the ExampleSet, it is equivalent to the X-Prediction operator with the *leave one out* parameter set to true.
1. Process Control

sampling type (selection) The X-Prediction operator can use several types of sampling for building the subsets. Following options are available:

- **linear_sampling** The Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.

- **shuffled_sampling** The Shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.

- **stratified_sampling** The Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class labels.

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomizing examples of a subset. Using the same value of the local random seed will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

Related Documents

X-Validation (896)
1.2. Loop

Tutorial Processes

Introduction to the X-Prediction operator

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to uniquely identify examples. This is done so that you can understand this process easily; otherwise IDs are not required here. A breakpoint is added after this operator so that you can preview the data before the X-Prediction operator is applied. Double click the X-Prediction operator, you will see the training and testing subprocesses. The Decision Tree operator is used in the training subprocess. The trained model (i.e. Decision Tree) is passed to the testing subprocess through the model ports. The testing subprocess receives testing data from the testing port.

Now, have a look at the parameters of the X-Prediction operator. The number of validations parameter is set to 3 and the sampling type parameter is set to linear sampling. Remaining parameters have default values. The no of validations is set to 3, which implies that 3 subsets of the 'Golf' data set will be created. You will observe later that these three subsets are created:

- sub1: examples with IDs 1 to 5 (5 examples)
- sub2: examples with IDs 6 to 9 (4 examples)
- sub3: examples with IDs 10 to 14 (5 examples)

You can see that all examples in a subset are consecutive (i.e. with consecutive IDs). This is because linear sampling is used. Also note that all subsets have almost an equal number of elements. An exactly equal number of elements was not possible because 14 examples could not be divided equally in 3 subsets.

As the no of validations parameter is set to 3, there will be three iterations.

- Iteration 1: A model (decision tree) will be trained on sub2 and sub3 during the training subprocess. The trained model will be applied on sub1 during
1. Process Control

testing subprocess.

- **Iteration 2:** A model (decision tree) will be trained on sub1 and sub3 during the training subprocess. The trained model will be applied on sub2 during testing subprocess.

- **Iteration 3:** A model (decision tree) will be trained on sub1 and sub2 during the training subprocess. The trained model will be applied on sub3 during testing subprocess.

*Breakpoints* are inserted to make you understand the process. Here is what happens when you run the process:

- First the 'Golf' data set is displayed with all rows uniquely identified using the ID attribute. There are 14 rows with ids 1 to 14. Press the green-colored Run button to continue.

- Now a Decision tree is shown. This was trained on a subset (combination of sub2 and sub3) of the 'Golf' data set. Press the Run button to continue.

- The Decision Tree was applied on the testing data. Testing data for this iteration was sub1. Here you can see the results after application of the Decision Tree model. Have a look at the IDs of the testing data here. They are 1 to 5. This means that the tree was trained on the remaining examples i.e. examples with IDs 6 to 14 i.e. sub2 + sub3.

- Now you can see a different Decision tree. It was trained on another subset of 'Golf' data set that is why it is different from the previous decision tree. Press the Run button and you will see testing data for this tree. This process will repeat 3 times meaning three iterations because the *number of validations* parameter was set to 3.

- At the end of 3 iterations, you will see the complete labeled ExampleSet in the Results Workspace.

You can run the same process with different values of the *sampling type* parameter. If *linear sampling* is used, as in our example process, you will see that the IDs of the examples in the subsets will be consecutive values. If *shuffled sampling*
is used you will see that the IDs of the examples in the subsets will be randomized. If *stratified sampling* is used you will also see randomized IDs but the class distribution in the subsets will be nearly the same as in the whole 'Golf' data set.

### Branch

This operator consists of two subprocesses but it executes only one subprocess at a time depending upon the condition. This operator is similar to the 'if-then-else' statement, where one of the two options is selected depending upon the results of the specified condition. It is important to have understanding of subprocesses in order to use this operator effectively.

#### Description

The Branch operator tests the condition specified in the parameters (mostly through the *condition type* and *condition value* parameters) on the object supplied at the *condition* input port. If the condition is satisfied, the first subprocess i.e. the 'Then' subprocess is executed otherwise the second subprocess i.e. the 'Else' subprocess is executed.

It is very important to have a good understanding of use of subprocesses in RapidMiner to understand this operator completely. A subprocess introduces a
1. Process Control

process within a process. Whenever a subprocess is reached during a process execution, first the entire subprocess is executed. Once the subprocess execution is complete, the flow is returned to the process (the parent process). A subprocess can be considered as a small unit of a process, like in a process, all operators and combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied. For more detail about subprocesses please study the Subprocess operator.

Double-click on the Branch operator to go inside and view the subprocesses. The subprocesses are then shown in the same Process View. Here you can see two subprocesses: 'Then' and 'Else' subprocesses. The 'Then' subprocess is executed if the condition specified in the parameters results true. The 'Else' subprocess is executed if the condition specified in the parameters results false. To go back to the parent process, click the blue-colored up arrow button in the Process View toolbar. This works like files and folders work in operating systems. Subprocesses can have subprocesses in them just like folders can have folders in them.

The Branch operator is similar to the Select Subprocess operator because they both have multiple subprocesses but only one subprocess is executed at a time. The Select Subprocess operator can have more than two subprocesses and the subprocess to be executed is specified in the parameters. On the contrary, The Branch operator has only two subprocesses and the subprocess to be executed depends upon the result of the condition specified in the parameters. The condition is specified through the condition type and condition value parameters. Macros can be provided in the condition value parameter. Thus the subprocess to be executed can be controlled by using macros. If this operator is placed in any Loop operator this operator will be executed multiple number of times. True power of this operator comes into play when it is used with other operators like various Macro and Loop operators. For example, if this operator is placed in any Loop operator and the condition value parameter is controlled by a macro then this operator can be used to dynamically change the process setup. This might be useful in order to test different layouts.
1.3. Branch

Input Ports

**condition** \((con)\) Any object can be supplied at this port. The condition specified in the parameters is tested on this object. If the condition is satisfied the 'Then' subprocess is executed otherwise the 'Else' subprocess is executed.

**input** \((inp)\) The Branch operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first input port of the Branch operator is available at the first input port of the nested chain (inside the subprocess). Don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Output Ports

**input** \((inp)\) The Branch operator can have multiple outputs. When one output is connected, another input port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first input port of subprocess is delivered at the first input of the Branch operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Parameters

**condition type** \((selection)\) The type of condition is selected through this parameter.

**condition value** The value of the selected condition type is specified through this parameter. The **condition type** and **condition value** parameters together specify the condition statement. This condition will be tested on the object provided at the **condition** input port.

**io object** \((selection)\) This parameter is only available when the **condition type**
1. Process Control

Parameter is set to 'input exists'. This parameter specifies the class of the object which should be checked for existence.

**Return inner output (boolean)** This parameter indicates if the outputs of the inner subprocess should be delivered through this operator.

Tutorial Processes

Applying different subprocesses on Golf data set depending upon the performance value

The 'Golf' data set is loaded using the Retrieve operator. The Default Model operator is applied on it. The resultant model is applied on the 'Golf-Testset' data set through the Apply Model operator. The performance of this model is measured by the Performance operator. A breakpoint is inserted here so that you can have a look at this performance vector. You can see that its accuracy value is 64.29%. It is provided at the condition port of the Branch operator. Thus the condition specified in the parameters of the Branch operator will be tested on this performance vector. The 'Golf' data set is also provided to the Branch operator (through the first input port).

Now have a look at the subprocesses of the Branch operator. The 'Then' subprocess simply connects the condition port to the input port without applying any operator. Thus If the condition specified in the parameters is true, the condition object i.e. the performance vector will be delivered by the Branch operator. The 'Else' subprocess does not use the object at the condition port. Instead, it applies the K-NN operator on the object at the first input port i.e. the 'Golf' data set. Thus If the condition specified in the parameters is false, the K-NN operator will be applied on the object at the first input port i.e. the 'Golf' data set and the resultant model will be delivered by the Branch operator.

Now have a look at the parameters of the Branch operator. The condition type parameter is set to 'min performance value' and the condition value parameter is set to 70. Thus if the performance of the performance vector is greater than 70,
the condition will be true.

Overall in this process, The Default Model is trained on the 'Golf' data set, if its performance on the 'Golf-Testset' data set is more than 70% the performance vector will be delivered otherwise the K-NN model trained on the 'Golf' data set will be delivered.

Select Subprocess

This operator consists of multiple subprocesses but it executes only one subprocess at a time. This operator is similar to a switch, where numerous options exist but only one option is selected at a time. It is important to have a good understanding of subprocesses in order to use this operator effectively.
1. Process Control

Description

It is very important to have a good understanding of the use of subprocesses in RapidMiner to understand this operator completely. A subprocess introduces a process within a process. Whenever a subprocess is reached during a process execution, first the entire subprocess is executed. Once the subprocess execution is complete, flow is returned to the process (the parent process). A subprocess can be considered as a small unit of a process, like in a process, all operators and combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied. For more details about subprocesses please study the Subprocess operator.

Double-click on the Select Subprocess operator to go inside and view the subprocesses. The subprocesses are then shown in the same Process View. Here you can see the options to add or remove subprocesses. To go back to the parent process, click the blue-colored up arrow button in the Process View toolbar. This works like files and folders work in operating systems. Subprocesses can have subprocesses in them just like folders can have folders in them.

The Select Subprocess operator consists of multiple subprocesses but it executes only one subprocess at a time. The number of subprocesses can be easily controlled. You can easily add or remove subprocesses. The process to be executed is selected by the select which parameter. Macros can be provided in the select which parameter. Thus the subprocess to be executed can be controlled by using macros. If this operator is placed in any Loop operator this operator will be executed multiple number of times. The true power of this operator comes into play when it is used with other operators like various Macro and Loop operators. For example, if this operator is placed in any Loop operator and the select which parameter is controlled by a macro then this operator can be used to dynamically change the process setup. This might be useful in order to test different layouts, e.g. the gain by using different preprocessing steps or the quality of a certain learner.
Input Ports

input \((inp)\) The Select Subprocess operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first input port of the Select Subprocess operator is available at the first input port of the nested chain (inside the subprocess). Don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Output Ports

output \((out)\) The Select Subprocess operator can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first output port of subprocess is delivered at the first output of the Select Subprocess operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports at all levels of the chain.

Parameters

select which \((integer)\) This parameter indicates which subprocess should be applied. True power of this operator comes into play when the select which parameter is specified through a macro.
1. Process Control

Tutorial Processes

Applying different classification operators on Golf data set using the Select Subprocess operator

The 'Golf' data set is loaded using the Retrieve operator. The Select Subprocess operator is applied on it. Double-click on the Select Subprocess operator to see the subprocesses in it. As you can see, there are four subprocesses:

- Subprocess 1: The k-NN operator is applied on the input and the resulting model is passed to the output.
- Subprocess 2: The Naive Bayes operator is applied on the input and the resulting model is passed to the output.
- Subprocess 3: The Decision Tree operator is applied on the input and the resulting model is passed to the output.
- Subprocess 4: The input is directly connected to the output.

Only one of these subprocesses can be executed at a time. The subprocess to be executed can be controlled by the select which parameter. The select which parameter is set to 1, thus the first subprocess will be executed. When you run the process you will see the model created by the k-NN operator in the Results workspace. To execute the second subprocess set the select which parameter to 2 and run the process again. You will see the model generated by the Naive Bayes operator in the Results Workspace. To execute the third subprocess set the select which parameter to 3 and run the process again. You will see the model generated by the Decision Tree operator in the Results Workspace. To execute the fourth subprocess set the select which parameter to 4 and run the process again. Now you will see the 'Golf' data set in the Results Workspace because no operator was applied in the fourth subprocess.
1.4. Collections

Collect

This operator combines multiple input objects into a single collection.

Description

The Collect operator combines a variable number of input objects into a single collection. It is important to know that all input objects should be of the same IOObject class. In the Process View, collections are indicated by double lines. If the input objects are collections themselves then the output of this operator would be a collection of collections. However if the unfold parameter is set to true then the output will be the union of all elements of the input collections. After combining objects into a collection, the Loop Collection operator can be used to iterate over this collection. The Select operator can be used to retrieve the required element of the collection.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator will allow you to collect the required objects into a single collection, the Loop Collection operator will allow you to iterate over all collections and finally you can separate the input objects from collection...
1. Process Control

by individually selecting the required element by using the Select operator.

Input Ports

input (inp) This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first input port of the Collect operator becomes the first element of the resultant collection. It is important to note that all input objects should be of the same IOObject class.

Output Ports

collection (col) All the input objects are combined into a single collection and the resultant collection is delivered through this port.

Parameters

unfold (boolean) This parameter is only applicable when the input objects are collections. This parameter specifies whether collections received at the input ports should be unfolded. If the input objects are collections themselves and the unfold parameter is set to false, then the output of this operator would be a collection of collections. However if the unfold parameter is set to true then the output will be the union of all elements of the input collections.

Tutorial Processes

Introduction to collections
This Example Process explains a number of important ideas related to collections. This Example Process shows how objects can be collected into a collection, then some preprocessing is applied on the collection and finally individual elements of the collection are separated as required.

The 'Golf' and 'Golf-Testset' data sets are loaded using the Retrieve operator. Both ExampleSets are provided as inputs to the Subprocess operator. The subprocess performs some preprocessing on the ExampleSets and then returns them through its output ports. The first output port returns the preprocessed 'Golf' data set which is then used as training set for the Decision Tree operator. The second output port delivers the preprocessed 'Golf-Testset' data set which is used as testing set for the Apply Model operator which applies the Decision Tree model. The performance of this model is measured and it is connected to the results port. The training and testing ExampleSets can also be seen in the Results Workspace.

Now have a look at the subprocess of the Subprocess operator. First of all, the Collect operator combines the two ExampleSets into a single collection. Note the double line output of the Collect operator which indicates that the result is a collection. Then the Loop Collection operator is applied on the collection. The Loop Collection operator iterates over the elements of the collection and performs some preprocessing (renaming an attribute in this case) on them. You can see in the subprocess of the Loop Collection operator that the Rename operator is used for changing the name of the Temperature attribute to 'New Temperature'. It is important to note that this renaming is performed on both ExampleSets of the collection. The resultant collection is supplied to the Multiply operator which generates two copies of the collection. The first copy is used by the Select operator (with index parameter = 1) to select the first element of collection i.e. the preprocessed 'Golf' data set. The second copy is used by the second Select operator (with index parameter = 2) to select the second element of the collection i.e. the preprocessed 'Golf-Testset' data set.
1. Process Control

Select

This operator returns the specified single object from the given collection of objects.

Description

Operators like the Collect operator combine a variable number of input objects into a single collection. In the Process View, collections are indicated by double lines. The Select operator can be used for selecting an individual object from this collection. The index parameter specifies the index of the required object. If the objects of the given collection are collections themselves the output of this
operator would be the collection at the specified index. However if the *unfold* parameter is set to true the index refers to the index in the flattened list, i.e. the list obtained from the input list by replacing all nested collections by their elements.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator will allow you to collect the required objects into a single collection, the Loop Collection operator will allow you to iterate over all collections and finally you can separate the input objects from a collection by individually selecting the required element by using the Select operator.

**Input Ports**

collection (*col*) This port expects a collection of objects as input. Operators like the Collect operator combine a variable number of input objects into a single collection.

**Output Ports**

selected (*sel*) The object at the index specified by the *index* parameter is returned through this port.

**Parameters**

*index* (*integer*) This parameter specifies the index of the required object within the collection of objects.

*unfold* (*boolean*) This parameter is only applicable when the objects of the given collection are collections themselves. This parameter specifies whether collections received at the input ports should be considered unfolded for selection. If the input objects are collections themselves and the *unfold* parameter is set to false,
1. Process Control

then the index parameter will refer to the collection at the specified index. However if the unfold parameter is set to true then the index refers to the index in the flattened list, i.e. the list obtained from the input list by replacing all nested collections by their elements.

Tutorial Processes

Introduction to collections

This Example Process explains a number of important ideas related to collections. It shows how objects can be collected into a collection, then some preprocessing is applied on the collection and finally individual elements of the collection are separated as required.

The 'Golf' and 'Golf-Testset' data sets are loaded using the Retrieve operator. Both ExampleSets are provided as inputs to the Subprocess operator. The subprocess performs some preprocessing on the ExampleSets and then returns them through its output ports. The first output port returns the preprocessed 'Golf' data set which is then used as training set for the Decision Tree operator. The second output port delivers the preprocessed 'Golf-Testset' data set which is used as testing set for the Apply Model operator which applies the Decision Tree model. The performance of this model is measured and it is connected to the results port. The training and testing ExampleSets can also be seen in the Results Workspace.

Now have a look at the subprocess of the Subprocess operator. First of all, the Collect operator combines the two ExampleSets into a single collection. Note the double line output of the Collect operator which indicates that the result is a collection. Then the Loop Collection operator is applied on the collection. The Loop Collection operator iterates over the elements of the collection and performs some preprocessing (renaming an attribute in this case) on them. You can see in the subprocess of the Loop Collection operator that the Rename operator is used for changing the name of the Temperature attribute to 'New Temperature'. It is important to note that this renaming is performed on both ExampleSets
of the collection. The resultant collection is supplied to the Multiply operator which generates two copies of the collection. The first copy is used by the Select operator (with index parameter = 1) to select the first element of the collection i.e. the preprocessed 'Golf' data set. The second copy is used by the second Select operator (with index parameter = 2) to select the second element of the collection i.e. the preprocessed 'Golf-Testset' data set.

Loop Collection

This operator iterates over a collection of objects. It is a nested operator and its subprocess executes once for each object of the given collection.
1. Process Control

Description

Objects can be grouped into a collection using the Collect operator. In the Process View, collections are indicated by double lines. The Loop Collection operator loops over its subprocess once for every object in the input collection. The output of this operator is also a collection, any additional results of the subprocess can also be delivered through its output ports (as collections). If the unfold parameter is set to true then the output will be the union of all elements of the input collections.

Collections can be useful when you want to apply the same operations on a number of objects. The Collect operator will allow you to collect the required objects into a single collection, the Loop Collection operator will allow you to iterate over all collections and finally you can separate the input objects from collection by individually selecting the required element by using the Select operator.

Input Ports

collection (col) This input port expects a collection. It is the output of the Collect operator in the attached Example Process.

Output Ports

output (out) This operator can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object supplied at the first output port of the subprocess of the Loop Collection operator is delivered through the first output port of this operator. The objects are delivered as collections.
1.4. Collections

Parameters

unfold (boolean) This parameter specifies whether collections received at the input ports should be unfolded. If the unfold parameter is set to true then the output will be the union of all elements of the input collections.

Tutorial Processes

Introduction to collections

This Example Process explains a number of important ideas related to collections. This Example Process shows how objects can be collected into a collection, then some preprocessing is applied on the collection and finally individual elements of the collection are separated as required.

The 'Golf' and 'Golf-Testset' data sets are loaded using the Retrieve operator. Both ExampleSets are provided as inputs to the Subprocess operator. The subprocess performs some preprocessing on the ExampleSets and then returns them through its output ports. The first output port returns the preprocessed 'Golf' data set which is then used as training set for the Decision Tree operator. The second output port delivers the preprocessed 'Golf-Testset' data set which is used as testing set for the Apply Model operator which applies the Decision Tree model. The performance of this model is measured and it is connected to the results port. The training and testing ExampleSets can also be seen in the Results Workspace.

Now have a look at the subprocess of the Subprocess operator. First of all, the Collect operator combines the two ExampleSets into a single collection. Note the double line output of the Collect operator which indicates that the result is a collection. Then the Loop Collection operator is applied on the collection. The Loop Collection operator iterates over the elements of the collection and performs some preprocessing (renaming an attribute in this case) on them. You can see in the subprocess of the Loop Collection operator that the Rename operator is
1. Process Control

used for changing the name of the Temperature attribute to 'New Temperature'. It is important to note that this renaming is performed on both ExampleSets of the collection. The resultant collection is supplied to the Multiply operator which generates two copies of the collection. The first copy is used by the Select operator (with index parameter = 1) to select the first element of collection i.e. the preprocessed 'Golf' data set. The second copy is used by the second Select operator (with index parameter = 2) to select the second element of the collection i.e. the preprocessed 'Golf-Testset' data set.
2 Utility

Subprocess

This operator introduces a process within a process. Whenever a Subprocess operator is reached during a process execution, first the entire subprocess is executed. Once the subprocess execution is complete, flow is returned to the process (the parent process). Subprocess can be considered as small unit of process, like in process, all operators and combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied.

Description

Double click on the Subprocess operator to go inside the subprocess. Subprocess is then shown in the same Process View. To go back to the parent process, click the blue-colored up arrow button in Process View toolbar. This works like files and folders work in operating systems. Subprocesses can have Subprocesses in them just like folders can have folders in them. The order of execution in case of nested Subprocesses is the same as a depth-first-search through a tree structure. When a Subprocess is reached, all operators inside it are executed and then the execution flow returns to the parent process and the operator that is located after Subprocess operator (in parent process) is executed. This description can
be easily understood by studying the attached Example Process.

A Subprocess can be considered as a simple operator chain which can have an arbitrary number of inner operators. The operators are subsequently applied and their output is used as input for the succeeding operators. The input of the Subprocess is used as input for the first operator in the Subprocess and the output of the last operator in the Subprocess is used as the output of the Subprocess. Subprocesses make a process more manageable but don't forget to connect all inputs and outputs in correct order. Also make sure that you have connected the right number of ports at all levels of the chain.

Subprocesses are useful in many ways. They give a structure to the entire process. Process complexity is reduced and they become easy to understand and modify. Many operators have subprocess as their integral parts e.g. X-Validation operator. X-Validation operator is also shown in the attached Example Process. It should be noted that connecting the input of a Subprocess directly to its output without applying any operator in between or using an empty Subprocess gives no results.

Input Ports

*input* (*inp*) Subprocess can have multiple inputs. When one input is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at first *input* port of Subprocess is available at first *input* port of the nested chain (inside the subprocess). Subprocesses make a process more manageable but don't forget to connect all inputs in correct order. Make sure that you have connected right number of ports at all levels of chain.
Output Ports

output (out) A Subprocess can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at first output port of subprocess is delivered at first output of the outer process. Subprocesses make a process more manageable but don't forget to connect all outputs in correct order. Make sure that you have connected right number of ports at all levels of chain.

Tutorial Processes

Using subprocesses to structure a process

Golf dataset is loaded using Retrieve operator. It is attached to the first input of the Subprocess operator. Double click on the Subprocess operator to see what is inside this Subprocess. The first input of the Subprocess is attached with a Decision Tree operator. The output of the Decision Tree operator is given to the first output port. Now, go back to the main process. You will see that the first output port of the Subprocess operator is attached to the first result port. This explains 'Tree(decision tree(golf))' in the Results Workspace. This is how it works: Golf data set enters the subprocess through first input port, then the Decision Tree operator is applied on it in the subprocess, the resulting model is delivered to the results via first output port of the subprocess.

During the main process, Transactions data set is loaded using Retrieve operator. It is attached to the second input port of Subprocess operator. Double click on the Subprocess operator to see what is inside this Subprocess. The second input port of the Subprocess is attached directly to the second output port without applying any operator. Now, go back to the main process. You will see that the second output port of the Subprocess operator is attached to the second result port.
2. Utility

But, as no operator is applied to the Transactions data set in the Subprocess, it fails to produce any results (not even the result of Retrieve operator is shown in the Results Workspace). This explains why we have three results in Results Workspace despite attachment of four outputs to the results ports in the main process.

In the Subprocess, Iris data set is loaded using Retrieve operator. It is connected to the Decision Tree operator and the resultant model is attached to the third output port of the Subprocess, which is in turn attached to the third results port in the main process. This explains 'Tree (decision tree (Iris))' in the Results Workspace.

In the Subprocess, Weighting data set is loaded using Retrieve operator. It is connected to the X-Validation operator and the resultant Performance Vector is attached to the forth output port of the Subprocess, which is in turn attached to the forth results port in main process. This explains 'performanceVector (Performance)' in the Results Workspace. X-Validation operator itself is composed of a subprocess; double click on the X-Validation operator and you will see the subprocess within this operator. Explanation of what is going on inside X-Validation would be a diversion here. This operator was added here just to show how various operators can be composed of Subprocess. To know more about X-Validation operator you can read the description of X-Validation operator.

Note: This Example Process is just for highlighting different perspectives of Subprocess operator. It may not be very useful in real scenarios. The Example Process of Performance operator is also a good example of usage of Subprocess operator.
Set Macro

This operator can be used to define a macro which can be used by \%{macro_name} in parameter values of succeeding operators of the current process. The macro value will NOT be derived from any ExampleSet. A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define an existing macro.

Description

This operator can be used to define a macro which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter values in coming operators by writing the macro name in \%{macro_name} format in the parameter value where 'macro_name' is the name of the macro specified when the macro was
defined. In the Set Macro operator, the macro name is specified by the `macro` parameter and the macro value is specified by the `value` parameter. The macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define an existing macro.

This operator sets the value of a macro irrespective of any ExampleSet. That is why this operator can also exist on its own i.e. without being connected to any other operator. If you want to create a single macro from properties of a given input ExampleSet, the Extract Macro operator is the right operator.

**Macros**

A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. The macro is one of the advanced topics of RapidMiner, please study the attached Example Process to develop a better understanding of macros. The Example Processes of the Extract Macro operator are also useful for understanding the concepts related to the macros.

There are also some predefined macros:

- `%{process_name}`: will be replaced by the name of the process (without path and extension)
- `%{process_file}`: will be replaced by the file name of the process (with extension)
- `%{process_path}`: will be replaced by the complete absolute path of the process file
- Several other short macros also exist, e.g. `%{a}` for the number of times the current operator was applied.
Please note that other operators like many of the loop operators (e.g. Loop Values, Loop Attributes) also add specific macros.

During the runtime the defined macros can be observed in the macro viewer.

**Differentiation**

**Set Macros** The Set Macros operator is like the Set Macro operator with only one difference. The Set Macros operator can be used for setting values of multiple macros whereas the Set Macro operator can be used for setting value of just a single macro. See page 98 for details.

**Input Ports**

through *(thr)* It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first through input port of the Set Macro operator is available at the first through output port.

**Output Ports**

through *(thr)* Objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the macro value is set even if this port is left without connections. The Set Macro operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered
at the first *through* input port of the Set Macro operator is delivered at the first *through* output port.

**Parameters**

**macro** This parameter is used to specify the name of the macro. The macro can be accessed in succeeding operators of the current process by writing the macro's name in `{macro_name}` format, where 'macro_name' is the name of the macro specified in this parameter.

**value** This parameter is used to specify the value of the macro. When the macro is accessed in succeeding operators of the current process by writing the macro's name in `{macro_name}` format, it is replaced by the value of the macro specified by this parameter.

**Related Documents**

Set Macros (98)

**Tutorial Processes**

**Introduction to the Set Macro operator**

This is a very basic process that demonstrates the use of the Set Macro operator. The Set Macro operator is used first of all. One macro is defined using the *macro* and the *value* parameter. The macro is named 'number' and it is given the value 1. Note that this operator is not connected to any other operator; it can exist at its own. Always make sure that the macro is defined before it is used in the parameter values.

The 'Golf' data set is loaded using the Retrieve operator. The Select Subprocess
2.1. Macros

operator is applied on it. Double-click on the Select Subprocess operator to see the subprocesses in it. As you can see, there are four subprocesses:

- **Subprocess 1:** The k-NN operator is applied on the input and the resulting model is passed to the output.

- **Subprocess 2:** The Naive Bayes operator is applied on the input and the resulting model is passed to the output.

- **Subprocess 3:** The Decision Tree operator is applied on the input and the resulting model is passed to the output.

- **Subprocess 4:** The input is directly connected to the output.

Only one of these subprocesses can be executed at a time. The subprocess to be executed can be controlled by the `select which` parameter of the Select Subprocess operator. The `select which` parameter is set using the 'number' macro defined by the Set Macro operator. The `select which` parameter is set to 'number'. When the process will be executed, 'number' will be replaced with the value of the 'number' macro i.e. 'number' will be replaced by 1. Thus the `select which` parameter is set to 1, thus the first subprocess will be executed. When you run the process you will see the model created by the k-NN operator in the Results workspace. As the value of the `select which` parameter is provided by the macro created by the Set Macro operator, changing the value of the macro will change the value of the `select which` parameter. To execute the second subprocess set the `value` parameter of the Set Macro operator to 2 and run the process again. You will see the model generated by the Naive Bayes operator in the Results Workspace. To execute the third subprocess set the `value` parameter of the Set Macro operator to 3 and run the process again. You will see the model generated by the Decision Tree operator in the Results Workspace. To execute the fourth subprocess set the `value` parameter of the Set Macro operator to 4 and run the process again. Now you will see the 'Golf' data set in the Results Workspace because no operator was applied in the fourth subprocess.
2. Utility

Set Macros

This operator can be used to define multiple macros which can be used by \%{\textit{macro\_name}} in parameter values of succeeding operators of the current process. The macro values will NOT be derived from any ExampleSet. A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define existing macros.

Description

This operator can be used to define multiple macros which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter values in coming operators by writing the macro name in \%{\textit{macro\_name}} format in a parameter value where 'macro\_name' is the name of the macro specified when the macro was defined. In the Set Macros operator, the macro name and value is specified by the \textit{macros} parameter. A macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define existing macros.
2.1. Macros

This operator sets the value of multiple macros irrespective of any ExampleSet. That is why this operator can also exist on its own i.e. without being connected to any other operator. If you want to create a single macro from properties of a given input ExampleSet, the Extract Macro operator is the right operator.

Macros

A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. The macro is one of the advanced topics of RapidMiner, please study the attached Example Process to develop a better understanding of macros. The Example Processes of the Extract Macro operator are also useful for understanding the concepts related to the macros.

There are also some predefined macros:

- \%{process\_name}: will be replaced by the name of the process (without path and extension)

- \%{process\_file}: will be replaced by the file name of the process (with extension)

- \%{process\_path}: will be replaced by the complete absolute path of the process file

- Several other short macros also exist, e.g. \%{a} for the number of times the current operator was applied.

Please note that other operators like many of the loop operators (e.g. Loop Values, Loop Attributes) also add specific macros.
Input Ports

**through (thr)** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Set Macros operator is available at the first *through* output port.

Output Ports

**through (thr)** Objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the macro value is set even if this port is left without connections. The Set Macros operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Set Macros operator is delivered at the first *through* output port.

Parameters

**macros** This parameter is used to specify the names and values of the macros. Macros can be accessed in succeeding operators of the current process by writing the macro's name in `%{macro_name}` format, where 'macro_name' is the name of the macro specified in this parameter.
2.1. Macros

Tutorial Processes

Introduction to the Set Macros operator

This is a very basic process that demonstrates the use of macros and the Set Macros operator. The Set Macros operator is used first of all. Two macros are defined in the *macros* parameter. They are named 'min' and 'max'. 'min' is given the value 1 and 'max' is given the value 4. Note that this operator is not connected to any other operator; it can exist at its own. Always make sure that the macro is defined before it is used in parameter values.

The 'Golf' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it. The *first example* parameter is set to '%{min}' and the *last example* parameter is set to '%{max}'. When the process will be executed, '%{min}' and '%{max}' will be replaced with the values of the respective macros i.e. '%{min}' and '%{max}' will be replaced by 1 and 4 respectively. Thus the Filter Examples Range operator will deliver only the first four examples of the 'Golf' data set.

At the end, the Write CSV operator is applied to store the output of the Filter Example Range operator in a CSV file. Note that the *csv file* parameter is set to 'D:\%{process_name}'. Here '%{process_name}' is a predefined macro which will be replaced by the name of the current process (without file extension). Thus the output of the Filter Example Range operator will be written in a csv file in the D drive of your computer. The name of the file will be the same as the name of this process.
2. Utility

Generate Macro

This operator can be used to calculate new macros from existing macros. A macro can be used by writing \%\{macro\_name\} in parameter values of succeeding operators of the current process. A Macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define existing macros.

Description

This operator can be used to define macros which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter value in coming operators by writing the macro name in \%\{macro\_name\} format in the parameter value where 'macro\_name' is the name of the macro specified when it was defined. In Generate Macro operator macro names and function descriptions are specified in the function descriptions parameters. The macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define existing macros by specifying the name of that macro as name in the function descriptions parameter.
A large number of operations are supported, which allows you to write rich expressions. Here is a list of supported expressions along with its explanation and examples. In these examples M, N and O are used as macro names. These are just simple examples; more complicated expressions can be created by using multiple operations. Parenthesis can be used to nest operations. The description of all operations follows this format:

Name of operation (syntax of operation): brief description; examples: example1, example2

The following basic operations are supported:

- **Addition (+):** Calculates the addition of the two terms surrounding this operator; examples: \( \%\{M\} + 7 \), \( \%\{M\} + \%\{N\} \), \( \%\{M\} + \%\{N\} + 9 \)

- **Subtraction(-):** Calculates the subtraction of the first term from the second one; examples: \( 15 - \%\{M\} \), \( \%\{M\} - \%\{N\} \), \( (\%\{M\} - \%\{N\}) - \%\{O\} \)

- **Multiplication (*):** Calculates the multiplication of the two terms surrounding this operator; examples: \( 5 \times \%\{N\} \), \( \%\{M\} \times \%\{N\} \), \( \%\{M\} \times \%\{N\} \times \%\{O\} \)

- **Division(/):** Calculates the division of the first term by the second one; examples: \( \%\{N\} / 4 \), \( \%\{M\} / \%\{N\} \), \( 4 / 2 \)

- **Power(^):** Calculates the first term to the power of the second one; examples: \( 2^3 \), \( \%\{M\}^2 \), \( \%\{M\}^\%\{N\} \)

- **Modulus(%):** Calculates the modulus of the first term by the second one; examples: \( 11 \%\{2\} \), \( \%\{M\} \%\{5\} \), \( \%\{N\} \%\%\{O\} \)

- **Less Than(<):** Delivers true if the first term is less than the second; examples: \( \%\{M\} < 4 \), \( \%\{N\} < \%\{M\} \)

- **Greater Than(>):** Delivers true if the first term is greater than the second; examples: \( \%\{M\} > 3 \), \( \%\{N\} > \%\{O\} \)

- **Less or Equal(<=):** Delivers true if the first term is less than or equal to the second; examples: \( \%\{M\} <=5 \), \( \%\{O\} <= \%\{N\} \)
2. Utility

- More or Equal (\(\geq\)): Delivers true if the first term is greater than or equal to the second; examples: \(\{M\}\geq4, 78\geq\{N\}\)

- Equal (\(\equiv\)): Delivers true if the first term is equal to the second; examples: \(\{M\}\equiv\{N\}, \{M\}\equiv12\)

- Not Equal (\(!=\)): Delivers true if the first term is not equal to the second; examples: \(\{M\}!\equiv\{N\}, \{M\}!=25\)

- Boolean Not (\(!\)): Delivers true if the following term is false or vice versa; examples: !(\(\{M\}>2\)), !(\(\{N\}<\equiv25\))

- Boolean And (\(&\ &\)): Delivers true if both surrounding terms are true; examples: \(\{M\}>2\)& & \(\{N\}<\equiv\{O\}\)

- Boolean Or (\(||\)): Delivers true if at least one of the surrounding terms is true; examples: \(\{M\}>2\)||\(\{N\}<\equiv\{O\}\)

The following log and exponential functions are supported:

- Natural Logarithm (\(\ln(x)\)): Calculates the logarithm of the argument to the base e; examples: \(\ln(5), \ln(\{M\})\)

- Logarithm Base 10 (\(\log(x)\)): Calculates the logarithm of the argument to the base 10; examples: \(\log(25), \log(\{M\})\)

- Logarithm Base 2 (\(\log_2(x)\)): Calculates the logarithm of the argument to the base 2, it is also called logarithm dualis; examples: \(\log_2(24), \log_2(\{M\})\)

- Exponential (\(\exp(x)\)): Calculates the value of the constant e to the power of the argument, it is equivalent to \(\exp(x)\); examples: \(\exp(12), \exp(\{M\})\)

- Power (\(\text{pow}(x,y)\)): Calculates the first term to the power of the second one; examples: \(\text{pow}(\{M\},\ B), \text{pow}(\{M\},\ 3)\)

The following trigonometric functions are supported:

- Sine (\(\sin(x)\)): Calculates the sine of the given argument; examples: \(\sin(\{M\}), \sin(45)\)
2.1. Macros

- **Cosine (cos(x))**: Calculates the cosine of the given argument; examples: \( \cos(\%\{M\}) \), \( \cos(45) \)

- **Tangent (tan(x))**: Calculates the tangent of the given argument; examples: \( \tan(\%\{M\}) \), \( \tan(45) \)

- **Arc Sine (asin(x))**: Calculates the inverse sine of the given argument; examples: \( \text{asin}(\%\{M\}) \), \( \text{asin}(0.50) \)

- **Arc Cosine (acos(x))**: Calculates the inverse cosine of the given argument; examples: \( \text{acos}(\%\{M\}) \), \( \text{acos}(0.50) \)

- **Arc Tangent (atan(x))**: Calculates the inverse tangent of the given argument; examples: \( \text{atan}(\%\{M\}) \), \( \text{atan}(1) \)

- **Arc Tangent with 2 parameters (atan2(x,y))**: Calculates the inverse tangent based on the two given arguments; examples: \( \text{atan}(\%\{M\}, 0.5) \)

- **Hyperbolic Sine (sinh(x))**: Calculates the hyperbolic sine of the given argument; examples: \( \text{sinh}(\%\{M\}) \)

- **Hyperbolic Cosine (cosh(x))**: Calculates the hyperbolic cosine of the given argument; examples: \( \text{cosh}(\%\{M\}) \)

- **Hyperbolic Tangent (tanh(x))**: Calculates the hyperbolic tangent of the given argument; examples: \( \text{tanh}(\%\{M\}) \)

- **Inverse Hyperbolic Sine (asinh(x))**: Calculates the inverse hyperbolic sine of the given argument; examples: \( \text{asinh}(\%\{M\}) \)

- **Inverse Hyperbolic Cosine (acosh(x))**: Calculates the inverse hyperbolic cosine of the given argument; examples: \( \text{acosh}(\%\{M\}) \)

- **Inverse Hyperbolic Tangent (atanh(x))**: Calculates the inverse hyperbolic tangent of the given argument; examples: \( \text{atanh}(\%\{M\}) \)

The following *statistical* functions are supported:

- **Round (round(x))**: Rounds the given number to the next integer. If two
arguments are given, the first one is rounded to the number of digits indicated by the second argument; examples: round(\{M\}), round(\{M\}, 3). round(\{M\}, 3) rounds macro '%{M}' to 3 decimal places.

- **Floor (floor(x))**: Calculates the first integer less than the given argument, e.g. floor(4.7) would be 4; examples: floor(\{M\}), floor(23.34)

- **Ceiling (ceil(x))**: Calculates the next integer greater than the given argument e.g. ceil(23.4) would be 24; examples: ceil(\{M\}), ceil(23.34)

- **Average (avg(x,y,z...))**: Calculates the average of the given arguments; examples: avg(\{M\},\{N\}), avg(\{M\},\{N\},\{O\}), avg(\{M\},\{N\},25)

- **Minimum (min(x,y,z...))**: Calculates the minimum of the given arguments; examples: min(\{M\},\{N\}), min(\{M\},\{N\},\{O\}) , min(\{M\},0,\{N\})

- **Maximum (max(x,y,z...))**: Calculates the maximum of the given arguments; examples: max(\{M\},\{N\}), max(\{M\},\{N\},\{O\}) , max(\{M\},\{N\},45)

The following *miscellaneous* functions are supported:

- **If-Then-Else (if(condition,true-evaluation, false-evaluation))**: The first argument specifies the condition. If the condition is evaluated as true then the result of the second argument is delivered otherwise the result of the third argument is delivered; example: if(\{M\}>5, 7*\{M\}, \{N\}/2)

- **Absolute (abs(a))**: Delivers the absolute value of the argument; example: const(\{M\})

- **Square Root (sqrt(x))**: Delivers the square root of the given argument; examples: sqrt(16), sqrt(\{M\})

- **Signum (sgn(x))**: Delivers -1 or +1 depending on the sign of the argument; example: sgn(-5) delivers -1.

- **Random (rand())**: Delivers a random number between 0 and 1. It can be used to generate random numbers in any range when used with other functions like multiplication, addition, floor etc; example: floor((rand()\*10) + 15) produces random integers between 15 and 25.
2.1. Macros

- **Modulus (mod(x, y))**: Calculates the modulus of the first term by the second one; example: $11\%2$, $\%\{M\}\%3$

- **Sum (sum(x, y, z, ...))**: Calculates the sum of all arguments; example: $\text{sum}(\%\{M\}, \%\{N\}, 42)$

- **Binomial (binom(x, y))**: Calculates the binomial coefficients; example: $\text{binom}(5, 2)$

- **Number to string (str(a))**: Changes the number given as argument to a string; example: $\text{str}(\%\{M\})$

This operator also supports the constants 'pi' and 'e' if the *use standard constants* parameter is set to true. You can also use strings in operations but the string values should be enclosed in double quotes (" ").

The functions available in the Generate Macro operator behave similar to the functions of the Generate Attributes operator. Please study the Example Process of the Generate Attributes operator to understand the use of these functions.

**Macros**

Macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. Macro is one of the advanced topics of RapidMiner, please study the attached Example Processes to develop a better understanding of macros.

There are also some predefined macros:

- $\%\{process\_name\}$: will be replaced by the name of the process (without path and extension)

- $\%\{process\_file\}$: will be replaced by the file name of the process (with ex-
2. Utility

tension)

- \%\{process_path\}: will be replaced by the complete absolute path of the process file

- Several other short macros also exist, e.g. \%\{a\} for the number of times the current operator was applied.

Please note that other operators like many of the loop operators (e.g. Loop Values, Loop Attributes) also add specific macros.

Input Ports

through (thr) It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at first through input port of the Generate Macro operator is available at the first through output port.

Output Ports

through (thr) The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the macro value is calculated even if this port is left without connections. The Generate Macro operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Generate Macro operator is delivered at the first through output port.
2.1. Macros

Parameters

**function descriptions** The list of macro names together with the expressions which define the new macros are specified through this parameter.

**use standard constants** (*boolean*) This parameter indicates if standard constants like 'e' or 'pi' should be available. If checked, these constants can be used in expressions for generating new macros.

Tutorial Processes

Generating a new macro from an already existing macro

This Example Process discusses an imaginary scenario to explain how the Generate macro operator can be used to define a new macro from an existing macro. Suppose we want to apply the K-NN operator on an ExampleSet such that the value of the \( k \) parameter of the K-NN operator is set dynamically to some ratio of number of examples in the input ExampleSet. Let us assume that the required ratio is \( k = n * 0.025 \) where \( n \) is the number of examples in the input ExampleSet.

The Polynomial data set is loaded using the Retrieve operator. The Extract Macro operator is applied on it to create a new macro. The *macro* parameter is set to 'example_count' and the *macro type* parameter is set to 'number of examples'. Thus a macro is created with the name 'example_count'. The value of this macro is equal to the number of examples in the Polynomial data set. Then the Generate Macro operator is applied. Only one macro is defined in the *function descriptions* parameter. The macro is named 'value_k' and it is defined by this expression: \( \text{ceil}( \%\{\text{example\_count}\} \times 0.025) \). Note the use of the 'example_count' macro here. When this process is executed '\%\{example\_count\}' is replaced by the value of the 'example_count' macro which is equal to the number of examples in the input ExampleSet which is 200. Thus at run-time the expression that defines 'value_k' macro will be evaluated as \( \text{ceil}(200 \times 0.025) \). The result of this
The K-NN operator is applied on the Polynomial data set with the parameter $k$ set to $\{\text{value}_k\}$. At run-time $\{\text{value}_k\}$ will be replaced by the actual value of the 'value_k' macro which is 5. Thus K-NN operator will be applied on the Polynomial data set with the parameter $k$ set to 5. This can be verified by running the process and viewing the results in the Results Workspace.

**Extract Macro**

This operator can be used to define a single macro which can be used by $\{\text{macro}_name\}$ in parameter values of succeeding operators of the current process. The macro value will be derived from the input ExampleSet. A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. This operator can also be used to re-define an existing macro.

**Description**

This operator can be used to define a single macro which can be used in parameter values of succeeding operators of the current process. Once the macro has been defined, the value of that macro can be used as parameter values in coming operators by writing the macro name in $\{\text{macro}_name\}$ format in the
parameter value where 'macro_name' is the name of the macro specified when it was defined. In the Extract Macro operator the macro name is specified by the \textit{macro} parameter. The macro will be replaced in the value strings of parameters by the macro's value. This operator can also be used to re-define an existing macro.

This operator sets the value of a single macro from properties of a given input ExampleSet. This includes properties like the number of examples or number of attributes of the input ExampleSet. Specific data value of the input ExampleSet can also be used to set the value of the macro which can be set using various statistical properties of the input ExampleSet e.g. average, min or max value of an attribute. All these options can be understood by studying the parameters and the attached Example Processes. The Set Macro operator can also be used to define a macro but it does not set the value of the macro from properties of a given input ExampleSet.

\textbf{Macros}

A macro can be considered as a value that can be used by all operators of the current process that come after it has been defined. Whenever using macros, make sure that the operators are in the correct sequence. It is compulsory that the macro should be defined before it can be used in parameter values. The macro is one of the advanced topics of RapidMiner, please study the attached Example Processes to develop a better understanding of macros.

There are also some predefined macros:

- \%\{\texttt{process\_name}\}: will be replaced by the name of the process (without path and extension)
- \%\{\texttt{process\_file}\}: will be replaced by the file name of the process (with extension)
- \%\{\texttt{process\_path}\}: will be replaced by the complete absolute path of the
2. Utility

process file

- Several other short macros also exist, e.g. \%\{a\} for the number of times the current operator was applied.

Please note that other operators like many of the loop operators (e.g. Loop Values, Loop Attributes) also add specific macros.

Input Ports

def example set input (exa) This input port expects an ExampleSet. The macro value will be extracted from this ExampleSet

Output Ports

def example set output (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators. It is not compulsory to attach this port to any other port, Macro value is set even if this port is left without connections.

Parameters

macro (string) This parameter is used to name the macro and can be accessed in succeeding operators of current process by writing the macro name in \%\{macro_name\} format, where 'macro_name' is the name of the macro specified by this parameter.

macro type This parameter indicates the way the input ExampleSet should be used to define the macro.

- number_of_examples If this option is selected, the macro value is set to the total number of examples in the input ExampleSet.
2.1. Macros

- **number_of_attributes** If this option is selected, the macro value is set to the total number of attributes in the input ExampleSet.

- **data_value** If this option is selected, the macro value is set to the value of the specified attribute at the specified index. The attribute is specified using the `attribute name` parameter and the index is specified using the `example index` parameter.

- **statistics** If this option is selected, the macro value is set to the value obtained by applying the selected statistical operation on the specified attribute. The attribute is specified using the `attribute name` parameter and the statistical operator is selected using the `statistics` parameter.

**statistics** This parameter is only available when the `macro type` parameter is set to 'statistics'. This parameter allows you to select the statistical operator to be applied on the attribute specified by the `attribute name` parameter.

**attribute name** *(string)* This parameter is only available when the `macro type` parameter is set to 'statistics' or 'data value'. This parameter allows you to select the required attribute.

**attribute value** *(string)* This parameter is only available when the `macro type` parameter is set to 'statistics' and the `statistics` parameter is set to 'count'. This parameter is used to specify a particular value of the specified attribute. The macro value will be set to the number of occurrences of this value in the specified attribute. The attribute is specified by the `attribute name` parameter.

**example index** *(integer)* This parameter is only available when the `macro type` parameter is set to 'data value'. This parameter allows you to select the index of the required example of the attribute specified by the `attribute name` parameter and the optional `additional macros` parameter.

**additional macros** This parameter is only available when the `macro type` parameter is set to 'data value'. This optional parameter allows you to add an unlimited amount of additional macros. Note that the value for the `example index` parameter is used for all macros in this list.
### Tutorial Processes

#### Introduction to the Extract Macro operator

This is a very basic process that demonstrates the use of macros and the Extract Macro operator. The 'Golf' data set is loaded using the Retrieve operator. The Extract Macro operator is applied on it. The macro is named 'avg_temp'. The macro type parameter is set to 'statistics', the statistics parameter is set to 'average' and the attribute name parameter is set to 'Temperature'. Thus the value of the avg_temp macro is set to the average of values of the 'Golf' data set's Temperature attribute. Which in all 14 examples of the 'Golf' data set is 73.571. Thus the value of the avg_temp macro is set to 73.571. In this process, wherever \%{avg_temp} is used in parameter values, it will be replaced by the value of the avg_temp macro i.e. 73.571. Note that the output port of the Extract Macro operator is not connected to any other operator but still the avg_temp macro has been created.

The 'Golf-Testset' data set is loaded using the Retrieve operator. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter'. The parameter string parameter is set to 'Temperature > \%{avg_temp}'. Note the use of the avg_temp macro. When this process is run, \%{avg_temp} will be replaced by the value of the avg_temp macro i.e. 73.571. Thus only those examples of the Golf-Testset data set will make it to the output port where the value of the Temperature attribute is greater than average value of the Temperature attribute values of the Golf data set (i.e. 73.571). You can clearly verify this by seeing the results in the Results Workspace.
Redefining a macro using the Extract Macro operator

The focus of this Example Process is to show how macros can be redefined using the Extract Macro operator. This process is almost the same as the first Example Process. The only difference is that after the `avg_temp` macro has been defined, the same macro is redefined using the 'Golf' data set and the Extract Macro operator. The 'Golf' data set is loaded again and it is provided to the second Extract Macro operator. In this Extract Macro operator the macro parameter is set to 'avg_temp' and the macro type parameter is set to 'number of examples'. As the `avg_temp` macro already exists, no new macro is created; the already existing macro is redefined. As the number of examples in the 'Golf' data set is 14, `avg_temp` is redefined as 14. Thus in the Filter Examples operator the value of the Temperature attribute of the 'Golf-Testset' data set is compared with 14 instead of 73.571. This can be verified by seeing the results in the Results workspace. Please note that macros are redefined depending on their order of execution.
use of Extract Macro in complex preprocessing

This Example Process is also part of the RapidMiner tutorial. It is included here to show the usage of the Extract Macro operator in complex preprocessing. This process will cover a number of concepts of macros including redefining macros, the macro of the Loop Values operator and the use of the Extract Macro operator. This process starts with a subprocess which is used to generate data. What is happening inside this subprocess is not relevant to the use of macros, so it is not discussed here. A breakpoint is inserted after this subprocess so that you can view the ExampleSet. You can see that the ExampleSet has 12 examples and 2 attributes: 'att1' and 'att2'. 'att1' is nominal and has 3 possible values: 'range1', 'range2' and 'range3'. 'att2' has real values.

The Loop Values operator is applied on the ExampleSet and iterates over the values of the specified attribute (i.e. att1) and applies the inner operators on the given ExampleSet while the current value can be accessed via the macro defined by the iteration macro parameter which is set to 'loop_value', thus the current value can be accessed by specifying $\{loop_value\}$ in the parameter values. As att1 has 3 possible values, Loop Values will iterate 3 times, once for each possible value of att1.
2.1. Macros

Here is an explanation of what happens inside the Loop Values operator. It is provided with an ExampleSet as input. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter' and the parameter string is set to 'att1 = %{loop.value}'. Note the use of the loop_value macro here. Only those examples are selected where the value of att1 is equal to the value of the loop_value macro. A breakpoint is inserted here so that you can view the selected examples. Then the Aggregation operator is applied on the selected examples. It is configured to take the average of the att2 values of the selected examples. This average value is stored in a new ExampleSet in the attribute named 'average(att2)'. A breakpoint is inserted here so that you can see the average of the att2 values of the selected examples. The Extract Macro operator is applied on this new ExampleSet to store this average value in a macro named 'current_average'. The originally selected examples are passed to the Generate Attributes operator that generates a new attribute named 'att2.abs_avg' which is defined by the expression 'abs(att2 - %{current.average})'. Note the use of the current_average macro here. Its value is subtracted from all values of att2 and stored in a new attribute named 'att2.abs_avg'. The Resultant ExampleSet is delivered at the output of the Loop Values operator. A breakpoint is inserted here so that you can see the ExampleSet with the 'att2.abs_avg' attribute. This output is fed to the Append operator in the main process. It merges the results of all the iterations into a single ExampleSet which is visible at the end of this process in the Results Workspace.

Here is what you see when you run the process.

- ExampleSet generated by the Generate Data subprocess. Then the process enters the Loop Value operator and iterates 3 times.

Iteration 1:

- ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range1'

- Average of 'att2' values for the selected examples. The average is -1.161.

- ExampleSet with 'att2.abs_avg' attribute for iteration 1.
2. Utility

Iteration 2:

- ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range2'

- Average of 'att2' values for the selected examples. The average is -1.656.

- ExampleSet with 'att2_abs_avg' attribute for iteration 2.

Iteration 3:

- ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range3'

- Average of 'att2' values for the selected examples. The average is 1.340.

- ExampleSet with 'att2_abs_avg attribute' for iteration 3.

Now the process comes out of the Loop Values operator and the Append operator merges the final ExampleSets of all three iterations into a single ExampleSet that you can see in the Results Workspace.

Log

This operator stores information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The stored information can be
plotted by the GUI when the process execution is complete. Moreover, the information can also be written into a file.

**Description**

The Log operator is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example, you want to see values of different parameters in all iterations of any Loop operator. In such scenarios the ideal operator is the Log operator. A large variety of information can be stored using this operator. Values of all parameters of all operators in the current process can be stored. Other information like apply-count, cpu-time, execution-time, loop-time etc can also be stored. The information stored in the log table can be viewed using the Table View in the Results Workspace. The information can also be analyzed in form of various graphs using the Plot View in the Results Workspace. The information can also be written directly into a file using the *filename* parameter.

The *log* parameter is used for specifying the information to be stored. The *column name* option specifies the name of the column in the log table (and/or file). Then you can select any operator from the drop down menu. Once you have selected an operator, you have two choices. You can either store a parameter value or store other values. If you opt for the parameter value, you can choose any parameter of the selected operator through the drop down menu. If you opt for other values, you can choose any value like apply-count, cpu-time etc from the last drop down menu.

Each time the Log operator is applied, all the values and parameters specified by the *log* parameter are collected and stored in a data row. When the process finishes, the operator writes the collected data rows into a file (if the *filename* parameter has a valid path). In GUI mode, 2D or 3D plots are automatically generated and displayed in the Results Workspace. Please study the attached Example Processes to understand working of this operator.
2. Utility

Input Ports

through (thr) It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first through input port of the Log operator is available at the first through output port.

Output Ports

through (thr) The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port. The Log operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Log operator is delivered at the first through output port.

Parameters

filename (filename) This parameter is used if you want to write the stored values into a file. The path of the file is specified here.

log (list) This is the most important parameter of this operator. It is used for specifying the values that should be stored by the Log operator. The log parameter is used for specifying the information to be stored. The column name option specifies the name of the column in the log table (and/or file). Then you can select any operator from the drop down menu. Once you have selected an operator, you have two choices. You can either store a parameter value or store other values. If you opt for the parameter value, you can choose any parameter
of the selected operator through the drop down menu. If you opt for other values, you can choose any value like apply-count, cpu-time etc from the last drop down menu.

**sorting type (selection)** This parameter indicates if the logged values should be sorted according to the specified dimension.

**sorting dimension (string)** This parameter is only available when the sorting type parameter is set to 'top-k' or 'bottom-k'. This parameter is used for specifying the dimension that is to be used for sorting.

**sorting k (integer)** This parameter is only available when the sorting type parameter is set to 'top-k' or 'bottom-k'. Only k results will be kept.

**persistent (boolean)** This is an expert parameter. This parameter indicates if the results should be written to the specified file immediately.

---

### Tutorial Processes

#### Introduction to the Log operator

This Example Process shows usage of the Log and Extract Macro operator in complex preprocessing. Other than concepts related to the Log operator, this process will cover a number of concepts of macros including redefining macros, macro of Loop Values operator and use of the Extract Macro operator. This process starts with a subprocess which is used to generate data. What is happening inside this subprocess is not relevant to the use of the Log operator, so it is not discussed here. A breakpoint is inserted after this subprocess so that you can view the ExampleSet. You can see that the ExampleSet has 12 examples and 2 attributes: 'att1' and 'att2'. 'att1' is nominal and has 3 possible values: 'range1', 'range2' and 'range3'. 'att2' has real values.

The Loop Values operator is applied on the ExampleSet. It iterates over the values of the specified attribute (i.e. att1) and applies the inner operators on the given ExampleSet while the current value can be accessed via the macro defined by the iteration macro parameter. The iteration macro parameter is set...
2. Utility

To 'loop_value', thus the current value can be accessed by specifying %\{loop_value\} in the parameter values. As att1 has 3 possible values, the Loop Values operator will iterate 3 times, once for each possible value of att1.

Here is an explanation of what happens inside the Loop Values operator. The Loop Values operator is provided with an ExampleSet as input. The Filter Examples operator is applied on it. The condition class parameter is set to 'attribute value filter' and the parameter string is set to 'att1 = %\{loop_value\}'. Note the use of the loop_value macro here. Only those examples are selected where the value of att1 is equal to the value of the loop_value macro. A breakpoint is inserted here so that you can view the selected examples. Then Aggregation operator is applied on the selected examples. It is configured to take the average of the att2 values of the selected examples. This average value is stored in a new ExampleSet in an attribute named 'average(att2)'. A breakpoint is inserted here so that you can see the average of the att2 values of the selected examples. The Extract Macro operator is applied on this new ExampleSet to store this average value in a macro named 'current_average'. The originally selected examples are passed to the Generate Attributes operator that generates a new attribute named 'att2_abs_avg'. This attribute is defined by the expression 'abs(att2 - %\{current_average\})'. Note the use of the current_average macro here. Value of the current_average macro is subtracted from all values of att2 and stored in a new attribute named 'att2_abs_avg'. The Resultant ExampleSet is delivered at the output of the Loop Values operator. A breakpoint is inserted here so that you can see the ExampleSet with the 'att2_abs_avg' attribute. This output is fed to the Append operator in the main process. The Append operator merges the results of all the iterations into a single ExampleSet which is visible at the end of this process in the Results Workspace.

Note the Log operator in the subprocess of the Loop Values operator. Three columns are created using the log parameter. The 'Average att2' column stores the value of the macro of the Extract Macro operator. The 'Iteration' column stores the apply-count of the Aggregate operator which is the same as the number of iterations of the Loop Values operator. The 'att1 value' column stores the value of att1 in the current iteration. At the end of the process, you will see that the Log operator stores a lot of information that was not directly accessible. Moreover, it
2.2. Logging

displays all the required information at the end of the process, thus breakpoints are not required.

Also note that the filename parameter of the Log operator is set to: 'D:\log.txt'. Thus a text file named 'log' is created in your 'D' drive. This file has the information stored during this process by the Log operator.

Here is what you see when you run the process:

- The ExampleSet generated by the Generate Data subprocess. Then the process enters the Loop Value operator and iterates 3 times.

Iteration 1:

- The ExampleSet where the 'att1' value is equal to the current value of the loop_value macro i.e. 'range1'
- The average of the 'att2' values for the selected examples. The average is -1.161.
- The ExampleSet with the 'att2_abs_avg' attribute for iteration 1.

Iteration 2:

- The ExampleSet where the 'att1' value is equal to the current value of loop_value macro i.e. 'range2'
- The Average of the 'att2' values for the selected examples. The average is -1.656.
- The ExampleSet with the 'att2_abs_avg' attribute for iteration 2.

Iteration 3:

- The ExampleSet where the 'att1' value is equal to the current value of loop_value macro i.e. 'range3'
- The Average of the 'att2' values for the selected examples. The average is 1.340.
2. Utility

- The ExampleSet with the 'att2_abs_avg attribute' for iteration 3.

Now the process comes out of the Loop Values operator and the Append operator merges the final ExampleSets of all three iterations into a single ExampleSet that you can see in the Results Workspace.

Now have a look at the results of the Log operator. You can see all the required values in tabular form using the Table View. You can see that all the values that were viewed using breakpoints are available in a single table. You can see the results in the Plot View as well. Also have a look at the file stored in the 'D' drive. This file has exactly the same information.

Viewing Training vs Testing error using the Log operator

The 'Weighting' is loaded using the Retrieve operator. The Loop Parameters operator is applied on it. The parameters of the Loop Parameters operator are set such that this operator loops 25 times. Thus its subprocess is executed 25 times. In every iteration, the value of the $C$ parameter of the SVM(LibSVM) operator is changed. The value of the $C$ parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration.

Have a look at the subprocess of the Loop Parameters operator. First the data is split into two equal partitions using the Split Data operator. The SVM (Lib-SVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.
2.2. Logging

The log parameter of the Log operator stores four things.

1. The iterations of the Loop Parameter operator are counted by apply-count of the SVM operator. This is stored in a column named 'Count'.

2. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'.

3. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'.

4. The value of the $C$ parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'.

Also note that the stored information will be written into a file as specified in the filename parameter.

Run the process and turn to the Results Workspace. You can see all the values in the Table View. This table can be used to study how classification errors in training and testing partitions behave with the increase in the value of the $C$ parameter of the SVM(LibSVM) operator. To view these results in graphical form, switch to the Plot View. Select an appropriate plotter. You can use 'Series Multiple' plotter with 'SVM-C' as the 'Index Dimension'. Select 'Training Error' and 'Testing Error' in the 'Plot Series'. The 'scatter multiple' plotter can also be used. Now you can analyze how the training and testing error behaved with the increase in the parameter $C$. 
Provide Macro as Log Value

This operator reads the value of the specified macro and provides the macro value for logging. This operator is only useful if the required macro value cannot be logged directly.

Description

The Provide Macro as Log Value operator can be used to log the current value of the specified macro. The name of the required macro is specified in the *macro name* parameter. Most operators provide the macro they define as a loggable value and in these cases this value can be logged directly. But in all other cases where the operator does not provide a loggable value for the defined macro, this operator may be used to do so. Please note that the value will be logged as a nominal value even if it is actually a numerical value.

You must be familiar with the basic concepts of macros and logging in order to understand this operator completely. Some basics of macros and logging are discussed in the coming paragraphs.

A macro can be considered as a value that can be used by all operators of the current process that come after the macro has been defined. Once the macro has been defined, the value of that macro can be used as a parameter value in coming operators by writing the macro name in `%(macro_name)` format in the parameter value where 'macro_name' is the name of the macro. Please note that many operators like many of the loop operators (e.g. Loop Values, Loop Attributes) also add specific macros. For more information regarding macros please study the Set Macro operator.

Logging and log-related operators store information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The Log is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example you want to see values of different parameters
in all iterations of any Loop operator. For more information regarding logging please study the Log operator.

Input Ports

through \((thr)\) It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first through input port of the Provide Macro as Log Value operator is available at the first through output port.

Output Ports

through \((thr)\) The object that was given as input is passed without changing to the output through this port. It is not compulsory to attach this port to any other port. The Provide Macro as Log Value operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Provide Macro as Log Value operator is delivered at the first through output port

Parameters

macro name \((string)\) This parameter specifies the name of the macro whose value should be provided for logging.
Tutorial Processes

Logging the value of a macro with or without the Provide Macro as Log Value operator

This Example Process shows how macro values can be logged with or without the Provide Macro as Log Value operator. The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. As you can see, there are 14 examples in the ExampleSet. The Extract Macro operator is applied on the ExampleSet. The macro of the extract macro operator is named 'eCount' and it stores the number of examples in the ExampleSet i.e. 14. The Provide Macro as Log Value operator is applied to provide the value of the 'eCount' macro as a loggable value. Finally the Log operator is applied to store the value of the 'eCount' macro in the log table. Have a look at the log parameter settings in the Log operator. Two columns have been defined: 'Direct' and 'Indirect'. The 'Direct' column gets the value of the 'eCount' macro directly from the Extract Macro operator. This only works if the macro is provided in loggable form by the operator. The 'Indirect' column gets the value of the 'eCount' macro from the Provide Macro as Log Value operator. This is how a macro value should be logged if it cannot be logged directly. Run the process and you will see two columns in the Table View of the results of the Log operator. Both columns have the value of the 'eCount' macro (14).
### 2.2. Logging

#### Log to Data

This operator transforms the data generated by the Log operator into an ExampleSet which can then be used by other operators of the process.

### Description

The Log operator stores information into the log table. This information can be almost anything including parameter values of operators, apply-count of operators, execution time etc. The Log operator is mostly used when you want to see the values calculated during the execution of the process that are otherwise not visible. For example you want to see values of different parameters in all iterations of any Loop operator. In such scenarios the ideal operator is the Log operator. A large variety of information can be stored using this operator. The information stored in the log table can be viewed using the Table View in the Results Workspace. But this information is not directly accessible in the process. To solve this problem, the Log to Data operator provides the information in the log table in form of an ExampleSet. This ExampleSet can be used in the process like any other ExampleSet. RapidMiner automatically guesses the type of attributes of this ExampleSet and all attributes have regular role. The type and role can be changed by using the corresponding operators.

### Input Ports

- **through (thr)** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first through input port of the Log to Data operator is available at the first
2. Utility

through output port.

Output Ports

data generates by the Log operator is delivered as an ExampleSet through this port.

to output through this port. It is not compulsory to attach this port to any other port. The Log to Data operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Log to Data operator is delivered at the first through output port.

Parameters

log name (string) This parameter specifies the name of the Log operator that generated the log data which should be returned as an ExampleSet. If this parameter is left blank then the first found data table is returned as an ExampleSet.

Tutorial Processes

Accessing Training vs Testing error using the Log and Log to Data operators

The 'Weighting' data set is loaded using the Retrieve operator. The Loop Parameters operator is applied on it. The parameters of the Loop Parameters operator are set such that this operator loops 25 times. Thus its subprocess is executed 25 times. In every iteration, the value of the C parameter of the SVM(LibSVM)
2.2. Logging

operator is changed. The value of the $C$ parameter is 0.001 in the first iteration. The value is increased logarithmically until it reaches 100000 in the last iteration.

Have a look at the subprocess of the Loop Parameters operator. First the data is split into two equal partitions using the Split Data operator. The SVM (LibSVM) operator is applied on one partition. The resultant classification model is applied using two Apply Model operators on both the partitions. The statistical performance of the SVM model on both testing and training partitions is measured using the Performance (Classification) operators. At the end the Log operator is used to store the required results.

The $log$ parameter of the Log operator stores four things.

1. The iterations of the Loop Parameter operator are counted by apply-count of the SVM operator. This is stored in a column named 'Count'.

2. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Training partition is stored in a column named 'Training Error'.

3. The value of the classification error parameter of the Performance (Classification) operator that was applied on the Testing partition is stored in a column named 'Testing Error'.

4. The value of the $C$ parameter of the SVM (LibSVM) operator is stored in a column named 'SVM C'.

In the main process, the Log to Data operator is used for providing the log values in form of an ExampleSet. The resultant ExampleSet is connected to the result port of the process and it can be seen in the Results Workspace. You can see the meta data of the ExampleSet in the Meta Data View and the values of the ExampleSet can be seen in the Data View. This ExampleSet can be used to study how classification errors in training and testing partitions behave with the increase in the value of the $C$ parameter of the SVM(LibSVM) operator. To view these results in graphical form, switch to the Plot View. Select an appropriate plotter. You can use 'Series Multiple' plotter with 'SVM-C' as the 'Index Dimension'. Select 'Training Error' and 'Testing Error' in the 'Plot Series'.
2. Utility

The 'scatter multiple' plotter can also be used. Now you can analyze how the training and testing error behaved with the increase in the parameter $C$. More importantly this ExampleSet is available in the process, so information stored in it can be used by other operators of the process.

![Main Process Diagram]

**Execute Process**

This operator embeds a complete process (previously written into a file) into the current process.

**Description**

This operator can be used to embed a complete process definition of a saved process into the current process definition. The saved process will be loaded and executed when the current process reaches this operator. Optionally, the input of this operator can be used as input for the embedded process. In both cases, the output of the saved process will be delivered as output of this operator. Please note that validation checks will not work for a process containing an operator of this type since the check cannot be performed without actually loading the process. The use of this operator can be easily understood by studying the attached Example Process.
Input Ports

**input** (*inp*) The Execute Process operator can have multiple inputs. When one *input* port is connected, another *input* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *input* port of the Execute Process operator is available at the first *input* port of the embedded process. Don't forget to connect all inputs in correct order. Make sure that you have connected the right number of ports.

Output Ports

**result** (*res*) The Execute Process operator can have multiple outputs. When one *result* port is connected, another *result* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The Object delivered at the first *output* port of the embedded process is delivered at the first *result* port of the Execute Process operator. Don't forget to connect all outputs in correct order. Make sure that you have connected the right number of ports.

Parameters

**process location** The location of the process to be embedded is provided here.

**use input** (*boolean*) This is an expert parameter. It indicates if the input of this operator should be used as input for the embedded process. This should always be set to true if you want to provide input to the embedded process through the current process.

**store output** (*boolean*) This is an expert parameter. It indicates if the operator output should be stored. This applies only if the context of the embedded process defines output locations.

**propagate metadata recursively** (*boolean*) This is an expert parameter. It determines whether meta data is propagated through the included process.
2. Utility

**cache process** *(boolean)* This is an expert parameter. It determines if the process should be loaded during execution. If it is checked, the process will not be loaded during the execution.

**macros** This is an expert parameter. It defines macros for this sub-process.

## Tutorial Processes

**The process to be used as an embedded process in the next Example Process**

This process does not use the Execute Process operator, rather it is used as an embedded process in the second Example Process. This process uses the Decision Tree operator twice. In both cases input is not provided to the operators in this process. The input ports of the operators are connected with the input ports of the process, thus these operators will receive inputs via another process as we shall see soon. Such a process cannot work at its own, because inputs are missing. Only the *mod* port is connected to the output in both operators. Always make sure that the input ports of this process are connected in the right way whenever you want this process to receive inputs from other processes.

![Diagram of the process](image_url)

### Executing an embedded process
2.3. Execution

The Execute Process operator is used in this process. The *process location* parameter supplies the location of the first Example Process. Make sure you provide the same location here that you used to save the first Example Process. The Retrieve operator is used twice, first it loads the 'Golf' data set and then it is used to load the 'Labor-Negotiations' data set. These data sets are sent as input to the embedded process. The object connected to the first *input* port of the Execute Process operator is received at the first *input* port of the embedded process. Thus the 'Golf' data set is provided as input to the first Decision Tree operator. Similarly, the 'Labor-negotiations' data set is provided as input to the second Decision Tree operator. Passing input to the embedded process was possible because the *use input* parameter was checked. If you uncheck it and run the process again, you will get an error message. Outputs of the embedded process are delivered as outputs of the Execute Process operator in the current process. The order of outputs remains the same. The Decision Tree model of the Labor-negotiations data set is connected to the second *res* port of the first Example Process, thus it is available at the second *res* port of the Execute Process operator in the current process.

![Diagram showing the process flow](image)

**Execute Script**

This operator executes Java code and/or Groovy scripts. This basically means that users can write their own operators directly within
the process by specifying Java code and/or a Groovy script which will be interpreted and executed during the process runtime.

Description

This is a very powerful operator because it allows you to write your own script. This operator should be used if the task you want to perform through your script cannot be performed by existing RapidMiner operators because writing scripts can be time-consuming and error-prone.

Groovy is an agile and dynamic language for the Java Virtual Machine. It builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk. Groovy integrates well with all existing Java classes and libraries because it compiles straight to Java bytecode so you can use it anywhere you can use Java. For a complete reference of Groovy scripts please refer to http://groovy.codehaus.org/.

In addition to the usual scripting code elements from Groovy, the RapidMiner scripting operator defines some special scripting elements:

- If the standard imports parameter is set to true, all important types like Example, ExampleSet, Attribute, Operator etc as well as the most important Java types like collections etc are automatically imported and can directly be used within the script. Hence, there is no need for importing them in your script. However, you can import any other class you want and use it in your script.

- The current operator (the scripting operator for which you define the script) is referenced by operator.
  - Example: operator.log(ttext"

- All operator methods like log (see above) that access the input or the complete process can directly be used by writing a preceding operator.
  - Example: operator.getProcess()
2.3. Execution

- Input of the operator can be retrieved via the input method `getInput(Class)` of the surrounding `operator`.
  - Example: `ExampleSet exampleSet = operator.getInput(ExampleSet.class)`

- You can iterate over examples with the following construct:
  - `for (Example example : exampleSet) { ... }`

- You can retrieve example values with the shortcut:
  - In case of non-numeric values: `String value = example["attribute_name"];
  - In case of numeric values: `double value = example["attribute_name"];

- You can set example values with the shortcut:
  - In case of non-numeric values: `example["attribute_name"] = "value";`
  - In case of numeric values: `example["attribute_name"] = 5.7;`

Please study the attached Example Processes for better understanding. Please note that Scripts written for this operator may access Java code. Scripts may hence become incompatible in future releases of RapidMiner.

Input Ports

`input (inp)` The Script operator can have multiple inputs. When one input is connected, another `input` port becomes available which is ready to accept another input (if any).

Output Ports

`output (out)` The Script operator can have multiple outputs. When one output is connected, another `output` port becomes available which is ready to deliver another output (if any).
2. Utility

Parameters

**script** The script to be executed is specified through this parameter.

**standard imports** *(boolean)* If the *standard imports* parameter is set to true, all important types like Example, ExampleSet, Attribute, Operator etc as well as the most important Java types like collections etc are automatically imported and can directly be used within the script. Hence, there is no need for importing them in your script. However, you can import any other class you want and use it in your script.

Tutorial Processes

Iterating over attributes for changing the attribute names to lower case

The 'Transactions' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can view the ExampleSet. Note that the names of all attributes of the ExampleSet are in upper case letters. The Script operator is applied on the ExampleSet. The script changes the attribute names to lower case letters. This can be verified by viewing the results in the Results Workspace.

Here is a brief description of what happens in the script. First the input of the operator is retrieved via the input method *getInput(Class)*. Then the *for loop* iterates for all attributes and uses the *toLowerCase()* method to change the names of the attributes to lower case letters. At the end, the modified ExampleSet is returned.

Please note that this is a very simple script, it was included here just to introduce you with working of this operator. This operator can be used to perform very complex tasks.
2.3. Execution

Iterating over all examples for changing the attribute values to upper case

The 'Transactions' data set is loaded using the Retrieve operator. A Breakpoint is inserted here so that you can view the ExampleSet. Note that the values of all attributes of the ExampleSet are in lower case letters. The Script operator is applied on the ExampleSet. The script changes the attribute values to upper case letters. This can be verified by viewing the results in the Results Workspace.

Here is a brief description of what happens in the script. First the input of the operator is retrieved via the input method getInput(Class). Then the outer for loop iterates for all attributes and stores the name of the current attribute in a string variable. Then the inner for loop iterates over all the examples of the current attribute and changes the values from lower to upper case using the toUpperCase() method. At the end, the modified ExampleSet is returned.

Please note that this is a very simple script, it was included here just to introduce you with working of this operator. This operator can be used to perform very complex tasks.
2. Utility

Subtracting mean of numerical attributes from attribute values

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. Note the values of the 'Temperature' and 'Humidity' attributes. The Script operator is applied on the ExampleSet. The script subtracts the mean of each numerical attribute from all values of that attribute. This can be verified by viewing the results in the Results Workspace.

Here is a brief description of what happens in the script. First the input of the operator is retrieved via the input method `getInput(Class)`. Then the outer `for loop` iterates for all attributes and stores the name of the current attribute in a string variable and the mean of this attribute in a double type variable. Then the inner `for loop` iterates over all the examples of the current attribute and subtracts the mean from the current value of the example. At the end, the modified ExampleSet is returned.

Please note that this is a very simple script, it was included here just to introduce you with working of this operator. This operator can be used to perform very complex tasks.
2.3. Execution

Execute SQL

This operator executes the specified SQL statement on the specified database.

Description

The Execute SQL operator executes the specified SQL statement on the specified SQL database. The SQL query can be specified through the query parameter. If the SQL query is in a file then the path of that file can be specified through the query file parameter. Please note that this operator cannot be used for loading data from databases. It can be used for executing SQL statements like CREATE or ADD etc. In order to load data from an SQL database, please use the Read Database operator. You need to have at least a basic understanding of databases, database connections and queries in order to use this operator properly. Please go through the parameters and the attached Example Process to understand the working of this operator.
Differentiation

**Read Database** The Read Database operator is used for loading data from a database into RapidMiner. The Execute SQL operator cannot be used for loading data from databases. It can be used for executing SQL statements like CREATE or ADD etc on the database. See page 210 for details.

**Input Ports**

*through (thr)* It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Execute SQL operator is available at the first *through* output port.

**Output Ports**

*through (thr)* The objects that were given as input are passed without changing to the output through this port. It is not compulsory to connect this port to any other port; the SQL command is executed even if this port is left without connections. The Execute SQL operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Execute SQL operator is delivered at the first *through* output port.
Parameters

**define connection** *(selection)* This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

**connection** *(selection)* This parameter is only available when the **define connection** parameter is set to **predefined**. This parameter is used for connecting to a database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down list. You can add a new connections or modify previous connections using the button next to the drop down list. You may also accomplish this by clicking on **Manage Database Connections**... from the **Tools** menu in the main window. A new window appears. This window asks for several details e.g. **Host**, **Port**, **Database system**, **schema**, **username** and **password**. The **Test** button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down list of the **connection** parameter. You need to have a basic understanding of databases for configuring a connection.

**database system** *(selection)* This parameter is only available when the **define connection** parameter is set to **url**. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

**database url** *(string)* This parameter is only available when the **define connection** parameter is set to **url**. This parameter is used for defining the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

**username** *(string)* This parameter is only available when the **define connection** parameter is set to **url**. This parameter is used for specifying the username of the database.

**password** *(string)* This parameter is only available when the **define connection** parameter is set to **url**. This parameter is used for specifying the password of the database.

**jndi name** *(string)* This parameter is only available when the **define connection** parameter is set to **jndi**. This parameter is used for specifying the JNDI name
2. Utility

for a data source.

**query** *(string)* This parameter is used for specifying the SQL query which will be executed on the specified database.

**query file** *(filename)* This parameter is used for selecting the file that contains the SQL query which will be executed on the specified database. Long queries are usually stored in files. Storing queries in files can also enhance reusability.

**prepare statement** *(boolean)* If checked, the statement is prepared, and '?' can be filled in using the **parameters** parameter.

**parameters** *(enumeration)* This parameter specifies the Parameters to insert into '?' placeholders when the statement is prepared.

**Related Documents**

*Read Database* (210)

**Tutorial Processes**

**Creating a new table in mySQL database**

The Execute SQL operator is used for creating a new table in an existing mySQL database. The **define connection** parameter is set to predefined. The **define connection** parameter was configured using the button next to the drop down list. The name of the connection was set to 'mySQLconn'. The following values were set in the connection parameter's wizard. The **Database system** was set to 'mySQL'. The **Host** was set to 'localhost'. The **Port** was set to '3306'. The **Database scheme** was set to 'golf'; this is the name of the database. The **User** was set to 'root'. No password was provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The **query** parameter is set to the following query: 'CREATE TABLE Weather(Temperature
2.3. Execution

INTEGER). This query creates a new table named Weather in the 'golf' database. This table has one integer attribute named Temperature. Run the process, you will not see any results in RapidMiner because this operator did not return anything. It simply executed the query on the specified database. So, in order to see the changes you can open the database and verify that a new table has been created.

![Diagram of a process flowchart]

**Execute Program**

This operator simply executes a command in the shell of the underlying operating system. It can execute any system command or external program.

**Description**

This operator executes a system command. The command and all its arguments are specified by the *command* parameter. Please note that the command is system dependent. The standard output stream of the process can be redirected to the log file by enabling the *log stdout* parameter. The standard error stream of the process can be redirected to the log file by enabling the *log stderr* parameter.

In Windows / MS DOS, simple commands should be preceded by 'cmd /c' call, e.g. 'cmd /c notepad'. Just writing 'notepad' in the *command* parameter will also work in case you are executing a program and not just a shell command. Then Windows opens a new shell, executes the command, and closes the shell again.
2. Utility

However, Windows 7 may not open a new shell, it just executes the command. Another option would be to precede the command with 'cmd /c start' which opens the shell and keeps it open. The rest of the process will not be executed until the shell is closed by the user. Please study the attached Example Processes for more information.

The java Runtime.exec(String) method is used for executing the command. Characters that have special meaning on the shell e.g. the pipe symbol or brackets and braces do not have a special meaning to Java. Please note, that this Java method parses the string into tokens before it is executed. These tokens are not interpreted by a shell. If the desired command involves piping, redirection or other shell features, it is best to create a small shell script to handle this.

Input Ports

**through (thr)** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first through input port of the Execute Program operator is available at the first through output port.

Output Ports

**through (thr)** The objects that were given as input are passed without changing to the output through this port. It is not compulsory to connect this port to any other port, the command is executed even if this port is left without connections. The Execute Program operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Execute Program operator
is delivered at the first *through* output port

## Parameters

*command* (*string*) This parameter specifies the command to be executed.

*log stdout* (*boolean*) If set to true, the *stdout* stream (standard output stream) of the command is redirected to the log file.

*log stderr* (*boolean*) If set to true, the *stderr* stream (standard error stream) of the command is redirected to the log file.

## Tutorial Processes

### Introduction to the Execute Program operator

This Example Process uses the Execute Program operator to execute commands in the shell of Windows 7. Two Execute Program operators are used. The *command* parameter of the first Execute Program operator is set to 'cmd /c java -version'. The *command* parameter of the second Execute Program operator is set to 'cmd /c notepad'. When the process is executed, first the java version is described in the log window. Then the notepad is opened. The process waits for the notepad to close. The process proceeds when the notepad is closed by the user. Please note that setting the *command* parameter to just 'notepad' would have also worked here.
2. Utility

Opening Internet Explorer by the Execute Program operator

This Example Process uses the Execute Program operator to open the Internet Explorer browser by using the shell commands of Windows 7. The *command* parameter of the Execute Program operator is set to 'cmd /c start C:\Program Files\Internet Explorer\iexplore.exe'. When the process is executed, the Internet Explorer browser opens. The process waits for the Internet Explorer browser to be closed by the user. The process proceeds when the Internet Explorer browser is closed.
2.4. Files

Write as Text

This operator writes the given results to the specified file. This operator can be used at each point in an operator chain to write results at every step of the process into a file.

Description

The Write as Text operator writes the given results to the specified file. The file is specified through the result file parameter. This operator does not modify the results; it just writes them to the file and then delivers the unchanged results through its output ports. Every input object which implements the ResultObject interface (which is the case for almost all objects generated by the core RapidMiner operators) will write its results to the file specified by the result file parameter. If the result file parameter is not set then the global result file parameter with the same name of the ProcessRootOperator (the root of the process) will be used. If this file is also not specified then the results are simply written to the console (standard out).

Input Ports

input (inp) Any results connected at this port are written to the specified file and then delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The result supplied at the first input port of the Write as Text operator is available at its first output port.
Output Ports

**input (inp)** The results that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the results are written into the file even if this port is left without connections. The Write as Text operator can have multiple outputs. When one output is connected, another output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The result connected at the first input port of the Write as Text operator is delivered through the first output port.

Parameters

**result file (filename)** The results are written into the file specified through this parameter.

**encoding (selection)** This is an expert parameter. There are different options, users can choose any of them.

Tutorial Processes

Writing multiple results into a file

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it with default values of all parameters. The Default Model operator trains a model on the 'Golf' data set in the training subprocess. The trained model is applied on the testing data set in the testing subprocess by the Apply Model operator. The Performance operator measures the performance of the model. The Split Validation operator delivers multiple results i.e. a trained model, the input ExampleSet and the performance vector. All these results are connected to the Write as Text operator to write them into a file. The
result file parameter is set to 'D:\results.txt', thus a text file named 'results' is created in the D drive of the computer. If the file already exists then the results are appended to the file. The results are written into the specified file but they are not as detailed as the results in the Results Workspace.

Copy File

This operator copies the chosen file to the specified destination.

Description

The Copy File operator copies the file specified in its parameters to the indicated destination. If the inserted path does not already exist the needed folders and the file are created. It is also possible to overwrite an existing document.

Input Ports

through (thr) It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the
first through input port of the Copy File operator is available at the first through output port.

Output Ports

through (thr) The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is copied even if this port is left without connections. The Copy File operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Copy File operator is delivered at the first through output port.

Parameters

source file (file) The file that should be copied is specified through this parameter.

new file (file) The copied file is saved as the file and at the target specified through this parameter.

overwrite (boolean) If this parameter is set to true a file specified in the new file parameter that already existed is replaced by the file specified in the source file parameter.

Tutorial Processes

Writing the Labor-Negotiations data set into an Excel file and copying it
This Example Process shows how the Copy File operator can be used to copy a specified file. For this Example Process you first need a file to copy. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. For further understanding of this operator please read the description of the Write Excel operator.

The Copy File operator is inserted without any connections. The source file parameter is set to 'D:\Labor data set.xls' and the new file parameter to 'D:\test\Labor data set.xls'. Run the process and two files are created in your D drive, the 'Labor data set.xls' of the Write Excel operator and the 'Labor data set.xls' in a separate folder named 'test'.

### Rename File

This operator renames a file or a folder.
2. Utility

Description

The Rename File operator allocates a new name to a selected file or folder. Please ensure that the new name of your file has the right ending, e.g. `.xls` as in our Example Process.

Input Ports

*through (thr)* It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Rename File operator is available at the first *through* output port.

Output Ports

*through (thr)* The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is renamed even if this port is left without connections. The Rename File operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Rename File operator is delivered at the first *through* output port.
2.4. Files

Parameters

file *(file)* The file or folder that should be renamed is specified through this parameter.
new name *(string)* The new name of the file or folder is specified through this parameter.

Tutorial Processes

Writing the Labor-Negotiations data set into an Excel file and renaming it

This Example Process shows how the Rename File operator can be used to rename a specified file. For this Example Process you first need a file to rename. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. A breakpoint is inserted so you can have a look at the Excel file in your D drive. For further understanding of this operator please read the description of the Write Excel operator.

The Rename File operator is inserted without any connections. The file parameter is set to 'D:\Labor data set.xls' and the new name parameter to 'labor_data_set.xls'. Run the process and the file in your D drive which was formerly named as 'Labor data set.xls' will now be named as labor_data_set.xls'.
Delete File

This operator deletes a file at the specified location.

Description

The Delete File operator deletes the selected file if possible otherwise an error message is shown. You can also specify that an error message should be shown if the file that should be deleted does not exist.

Input Ports

through (thr) It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected,
another *through* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first *through* input port of the Delete File operator is available at the first *through* output port.

**Output Ports**

*through* (*thr*) The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is deleted even if this port is left without connections. The Delete File operator can have multiple outputs. When one output is connected, another *through* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first *through* input port of the Delete File operator is delivered at the first *through* output port.

**Parameters**

*file* (*file*) The file that should be deleted is specified through this parameter.

*fail if missing* (*boolean*) Determines whether an exception should be generated if the file is missing, e.g. because it already got deleted in the last run. If set to false nothing happens if this error occurs.

**Tutorial Processes**

**Writing the Labor-Negotiations data set into an Excel file and copying it**

This Example Process shows how the Delete File operator can be used to delete
a specified file. For this Example Process you first need a file to delete. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. A breakpoint is inserted so you can have a look at the Excel file in your D drive. For further understanding of this operator please read the description of the Write Excel operator.

The Delete File operator is inserted without any connections. The file parameter is set to 'D:\Labor data set.xls' and the fail if missing parameter to true. Run the process and the file in your D drive will be deleted. If you delete the Retrieve and the Write Excel operator and run the process again an error message will be shown telling you that the file could not be deleted as it does not exist.

### Move File

This operator moves the chosen file to the specified destination.
Description

The Move File operator moves the selected file from its original directory to the chosen location. If the inserted path does not already exist the needed folders and the file are created. It is also possible to overwrite an existing document.

Input Ports

through (thr) It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another through input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The object supplied at the first through input port of the Move File operator is available at the first through output port.

Output Ports

through (thr) The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the file is moved even if this port is left without connections. The Move File operator can have multiple outputs. When one output is connected, another through output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first through input port of the Move File operator is delivered at the first through output port.
Parameters

file (file) The file that should be moved.
destination (file) The new location of the file.
overwrite (boolean) Determines whether an already existing file should be over-written.

Tutorial Processes

Writing the Labor-Negotiations data set into an Excel file and moving it

This Example Process shows how the Move File operator can be used to move a specified file. For this Example Process you first need a file to move. An Excel file is created by loading the 'Labor-Negotiations' data set with the Retrieve operator and writing it into an Excel file with the Write Excel operator. The file is saved as 'D:\Labor data set.xls' if this file does not already exist. A breakpoint is inserted so you can have a look at the Excel file in your D drive. For further understanding of this operator please read the description of the Write Excel operator.

The Move File operator is inserted without any connections. The file parameter is set to 'D:\Labor data set.xls' and the destination parameter to 'C:\Data\Labor data set.xls'. Run the process and the file in your D drive will be moved to a newly created folder named 'Data' in your C drive.
2.4. Files

Create Directory

This operator creates a directory at the specified location.

Description

The Create Directory operator creates a directory at the chosen location in the file system, if the folder does not already exist. If the inserted path does not exist the needed folders are created.

Input Ports

**through (thr)** It is not compulsory to connect any object with this port. Any object connected at this port is delivered without any modifications to the output port. This operator can have multiple inputs. When one input is connected, another **through** input port becomes available which is ready to accept another
input (if any). The order of inputs remains the same. The object supplied at the first \textit{through} input port of the Create Directory operator is available at the first \textit{through} output port.

**Output Ports**

\textit{through (thr)} The objects that were given as input are passed without changing to the output through this port. It is not compulsory to attach this port to any other port, the directory is created even if this port is left without connections. The Create Directory operator can have multiple outputs. When one output is connected, another \textit{through} output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object delivered at the first \textit{through} input port of the Create Directory operator is delivered at the first \textit{through} output port.

**Parameters**

\textit{location (file)} The location where the new directory is build is specified through this parameter.

\textit{name (string)} The name of the new directory is specified through this parameter.

**Tutorial Processes**

Creating a new directory in the D drive

The Create directory operator is inserted in the process. The \textit{location} parameter is set to 'D:\new' and the \textit{name} parameter to 'folder'. Run the process and a folder named 'folder' is created in your D drive inside the newly created folder 'new'.

162
2.5. Data Generation

Generate Data

This operator generates an ExampleSet based on numerical attributes. The number of attributes, number of examples, lower and upper bounds of attributes, and target function can be specified by the user.

Description

The Generate Data operator generates an ExampleSet with a specified number of numerical attributes which is controlled by the number of attributes parameter. Please note that in addition to the specified number of regular attributes, the label attribute is automatically generated by applying the function selected by the target function parameter. The selected target function is applied on the attributes to generate the label attribute. For example if the number of attributes parameter is set to 3 and the target function is set to 'sum'. Then three regular numerical attributes will be created. In addition to these regular attributes a label attribute will be generated automatically. As the target function is set to 'sum', the label attribute value will be the sum of all three regular attribute values.
2. Utility

Output Ports

output (out) The Generate Data operator generates an ExampleSet based on numerical attributes which is delivered through this port. The meta data is also delivered along with the data. This output is same as the output of the Retrieve operator.

Parameters

target function (selection) This parameter specifies the target function for generating the label attribute. There are different options; users can choose any of them.

number examples (integer) This parameter specifies the number of examples to be generated.

number of attributes (integer) This parameter specifies the number of regular attributes to be generated. Please note that the label attribute is generated automatically besides these regular attributes.

attributes lower bound (real) This parameter specifies the minimum possible value for the attributes to be generated. In other words this parameter specifies the lower bound of the range of possible values of regular attributes.

attributes upper bound (real) This parameter specifies the maximum possible value for the attributes to be generated. In other words this parameter specifies the upper bound of the range of possible values of regular attributes.

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

data management (selection) This is an expert parameter. A long list is pro-
2.5. Data Generation

vided; users can select any option from this list.

Tutorial Processes

Introduction to the Generate Data operator

The Generate Data operator is applied for generating an ExampleSet. The target function parameter is set to 'sum', thus the label attribute will be the sum of all attributes' values. The number examples parameter is set to 100, thus the ExampleSet will have 100 examples. The number of attributes parameter is set to 3, thus three numerical attributes will be generated beside the label attribute. The attributes lower bound and attributes upper bound parameters are set to -10 and 10 respectively, thus values of the regular attributes will be within this range. You can verify this by viewing the results in the Results Workspace. The use local random seed parameter is set to false in this Example process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.

Generate Nominal Data

This operator generates an ExampleSet based on nominal attributes. The number of examples, number of attributes, and number of values can be specified by the user.
2. Utility

Description

The Generate Nominal Data operator generates an ExampleSet with the specified number of nominal attributes which is controlled by the number of attributes parameter. Please note that in addition to the specified number of regular attributes, the label attribute is automatically generated. The label attribute generated by this operator has only two possible values i.e. positive and negative. This operator is used for generating a random ExampleSet for testing purposes.

Output Ports

output (out) The Generate Nominal Data operator generates an ExampleSet based on nominal attributes which is delivered through this port. The meta data is also delivered along with the data. This output is same as the output of the Retrieve operator.

Parameters

number examples (integer) This parameter specifies the number of examples to be generated.

number of attributes (integer) This parameter specifies the number of regular attributes to be generated. Please note that the label attribute is generated automatically besides these regular attributes.

number of values (integer) This parameter specifies the number of unique values of the attributes.

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

local random seed (integer) This parameter specifies the local random seed.
2.5. Data Generation

This parameter is only available if the *use local random seed* parameter is set to true.

Tutorial Processes

Introduction to the Generate Nominal Data operator

The Generate Nominal Data operator is applied for generating an ExampleSet. The *number examples* parameter is set to 100, thus the ExampleSet will have 100 examples. The *number of attributes* parameter is set to 3, thus three nominal attributes will be generated beside the label attribute. The *number of values* parameter is set to 5, thus each attribute will have 5 possible values. You can verify this by viewing the results in the Results Workspace. The *use local random seed* parameter is set to false in this Example process. Set the *use local random seed* parameter to true and run the process with different values of *local random seed*. You will see that changing the values of *local random seed* changes the randomization.

Generate Direct Mailing Data

This operator generates an ExampleSet that represents direct mailing data. The number of examples can be specified by the user.
2. Utility

Description

The Generate Direct Mailing Data operator generates an ExampleSet that represents direct mailing data. This ExampleSet can be used when you do not have a data set that represents a real direct mailing data. This ExampleSet can be used as a placeholder for such a requirement. This data set has 8 regular attributes and 1 special attribute. The regular attributes are name (nominal), age (integer), lifestyle (nominal), zip code (integer), family status (nominal), car (nominal), sports (nominal) and earnings (integer). The special attribute is label (nominal). The number of examples in the data set can be set by the *number examples* parameter. To have a look at this ExampleSet, just run the attached Example Process.

Output Ports

*output (out)* The Generate Direct Mailing Data operator generates an ExampleSet which is delivered through this port. The meta data is also delivered along with the data. This output is same as the output of the Retrieve operator.

Parameters

number examples *(integer)* This parameter specifies the number of examples to be generated.

use local random seed *(boolean)* This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.

local random seed *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.
Tutorial Processes

Introduction to the Generate Direct Mailing Data operator

The Generate Direct Mailing Data operator is applied for generating an ExampleSet that represents direct mailing data. The number examples parameter is set to 10000, thus the ExampleSet will have 10000 examples. You can see the ExampleSet in the Results Workspace. The use local random seed parameter is set to false in this Example Process. Set the use local random seed parameter to true and run the process with different values of local random seed. You will see that changing the values of local random seed changes the randomization.

Generate Sales Data

This operator generates an ExampleSet that represents sales data. The number of examples can be specified by the user.
Description

The Generate Sales Data operator generates an ExampleSet that represents sales data. This ExampleSet can be used when you do not have a data set that represents actual sales data. This ExampleSet can be used as a placeholder for such a requirement. This data set has 7 regular attributes and 1 special attribute. The regular attributes are store_id (nominal), customer_id (nominal), product_id (integer), product_category (nominal), date (date), amount (integer) and single_price (real). The special attribute is transaction_id (integer) which is an id attribute. The number of examples in the data set can be set by the number examples parameter. To have a look at this ExampleSet, just run the attached Example Process.

Output Ports

output (out) The Generate Sales Data operator generates an ExampleSet which is delivered through this port. The meta data is also delivered along with the data. This output is same as the output of the Retrieve operator.

Parameters

number examples (integer) This parameter specifies the number of examples to be generated.
use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of values.
local random seed (integer) This parameter specifies the local random seed. This parameter is available only if the use local random seed parameter is set to true.
Tutorial Processes

Introduction to the Generate Sales Data operator

The Generate Sales Data operator is applied for generating an ExampleSet that represents sales data. The *number examples* parameter is set to 10000, thus the ExampleSet will have 10000 examples. You can see the ExampleSet in the Results Workspace. The *use local random seed* parameter is set to false in this Example Process. Set the *use local random seed* parameter to true and run the process with different values of *local random seed*. You will see that changing the values of *local random seed* changes the randomization.

Add Noise

This operator adds noise in the given ExampleSet by adding random attributes to the ExampleSet and by adding noise in the existing attributes.

Description

The Add Noise operator provides a number of parameters for selecting the attributes for adding noise in them. This operator can add noise to the label attribute or to the regular attributes separately. In case of a numerical label the given label noise (specified by the *label noise* parameter) is the percentage of
the label range which defines the standard deviation of normal distributed noise which is added to the label attribute. For nominal labels the label noise parameter defines the probability to randomly change the nominal label value. In case of adding noise to regular attributes the default attribute noise parameter simply defines the standard deviation of normal distributed noise without using the attribute value range. Using the parameter list is also possible for setting different noise levels for different attributes (by using the noise parameter). However, it is not possible to add noise to nominal attributes.

The Add Noise operator can add random attributes to the ExampleSet. The number of random attributes is specified by the random attributes parameter. New random attributes are simply filled with random data which is not correlated to the label at all. The offset and linear factor parameters are available for adjusting the values of new random attributes.

Input Ports

data

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

data

example set output (exa) Noise is added to the given ExampleSet and the resultant ExampleSet is delivered through this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.
2.5. Data Generation

Parameters

attribute filter type *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter *(attribute)* becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters *(regular expression, use except expression)* become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters *(value type, use value type exception)* become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters *(block type, use block type exception)* become visible in the Parameter View.
2. Utility

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** (*string*) The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** (*string*) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** (*string*) The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** (*boolean*) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameter View.

**except regular expression** (*string*) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** (*selection*) The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.
2.5. Data Generation

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like ' ( > 0 & & < 2 ) || ( > 10 & & < 12 )' are not allowed because they use both & & and ||. Use a blank space after ' > ', ' = ' and ' < ' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**random attributes** *(integer)* This parameter specifies the required number of
new random attributes to add to the input ExampleSet.

**label noise (real)** This parameter specifies the noise to be added in the label attribute. In case of a numerical label the given label noise is the percentage of the label range which defines the standard deviation of normal distributed noise which is added to the label attribute. For nominal labels the *label noise* parameter defines the probability to randomly change the nominal label value.

**default attribute noise (real)** This parameter specifies the default noise for all the selected regular attributes. The *default attribute noise* parameter simply defines the standard deviation of normal distributed noise without using the attribute value range.

**noise (list)** This parameter gives the flexibility of adding different noises to different attributes by providing a list of noises for all attributes.

**offset (real)** The *offset* value is added to the values of all the random attributes created by this operator.

**linear factor (real)** The *linear factor* value is multiplied with the values of all the random attributes created by this operator.

**use local random seed (boolean)** This parameter indicates if a *local random seed* should be used for randomization.

**local random seed (integer)** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

## Tutorial Processes

### Adding noise to the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Add Noise operator is applied on it. The *attribute filter type* parameter is set to 'all', thus noise will be added to all the attributes of the ExampleSet. The *label noise* and *default attribute noise* parameters are set to 0.05 and 0.06 respectively. The *random attributes* parameter is set to 1, thus a new random attribute will be added to the ExampleSet. The ExampleSet with noise and the original ExampleSet are
connected to the result ports of the process. Both ExampleSets can be seen in the Results Workspace. You can see that there is a new random attribute in the ExampleSet generated by the Add Noise operator. By comparing the values of both ExampleSets you can see how noise was added by the Add Noise operator.

Materialize Data

This operator creates a fresh and clean copy of the data in the memory.

Description

The Materialize Data operator creates a fresh and clean copy of the data in the memory. It might be useful after large preprocessing chains with a lot of views or even data copies. In such cases, it can be especially useful in combination with a memory cleanup operator e.g. the Free Memory operator.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input.
2. Utility

Output Ports

example set output *(exa)* The fresh and clean copy of the ExampleSet is delivered through this port.

original *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

datamanagement *(selection)* This is an expert parameter. There are different options, users can choose any of them

Tutorial Processes

Creating fresh copy of an ExampleSet

This is a very simple Example Process which just shows how to use the Materialize Data operator. The 'Labor-Negotiations' data set is loaded using the Retrieve operator. The Subprocess operator is applied on it. No operator is applied in the Subprocess operator because it is used as a dummy operator here. Suppose we have large preprocessing chains with a lot of views or even data copies in the subprocess and we want a fresh copy of data after the subprocess is complete. The Materialize Data operator is applied after the Subprocess operator to create a fresh and clean copy of the data. No large preprocessing tasks were performed in this Example Process because this Process was intended to discuss only the way this operator can be applied.
Free Memory

This operator frees unused memory. It might be useful after large preprocessing chains with a lot of currently unused views or even data copies.

Description

The Free Memory operator cleans up unused memory resources. It might be very useful in combination with the Materialize Data operator after large preprocessing trees using a lot of views or data copies. It can be very useful after the data set was materialized in memory. Internally, this operator simply invokes a garbage collection from the underlying Java programming language.

Please note that RapidMiner keeps data sets in memory as long as possible. So if there is any memory left, RapidMiner will not discard previous results of the process or data at the port. The Free Memory operator can be useful if you get the OutOfMemoryException. Also note that operators like the Remember operator put the objects in the Store. The Free Memory operator does not clean up the store. This operator will only free memory which is no longer needed which is not the case if the object is in the Store.

After process execution has been completed, everything including the Stores is freed, but only if needed! So you can take a look at your previous results in the Result History as long as they fit into the memory. RapidMiner will automatically
discard all these Results if a currently running process or new result needs free
memory. So the memory usage will constantly grow with the time until it has
reached a peak value and RapidMiner starts to discard previous results.

Output Ports

through (thr) The objects that were given as input are passed without changing
to the output through this port. It is not compulsory to attach this port to any
other port, the memory is freed even if this port is left without connections. The
Free Memory operator can have multiple outputs. When one output is connected,
another through output port becomes available which is ready to deliver another
output (if any). The order of outputs remains the same. The object delivered at
the first through input port of the Free Memory operator is delivered at the first
through output port.

Tutorial Processes

Introduction to the Free Memory operator
This is a very simple Example Process which just shows how to use the Free Memory operator. First the Subprocess operator is applied. Suppose we have a memory intensive task in the subprocess and we want to free unused memory after the subprocess is complete. The Free Memory operator is applied after the Subprocess operator to free the unused memory. No memory intensive task is performed in this Example Process. This Process was intended to discuss only the way this operator can be applied. Please make sure that the operators are applied in correct sequence. Also note that the Free Memory operator is not connected to any other operator but still it can perform its task.
3 Repository Access

Retrieve

This operator reads an object from the data repository.

Description

This operator can be used to access the repositories. It should replace all file access, since it provides full meta data processing, which eases the usage of Rapid-Miner a lot. In contrast to accessing a raw file, it provides the complete meta data of the data, so all meta data transformations are possible.

An easier way to load an object from the repository is to drag and drop the required object from the Repositories View. This will automatically insert a Retrieve operator with correct path of the desired object.

This operator has no input port. All it requires is a valid value in repository entry parameter.

Output Ports

output (out) It returns the object whose path was specified in repository entry parameter.
3. Repository Access

Parameters

**repository entry (string)** A valid path should be specified here in order to load an object. This parameter references an entry in the repository which will be returned as the output of this operator. Repository locations are resolved relative to the repository folder containing the current process. Folders in the repository are separated by a forward slash (/), a "." references the parent folder. A leading forward slash references the root folder of the repository containing the current process. A leading double forward slash is interpreted as an absolute path starting with the name of a repository.

- 'MyData' looks up an entry 'MyData' in the same folder as the current process.
- '../Input/MyData' looks up an entry 'MyData' located in a folder 'Input' next to the folder containing the current process.
- '/data/Model' looks up an entry 'Model' in a top-level folder 'data' in the repository holding the current process.
- '//'Samples/data/Iris' looks up the Iris data set in the 'Samples' repository.

Tutorial Processes

Retrieving Golf from Repository

The Example Process loads Golf data set from repository. **Repository entry** parameter is provided with path '//'Samples/data/Golf', thus Golf data set is returned from Samples repository. As it can be seen in Results Workspace, Retrieve operator loads both data and meta data.
Store

This operator stores an IO Object in the data repository.

Description

This operator stores an IO Object at a location in the data repository. The location of the object to be stored is specified through the repository entry parameter. The stored object can be used by other processes by using the Retrieve operator. Please see the attached Example Processes to understand the basic working of this operator. The Store operator is used to store an ExampleSet and a model in the Example Processes.

Input Ports

input (inp) This port expects an IO Object. In the attached Example Processes an ExampleSet and a model are provided as input.

Output Ports

through (thr) The IO Object provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same IO Object in further operators of the process.
3. Repository Access

Parameters

**Repository entry (string)** This parameter is used to specify the location where the input IO Object is to be stored.

Tutorial Processes

Storing an ExampleSet using the Store operator

This Process shows how the Store operator can be used to store an ExampleSet. The 'Golf' data set and the 'Golf-Testset' data set are loaded using the Retrieve operator. These ExampleSets are merged using the Append operator. The resultant ExampleSet is named 'Golf-Complete' and stored using the Store operator. The stored ExampleSet is used in the third Example Process.

Storing a model using the Store operator

This Process shows how the Store operator can be used to store a model. The
'Golf' data set is loaded using the Retrieve operator. The Naive Bayes operator is applied on it and the resultant model is stored in the repository using the Store operator. The model is stored with the name 'Golf-Naive-Model'. The stored model is used in the third Example Process.

Using the objects stored by the Store operator

This Process shows how a stored IO Object can be used. The 'Golf-Complete' data set stored in the first Example Process and the 'Golf-Naive-Model' stored in the second Example Process is loaded using the Retrieve operator. The Apply Model operator is used to apply the 'Golf-Naive-Model' on the 'Golf-Complete' data set. The resultant labeled ExampleSet can be viewed in the Results Workspace.
3. Repository Access
4 Import

Read csv

This operator is used to read CSV files.

Description

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form. CSV files have all values of an example in one line. Values for different attributes are separated by a constant separator. It may have many rows. Each row uses a constant separator for separating attribute values. CSV name suggests that the attributes values would be separated by commas, but other separators can also be used.

For complete understanding of this operator read the parameters section thoroughly. The easiest and shortest way to import a CSV file is to use import configuration wizard from Parameters View. The best way, which may require some extra effort, is to first set all the parameters in Parameters View and then use the wizard. Please make sure that the CSV file is read correctly before building a process using it.
4. Import

Output Ports

output (out) this port delivers the CSV file in tabular form along with the meta data. This output is similar to the output of Retrieve operator.

Parameters

Import Configuration Wizard (menu) This option allows you to configure this operator by means of a wizard. This user-friendly wizard makes use of this operator easy.

csv file (string) path of the CSV file is specified here. It can be selected using choose a file button.

column separators (string) Column separators for CSV files can be specified here in a regular expression format. A good understanding of regular expression can be developed from studying Select Attributes operator's description and Example Processes.

trim lines (boolean) This option indicates if lines should be trimmed (empty spaces are removed at the beginning and the end) before the column split is performed. This option might be problematic if TABs are used as separators.

use quotes (boolean) This option indicates if quotes should be regarded. Quotes can be used to store special characters like column separators. For example if (,) is set as column separator and (") is set as quotes character. (a,b,c,d) will be translated as 4 values for 4 columns. On the other hand (ä,b,c,d") will be translated as a single column value a,b,c,d. If this option is set to false, quotes character parameter and escape character parameter for quotes cannot be defined.

quotes character (char) This option defines the quotes character.

escape character for quotes (char) This is the character that is used to escape quotes. For example if (") is used as quotes character and (\) is used as escape character. ("yes") will be translated as (yes) and (\"yes\") will be translated as ("yes").

skip comments (boolean) Skip comments option is used to ignore comments in
csv file. This is useful only if the csv file has comments. If this option is set to true, a comment character should be defined using comment characters parameter.

**comment characters** *(string)* Lines beginning with these characters are ignored. If this character is present in the middle of the line, anything that comes in that line after this character is ignored. Remember that comment character itself is also ignored.

**parse numbers** *(boolean)* Specifies whether numbers are parsed or not.

**decimal character** *(char)* This character is used as the decimal character.

**grouped digits** *(boolean)* This option decides whether grouped digits should be parsed or not. If this option is set to true, grouping character parameter should be specified.

**grouping character** *(char)* This character is used as the grouping character. If this character is found between numbers, the numbers are combined and this character is ignored. For example if ”22-14¨ is present in csv file and ¨ is set as grouping character, then ”2214“ will be stored.

**date format** *(string)* The date and time format is specified here. Many predefined options exist; users can also specify a new format. If text in a csv file column matches this date format, that column is automatically converted to date type. Some corrections are automatically made in date type values. For example a value '32-March' will automatically be converted to '1-April'. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the date format parameter. If they do, this column of the csv file will be automatically parsed as date and the according attribute will be of date type.

**first row as names** *(boolean)* If this option is set to true, it is assumed that the first line of the csv file has the names of attributes. Then attributes are automatically named and first line of csv file is not treated as a data line.

**annotations** *(menu)* If first row as names is not set to true, annotations can be added using 'Edit List' button of this parameter which opens a new menu. This menu allows you to select any row and assign an annotation to it. Name, Comment and Unit annotations can be assigned. If row 0 is assigned Name annotation, it is equivalent to setting first row as names parameter to true. If you want to ignore any rows you can annotate them as Comment. Remember row
4. Import

number in this menu does not count commented lines.

time zone *(selection)* This is an expert parameter. A long list of time zones is provided; users can select any of them.

locale *(selection)* This is an expert parameter. A long list of locales is provided; users can select any of them.

encoding *(selection)* This is an expert parameter. A long list of encodings is provided; users can select any of them.

data set meta data information *(menu)* This option is an important one. It allows you to adjust the meta data of the csv file. Column index, name, type and role can be specified here. Read CSV operator tries to determine an appropriate type of the attributes by reading the first few lines and checking the occurring values. If all values are integers, the attribute will become an integer. Similarly if all values are real numbers, the attribute will become of type real. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the date format parameter. If they do, this column of the csv file will be automatically parsed as date and the according attribute will be of type date. Automatically determined types can be overridden using this parameter.

read not matching values as missings *(boolean)* If this value is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?'. For example if 'abc' is written in an integer column, it will be treated as a missing value. Question mark (?) in CSV file is also read as missing value.

data management *(selection)* This is an expert parameter. A long list is provided; users can select any option from this list.

Tutorial Processes

Reading a CSV file

Save the following text in a text file and load it with the given Read CSV Example Process. Run the process and compare the results in Results Workspace
4.1. Data

(data view) with the csv file.

att1,att2,att3,att4 # row 1

80.6, yes , 1996.JAN.21 ,22-14 # row 2


13.5,\nno\",1998.AUG.22,23-14 # row 4

23.3,yes,1876.JAN.32,42-65# row 5

21.6,yes,2001.JUL.12,xyz # row 6

12.56,",?",2002.SEP.18,15-90# row 7

Here is some explanation of what happens in this process:

- '#' is defined as comment character so 'row no.' is ignored in all rows.

- As first row as names parameter is set to true att1,att2,att3 and att4 are set as names of attributes

- att1 is set as real , att2 as polynomial, att3 as date and att4 as real

- in attribute att4 ,'.' are ignored because grouped digits parameter is set to true and grouping character = '.'

- In row 2 white spaces at start and at end of values are ignored because trim lines parameter is set to true.

- In row 3 quotes are used but they are ignored because escape character is not used.

- In row 4 escape quote is used, so quotes are not ignored.

- In row 5 date value is automatically corrected, 'jan.32' is changed to 'feb.1'.

- In row 6 invalid real value in forth column is replaced by '?' because read not matching values as missings parameter is set to true.
4. Import

- In row 7 quotes are used to store special characters including \textit{column separator} and question mark.

\textbf{Read Excel}

This operator reads an ExampleSet from the specified Excel file.

\textbf{Description}

This operator can be used to load data from Microsoft Excel spreadsheets. This operator is able to read data from Excel 95, 97, 2000, XP, and 2003. The user has to define which of the spreadsheets in the workbook should be used as data table. The table must have a format such that each row is an example and each column represents an attribute. Please note that the first row of the Excel sheet might be used for attribute names which can be indicated by a parameter. The data table can be placed anywhere on the sheet and can contain arbitrary formatting instructions, empty rows and empty columns. Missing data values in Excel should be indicated by empty cells or by cells containing only "?".

For complete understanding of this operator read the parameters section. The easiest and shortest way to import an Excel file is to use the \textit{import configuration wizard} from the Parameters View. The best way, which may require some extra effort, is to first set all the parameters in the Parameters View and then use the \textit{wizard}. Please make sure that the Excel file is read correctly before building a process using it.
4.1. Data

Output Ports

output (out) This port delivers the Excel file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

import configuration wizard This option allows you to configure this operator by means of a wizard. This user-friendly wizard makes the use of this operator easy.

excel file The path of the Excel file is specified here. It can be selected using the choose a file button.

sheet number (integer) The number of the sheet which you want to import should be specified here.

imported cell range This is a mandatory parameter. The range of cells to be imported from the specified sheet is given here. It is specified in 'xm:yn' format where 'x' is the column of the first cell of range, 'm' is the row of the first cell of range, 'y' is the column of the last cell of range, 'n' is the row of the last cell of range. 'A1:E10' will select all cells of the first five columns from row 1 to 10.

first row as names (boolean) If this option is set to true, it is assumed that the first line of the Excel file has the names of attributes. Then the attributes are automatically named and the first line of Excel file is not treated as a data line.

annotations If the first row as names parameter is not set to true, annotations can be added using the 'Edit List' button of this parameter which opens a new menu. This menu allows you to select any row and assign an annotation to it. Name, Comment and Unit annotations can be assigned. If row 0 is assigned Name annotation, it is equivalent to setting the first row as names parameter to true. If you want to ignore any rows you can annotate them as Comment.

date format The date and time format is specified here. Many predefined options exist; users can also specify a new format. If text in an Excel file column matches this date format, that column is automatically converted to date type. Some corrections are automatically made in the date type values. For example a
4. Import

value '32-March' will automatically be converted to '1-April'. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the date format parameter. If they do, this column of the Excel file will be automatically parsed as date and the according attribute will be of date type.

time zone This is an expert parameter. A long list of time zones is provided; users can select any of them.

locale This is an expert parameter. A long list of locales is provided; users can select any of them.

data set meta data information This option is an important one. It allows you to adjust the meta data of the ExampleSet created from the specified Excel file. Column index, name, type and role can be specified here. The Read Excel operator tries to determine an appropriate type of the attributes by reading the first few lines and checking the occurring values. If all values are integers, the attribute will become an integer. Similarly if all values are real numbers, the attribute will become of type real. Columns containing values which can't be interpreted as numbers will be interpreted as nominal, as long as they don't match the date and time pattern of the date format parameter. If they do, this column of the Excel file will be automatically parsed as date and the according attribute will be of type date. Automatically determined types can be overridden using this parameter.

read not matching values as missings (boolean) If this value is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?'. For example if 'abc' is written in an integer column, it will be treated as a missing value. A question mark (?) or an empty cell in the Excel file is also read as a missing value.

data management This is an expert parameter. A long list is provided; users can select any option from this list.

Tutorial Processes

Reading an ExampleSet from an Excel file
4.1. Data

For this Example Process you need an Excel file first. The one of this Example Process was created by copying the 'Golf' data set present in the Repositories into a new Excel file which was named 'golf'. The data set was copied on sheet 1 of the Excel file thus the sheet number parameter is given value 1. Make sure that you provide the correct location of the file in the Excel file parameter. The first cell of the sheet is A1 and last required cell is E15, thus the imported cell range parameter is provided value 'A1:E15'. As the first row of the sheet contains names of attributes, the first row as names parameter is checked. The remaining parameters were used with default values. Run the process, you will see almost the same results as you would have gotten from using the Retrieve operator to retrieve the 'Golf' data set from the Repository. You will see a difference in the meta data though, for example here the types and roles of attributes are different from those in the 'Golf' data set. You can change the role and type of attributes using the data set meta data information parameter. It is always good to make sure that all attributes are of desired role and type. In this example one important change that you would like to make is to change the role of the Play attribute. Its role should be changed to label if you want to use any classification operators on this data set.

Read SAS

This operator is used for reading an SAS file.
4. Import

Description

This operator can read SAS (Statistical Analysis System) files. Please study the attached Example Process for understanding the use of this operator. Please note that when an SAS file is read, the roles of all the attributes are set to regular. Numeric columns use the rreal data type, nominal columns use the ppolynomial data type in RapidMiner.

Output Ports

output (out) This port delivers the SAS file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

file (string) The path of the SAS file is specified here. It can be selected using the choose a file button.

Tutorial Processes

Use of the SAS operator

an SAS file is loaded using the Open File operator and then read via the Read SAS operator.
4.1. Data

Read Access

This operator reads an ExampleSet from a Microsoft Access database.

Description

The Read Access operator is used for reading an ExampleSet from the specified Microsoft Access database (.mdb extension). You need to have at least basic understanding of databases, database connections and queries in order to use this operator properly. Go through the parameters and Example Process to understand the flow of this operator.

Output Ports

output (out) This port delivers the result of the query on database in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

username (string) This parameter is used to specify the username of the database (if any).

password (string) This parameter is used to specify the password of the database.
4. Import

(if any).

**define query (selection)** Query is a statement that is used to select required data from the database. This parameter specifies whether the database query should be defined directly, through a file or implicitly by a given table name. The SQL query can be auto generated giving a table name, passed to Rapid-Miner via a parameter or, in case of long SQL statements, in a separate file. The desired behavior can be chosen using the **define query** parameter. Please note that column names are often case sensitive and might need quoting.

**query (string)** This parameter is only available when the **define query** parameter is set to 'query'. This parameter is used to define the SQL query to select desired data from the specified database.

**query file (filename)** This parameter is only available when the **define query** parameter is set to 'query file'. This parameter is used to select a file that contains the SQL query to select desired data from the specified database. Long queries are usually stored in files. Storing queries in files can also enhance reusability.

**table name (string)** This parameter is only available when the **define query** parameter is set to 'table name'. This parameter is used to select the required table from the specified database.

**database file (filename)** This parameter specifies the path of the Access database i.e. the mdb file.

**Tutorial Processes**

**Writing and then reading data from an Access database**

Before the execution of this process a database was manually created in Microsoft Access. The name of database is 'golf_db.mdb' and it is placed in the D drive of the system. An empty table named 'golf' is also created in the database.

The 'Golf' data set is loaded using the Retrieve operator. The Write Access operator is used for writing this ExampleSet into the golf table of the 'golf_db.mdb' database. The **database file** parameter is provided with the path of the database
4.1. Data

file 'D:\golf.db.mdb' and the name of the desired table is specified in the table name parameter (i.e. it is set to 'golf'). A breakpoint is inserted here. No results are visible in RapidMiner at this stage but you can see that at this point of the execution the previously empty golf table has been filled with the examples of the 'Golf' data set.

Now the Read Access operator is used for reading the golf table from the 'golf.db.mdb' database. The database file parameter is provided with the path of the database file 'D:\golf.db.mdb'. The define query parameter is set to 'table name'. The table name parameter is set to 'golf' which is the name of the required table. Continue the process, you will see the entire golf table in the Results Workspace. The define query parameter is set to 'table name' if you want to read an entire table from the database. You can also read a selected portion of the database by using queries. Set the define query parameter to 'query' and specify a query in the query parameter.
4. Import

Read AML

This operator reads an ExampleSet from one or more file. This operator can be configured to read almost all line-based file formats.

Description

Files with '.aml' extension are attribute description files. They store the meta data for an ExampleSet in standard XML format. This operator reads an ExampleSet from one or more files. You can use the wizard of this operator or the Attribute Editor tool in order to create '.aml' files that contain meta data for your datasets. The Attribute Editor tool can be accessed by clicking the button right next to 'attributes' parameter textbox. This operator supports the reading of data from multiple source files. Each attribute (including special attributes like labels, weights etc) might be read from another file. Please note that only the minimum number of lines of all files will be read, i.e. if one of the data source files has fewer lines than the others, only this number of examples will be read from all files. Go through the parameters and Example Process to get a better understanding of this operator.

Output Ports

output (out) this port delivers the required file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

start data loading wizard (menu) This option allows you to configure this operator by means of a wizard. This user-friendly wizard makes use of this operator easy. This wizard guides you through the process of data loading and meta data
4.1. Data

attributes The path and filename for the XML attribute description file is specified here. An attribute description file (extension: .aml) is required to retrieve meta data of the ExampleSet. This file is a simple XML document defining the properties of the attributes (like their name and range) and their source files. The data may be spread over several files. This file also contains the names of the files to read the data from. Therefore, the actual data files do not have to be specified as a parameter of this operator. For further information on file formats please study the First steps/File formats section.

The Attribute Editor is a very useful tool placed next to the attributes parameter textbox. This tool allows you to open existing '.aml' files, modify meta data, save current meta data description in an '.aml' file, append multiple data files together, save content in a data file, select the range of attributes and examples.

sample ratio (real) This parameter is used only when the sample size parameter is set to -1. This parameter specifies the fraction of the data set which should be read. It can have value from 0 to 1. Value=1 would read complete data set, value=0.5 would read half of data set and so on.

sample size (integer) This parameter specifies the exact number of examples which should be read. If it is set to -1, then sample ratio parameter is used to select fraction of examples from ExampleSet. If value of sample size parameter is not -1, sample ratio parameter will be ignored and number of examples specified in sample size parameter would be read.

permute (boolean) Indicates if the loaded data should be permuted. If this parameter is set to true, the sequence of examples is changed in the resulting ExampleSet.

decimal point character (string) This character is used as the decimal character.

column separators (string) Column separators for files can be specified here in a regular expression format. A good understanding of regular expression can be developed from studying the Select Attributes operator's description and Example Processes. The default split parameter \";\s*\|\s*\|\s+\" should work for most file formats. This regular expression describes the following column separators:

- the character \", followed by a whitespace of arbitrary length (also no white
4. Import

- the character “;” followed by a whitespace of arbitrary length (also no white space)
- a whitespace of arbitrary length (minimum length: 1)

A logical XOR is defined by “|”. Other useful separators might be “\t” for tabs, for a single whitespace, and “\s” for any number of whitespaces.

use comment characters (boolean) Skip comments option is used to ignore comments in a file. This is useful only if the file has comments. If this option is set to true, a comment character should be defined using comment chars parameter.

comment chars (string) Lines beginning with these characters are ignored. Remember that comment character itself is also ignored.

use quotes (boolean) This option indicates if quotes should be regarded. Quotes can be used to store special characters like column separators. For example if (,) is set as column separator and (“) is set as quotes character. (a,b,c,d) will be translated as 4 values for 4 columns. On the other hand (ä,b,c,d) will be translated as a single column value i.e. a,b,c,d. If this option is set to true, quotes character parameter and escape character parameter for quotes can be defined.

quote character (char) This option defines the quotes character.

quoting escape character (char) This is the character that is used to escape quotes. For example if (“) is used as quotes character and (\) is used as escape character. (“yes”) will be translated as (yes) and (\ “yes\ “) will be translated as (“yes”).

trim lines (boolean) This option indicates if lines should be trimmed (empty spaces are removed at the beginning and the end) before the column split is performed. This option might be problematic if TABs are used as separators.

skip error lines (boolean) If this parameter is set to true, lines which cannot be read will be skipped instead of letting this operator fail its execution.

data management (selection) This is an expert parameter. A long list is provided; users can select any option from this list.

encoding (selection) This is an expert parameter. A long list of encoding is provided; users can select any one of them.
4.1. Data

**use local random seed** *(boolean)* Indicates if a *local random seed* should be used for randomizing examples of an ExampleSet. Using the same value of *local random seed* will produce the same ExampleSet. Changing the value of this parameter changes the way examples are randomized, thus the ExampleSet will have a different set of examples.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

### Tutorial Processes

#### Writing a file

This process writes an 'aml' file. This 'aml' file will be read in the second Example Process using the Read AML operator. This process is shown here just to show you an attribute description file and how it is used in the Read AML operator. The Retrieve operator is used to retrieve the 'golf' data set from the Repository. Then the Write AML operator is used to write the 'golf' data set into a file. Create two text files before moving further. In this Example Process a file named 'golf\_data' and another file named 'golf\_att' was created on the desktop before execution of this process. The 'golf\_data' file will be used to store data of the 'golf' data set and the 'golf\_att' file will store information regarding meta data of the 'golf' data set. In other words, 'golf\_data' will become our data file and 'golf\_att' will be the attribute description file. Give the path and name of the data file in *example set file* parameter and give the path and name of the attribute description file in *attribute description file* parameter. Now run the process. Now have a look at both files. You can see that the 'golf\_data' file has examples of the 'golf' data set whereas the 'golf\_att' file has meta data of the 'golf' data set.
Reading a file

In this Example Process, the 'golf_data' and 'golf_att' files created in the first Example Process will be read using the Read AML operator. The easiest way to configure this operator is to use the start data loading wizard. Path and file name of the 'golf_data' file are provided as path of the data file in the start of wizard. Path and filename of 'golf_att' file are provided as path of the attribute description file at the last step of the wizard. The sample ratio parameter was set to '0.5', thus half of the data set would be read. The permute parameter is also checked, thus examples of the resultant data set would be shuffled. All other parameters were used with default values. Run the process and you will see that half of the 'golf_data' file is read and the examples are shown out of sequence.

Read ARFF

This operator is used for reading an ARFF file.
4.1. Data

Description

This operator can read ARFF (Attribute-Relation File Format) files known from the machine learning library Weka. An ARFF file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software. Please study the attached Example Process for understanding the basics and structure of the ARFF file format. Please note that when an ARFF file is written, the roles of the attributes are not stored. Similarly when an ARFF file is read, the roles of all the attributes are set to regular.

Output Ports

output (out) This port delivers the ARFF file in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

data file (filename) The path of the ARFF file is specified here. It can be selected using the choose a file button.

encoding (selection) This is an expert parameter. A long list of encoding is provided; users can select any of them.

read not matching values as missings (boolean) This is an expert parameter. If this parameter is set to true, values that do not match with the expected value type are considered as missing values and are replaced by '?' . For example if 'abc' is written in an integer column, it will be treated as a missing value. Question mark (?) in ARFF file is also read as missing value.

decimal character (char) This character is used as the decimal character.

grouped digits (boolean) This parameter decides whether grouped digits should be parsed or not. If this parameter is set to true, the grouping character parameter
4. Import

should be specified.

**grouping character** \( (\text{char}) \) This parameter is available only when the **grouped digits** parameter is set to true. This character is used as the grouping character. If it is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the ARFF file and it is set as grouping character, then "2214" will be stored.

**Tutorial Processes**

**The basics of the ARFF**

The 'Iris' data set is loaded using the Retrieve operator. The Write ARFF operator is applied on it to write the 'Iris' data set into an ARFF file. The *example set file* parameter is set to 'D:\Iris'. Thus an ARFF file is created in the 'D' drive of your computer with the name 'Iris'. Open this file to see the structure of an ARFF file.

ARFF files have two distinct sections. The first section is the Header information, which is followed by the Data information. The Header of the ARFF file contains the name of the Relation and a list of the attributes. The name of the Relation is specified after the @RELATION statement. The Relation is ignored by RapidMiner. Each attribute definition starts with the @ATTRIBUTE statement followed by the attribute name and its type. The resultant ARFF file of this Example Process starts with the Header. The name of the relation is 'RapidMinerData'. After the name of the Relation, six attributes are defined.

Attribute declarations take the form of an ordered sequence of @ATTRIBUTE statements. Each attribute in the data set has its own @ATTRIBUTE statement which uniquely defines the name of that attribute and its data type. The order of declaration of the attributes indicates the column position in the data section of the file. For example, in the resultant ARFF file of this Example Process the 'label' attribute is declared at the end of all other attribute declarations. Therefore values of the 'label' attribute are in the last column of the Data section.

208
The possible attribute types in ARFF are:

- numeric
- integer
- real
- \{nominalValue1,nominalValue2,...\} for nominal attributes
- string for nominal attributes without distinct nominal values (it is however recommended to use the nominal definition above as often as possible)
- date [date-format] (currently not supported by RapidMiner)

You can see in the resultant ARFF file of this Example Process that the attributes 'a1', 'a2', 'a3' and 'a4' are of real type. The attributes 'id' and 'label' are of nominal type. The distinct nominal values are also specified with these nominal attributes.

The ARFF Data section of the file contains the data declaration line @DATA followed by the actual example data lines. Each example is represented on a single line, with carriage returns denoting the end of the example. Attribute values for each example are delimited by commas. They must appear in the order that they were declared in the Header section (i.e. the data corresponding to the n-th @ATTRIBUTE declaration is always the n-th field of the example line). Missing values are represented by a single question mark (?).

A percent sign (%) introduces a comment and will be ignored during reading. Attribute names or example values containing spaces must be quoted with single quotes ('). Please note that in RapidMiner the sparse ARFF format is currently only supported for numerical attributes. Please use one of the other options for sparse data files provided by RapidMiner if you also need sparse data files for nominal attributes.
4. Import

Reading an ARFF file using the Read ARFF operator

The ARFF file that was written in the first Example Process using the Write ARFF operator is retrieved in this Example Process using the Read ARFF operator. The data file parameter is set to 'D:\Iris'. Please make sure that you specify the correct path. All other parameters are used with default values. Run the process. You will see that the results are very similar to the original Iris data set of RapidMiner repository. Please note that the role of all the attributes is regular in the results of the Read ARFF operator. Even the roles of 'id' and 'label' attributes are set to regular. This is so because the ARFF files do not store information about the roles of the attributes.

Read Database

This operator reads an ExampleSet from a SQL database.
4.1. Data

Description

The Read Database operator is used for reading an ExampleSet from the specified SQL database. You need to have at least basic understanding of databases, database connections and queries in order to use this operator properly. Go through the parameters and Example Process to understand the flow of this operator.

When this operator is executed, the table delivered by the query will be copied into the memory of your computer. This will give all subsequent operators a fast access on the data. Even learning schemes like the Support Vector Machine with their high number of random accesses will run fast. If the table is too big for your main memory, you may use the Stream Database operator. It will hold only a part of the table in memory for the cost of several magnitudes slower access if the desired example isn't cached.

The java ResultSetMetaData interface does not provide information about the possible values of nominal attributes. The internal indices the nominal values are mapped to, will depend on the ordering they appear in the table. This may cause problems only when processes are split up into a training process and a testing process. This is not a problem for learning schemes which are capable of handling nominal attributes. If a learning scheme like the SVM is used with nominal data, RapidMiner pretends that nominal attributes are numerical and uses indices for the nominal values as their numerical value. The SVM may perform well if there are only two possible values. If a test set is read in another process, the nominal values may be assigned different indices, and hence the SVM trained is useless. This is not a problem for the label attributes, since the classes can be specified using the classes parameter and hence all learning schemes intended to use with nominal data are safe to use. You might avoid this problem if you first combine both ExampleSets using the Append operator and then split it again using two Filter Examples operators.
Differentiation

Execute SQL The Read Database operator is used for loading data from a database into RapidMiner. The Execute SQL operator cannot be used for loading data from databases. It can be used for executing SQL statements like CREATE or ADD etc on the database. See page 141 for details.

Stream Database In contrast to the Read Database operator, which loads the data into the main memory, the Stream Database operator keeps the data in the database and performs the data reading in batches. This allows RapidMiner to access data sets of arbitrary sizes without any size restrictions. See page 215 for details.

Output Ports

output (out) This port delivers the result of the query on database in tabular form along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

define connection (selection) This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

collection (string) This parameter is only available when the define connection parameter is set to predefined. This parameter is used to connect to the database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down box. You can add a new connection or modify previous connections using the button next to the drop down box. You may also accomplish this by clicking on the Manage Database Connections... from the Tools menu in the main window. A new window appears. This window asks for several details e.g. Host, Port, Database system, schema, username and password. The Test button in this new window will allow you to check whether
the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the connection parameter. You need to have basic understanding of databases for configuring a connection.

**database system** *(selection)* This parameter is only available when the define connection parameter is set to *url*. This parameter is used to select the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

**database url** *(string)* This parameter is only available when the define connection parameter is set to *url*. This parameter is used to define the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

**username** *(string)* This parameter is only available when the define connection parameter is set to *url*. This parameter is used to specify the username of the database.

**password** *(string)* This parameter is only available when the define connection parameter is set to *url*. This parameter is used to specify the password of the database.

**jndi name** *(string)* This parameter is only available when the define connection parameter is set to *jndi*. This parameter is used to give the JNDI a name for a data source.

**define query** *(selection)* Query is a statement that is used to select required data from the database. This parameter specifies whether the database query should be defined directly, through a file or implicitly by a given table name. The SQL query can be auto generated giving a table name, passed to Rapid-Miner via a parameter or, in case of long SQL statements, in a separate file. The desired behavior can be chosen using the define query parameter. Please note that column names are often case sensitive and might need quoting.

**query** *(string)* This parameter is only available when the define query parameter is set to *query*. This parameter is used to define the SQL query to select desired data from the specified database.

**query file** *(filename)* This parameter is only available when the define query parameter is set to *query file*. This parameter is used to select a file that contains the SQL query to select desired data from the specified database. Long queries
4. Import

are usually stored in files. Storing queries in files can also enhance reusability.

**table name** *(string)* This parameter is only available when the define query parameter is set to *table name*. This parameter is used to select the required table from the specified database.

**prepare statement** *(boolean)* If checked, the statement is prepared, and '?' can be filled in using the parameters parameter.

**parameters** *(enumeration)* Parameters to insert into '?' placeholders when statement is prepared.

Related Documents

Execute SQL (141)
Stream Database (215)

Tutorial Processes

Reading ExampleSet from a mySQL database

The Read Database operator is used to read a mySQL database. The define connection parameter is set to predefined. The define connection parameter was configured using the button next to the drop down box. The name of the connection was set to 'mySQLconn'. The following values were set in the connection parameter's wizard. The Database system was set to 'mySQL'. The Host was set to 'localhost'. The Port was set to '3306'. The Database scheme was set to 'golf'; this is the name of the database. The User was set to 'root'. No password was provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The define query parameter was set to 'table name'. The table name parameter was set to 'golf_table' which is the name of the required table in the 'golf' database. Run the process, you will see the entire 'golf_table' in the Results.
4.1. Data

Workspace. The define query parameter is set to 'table name' if you want to read an entire table from the database. You can also read a selected portion of the database by using queries. Set the define query parameter to 'query' and specify a query in the query parameter. One sample query is already defined in this example. This query reads only those examples from 'golf_table' where the 'Outlook' attribute has the value 'sunny'.

Stream Database

This operator reads an ExampleSet from an SQL database by incrementally caching it (recommended).

Description

The Stream Database operator is used for reading an ExampleSet from the specified SQL database. You need to have at least basic understanding of databases and database connections in order to use this operator properly. Please go through the parameter description and the attached Example Process to understand the working of this operator.

This operator reads an ExampleSet from an SQL database. The data is loaded from a single table which is defined by the table name parameter. Please note that table and column names are often case sensitive. The most convenient way of defining the necessary parameters is through the configuration wizard. The most important parameters (database URL and username) will be automatically determined by this wizard. You can define the special attributes like labels, ids
4. Import

and weights through corresponding parameters.

In contrast to the Database operator, which loads the data into the main memory, this operator keeps the data in the database and performs the data reading in batches. This allows RapidMiner to access data sets of arbitrary sizes without any size restrictions.

Please note the following important restrictions and notes:

- Only manifested tables (no views) are allowed as the base for this data caching operator.

- If primary key and index are not present, a new column named RM_INDEX is created and it is automatically used as the primary key.

- If a primary key is already present in the specified table, a new table named RM_MAPPED_INDEX is created which maps a new index column RM_INDEX to the original primary key.

- The users can provide the primary key column RM_INDEX themselves. This column should be an integer valued index attribute, counting should start from 1, without any gaps or missing values for all rows.

Besides the new index column or the creation of mapping table, no writing actions are performed in the database. Moreover, data sets built on top of a cached database table do not support writing actions at all. Users have to materialize the data, change it, and write it back into a new table of the database (e.g. with the Write Database operator.

Differentiation

Execute SQL The Stream Database operator is used for reading data from a database. The Execute SQL operator cannot be used for reading data from databases. It can be used for executing SQL statements like CREATE or ADD etc on the database. See page 141 for details.

Read Database In contrast to the Read Database operator, which loads the
data into the main memory, the Stream Database operator keeps the data in the database and performs the data reading in batches. This allows RapidMiner to access data sets of arbitrary sizes without any size restrictions. See page 210 for details.

Output Ports

**output (out)** This port delivers the database table in form of an ExampleSet along with the meta data. This output is similar to the output of the Retrieve operator.

Parameters

**define connection (selection)** This parameter indicates how the database connection should be specified. The following options are available: predefined, url and jndi.

**connection (string)** This parameter is only available when the **define connection** parameter is set to 'predefined'. This parameter is used for connecting to the database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down box. You can add a new connection or modify previous connections using the button next to the drop down box. You may also accomplish this by clicking on the **Manage Database Connections...** from the Tools menu in the main window. A new window appears. This window asks for several details e.g. Host, Port, Database system, schema, username and password. The **Test** button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the **connection** parameter. You need to have basic understanding of databases for configuring a connection.

**database system (selection)** This parameter is only available when the **define connection** parameter is set to 'url'. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, Post-
4. Import

greSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

database url (string) This parameter is only available when the define connection parameter is set to 'url'. This parameter is used for defining the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

username (string) This parameter is only available when the define connection parameter is set to 'url'. This parameter is used to specify the username of the database.

password (string) This parameter is only available when the define connection parameter is set to 'url'. This parameter is used for specifying the password of the database.

jndi name (string) This parameter is only available when the define connection parameter is set to 'jndi'. This parameter is used for specifying the JNDI name for a data source.

table name (string) This parameter is used for selecting the required table from the specified database.

recreate index (boolean) This parameter indicates if recreation of the index or index mapping table should be forced.

label attribute (string) The name (case sensitive) of the label attribute is specified through this parameter.

id attribute (string) The name (case sensitive) of the id attribute is specified through this parameter.

weight attribute (string) The name (case sensitive) of the weight attribute is specified through this parameter.

Related Documents

Execute SQL (141)
Read Database (210)
4.1. Data

Tutorial Processes

Reading an ExampleSet from a mySQL database

The Stream Database operator is used in this Example Process for reading a mySQL database. The define connection parameter is set to predefined. The define connection parameter was configured using the button next to the drop down box. The name of the connection was set to 'mySQLconn'. The following values were set in the connection parameter's wizard. The Database system was set to 'mySQL'. The Host was set to 'localhost'. The Port was set to '3306'. The Database scheme was set to 'golf'; this is the name of the database. The User was set to 'root'. No password was provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The table name parameter is set to 'golf_table' which is the name of the required table in the 'golf' database. The label attribute parameter is set to 'Play'. Run the process, you will see the entire 'golf_table' in the Results Workspace.

Read SPSS

This operator is used for reading SPSS files.
4. Import

Description

The Read SPSS operator can read the data files created by SPSS (Statistical Package for the Social Sciences), an application used for statistical analysis. SPSS files are saved in a proprietary binary format and contain a dataset as well as a dictionary that describes the dataset. These files save data by 'cases' (rows) and 'variables' (columns).

These files have a '.SAV' file extension. SAV files are often used for storing datasets extracted from databases and Microsoft Excel spreadsheets. SPSS datasets can be manipulated in a variety of ways, but they are most commonly used to perform statistical analysis tests such as regression analysis, analysis of variance, and factor analysis.

Input Ports

file (fil) This optional port expects a file object.

Output Ports

output (out) Data from the SPSS file is delivered through this port mostly in form of an ExampleSet.

Parameters

filename (filename) This parameter specifies the path of the SPSS file. It can be selected using the choose a file button.

datamanagement (selection) This parameter determines how the data is represented internally. This is an expert parameter. There are different options, users can choose any of them.
attribute naming mode (selection) This parameter determines which SPSS variable properties should be used for naming the attributes.

use value labels (boolean) This parameter specifies if the SPSS value labels should be used as values.

recode user missings (boolean) This parameter specifies if the SPSS user missings should be recoded to missing values.

sample ratio (real) This parameter specifies the fraction of the data set which should be read. If it is set to 1, the complete data set is read. If it is set to -1 then the sample size parameter is used for determining the size of the data to read.

sample size (integer) This parameter specifies the exact number of samples which should be read. If it is set to -1, then the sample ratio parameter is used for determining the size of data to read. If both are set to -1 then the complete data set is read.

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

Tutorial Processes

Reading an SPSS file

You need to have an SPSS file for this process. In this process, the name of the SPSS file is airline_passengers.sav and it is placed in the D drive of the computer. The file is read using the Read SPSS operator. All parameters are used with default values. After execution of the process you can see the resultant ExampleSet in the Results Workspace.
Read Model

This operator reads a model from a file. This operator is often used for reading models that were written using the Write Model operator.

Description

The Read Model operator reads a model from the file specified by the model file parameter. Since models are often written into files and then read for applying them in other processes or applications, this operator can read models written in different writing modes. The Write Model operator can write models in three modes: XML, XML Zipped and Binary. Once a model is read, it can be applied on any ExampleSet using the Apply Model operator.

Output Ports

output (out) The model is read from the specified file and the resultant model is delivered through this port.
Parameters

`model file (filename)` This parameter is used for specifying the model file.

Tutorial Processes

Writing the K-NN model into an XML file

This Example Process shows how the Write Model operator can be used for writing a model. The model file written in this Example Process will be read in the next Example Process. The 'Golf' data set is loaded using the Retrieve operator. The K-NN operator is applied on it. The resultant model is provided as input to the Write Model operator. The `model file` parameter is set to 'D:\model' and the `output type` parameter is set to XML. Thus an XML file named 'model' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written model file and make changes in it. The model file can be used in RapidMiner by the Read Model operator.

Reading the K-NN model from a file
4. Import

This Example Process shows how the Read Model operator can be used to read a model from a file. This Example Process reads the model file written by the previous Example Process. The 'Golf-Testset' data set is loaded using the Retrieve operator. The K-NN model written in the previous Example Process is loaded using the Read Model operator. The model file parameter is set to 'D:\model' to select the model file written in the previous process. The resultant model is provided as input to the Apply Model operator which applies it on the 'Golf-Testset' data set and delivers the labeled data and model which can be seen in the Results Workspace.

Read Weights

This operator reads the weights of all attributes of an ExampleSet from the specified file. This operator is often used for reading weights that were written using the Write Weights operator.
4.3. Attributes

Description

The Read Weights operator reads the weights for all attributes of an ExampleSet from the specified file which is specified through the \textit{attribute weights file} parameter. Each line of the file must hold the name and the weight of one attribute. This operator delivers these weights as a new \textit{AttributeWeights} IOObject. This object can be used for performing various weights related tasks e.g. in operators like Scale by Weights, Select by Weights etc.

Output Ports

\texttt{output (out)} The attribute weights are read from the specified file and the resultant \textit{AttributeWeights} IOObject is delivered through this port.

Parameters

\texttt{attribute weights file (filename)} This parameter is used for specifying the attribute weights file.

Tutorial Processes

Writing and then Reading attribute weights from a file

This Example Process shows how the Write Weights operator can be used for writing attribute weights and how the Read Weights operator can be used for reading attribute weights from this file. The 'Sonar' data set is loaded using the Retrieve operator. The Weight by Correlation operator is applied on it. A \textit{breakpoint} is inserted here so that you can have a look at the attribute weights. The resultant weights are provided as input to the Write Weights operator. The \textit{at-
4. Import

tribute weights file parameter is set to 'D:\sonar weights' thus a file named 'sonar weights' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written weights file and make changes in it (if required). The attribute weights file is then read by using the Read Weights operator. The attribute weights file parameter is set to 'D:\sonar weights' to read the same file that was written by the Write Weights operator. The resultant weights are connected to the result port of the process. The attribute weights can be seen in the Results Workspace.

![Diagram of the process](image-url)
5  Export

Write

This is the most generic export operator. It can write almost any kind of IO Object into a file.

Description

The Write operator can write almost any kind of IO Object into the specified file. The path of the file is specified through the object file parameter. The output type parameter specifies the type of the output. The Read operator can be used for reading almost any kind of IO Object from a file.

Input Ports

object (obj) This input port expects an IO Object. This IO Object will be written into the specified file.
5. Export

Output Ports

**object** *(obj)* The IO Object that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same performance vector in further operators of the process.

Parameters

**object file** *(filename)* The path of the IO Object file is specified here. It can be selected using the *choose a file* button.

**output type** *(selection)* This parameter specifies the type of the output. The following output types are supported: XML, XML Zipped, Binary

**continue on error** *(boolean)* This parameter specifies if the writing process should continue if there is an error during the writing process.

Tutorial Processes

Generic Write and Read operators

This Example Process demonstrates the use of the most generic writing and reading operators i.e. the Write and Read operators respectively. This Example Process shows how these operators can be used to write and read an ExampleSet but these operators are very generic and can be used for writing and reading almost any IO Object available in RapidMiner.

The 'Golf' data set is loaded using the Retrieve operator. This ExampleSet is provided as input to the Write operator. The **object file** parameter is set to 'D:\golf' thus a file named 'golf' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written file and make changes in it (if required). The Read operator is applied next. The **object file** parameter is set
to 'D:\golf' to read the file that was just written using the Write operator. The \textit{io object} parameter is set to 'ExampleSet' because this Read operator is being used for reading an ExampleSet. The resultant ExampleSet can be seen in the Results Workspace.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{process_diagram.png}
\caption{Process diagram showing the flow of data through the Write, AML, and Read operators.}
\end{figure}

\section*{Write AML}

This operator writes an ExampleSet into a file in \textit{dense} or \textit{sparse} format. The file written by this operator can be read by operators like the Read AML and Read Sparse operators.

\section*{Description}

The Write AML operator writes the values of all examples in an ExampleSet into a file. Both dense and sparse formats can be generated depending on the setting of the \textit{format} parameter. The files written in dense format can be read using the Read AML operator. The files written in sparse format can be read...
5. Export

using the Read Sparse operator. A brief description of dense and sparse formats is given here. For better understanding it is recommended that you study the First steps/File formats section.

dense format

Each line of the generated data file is of the form:

*regular attributes* < *special attributes>*

For example, each line could have the form:

*value1 value2 ... valueN* < *id* > < *label* > < *prediction* > ... < *confidences*>

The values in the parenthesis are optional and they are only written if they are available. The confidences are only given for nominal predictions. Other special attributes can also be written e.g. the example weight or the cluster number etc.

sparse format

Only non-zero values are written into the file, prefixed by a column index. If almost all the values in a data file are zero or have a default nominal value, sparse format may prove to be a suitable option.

Files with '.aml' extension are attribute description files. They store the meta data for an ExampleSet in standard XML format. Go through the parameters and Example Process to get a better understanding of this operator.
Input Ports

input \((\text{inp})\) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. This ExampleSet would be written into the specified file by this operator.

Output Ports

through \((\text{thr})\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

data file \((\text{filename})\) The ExampleSet is written into the file specified through this parameter. This is the only mandatory parameter of this operator. All other parameters are expert parameters.

attribute description file \((\text{filename})\) The Attribute descriptions are written into the file specified through this parameter. This file holds information about the meta data of the ExampleSet.

format \((\text{selection})\) This parameter specifies the format which should be used for writing the file. Basics of these formats have been discussed in the description of this operator.

zipped \((\text{boolean})\) This parameter indicates if the data file content should be zipped or not.

overwrite mode \((\text{selection})\) This parameter indicates if an existing file should be overwritten or data should be appended at the end of the file.

encoding \((\text{selection})\) This is an expert parameter. There are different options, users can choose any of them.
Tutorial Processes

Writing a file using the Write AML operator

This process writes an 'aml' file. This 'aml' file can be read in other processes using the Read AML operator. The Retrieve operator is used to retrieve the 'Golf' data set from the Repository. Then the Write AML operator is used to write the 'Golf' data set into a file. Give the path and name of the data file in the example set file parameter and give the path and name of the attribute description file in the attribute description file parameter. In this Example Process 'D:\golf_data.txt' and 'D:\golf_att.txt' are specified in the example set file and attribute description file parameters respectively. The 'golf_data' file will be used to store the data of the 'Golf' data set and the 'golf_att' file will store information regarding the meta data of the 'Golf' data set. In other words, 'golf_data' will become our data file and 'golf_att' will be the attribute description file. Now run the process and have a look at both files. You can see that the 'golf_data' file has examples of the 'Golf' data set whereas the 'golf_att' file has the meta data of the 'Golf' data set.

Write Arff

This operator is used for writing an ARFF file.
5.1. Data

Description

This operator can write data in form of ARFF (Attribute-Relation File Format) files known from the machine learning library Weka. An ARFF file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software. Please study the attached Example Processes for understanding the basics and structure of the ARFF file format. Please note that when an ARFF file is written, the roles of the attributes are not stored. Similarly when an ARFF file is read, the roles of all the attributes are set to regular.

Input Ports

input (inp) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

Output Ports

through (thr) The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.
file (fil) This port buffers the file object for passing it to the reader operators

Parameters

example set file (filename) The path of the ARFF file is specified here. It can be selected using the choose a file button.
encoding (selection) This is an expert parameter. A long list of encoding is provided; users can select any of them.
Tutorial Processes

The basics of ARFF

The 'Iris' data set is loaded using the Retrieve operator. The Write ARFF operator is applied on it to write the 'Iris' data set into an ARFF file. The example set file parameter is set to 'D:\Iris.txt'. Thus an ARFF file is created in the 'D' drive of your computer with the name 'Iris'. Open this file to see the structure of an ARFF file.

ARFF files have two distinct sections. The first section is the Header information, which is followed by the Data information. The Header of the ARFF file contains the name of the Relation and a list of the attributes. The name of the Relation is specified after the @RELATION statement. The Relation is ignored by RapidMiner. Each attribute definition starts with the @ATTRIBUTE statement followed by the attribute name and its type. The resultant ARFF file of this Example Process starts with the Header. The name of the relation is 'RapidMinerData'. After the name of the Relation, six attributes are defined.

Attribute declarations take the form of an ordered sequence of @ATTRIBUTE statements. Each attribute in the data set has its own @ATTRIBUTE statement which uniquely defines the name of that attribute and its data type. The order of declaration of the attributes indicates the column position in the data section of the file. For example, in the resultant ARFF file of this Example Process the 'label' attribute is declared at the end of all other attribute declarations. Therefore values of the 'label' attribute are in the last column of the Data section.

The possible attribute types in ARFF are:

- numeric
- integer
- real
5.1. Data

- \{\text{nominalValue1,nominalValue2,\ldots}\} for nominal attributes
- string for nominal attributes without distinct nominal values (it is however recommended to use the nominal definition above as often as possible)
- date [date-format] (currently not supported by RapidMiner)

You can see in the resultant ARFF file of this Example Process that the attributes 'a1', 'a2', 'a3' and 'a4' are of real type. The attributes 'id' and 'label' are of nominal type. The distinct nominal values are also specified with these nominal attributes.

The ARFF Data section of the file contains the data declaration line @DATA followed by the actual example data lines. Each example is represented on a single line, with carriage returns denoting the end of the example. Attribute values for each example are delimited by commas. They must appear in the order that they were declared in the Header section (i.e. the data corresponding to the n-th @ATTRIBUTE declaration is always the n-th field of the example line). Missing values are represented by a single question mark (?).

A percent sign (%) introduces a comment and will be ignored during reading. Attribute names or example values containing spaces must be quoted with single quotes ('). Please note that in RapidMiner the sparse ARFF format is currently only supported for numerical attributes. Please use one of the other options for sparse data files provided by RapidMiner if you also need sparse data files for nominal attributes.

Reading an ARFF file using the Read ARFF operator
5. Export

The ARFF file that was written in the first Example Process using the Write ARFF operator is retrieved in this Example Process using the Read ARFF operator. The *data file* parameter is set to 'D:\Iris.txt'. Please make sure that you specify the correct path. All other parameters are used with default values. Run the process. You will see that the results are very similar to the original Iris data set of RapidMiner repository. Please note that the role of all the attributes is regular in the results of the Read ARFF operator. Even the roles of 'id' and 'label' attributes are set to regular. This is so because the ARFF files do not store information about the roles of the attributes.

![Main Process](image)

Write Database

This operator writes an ExampleSet to an SQL database.

Description

The Write Database operator is used for writing an ExampleSet to the specified SQL database. You need to have at least basic understanding of databases and database connections in order to use this operator properly. Go through the parameters and the attached Example Process to understand the flow of this operator.

The user can specify the database connection and a table name. Please note that the table will be created during writing if it does not exist. The most convenient way of defining the necessary parameters is the *Manage Database Connections* wizard. The most important parameters (database URL and user name) will be...
5.1. Data

automatically determined by this wizard. At the end, you only have to define the
table name. This operator only supports the writing of the complete ExampleSet
consisting of all regular and special attributes and all examples. If this is not
desired, perform some preprocessing operators like the Select Attributes or Filter
Examples operators before applying the Write Database operator. Data from
database tables can be read in RapidMiner by using the Read Database operator.

**Input Ports**

**input (inp)** This input port expects an ExampleSet. It is output of the Retrieve
operator in the attached Example Process.

**Output Ports**

**through (thr)** The ExampleSet that was provided at the input port is delivered
through this output port without any modifications. This is usually used to reuse
the same ExampleSet in further operators of the process.

**Parameters**

**define connection (selection)** This parameter indicates how the database con-
nnection should be specified. It gives you three options: predefined, url and jndi.
**connection (string)** This parameter is only available when the **define connection**
parameter is set to **predefined**. This parameter is used for connecting to
the database using a predefined connection. You can have many predefined
connections. You can choose one of them using the drop down box. You can add
a new connection or modify previous connections using the button next to the
drop down box. You may also accomplish this by clicking on **Manage Database
Connections...** from the **Tools** menu in the main window. A new window ap-
ppears. This window asks for several details e.g. **Host, Port, Database system,**
schema, username and password. The Test button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the connection parameter. You need to have basic understanding of databases for configuring a connection.

database system (selection) This parameter is only available when the define connection parameter is set to url. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQldb, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

database url (string) This parameter is only available when the define connection parameter is set to url. This parameter is used for defining the URL connection string for the database, e.g. 'jdbc:mysql://foo.bar:portnr/database'.

username (string) This parameter is only available when the define connection parameter is set to url. This parameter is used for specifying the username of the database.

password (string) This parameter is only available when the define connection parameter is set to url. This parameter is used for specifying the password of the database.

jndi name (string) This parameter is only available when the define connection parameter is set to jndi. This parameter is used for giving the JNDI a name for a data source.

table name This parameter is used for selecting the required table from the specified database. Please note that you can also write a table name here, if the table does not exist it will be created during writing.

overwrite mode (selection) This parameter indicates if an existing table should be overwritten or data should be appended to the existing data.

set default varchar length (boolean) This parameter allows you to set varchar columns to default length.

default varchar length (integer) This parameter is only available when the set default varchar length parameter is set to true. This parameter specifies the default length of varchar columns.

add generated primary keys (boolean) This parameter indicates whether a new attribute holding the auto generated primary keys should be added to the
5.1. Data

table in the database.

db key attribute name (string) This parameter is only available when the add generated primary keys parameter is set to true. This parameter specifies the name of the attribute for the auto generated primary keys.

batch size (integer) This parameter specifies the number of examples which are written at once with one single query to the database. Larger values can greatly improve the speed. However, too large values can drastically decrease the performance. Moreover, some databases have restrictions on the maximum number of values written at once.

Tutorial Processes

Writing an ExampleSet to a mySQL database

The 'Golf' data set is loaded using the Retrieve operator. The Write Database operator is used for writing this data set to a mySQL database. The define connection parameter is set to predefined and it is configured using the button next to the drop down box. The name of the connection is set to 'mySQLconn'. The following values are set in the connection parameter's wizard: the Database system is set to 'mySQL'. The Host is set to 'localhost'. The Port is set to '3306'. The Database scheme is set to 'golf'; this is the name of the database. The User is set to 'root'. No password is provided. You will need a password if your database is password protected. Set all the values and test the connection. Make sure that the connection works.

The table name parameter is set to 'golf_table' which is the name of the required table in the 'golf' database. Run the process, you will see the entire 'golf_table' in the Results Workspace. You can also check the 'golf' database in phpmyadmin to see the 'golf_table'. You can read this table from the database using the Read Database operator. Please study the Example Process of the Read Database operator for more information.
5. Export

Update Database

This operator updates the values of all examples with matching ID values in a database.

Description

The Update Database operator is used for updating an existing table in the specified SQL database. You need to have at least basic understanding of databases and database connections in order to use this operator properly. Go through the parameters and the attached Example Process to understand the flow of this operator.

The user can specify the database connection, a table name and ID column names. The most convenient way of defining the necessary parameters is the Manage Database Connections wizard. The most important parameters (database URL and user name) will be automatically determined by this wizard.

The row(s) to update are specified via the db id attribute name parameter. If the id columns of the table do not match all the id values of any given example, the row will be inserted instead. The ExampleSet attribute names must be a subset
of the table column names, otherwise the operator will fail.

**Input Ports**

**input (inp)** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

**Output Ports**

**through (thr)** The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

**Parameters**

**define connection (selection)** This parameter indicates how the database connection should be specified. It gives you three options: predefined, url and jndi.

**connection (string)** This parameter is only available when the **define connection** parameter is set to **predefined**. This parameter is used for connecting to the database using a predefined connection. You can have many predefined connections. You can choose one of them using the drop down box. You can add a new connection or modify previous connections using the button next to the drop down box. You may also accomplish this by clicking on **Manage Database Connections...** from the **Tools** menu in the main window. A new window appears. This window asks for several details e.g. **Host, Port, Database system, schema, username** and **password**. The **Test** button in this new window will allow you to check whether the connection can be made. Save the connection once the test is successful. After saving a new connection, it can be chosen from the drop down box of the **connection** parameter. You need to have basic understanding of databases for configuring a connection.
5. Export

database system *(selection)* This parameter is only available when the `define connection` parameter is set to `url`. This parameter is used for selecting the database system in use. It can have one of the following values: MySQL, PostgreSQL, Sybase, HSQLDB, ODBC Bridge (e.g. Access), Microsoft SQL Server (JTDS), Ingres, Oracle.

database url *(string)* This parameter is only available when the `define connection` parameter is set to `url`. This parameter is used for defining the URL connection string for the database, e.g. `jdbc:mysql://foo.bar:portnr/database`.

username *(string)* This parameter is only available when the `define connection` parameter is set to `url`. This parameter is used for specifying the username of the database.

password *(string)* This parameter is only available when the `define connection` parameter is set to `url`. This parameter is used for specifying the password of the database.

jndi name *(string)* This parameter is only available when the `define connection` parameter is set to `jndi`. This parameter is used for giving the JNDI a name for a data source.

table name This parameter is used for selecting the required table from the specified database. Please note that you can also write a table name here, if the table does not exist it will be created during writing.

attribute filter type *(selection)* This parameter allows you to select the ID attribute which values ALL have to match in the example set and the database for the row to be updated. It has the following options:

- **all** Does not make sense in this context so do not use, will break the process.

- **single** This option allows the selection of a single id attribute.

- **subset** This option allows the selection of multiple id attributes through a list. This option will not work if the meta data is not known.

- **regular_expression** This option allows you to specify a regular expression for the id attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.
5.1. Data

- **value_type** This option allows selection of all the id attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows the selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

### Tutorial Processes

#### Updating an ExampleSet in a mySQL database

The 'Iris' data set is loaded using the Retrieve operator. The Update Database operator is used to update an existing database table named TTest in the MM-y connectionSSQL database. Rows in the example set and table which match on their ID column will be updated. If no match can be found, the row will be
5. Export

inserted instead.

![Diagram]

**Write Special Format**

This operator writes an ExampleSet or subset of an ExampleSet in a special user defined format.

**Description**

The path of the file is specified through the *example set file* parameter. The *special format* parameter is used for specifying the exact format. The character following the $ character introduces a command. Additional arguments to this command may be supplied by enclosing them in square brackets. The following commands can be used in the *special format parameter*:

- $a$ : This command writes all attributes separated by the default separator.

- $a[separator]$ : This command writes all attributes separated by a separator (the separator is specified as an argument in brackets).

- $s[separator][indexSeparator]$ : This command writes in sparse format. The separator and indexSeparator are provided as first and second arguments respectively. For all non zero attributes the following strings are concatenated: the column index, the value of the indexSeparator, the attribute value. The attributes are separated by the specified separator.

- $v[name]$ : This command writes the values of a single attribute. The
attribute name is specified as an argument. This command can be used for writing both regular and special attributes.

- `k[index]` : This command writes the values of a single attribute. The attribute index is specified as an argument. The indices start from 0. This command can be used for writing only regular attributes.

- `l` : This command writes the values of the label attribute.

- `p` : This command writes the values of the predicted label attribute.

- `d` : This command writes all prediction confidences for all classes in the form 'conf(class)=value'

- `d[class]` : This command writes the prediction confidences for the defined class as a simple number. The required class is provided as an argument.

- `i` : This command writes the values of the id attribute.

- `w` : This command writes the example weights.

- `b` : This command writes the batch number.

- `n` : This command writes the newline character i.e. newline is inserted when this character is reached.

- `t` : This command writes the tabulator character i.e. tab is inserted when this character is reached.

- `$$` : This command writes the dollar sign.

- `[` : This command writes the '[' character i.e. the opening square bracket.

- `]` : This command writes the ']' character i.e. the closing square bracket.

Please Make sure that the format string ends with `n` or the *add line separator* parameter is set to true if you want examples to be separated by newlines.
5. Export

Input Ports

input \((inp)\) This input port expects an ExampleSet. It is output of the Apply Model operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

through \((thr)\) The ExampleSet that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

Parameters

example set file \((filename)\) The ExampleSet is written into the file specified through this parameter.

special format \((string)\) This parameter specifies the exact format of the file. Many commands are available for specifying the format. These commands are discussed in the description of this operator.

fraction digits \((integer)\) This parameter specifies the number of fraction digits in the output file. This parameter is used for rounding off real numbers. Setting this parameter to -1 will write all possible digits i.e. no rounding off is done.

quote nominal values \((boolean)\) This parameter indicates if nominal values should be quoted with double quotes.

add line separator \((boolean)\) This parameter indicates if each example should be followed by a line break or not. If set to true, each example is followed by a line break automatically.

zipped \((boolean)\) This parameter indicates if the data file content should be zipped or not.

overwrite mode \((selection)\) This parameter indicates if an existing file should be overwritten or data should be appended.
5.1. Data

encoding (selection) This is an expert parameter. There are different options, users can choose any of them

Tutorial Processes

Writing labeled data set in a user-defined format

The k-NN classification model is trained on the 'Golf' data set. The trained model is then applied on the 'Golf-Testset' data set using the Apply Model operator. The resulting labeled data set is written in a file using the Write Special Format operator. Have a look at the parameters of the Write Special Format operator. You can see that the ExampleSet is written into a file named 'special'. The special format parameter is set to \$[ \$1 \$] \$t \$p \$t \$d[yes] \$t \$d[no].\n
This format string is composed of a number of commands, it can be interpreted as: '[label] predicted label confidence (yes) confidence (no)'. This format string states that four attributes shall be written in the file i.e. 'label', 'predicted label', 'confidence (yes)' and 'confidence (no)'. Each attribute should be separated by a tab. The label attribute should be enclosed in square brackets. Run the process and see the written file for verification.
Write Model

This operator writes a model into a file. The model can be written in three modes i.e. XML, XML Zipped and Binary.

Description

The Write Model operator writes the input model into the file specified by the \textit{model file} parameter. Since models are often written into files and then loaded for applying them in other processes or applications, this operator offers three different writing modes for models. The writing mode is controlled by the \textit{output type} parameter. Explanation of all the available writing modes is given in the parameters section. This operator is also able to keep old files if the \textit{overwrite existing file} parameter is set to false. However, this could also be achieved by using some of the parameter macros provided by RapidMiner like \texttt{%t} or \texttt{%a}. The model file written by this operator can be read by the Read Model operator.

Input Ports

\texttt{input (inp)} This port expects a model. In the attached Example Process the K-NN classification model is provided at the input.

Output Ports

\texttt{through (thr)} The model provided at the input port is delivered through this output port without any modifications. This port is usually used for reusing the same model in further operators of the process.
5.2. Models

Parameters

model file (filename) This parameter is used to specify the file where the input model is to be written.

overwrite existing file (boolean) If this parameter is set to true the file specified in the model file parameter is overwritten.

output type (selection) This operator offers three different writing modes for models

- **xml** If this mode is selected, the models are written as plain text XML files. The file size is usually the largest in this mode as compared to other modes. The file size may be several hundred mega bytes so you should be cautious. But this model type has the advantage that the user can inspect and change the model files.

- **xml_zipped** If this mode is selected, the models are written as zipped XML files. Users can simply unzip the files and read or change the contents. The file size is usually the smallest in this mode compared to other modes. For these reasons, this mode is the default writing mode for models. Please note that due to the XML parsing and unzipping the loading times for this mode are the longest compared to other modes.

- **binary** If this mode is selected, the models are written in binary format. The resulting model files cannot be inspected by the user and the file sizes are usually slightly larger than the zipped XML files. The loading time, however, is lesser than the time needed for the other modes.

Tutorial Processes

Writing the K-NN model into an XML file

This Example Process shows how the Write Model operator can be used to write
5. Export

a model. The Golf data set is loaded using the Retrieve operator. The K-NN operator is applied on it. The resultant model is provided as input to the Write Model operator. The model file parameter is set to 'D:\model' and the output type parameter is set to XML. Thus an XML file named 'model' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written model file and make changes in it. The model file can be used in RapidMiner by the Read Model operator.

Write Clustering

This operator writes the given cluster model into the specified file.

Description

The Write Clustering operator writes the cluster model provided at the input port into the specified file. The path of the file is specified through the cluster model file parameter. The Read Clustering operator can be used for reading the cluster model written by this operator. There are numerous clustering operators that create a cluster model e.g. the K-Means, K-Medoids etc. Clustering operators are located at 'Modeling/Clustering and Segmentation' in the Operators window.
5.2. Models

Input Ports

input (inp) This input port expects a cluster model. It is the output of the K-Means operator in the attached Example Process.

Output Ports

through (thr) The cluster model that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same cluster model in further operators of the process.

Parameters

cluster model file (filename) The path of the file where the cluster model is to be written is specified here. It can be selected using the choose a file button.

Tutorial Processes

Writing and then Reading a cluster model from a file

This Example Process shows how the Write Clustering operator can be used for writing a cluster model and how the Read Clustering operator can be used for reading the cluster model from this file. The 'Sonar' data set is loaded using the Retrieve operator. The K-Means operator is applied on it for generating a cluster model. A breakpoint is inserted here so that you can have a look at the cluster model. The resultant cluster model is provided as input to the Write Clustering operator. The cluster model file parameter is set to 'D:\cluster model' thus a file named 'cluster model' is created (if it does not already exist) in the
5. Export

'D' drive of your computer. This cluster model file is then read by using the Read Clustering operator. The cluster model file parameter is set to 'D:\cluster\model' to read the same file that was written by the Write Clustering operator. The resultant cluster model is connected to the result port of the process and it can be seen in the Results Workspace.

Write Weights

This operator writes the given attribute weights into the specified file.

Description

The Write Weights operator writes the attribute weights provided at the input port into the specified file. The path of the file is specified through the attribute weights file parameter. Each line in the file holds the name of one attribute and
its weight. The Read Weights operator can be used to read the weights written by this operator.

The attribute weights specify the relevance of the attributes with respect to the label attribute. There are numerous operators that create attribute weights e.g. the Weight by Correlation, Weight by PCA etc. Operators that generate attribute weights are located at 'Modeling/Attribute Weighting' in the Operators Window.

**Input Ports**

*input (inp)* This input port expects attribute weights. It is the output of the Weight by Correlation operator in the attached Example Process.

**Output Ports**

*through (thr)* The attribute weights that were provided at the input port are delivered through this output port without any modifications. This is usually used to reuse the same attribute weights in further operators of the process.

**Parameters**

*attribute weights file (filename)* The path of the file where the attribute weights are to be written is specified here. It can be selected using the *choose a file* button.

*encoding (selection)* This is an expert parameter. There are different options, users can choose any of them.
5. Export

Tutorial Processes

Writing and then Reading attribute weights from a file

This Example Process shows how the Write Weights operator can be used for writing attribute weights and how the Read Weights operator can be used for reading attribute weights from this file. The 'Sonar' data set is loaded using the Retrieve operator. The Weight by Correlation operator is applied on it. A breakpoint is inserted here so that you can have a look at the attribute weights. The resultant weights are provided as input to the Write Weights operator. The attribute weights file parameter is set to 'D:\sonar weights' thus a file named 'sonar weights' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written weights file and make changes in it (if required). The attribute weights file is then read by using the Read Weights operator. The attribute weights file parameter is set to 'D:\sonar weights' to read the same file that was written by the Write Weights operator. The resultant weights are connected to the result port of the process. The attribute weights can be seen in the Results Workspace.
5.3. Attributes

Write Constructions

This operator writes all attributes of the given ExampleSet into the specified file.

Description

The Write Constructions operator writes all attributes of the ExampleSet given at the input port into the specified file. The path of the file is specified through the attribute constructions file parameter. Each line in the file holds the construction description of one attribute. The Read Constructions operator can be used for reading this file.
5. Export

Input Ports

input (inp) This input port expects an ExampleSet as input. It is the output of the Retrieve operator in the attached Example Process.

Output Ports

through (thr) The ExampleSet the was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same ExampleSet in further operators of the process.

Parameters

attribute constructions file (filename) The path of the file where the attribute construction descriptions are to be written is specified here. It can be selected using the choose a file button.

encoding (selection) This is an expert parameter. There are different options, users can choose any of them.

Tutorial Processes

Introduction to the Write Constructions operator

This Example Process shows how the Write Constructions operator can be used for writing attribute constructions of an ExampleSet into a file. The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. This ExampleSet is provided as input to the Write Constructions operator. The attribute constructions file parameter
5.4. Results

is set to 'D:\attributes' thus a file named 'attributes' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written file and make changes in it (if required). This file can be read into RapidMiner by the Read Constructions operator.

Write Performance

This operator writes the given performance vector into a file.

Description

The Write Performance operator writes the performance vector provided at the input port into the specified file. The path of the file is specified through the performance file parameter. The files written by this operator are usually not very readable for humans. The Read Performance operator can be used to read the files written by this operator.

The performance vector has information about the statistical performance of a process or model. It stores information about various performance measuring
5. Export

criteria. There are numerous operators that produce performance vectors e.g. the Performance, Performance (Classification), Performance (User-Based) operators.

Input Ports

input (inp) This input port expects a performance vector. It is the output of the Performance (Classification) operator in the attached Example Process.

Output Ports

through (thr) The performance vector that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same performance vector in further operators of the process.

Parameters

performance file (filename) The path of the file where the performance vector is to be written is specified here. It can be selected using the choose a file button.

Tutorial Processes

Writing the performance of a classification model into a file

The 'Golf' data set is loaded using the Retrieve operator. The Default Model operator is applied on it. The Default Model operator generates a classification model. The Apply Model operator is used for applying this model on the 'Golf-Testset' data set. The Performance (Classification) operator is then used to
measure the performance of the classification model. The Performance (Classification) operator returns a performance vector that contains information about various performance measuring criteria. The Write Performance operator is applied on it to write this performance vector into a file. The performance file parameter is provided with this path: 'D:\performance.txt'. Thus a text file named 'performance' is created in the 'D' drive of your computer.

### Write Parameters

This operator writes the given set of parameters into the specified file.

### Description

The Write Parameters operator writes the parameter set provided at the input port into the specified file. The path of the file is specified through the parameter file parameter. Each line in the file holds the name of the operator, one
5. Export

parameter and its value. The Read Parameters operator can be used to read the parameter set written by this operator. There are numerous operators that generate a parameter set e.g. the Optimize Parameters (Grid), Optimize Parameters (Evolutionary) operators etc.

Input Ports

input (inp) This input port expects a parameter set. It is the output of the Optimize Parameters (Grid) operator in the attached Example Process.

Output Ports

through (thr) The parameter set that was provided at the input port are delivered through this output port without any modifications. This is usually used to reuse the same parameter set in further operators of the process.

Parameters

parameter file (filename) The path of the file where the parameter set is to be written is specified here. It can be selected using the choose a file button.
encoding (selection) This is an expert parameter. There are different options, users can choose any of them

Tutorial Processes

Writing and then Reading a parameter set from a file

This Example Process shows how the Write Parameters operator can be used
for writing a parameter set into a file and how the Read Parameters operator can be used for reading a parameter set from this file. The 'Weighting' data set is loaded using the Retrieve operator. The Optimize Parameters (Grid) operator is applied on it. A breakpoint is inserted here so that you can have a look at the parameter set generated by the Optimize Parameters (Grid) operator, it is not relevant for now how it generates this parameter set. For more information please study the documentation of the Optimize Parameters (Grid) operator. The parameter set is provided as input to the Write Parameters operator. The parameter file parameter is set to 'D:\parameters.txt' thus a file named 'parameters' is created (if it does not already exist) in the 'D' drive of your computer. You can open the parameter set file and make changes in it (if required). The parameter set file is then read by using the Read Parameters operator. The parameter file parameter is set to 'D:\parameters.txt' to read the same file that was written by the Write Parameters operator. The resultant parameter set is connected to the result port of the process. The parameter set can be seen in the Results Workspace.

Write Threshold

This operator writes the given threshold into the specified file.
5. Export

Description

The Write Threshold operator writes the threshold provided at the input port into the specified file. The path of the file is specified through the threshold file parameter. The file stores values of threshold, first class and second class parameters. The Read Threshold operator can be used to read the threshold written by this operator.

The threshold is used for crisp classification based on the prediction confidences (soft predictions). Operators like the Create Threshold operator create thresholds which can be applied on a labeled ExampleSet using the Apply Threshold operator. For more information about thresholds please study the Create Threshold operator.

Input Ports

input (inp) This input port expects a threshold. It is the output of the Create Threshold operator in the attached Example Process.

Output Ports

through (thr) The threshold that was provided at the input port is delivered through this output port without any modifications. This is usually used to reuse the same threshold in further operators of the process.

Parameters

threshold file (filename) The path of the file where the threshold is to be written is specified here. It can be selected using the choose a file button.
encoding (selection) This is an expert parameter. There are different options,
5.5. Other

users can choose any of them

Tutorial Processes

Writing and then Reading threshold from a file

This Example Process shows how the Write Threshold operator can be used for writing a threshold into a file and how the Read Threshold operator can be used for reading a threshold from this file. This process starts with the Create Threshold operator which is used for creating a threshold. The threshold parameter is set to 0.700 and the first class and second class parameters are set to 'negative' and 'positive' respectively. A breakpoint is inserted here so that you can see the threshold in the Results Workspace. This threshold is provided as input to the Write Threshold operator. The threshold file parameter is set to 'D:\threshold.txt' thus a file named 'threshold' is created (if it does not already exist) in the 'D' drive of your computer. You can open the written threshold file and make changes in it (if required). The threshold file is then read by using the Read Threshold operator. The threshold file parameter is set to 'D:\threshold.txt' to read the same file that was written by the Write Threshold operator. The resultant threshold is connected to the result port of the process and it can be seen in the Results Workspace.
5. Export
6 Data Transformation

Rename

This operator can be used to rename one or more attributes of an ExampleSet.

Description

The Rename operator is used for renaming one or more attributes of the input ExampleSet. Please keep in mind that attribute names must be unique. The Rename operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of integer type and regular role. Renaming the attribute to 'beta' will just change its name. It will retain its type integer and role regular. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.
6. Data Transformation

Input Ports

example set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with data for input because attributes are specified in its meta data. The Retrieve operator provides meta data along-with data.

Output Ports

example set (exa) The ExampleSet with renamed attributes is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

old name (string) This parameter is used to select the attribute whose name is to be changed.
new name (string) The new name of the attribute is specified through this parameter. Name can also include special characters.
rename additional attributes (string) To rename more than one attributes click on the Edit List button. Here you can select attributes and assign new names to them.
6.1. Name and Role Modification

Tutorial Processes

Renaming multiple attributes

The 'Golf' data set is used in this Example Process. The 'Play' attribute is renamed to 'Game' and the 'Wind' attribute is renamed to '#*#'. The 'Wind' attribute is renamed to '#*#', just to show that special characters can also be used to rename attributes. However, attribute names should always be meaningful and should be relevant to the type of information stored in them.

Rename by Replacing

This operator can be used to rename a set of attributes by replacing parts of the attribute names by a specified replacement.

Description

The Rename by Replacing operator replaces parts of the attribute names by the specified replacement. This operator is used mostly for removing unwanted parts of attribute names like whitespaces, parentheses, or other unwanted characters. The replace what parameter defines that part of the attribute name that should be replaced. It can be defined as a regular expression which is a very powerful
6. Data Transformation

The tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. The replace by parameter can be defined as an arbitrary string. Empty strings are also allowed. Capturing groups of the regular expression of the replace what parameter can be accessed with $1, $2, $3 etc. Please study the attached Example Process for more understanding.

Please keep in mind that attribute names must be unique. The Rename by Replacing operator has no impact on the type or role of an attribute. For example if you have an attribute named 'alpha' of integer type and regular role. Renaming the attribute to 'beta' will just change its name. It will retain its type integer and role regular. To change the role of an operator, use the Set Role operator. Many type conversion operators are available for changing the type of an attribute at 'Data Transformation/Type Conversion'.

Input Ports

default set input  (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

Output Ports

default set output  (exa) The ExampleSet with renamed attributes is output of this port.
original  (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
6.1. Name and Role Modification

Parameters

**attribute filter type (selection)** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (**attribute**) becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (**regular expression, use except expression**) become visible in the Parameter View.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (**value type, use value type exception**) become visible in the Parameter View.
- **block_type** This option is similar in working to the **value_type** option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example **value_series_start** and **value_series_end** block types both belong to the **value_series** block type. When this option is selected some other parameters (**block type**, **value_series_start**, **value_series_end**).
6. Data Transformation

*use block type exception* become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** (*string*) The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** (*string*) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression** (*string*) The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** (*boolean*) If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameter View.

**except regular expression** (*string*) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** (*selection*) The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** (*boolean*) If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except
6.1. Name and Role Modification

`value type)` becomes visible in the Parameter View.

`except value type (selection)` The attributes matching this type will not be 
selected even if they match the previously mentioned type i.e. `value type` parameter's value.

`block type (selection)` The block type of attributes to be selected can be chosen 
from a drop down list.

`use block type exception (boolean)` If enabled, an exception to the selected 
block type can be specified. When this option is selected another parameter (`except block type`) becomes visible in the Parameter View.

`except block type (selection)` The attributes matching this block type will be 
not be selected even if they match the previously mentioned block type i.e. `block type` parameter's value.

`numeric condition (string)` The numeric condition for testing examples of 
umeric attributes is specified here. For example the numeric condition `'> 6'` will 
keep all nominal attributes and all numeric attributes having a value of greater 
than 6 in every example. A combination of conditions is possible: `'> 6 & & < 11'` or `'< = 5 || < 0'`. But `& &` and `||` cannot be used together in one numeric 
condition. Conditions like `'(> 0 & & < 2) || (> 10 & & < 12)'` are not allowed 
because they use both `& &` and `||`. Use a blank space after `'> ', '= '` and `'< '` e.g. 
`'< 5'` will not work, so use `'< 5'` instead.

`include special attributes (boolean)` The special attributes are attributes with 
special roles which identify the examples. In contrast regular attributes simply 
describe the examples. Special attributes are: id, label, prediction, cluster, 
weight and batch. By default all special attributes are selected irrespective of the 
conditions in the Select Attribute operator. If this parameter is set to true, Special 
attributes are also tested against conditions specified in the Select Attribute 
operator and only those attributes are selected that satisfy the conditions.

`invert selection (boolean)` If this parameter is set to true, it acts as a NOT gate, 
it reverses the selection. In that case all the selected attributes are unselected 
and previously unselected attributes are selected. For example if attribute 'att1' 
is selected and attribute 'att2' is unselected prior to checking of this parameter. 
After checking of this parameter 'att1' will be unselected and 'att2' will be se-
lected.

`replace what (string)` The `replace what` parameter defines that part of the at-
6. Data Transformation

tribute name that should be replaced. It can be defined as a regular expression. Capturing groups of the regular expression of the replace what parameter can be accessed in the replace by parameter with $1, \$2, \$3$ etc.

**replace by (string)** The replace by parameter can be defined as an arbitrary string. Empty strings are also allowed. Capturing groups of the regular expression of the replace what parameter can be accessed with $1, \$2, \$3$ etc.

Tutorial Processes

Renaming attributes of the Sonar data set

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. You can see that the ExampleSet has 60 regular attributes with names like attribute_1, attribute_2 etc. The Rename by Replacing operator is applied on it. The attribute filter type parameter is set to 'all' thus all attributes can be renamed by this operator. The replace what parameter is set to the regular expression: '(attribute)'. The brackets are used for specifying the capturing group which can be accessed in the replace by parameter with $1$. The replace by parameter is set to '$1$'. Wherever 'attribute_' is found in names of the 'Sonar' attributes, it is replaced by the first capturing group and a dash i.e. 'att-'. Thus attributes are renamed to format att-1, att-2 and so on. This can be verified by seeing the results in the Results Workspace.
Set Role

This operator is used to change the role of one or more attributes.

Description

Role of an attribute reflects the part played by that attribute in an ExampleSet. Changing the role of an attribute may change the part played by that attribute in a process. One attribute can have exactly one role. This operator is used to change the role of one or more attributes of the input ExampleSet. This is a very simple operator, all you have to do is to select an attribute and select a new role for it. Different learning operators require attributes with different roles. This operator is frequently used to set the right roles for attributes before applying the desired operator. The change in role is only for the current process, i.e. Role of attribute is not changed permanently in the ExampleSet. Set Role operator should not be confused with Rename operator or Type Conversion operators. Rename operator is used to change the name of an attribute. Many type Conversion operators are available (at Data Transformation/Type conversion/) to change the type of attributes e.g. Nominal to Binominal operator, Numerical to Polynomial operator and many more.

Broadly roles are classified into two types i.e. regular and special. Regular attributes simply describe the examples. Regular attributes are usually used during learning processes. One ExampleSet can have numerous regular attributes. Special attributes are those which identify the examples separately. Special attributes have some specific task. Special roles are: label, id, prediction, cluster, weight, and batch. An ExampleSet can have numerous special attributes but one special role cannot be repeated. If one special role is assigned to more than one attributes in an ExampleSet, all such attributes will be dropped except the last one. This concept can be understood easily by studying the attached Example Process. Explanation of various roles is given in the parameters section.
6. Data Transformation

Input Ports

example set *(exa)* This input port expects an ExampleSet. It is output of Retrieve operator in our Example Process. Output of other operators may also be used as input. It is essential that meta data should be attached with data for input because role of an attribute is specified in its meta data of the ExampleSet. Retrieve operator provides meta data along-with data.

Output Ports

example set *(exa)* ExampleSet with modified role(s) is output of this port.
original *(ori)* ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

name *(string)* The name of the attribute whose role should be changed is specified through this parameter. You can select the attribute either from the drop down list or type it manually.

target role *(string)* The target role of the selected attribute is the new role assigned to it. Following target roles are possible:

- regular Attributes without a special role, i.e. those which simply describe the examples are called regular attributes and just leave out the role designation in most cases. Regular attributes are used as input variables for learning tasks

- id This is a special role, it acts as id attribute for the ExampleSet and it is usually unique in every example of the ExampleSet. Id role is used to clearly identify the examples of concerned ExampleSet. In this case the attribute adopts the role of an identifier and is called ID for short. Unique
ids can be given to all the examples using Generate ID operator.

- **label** This is a special role, it acts as a target attribute for learning operators e.g. Decision Tree operator. Labels identify the examples in any way and they must be predicted for new examples that are not yet characterized in such a manner. Label is also called 'goal variable'.

- **prediction** This is a special role, it acts as predicted attribute of a learning scheme. For example when a predictive model is learnt through any learning operator and then it is applied using Apply Model operator, in the output we have a new attribute with role *prediction* which holds the values of *label* predicted by the given model. *Label* and *prediction* attributes are also used for evaluating performance of a model.

- **cluster** This is a special role, it indicates the membership of an example of the ExampleSet to a particular cluster. For example, output of k-Mean operator adds a column with *cluster* role.

- **weight** This is a special role, it indicates the weight of the examples with regard to the *label*. Weights are used in learning processes to give different importance to examples with different weights. Attribute weights are used in numerous operators e.g. Select By Weights operator. Weights can also be used in evaluating performance of models e.g. Performance operator has *use example weights* parameter to consider *weight* of examples during performance evaluation process.

- **batch** This is a special role, it indicates the membership to an example batch.

- **user defined** Any role can be provided by directly typing in the textbox instead of selecting a role from the dropdown menu. If 'ignore' is written in the textbox, that attribute will be ignored by the coming operators in the process. This is also a special role, thus it needs to be unique. To ignore multiple attributes unique roles can be assigned like ignore01, ignore02, ignore03 and so on.
6. Data Transformation

set additional roles (menu) Click this button to modify roles of more than one attributes. A click on this button opens a new menu which allows you to select any attribute and assign any role to it. It also allows assigning multiple roles to the same attribute. But, as an attribute can have exactly one role, only the last role assigned to that attribute is actually assigned to it and all previous roles assigned to it are ignored.

Tutorial Processes

Setting roles of attributes

In this Example Process, Labor-Negotiation data set is loaded using Retrieve operator. Roles of its attributes are changed using Set Role operator. Here is an explanation of what happens when this process is executed:

- *name* and *shift-differential* attributes are dropped because *standby-pay* is also given the *label* role. As *label* is a special role and only one attribute of the same special role can exist, the first attributes are dropped and the last attribute (*standby-pay*) is assigned *label* role.

- *duration* is assigned *weight* role

- *wage-inc-1st*, *longterm-disability-assistance*, *pension*, *bereavement-assistance* and *wage-inc-2nd* are given *regular* role. They were regular attributes even before reassignment of the same role. Thus assigning the same role will not make any change. As there can be numerous regular attributes, no attribute is dropped.

- *wage-inc-3rd* and *working-hours* role were not modified. Thus they retain their original roles i.e. *regular* role.

- *col-adj* is assigned *id* role.

- *education-allowance* is assigned *batch* role.
6.1. Name and Role Modification

- *statutory-holidays* and *vacations* are assigned *ignore0* and *ignore1* roles respectively.

- *contrib-to-dental-plan* is assigned *prediction* role. *contrib-to-health-plan* is assigned *cluster* role.

Some attributes are dropped as explained earlier but note that the number of examples remains the same. Roles assigned in this Example Process were just to show how the Set Role operator works; in real scenarios such assignments of role may not be very useful. This also highlights another point that Set Role is not a context-aware operator. It assigns roles set by the users irrespective of its context. So users must have the knowledge of what role to be assigned in which scenario. Thanks to the Problems View and quick fixes, it becomes easy to set the right roles before applying different learning operators. Note that Problems View displays two warnings even in this Example Process.

**Exchange Roles**

This operator exchanges the roles of two attributes.
Description

The Exchange Roles operator exchanges the roles of the two specified attributes i.e. it assigns the role of the first attribute to the second attribute and vice versa. This can be useful, for example, to exchange the roles of a label with a regular attribute (or vice versa), or a label with a batch attribute, a label with a cluster etc. For more information about roles please study the description of the Set Role operator.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set output (exa) The roles of the specified attributes are exchanged and the resultant ExampleSet is delivered through this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

first attribute (string) This parameter specifies the name of the first attribute for the attribute role exchange.
second attribute (string) This parameter specifies the name of the second attribute for the attribute role exchange.
6.2. Type Conversion

Tutorial Processes

Exchanging roles of attributes of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the roles of the Play and Outlook attributes are label and regular respectively. The Exchange Roles operator is applied on the 'Golf' data set to exchange the roles of these attributes. The first attribute and second attribute parameters are set to 'Play' and 'Outlook' respectively. The resultant ExampleSet can be seen in the Results Workspace. You can see that now the role of the Play attribute is regular and the role of the Outlook attribute is label.

Numerical to Binominal

This operator changes the type of the selected numeric attributes to a binominal type. It also maps all values of these attributes to corresponding binominal values.
Description

The Numerical to Binominal operator changes the type of numeric attributes to a binominal type (also called binary). This operator not only changes the type of selected attributes but it also maps all values of these attributes to corresponding binominal values. Binominal attributes can have only two possible values i.e. 'true' or 'false'. If the value of an attribute is between the specified minimal and maximal value, it becomes 'false', otherwise 'true'. Minimal and maximal values can be specified by the min and max parameters respectively. If the value is missing, the new value will be missing. The default boundaries are both set to 0.0, thus only 0.0 is mapped to 'false' and all other values are mapped to 'true' by default.

Input Ports

element set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data. The ExampleSet should have at least one numeric attribute because if there is no such attribute, use of this operator does not make sense.

Output Ports

element set (exa) The ExampleSet with selected numeric attributes converted to binominal type is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
6.2. Type Conversion

Parameters

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to convert to binominal form. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type,
6. Data Transformation

*use block type exception* become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to binominal will take place; all other attributes will remain unchanged.

**regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type (selection)** The type of attributes to be selected can be chosen from
a drop down list.

use value type exception (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.

block type (selection) The block type of attributes to be selected can be chosen from a drop down list.

use block type exception (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

except block type (selection) The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

numeric condition (string) The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=', and '<' e.g. '< 5' will not work, so use '< 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Nominal to Binominal operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Nominal to Binominal operator and only those attributes are selected that satisfy the conditions.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1'
6. Data Transformation

is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected. 

**min (real)** This parameter is used to set the lower bound of the range. The **max** parameter is used to set the upper bound of the range. The attribute values that fell in this range are mapped to 'false'. The attribute values that do not fell in this range are mapped to 'true'.

**max (real)** This parameter is used to set the upper bound of the range. The **min** parameter is used to set the lower bound of the range. The attribute values that fell in this range are mapped to 'false'. The attribute values that do not fell in this range are mapped to 'true'.

Tutorial Processes

Converting numeric attributes of the Sonar data set to binominal attributes

This Example Process mostly focuses on the **min** and **max** parameters. All remaining parameters are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for selection of attributes. You can study the Example Process of the Select Attributes operator if you want an understanding of these parameters.

The 'Sonar' data set is loaded using the Retrieve operator. The Numerical to Binominal operator is applied on it. The **min** parameter is set to 0.0 and the **max** parameter is set to 0.01. All other parameters are used with default values. The **attribute filter type** parameter is set to 'all', thus all numeric attributes of the 'Sonar' data set will be converted to binominal type. As you can see in the Results Workspace, before application of the Numerical to Binominal operator, all attributes were of real type. After application of this operator they are now all changed to binominal type. All attribute values that fell in the range from 0.0 to 0.01 are mapped to 'false', all the other values are mapped to 'true'.
### 6.2. Type Conversion

#### Numerical to Polynominal

This operator changes the type of selected numeric attributes to a polynomial type. It also maps all values of these attributes to corresponding polynomial values. This operator simply changes the type of selected attributes; if you need a more sophisticated normalization method please use the discretization operators.

#### Description

The Numerical to Polynominal operator is used for changing the type of numeric attributes to a polynomial type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to corresponding polynomial values. It simply changes the type of selected attributes i.e. every new numerical value is considered to be another possible value for the polynomial attribute. In other words, each numerical value is simply used as nominal value of the new attribute. As numerical attributes can have a huge number of different values even in a small range, converting such a numerical attribute to polynomial form will generate a huge number of possible values for the new attribute. Such a polynomial attribute may not be a very useful one and it may increase memory usage significantly. If you need a more sophisticated normalization method please use the discretization operators. The Discretization operators are at: "Data Transformation/ Type Conversion/ Discretization".
6. Data Transformation

Input Ports

element set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data. The ExampleSet should have at least one numeric attribute because if there is no such attribute, use of this operator does not make sense.

Output Ports

example set (exa) The ExampleSet with selected numeric attributes converted to nominal type is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to convert to polynomial form. It has the following options:

- all This option simply selects all the attributes of the ExampleSet. This is the default option.
- single This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.
- subset This option allows selection of multiple attributes through a list.
6.2. Type Conversion

All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of parameter attribute if the meta data is known.
attributes (string) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to polynominal will take place; all other attributes will remain unchanged.

regular expression (string) The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

use except expression (boolean) If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

except regular expression (string) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

value type (selection) The type of attributes to be selected can be chosen from a drop down list.

use value type exception (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.

block type (selection) The block type of attributes to be selected can be chosen from a drop down list.

use block type exception (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

except block type (selection) The attributes matching this block type will be removed from the final output even if they matched the previously mentioned
block type.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' > ', '=' and '< ' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Nominal to Polynominal operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Nominal to Polynominal operator and only those attributes are selected that satisfy the conditions.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**Tutorial Processes**

**Converting numeric attributes of the Sonar data set to polynominal attributes**

This Example Process mostly focuses on the working of this operator. All parameters of this operator are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for selection of attributes. You can study the Example Process of the Select Attributes operator if you want
6. Data Transformation

an understanding of these parameters.

The 'Sonar' data set is loaded using the Retrieve operator. The Numerical to Polynominal operator is applied on it. All parameters are used with default values. The attribute filter type parameter is set to 'all', thus all numeric attributes of the 'Sonar' data set will be converted to nominal type. As you can see in the Results Workspace, before application of the Numerical to Polynominal operator, all attributes were of real type. After application of this operator they are now all changed to nominal type. But if you have a look at the examples, they are exactly the same i.e. just the type of the values has been changed not the actual values. Every new numerical value is considered to be another possible value for the polynominal attribute. In other words, each numerical value is simply used as nominal value of the new attribute. As there is a very large number of different values for almost all attributes in the 'Sonar' data set, converting these attributes to polynomial form generates a huge number of possible values for the new attributes. These new polynomial attributes may not be very useful and they may increase memory usage significantly. In such a scenario it is always better to use a more sophisticated normalization method i.e. the discretization operators.

Numerical to Real

This operator changes the type of the selected numerical attributes to real type. It also maps all values of these attributes to real values.
6.2. Type Conversion

Description

The Numerical to Real operator converts selected numerical attributes (especially the integer attributes) to real valued attributes. Each integer value is simply used as a real value of the new attribute. If the value is missing, the new value will be missing.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one non-real numerical attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

example set output (exa) The ExampleSet with selected numerical attributes converted to real type is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply numerical to real conversion. It has the following options:
6. Data Transformation

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (*value type, use value type exception*) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are
selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

attribute (string) The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

attributes (string) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

regular expression (string) The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

use except expression (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except regular expression (string) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

value type (selection) The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

use value type exception (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.
6. Data Transformation

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'.

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '>', 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '>' 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=' and '<' e.g. '<5' will not work, so use '<5' instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

---

**Tutorial Processes**

**Integer to real conversion of attributes of the Golf data set**

The 'Golf' data set is loaded using the Retrieve operator. A *breakpoint* is in-
6.2. Type Conversion

sorted here so that you can have a look at the ExampleSet. You can see that the type of the Humidity and Temperature attributes is integer. The Numerical to Real operator is applied on the 'Golf' data set to convert the type of these integer attributes to real. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that now the type of these attributes is real.

Real to Integer

This operator changes the type of the selected real attributes to integer type. It also maps all values of these attributes to integer values.

Description

The Real to Integer operator converts selected real attributes to integer valued attributes. Each real value is either cut or rounded off and then used as an integer value of the new attribute. This option is controlled by the round values parameter. If it is set to false, the decimal portion of the real value is simply truncated otherwise it is rounded off. If the real value is missing, the new integer value will be missing.
Input Ports

example set input \((exa)\) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one real attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

example set output \((exa)\) The ExampleSet with selected real attributes converted to integer type is output of this port.
original \((ori)\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type \((selection)\) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- all This option simply selects all the attributes of the ExampleSet. This is the default option.
- single This option allows selection of a single attribute. When this option is selected another parameter \((attribute)\) becomes visible in the Parameter View.
- subset This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not
known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will
remain unchanged.

**regular expression (string)** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameter View.

**except regular expression** (string) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** (selection) The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameter View.

**except value type** (selection) The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. *value type* parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** (selection) The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameter View.

**except block type** (selection) The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** (string) The numeric condition for testing examples of num-
6.2. Type Conversion

Numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' > ', '=' and '< ' e.g. '< 5' will not work, so use '< 5' instead.

Include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

Invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

Round values (boolean) This parameter indicates if the values should be rounded off for conversion from real to integer. If not set to true, then the decimal portion of real values is simply truncated to convert the real values to integer values.

Tutorial Processes

Real to integer conversion of attributes of the Iris data set

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has four real attributes i.e. a1, a2, a3 and a4. The Real to Integer operator is applied on the 'Iris' data set to convert the type of these real attributes to integer. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that now the type of these attributes is integer.
6. Data Transformation

Nominal to Binominal

This operator changes the type of selected nominal attributes to a binominal type. It also maps all values of these attributes to binominal values.

Description

The Nominal to Binominal operator is used for changing the type of nominal attributes to a binominal type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to binominal values i.e. true and false. For example, if a nominal attribute with name 'costs' and possible nominal values 'low', 'moderate', and 'high' is transformed, the result is a set of three binominal attributes 'costs = low', 'costs = moderate', and 'costs = high'. Only the value of one of these attributes is true for a specific example, the value of the other attributes is false. Examples of the original ExampleSet where the 'costs' attribute had value 'low', in the new ExampleSet these examples will have attribute 'costs=low' value set to 'true', value of 'cost=moderate' and 'cost=high' attributes will be 'false'. Numeric attributes of the input ExampleSet remain unchanged.
6.2. Type Conversion

Input Ports

**example set (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in its meta data. The Retrieve operator provides meta data along-with data. The ExampleSet should have at least one nominal attribute because if there is no such attribute, use of this operator does not make sense.

Output Ports

**example set (exa)** The ExampleSet with selected nominal attributes converted to binominal type is output of this port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model (pre)** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

**create view (boolean)** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type (selection)** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to convert to binominal form. It has the following options:
6. Data Transformation

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example, real and integer types both belong to numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example, value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (nu-
6.2. Type Conversion

numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

attribute (string) The required attribute can be selected from this option. The attribute name can be selected from the drop down box of parameter attribute if the meta data is known.

attributes (string) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to binominal will take place; all other attributes will remain unchanged.

regular expression (string) The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

use except expression (boolean) If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

except regular expression (string) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

value type (selection) The type of attributes to be selected can be chosen from a drop down list.

use value type exception (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.
block type (selection) The block type of attributes to be selected can be chosen from a drop down list.

use block type exception (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

except block type (selection) The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

numeric condition (string) The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ‘> 6’ will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ‘> 6 & & < 11’ or ‘< = 5 || < 0’. But & & and || cannot be used together in one numeric condition. Conditions like ‘(> 0 & & < 2) || (> 10 & & < 12)’ are not allowed because they use both & & and ||. Use a blank space after ‘>’, ‘=’ and ‘<’ e.g. ‘< 5’ will not work, so use ‘< 5’ instead.

include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Nominal to Binominal operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Nominal to Binominal operator and only those attributes are selected that satisfy the conditions.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

transform binominal (boolean) This parameter indicates if attributes which are already binominal should be dichotomized i.e. they should be split in two columns with values true and false.

use underscore in name (boolean) This parameter indicates if underscores should be used in the new attribute names instead of empty spaces and '='.
Although the resulting names are harder to read for humans it might be more appropriate to use these if the data should be written into a database system.

**Tutorial Processes**

**Nominal to Binominal conversion of attributes of Golf data set**

This Example Process mostly focuses on the *transform binominal* parameter. All remaining parameters are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for selection of attributes. You can study the Example Process of the Select Attributes operator if you want an understanding of these parameters.

The Retrieve operator is used to load the Golf data set. A *breakpoint* is inserted at this point so that you can have a look at the data set before application of the Nominal to Binominal operator. You can see that the 'Outlook' attribute has three possible values i.e. 'sunny', 'rain' and 'overcast'. The 'Wind' attribute has two possible values i.e. 'true' and 'false'. All parameters of the Nominal to Binominal operator are used with default values. Run the process. First you will see the Golf data set. Press the run button again and you will see the final results. You can see that the 'Outlook' attribute is replaced by three binominal attributes, one for each possible value of the original 'Outlook' attribute. These attributes are 'Outlook = sunny', 'Outlook = rain', and 'Outlook = overcast'. Only the value of one of these attributes is true for a specific example, the value of the other attributes is false. Examples whose 'Outlook' attribute had the value 'sunny' in the original ExampleSet, will have the attribute 'Outlook =sunny' value set to 'true'in the new ExampleSet, the value of the 'Outlook =overcast' and 'Outlook =rain' attributes will be 'false'. The numeric attributes of the input ExampleSet remain unchanged.

The 'Wind' attribute was not replaced by two binominal attributes, one for each possible value of the 'Wind' attribute because this attribute is already binominal.
Still if you want to break it into two separate binominal attributes, this can be done by setting the *transform binominal* parameter to true.

**Nominal to Text**

This operator changes the type of selected nominal attributes to text. It also maps all values of these attributes to corresponding string values.

**Description**

The Nominal to Text operator converts all nominal attributes to string attributes. Each nominal value is simply used as a string value of the new attribute. If the value is missing in the nominal attribute, the new value will also be missing.

**Input Ports**

*example set (exa)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.
Output Ports

example set \((exa)\) The ExampleSet with selected nominal attributes converted to text is output of this port.

original \((ori)\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type \((selection)\) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply nominal to text conversion. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter \((attribute)\) becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters \((regular\ expression, use\ except\ expression)\) become visible in the Parameter View.
- **value_type** This option allows selection of all the attributes of a particular
6. Data Transformation

type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** *(string)* The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** *(boolean)* If enabled, an exception to the first regular
6.2. Type Conversion

expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

except regular expression (string) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the regular expression parameter).

value type (selection) The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

use value type exception (boolean) If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (except value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will not be selected even if they match the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

block type (selection) The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'.

use block type exception (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

except block type (selection) The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. block type parameter's value.

numeric condition (string) The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' > ', '=' and '< ' e.g. '< 5' will not work, so use '< 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes sim-
6. Data Transformation

ply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**Tutorial Processes**

**Applying the Nominal to Text operator on the Golf data set**

The 'Golf' data set is loaded using the Retrieve operator. A *breakpoint* is inserted after the Retrieve operator so that you can have a look at the 'Golf' data set before application of the Nominal to Text operator. You can see that the 'Golf' data set has three nominal attributes i.e. 'Play', 'Outlook' and 'Wind'. The Nominal to Text operator is applied on this data set. The *attribute filter type* parameter is set to 'single' and the *attribute* parameter is set to 'Outlook'. Thus this operator converts the type of the 'Outlook' attribute to text. You can verify this by seeing the results in the Meta Data View in the Results Workspace.
Nominal to Numerical

This operator changes the type of selected non-numeric attributes to a numeric type. It also maps all values of these attributes to numeric values.

Description

The Nominal to Numerical operator is used for changing the type of non-numeric attributes to a numeric type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to numeric values. Binary attribute values are mapped to 0 and 1. Numeric attributes of input the ExampleSet remain unchanged. This operator provides three modes for conversion from nominal to numeric. This mode is selected by the coding type parameter. Explanation of these coding types is given in the parameters and they are also explained in the example process.

Input Ports

`example set (exa)`: This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with data for input because attributes are specified in its meta data. The Retrieve operator provides meta data along-with data. The ExampleSet should have at least one non-numeric attribute because if there is no such attribute, the use of this operator does not make sense.
Output Ports

example set (exa) The ExampleSet with selected non-numeric attributes converted to numeric types is output of this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

create view (boolean) It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply nominal to numeric conversion. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Pa-
6.2. Type Conversion

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.
6. Data Transformation

**regular expression** *(string)* The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file\_path.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file\_path.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single\_value'

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '*' > 6' will
6.2. Type Conversion

keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' > ', '=' and ' < ' e.g. ' < 5' will not work, so use ' < 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles. The special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

coding type (selection) This parameter indicates the coding which will be used for transforming nominal attributes to numerical attributes. There are three available options i.e. unique integers, dummy coding, effect coding. You can easily understand these options by studying the attached Example Process.

- unique integers If this option is selected, the values of nominal attributes can be seen as equally ranked, therefore the nominal attribute will simply be turned into a real valued attribute, the old values result in equidistant real values.

- dummy coding If this option is selected, for all values of the nominal attribute, excluding the comparison group, a new attribute is created. The comparison group can be defined using the comparison groups parameter. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the comparison group, all new attributes are set to 0. Note that the comparison group is an optional parameter with 'dummy coding'. If no comparison group is defined, in every example the new attribute which corresponds to
6. Data Transformation

the actual nominal value of that example gets value 1 and all other new attributes get value 0. In this case, there will be no example where all new attributes get value 0. This can be easily understood by studying the attached example process.

- **effect_coding** If this option is selected; for all values of the nominal attribute, excluding the *comparison group*, a new attribute is created. The *comparison group* can be defined using the *comparison groups* parameter. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the *comparison group*, all new attributes are set to -1. Note that the *comparison group* is a mandatory parameter with 'effect coding'. This can be easily understood by studying the attached example process.

**use comparison groups** *(boolean)* This parameter is available only when the *coding type* parameter is set to *dummy coding*. If checked, for each selected attribute in the ExampleSet a value has to be specified in the *comparison group* parameter. A separate new column for this value will not appear in the final result set. If not checked, all values of the selected attributes will result in an indicator attribute in the resultant ExampleSet.

**comparison groups** This parameter defines the *comparison group* for each selected non-numeric attribute. Only one *comparison group* can be specified for one attribute. When the *coding type* parameter is set to 'effect coding', it is compulsory to define a *comparison group* for all selected attributes.

**use underscore in name** *(boolean)* This parameter indicates if underscores should be used in the names of new attributes instead of empty spaces and '='. Although the resulting names are harder to read for humans but it might be more appropriate to use these if the data is to be written into a database system.

**Tutorial Processes**

**Nominal to Numeric conversion through different coding types**
This Example Process mostly focuses on the *coding type* and *comparison groups* parameters. All remaining parameters are mostly for selecting the attributes. The Select Attributes operator also has many similar parameters for the selection of attributes. You can study its Example Process if you want an understanding of these parameters.

The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it. The 'Outlook' and 'Wind' attributes are selected for this operator for changing them to numeric attributes. Initially, the *coding type* parameter is set to 'unique integers'. Thus, the nominal attributes will simply be turned into real valued attributes; the old values will result in equidistant real values. As you can see in the Results Workspace, all occurrences of value 'sunny' for the 'Outlook' attribute are replaced by 2. Similarly, 'overcast' and 'rain' are replaced by 1 and 0 respectively. In the same way, all occurrences of 'false' value in the 'Wind' attribute are replaced by 1 and occurrences of 'true' are replaced by 0.

Now, change the *coding type* parameter to 'dummy coding' and run the process again. As *dummy coding* is selected, for all values of the nominal attribute a new attribute is created. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. As you can see in the Results Workspace, 'Wind=true' and 'Wind=false' attributes are created in place of the 'Wind' attribute. In all examples where the 'Wind' attribute had value 'true', the 'Wind=true' attribute gets 1 and 'Wind=false' attribute gets 0. Similarly, all examples where the 'Wind' attribute had value 'false', the 'Wind=true' attribute gets value 0 and 'Wind=false' attribute gets value 1. The same principle applies to the 'Outlook' attribute.

Now, keep the *coding type* parameter as 'dummy coding' and also set the *use comparison groups* parameter to true. Run the process again. You can see in the *comparison groups* parameter that 'sunny' and 'true' are defined as *comparison groups* for the 'Outlook' and 'Wind' attributes respectively. As *dummy coding* is used and the *comparison groups* are also used thus for all values of the nominal
6. Data Transformation

attribute, excluding the *comparison group*, a new attribute is created. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the *comparison group*, all new attributes are set to 0. This is why 'Outlook=rain' and 'Outlook=overcast' attributes are created but 'Outlook=sunny' attribute is not created this time. In examples where the 'Outlook' attribute had value 'sunny', all new Outlook attributes get value 0. You can see this in the Results Workspace. The same rule is applied on the 'Wind' attribute.

Now, change the *coding type* parameter to 'effect coding' and run the process again. You can see in the *comparison groups* parameter that 'sunny' and 'true' are defined as *comparison groups* for the 'Outlook' and 'Wind' attributes respectively. As *effect coding* is selected thus for all values of the nominal attribute, excluding the *comparison group*, a new attribute is created. In every example, the new attribute which corresponds to the actual nominal value of that example gets value 1 and all other new attributes get value 0. If the value of the nominal attribute of this example corresponds to the *comparison group*, all new attributes are set to -1. This is why 'Outlook=rain' and 'Outlook = overcast' attributes are created but an 'Outlook=sunny' attribute is not created this time. In examples where the 'Outlook' attribute had value 'sunny', all new Outlook attributes get value -1. You can see this in the Results Workspace. The same rule is applied on the 'Wind' attribute.
6.2. Type Conversion

Nominal to Date

This operator converts the selected nominal attribute into the selected date time type. The nominal values are transformed into date and/or time values. This conversion is done with respect to the specified date format string.

Description

The Nominal to Date operator converts the selected nominal attribute of the input ExampleSet into the selected date and/or time type. The attribute is selected by the attribute name parameter. The type of the resultant date and/or time attribute is specified by the date type parameter. The nominal values are transformed into date and/or time values. This conversion is done with respect to the specified date format string that is specified by the date format parameter. The old nominal attribute will be removed and replaced by a new date and/or time attribute if the keep old attribute parameter is not set to true.

Date and Time Patterns

This section explains the date and time patterns. Understanding of date and time patterns is necessary specially for specifying the date format string in the date format parameter. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters that represent the components of a date or time. Text can be quoted using single quotes ('') to avoid interpretation as date or time components. All other characters are not interpreted as date or time components; they are simply matched against the input string during parsing.

Here is a brief description of the defined pattern letters. The format types like 'Text', 'Number', 'Year', 'Month' etc are described in detail after this section.
6. Data Transformation

- G: This pattern letter is the era designator. For example: AD, BC etc. This pattern letter follows the rules of 'Text' format type.

- y: This pattern letter represents year. yy represents year in two digits e.g. 96 and yyyy represents year in four digits e.g. 1996. This pattern letter follows the rules of the 'Year' format type.

- M: This pattern letter represents the month of the year. This pattern letter follows the rules of the 'Month' format type. Month can be represented as; for example; March, Mar or 03 etc.

- w: This pattern letter represents the week number of the year. This pattern letter follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the last week of December can be represented as 52.

- W: This pattern letter represents the week number of the month. This pattern letter follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the forth week of December can be represented as 04.

- D: This pattern letter represents the day number of the year. This pattern letter follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and last day of December can be represented as 365 (or 366 in case of a leap year).

- d: This pattern letter represents the day number of the month. This pattern letter follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and the last day of December can be represented as 31.

- F: This pattern letter represents the day number of the week. This pattern letter follows the rules of the 'Number' format type.

- E: This pattern letter represents the name of the day of the week. This pattern letter follows the rules of the 'Text' format type. For example, Tuesday or Tue etc.
6.2. Type Conversion

- **a**: This pattern letter represents the AM/PM portion of the 12-hour clock. This pattern letter follows the rules of the 'Text' format type.

- **H**: This pattern letter represents the hour of the day (from 0 to 23). This pattern letter follows the rules of the 'Number' format type.

- **k**: This pattern letter represents the hour of the day (from 1 to 24). This pattern letter follows the rules of the 'Number' format type.

- **K**: This pattern letter represents the hour of the day for 12-hour clock (from 0 to 11). This pattern letter follows the rules of the 'Number' format type.

- **h**: This pattern letter represents the hour of the day for 12-hour clock (from 1 to 12). This pattern letter follows the rules of the 'Number' format type.

- **m**: This pattern letter represents the minutes of the hour (from 0 to 59). This pattern letter follows the rules of the 'Number' format type.

- **s**: This pattern letter represents the seconds of the minute (from 0 to 59). This pattern letter follows the rules of the 'Number' format type.

- **S**: This pattern letter represents the milliseconds of the second (from 0 to 999). This pattern letter follows the rules of the 'Number' format type.

- **z**: This pattern letter represents the time zone. This pattern letter follows the rules of the 'General Time Zone' format type. Examples include Pacific Standard Time, PST, GMT-08:00 etc.

- **Z**: This pattern letter represents the time zone. This pattern letter follows the rules of the 'RFC 822 Time Zone' format type. Examples include -08:00 etc.

Please note that all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved. Pattern letters are usually repeated, as their number determines the exact presentation. Here is the explanation of various format types:

- **Text**: For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used (if available). For
6. Data Transformation

parsing, both forms are acceptable independent of the number of pattern letters.

- **Number**: For formatting, the number of pattern letters is the minimum number of digits. The numbers that are shorter than this minimum number of digits are zero-padded to this amount. For example if the minimum number of digits is 3 then the number 5 will be changed to 005. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.

- **Year**: If the underlying calendar is the Gregorian calendar, the following rules are applied:
  
  - For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a 'Number' format type.
  
  - For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern 'MM/dd/yyyy', the string '01/11/12' parses to 'Jan 11, 12 A.D'.

  - For parsing with the abbreviated year pattern ('y' or 'yy'), this operator must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the operator is created. For example, using a pattern of 'MM/dd/yy' and the operator created on Jan 1, 1997, the string '01/11/12' would be interpreted as Jan 11, 2012 while the string '05/04/64' would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that is not all digits (for example, '-1'), is interpreted literally. So '01/02/3' or '01/02/003' are parsed, using the same pattern, as 'Jan 2, 3 AD'. Likewise, '01/02/-3' is parsed as 'Jan 2, 4 BC'.

  Otherwise, if the underlying calendar is not the Gregorian calendar, calen-
6.2. Type Conversion

dar system specific forms are applied. If the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar short or abbreviated form is used.

- **Month:** If the number of pattern letters is 3 or more, the month is interpreted as 'Text' format type otherwise, it is interpreted as a 'Number' format type.

- **General time zone:** Time zones are interpreted as 'Text' format type if they have names. It is possible to define time zones by representing a GMT offset value. RFC 822 time zones are also acceptable.

- **RFC 822 time zone:** For formatting, the RFC 822 4-digit time zone format is used. General time zones are also acceptable.

This operator also supports localized date and time pattern strings by defining the *locale* parameter. In these strings, the pattern letters described above may be replaced with other, locale-dependent pattern letters.

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

- 'yyyy.MM.dd G'at' HH:mm:ss z': 2001.07.04 AD at 12:08:56 PDT
- 'EEE, MMM d, yy': Wed, Jul 4, '01
- 'h:mm a': 12:08 PM
- 'hh 'oclock' a, zzzz': 12 oclock PM, Pacific Daylight Time
- 'K:mm a, z': 0:08 PM, PDT
- 'yyyy.MMMMMM.dd GGG hh:mm aaa': 2001.July.04 AD 12:08 PM
- 'EEE, d MMM yyyy HH:mm:ss Z': Wed, 4 Jul 2001 12:08:56 -0700
- 'yyMMddHHmmsZ': 010704120856-0700
- 'yyyy-MM-dd'T'HH:mm:ss.SSSZ': 2001-07-04T12:08:56.235-0700
6. Data Transformation

Input Ports

**example set (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

**example set (exa)** The selected nominal attribute is converted to date type and the resultant ExampleSet is delivered through this port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**attribute name (string)** The name of the nominal attribute that is to be converted to date type is specified here.

**date type (selection)** This parameter specifies the type of the resultant attribute.

- **date** If the *date type* parameter is set to 'date', the resultant attribute will be of date type. The time portion (if any) of the nominal attribute will be ignored.

- **time** If the *date type* parameter is set to 'time', the resultant attribute will be of time type. The date portion (if any) of the nominal attribute will be ignored.

- **date_time** If the *date type* parameter is set to 'date_time', the resultant
6.2. Type Conversion

attribute will be of date_time type.

date format This is the most important parameter of this operator. It specifies the date time format of the selected nominal attribute. Date format strings are discussed in detail in the description of this operator.

time zone (selection) This is an expert parameter. A long list of time zones is provided; users can select any of them.

locale (selection) This is an expert parameter. A long list of locales is provided; users can select any of them.

keep old attribute (boolean) This parameter indicates if the original nominal attribute should be kept or if it should be discarded.

Tutorial Processes

Introduction to the Nominal to Date operator

This Example Process starts with a subprocess. The subprocess delivers an ExampleSet with just a single attribute. The name of the attribute is 'deadline_date'. The type of the attribute is nominal. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, all the examples of this attribute have both date and time information. The Nominal to Date operator is applied on this ExampleSet to change the type of the 'deadline_date' attribute from nominal to date type. Have a look at the parameters of the Nominal to Date operator. The attribute name parameter is set to 'deadline_date'. The date type parameter is set to 'date'. Thus the 'deadline_date' attribute will be converted from nominal to date type (not date_time) therefore the time portion of the value will not be available in the resultant attribute. The date format parameter is set to 'EEEE, MMMM d, yyyy h:m:s a z', here is an explanation of this date format string:

• 'E' is the pattern letter used for the representation of the name of the day of the week. As explained in the description, if the number of pattern letters is 4 or more, the full form is used. Thus 'EEEE' is used for representing
6. Data Transformation

the day of the week in full form e.g. Monday, Tuesday etc.

- 'M' is the pattern letter used for the representation of the name of the month of the year. As explained in the description, if the number of pattern letters is 4 or more, the full form is used. Thus 'MMMM' is used for representing the month of the year in full form e.g. January, February etc.

- 'y' is the pattern letter used for the representation of the year portion of the date. 'yyyy' represents year of date in four digits like 2011, 2012 etc.

- 'h' is the pattern letter used for the representation of the hour portion of the time. 'h' can represent multiple digit hours as well e.g. 10, 11 etc. The difference between 'hh' and 'h' is that 'hh' represents single digit hours by appending a 0 in start e.g. 01, 02 and so on. But 'h' represents single digits without any modifications e.g. 1, 2 and so on.

- 'm' is the pattern letter used for the representation of the minute portion of the time. 'm' can represent multiple digit minutes as well e.g. 51, 52 etc. The difference between 'mm' and 'm' is that 'mm' represents single digit minutes by appending a 0 in start e.g. 01, 02 and so on. But 'm' represents single digits without any modifications e.g. 1, 2 and so on.

- 's' is the pattern letter used for the representation of the second portion of the time. 's' can represent multiple digit seconds as well e.g. 40, 41 etc. The difference between 'ss' and 's' is that 'ss' represents single digit seconds by appending a 0 in start e.g. 01, 02 and so on. But 's' represents single digits without any modifications e.g. 1, 2 and so on.

- 'a' is the pattern letter used for the representation of the 'AM/PM' portion of the 12-hour date and time.

- 'z' is the pattern letter used for the representation of the time zone.

Please note that this date format string represents the date format of the nominal values of the selected nominal attribute of the input ExampleSet. The date format string helps RapidMiner to understand which portions of the nominal value represent which component of the date or time e.g. year, month etc.
6.2. Type Conversion

Text to Nominal

This operator changes the type of selected text attributes to nominal. It also maps all values of these attributes to corresponding nominal values.

Description

The Text to Nominal operator converts all text attributes to nominal attributes. Each text value is simply used as a nominal value of the new attribute. If the value is missing in the text attribute, the new value will also be missing.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The ExampleSet should have at least one text attribute because if there is no such attribute, the use of this operator does not make sense.
6. Data Transformation

Output Ports

example set output (exa) The selected text attributes are converted to nominal and the resultant ExampleSet is output of this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply text to nominal conversion. It has the following options:

- all This option simply selects all the attributes of the ExampleSet. This is the default option.

- single This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- subset This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if metadata is not known. When this option is selected another parameter becomes visible in the Parameter View.

- regular_expression This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- value_type This option allows selection of all the attributes of a particular
6.2. Type Conversion

It should be noted that types are hierarchical. For example, `real` and `integer` types both belong to the `numeric` type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (`value type, use value type exception`) become visible in the Parameter View.

- **block_type** This option is similar in working to the `value type` option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (`block type, use block type exception`) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (`numeric condition`) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of `attribute` parameter if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** *(string)* The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the `edit and preview regular expression` menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** *(boolean)* If enabled, an exception to the first regular
6. Data Transformation

expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. block type parameter's value.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '>', 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '>(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes sim-
6.2. Type Conversion

ply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

\textbf{invert selection (boolean)} If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

\section*{Tutorial Processes}

\subsection*{Introduction to the Text to Nominal operator}

This Example Process starts with the Subprocess operator which provides an ExampleSet. A \textit{breakpoint} is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has three text attributes i.e. 'att1', 'att2' and 'att3'. The Text to Nominal operator is applied on this data set. The \textit{attribute filter type} parameter is set to 'single' and the \textit{attribute} parameter is set to 'att1'. Thus this operator converts the type of the 'att1' attribute from text to nominal. You can verify this by seeing the results in the Meta Data View in the Results Workspace.
Date to Numerical

This operator changes the type of the selected date attribute to a numeric type. It also maps all values of this attribute to numeric values. You can specify exactly which component of date or time should be extracted. You can also specify relative to which date or time component information should be extracted.

Description

The Date to Numerical operator provides a lot of flexibility when it comes to selecting a component of date or time. The following components can be selected: millisecond, second, minute, hour, day, week, month, quarter, half year, and year. The most important thing is that these components can be selected relative to other components. For example it is possible to extract the day relative to the week, relative to the month or relative to the year. Suppose the date is 15/Feb/2012. Then the day relative to the month would be 15 because it is the 15th day of the month. And the day relative to the year would be 46 because this is the 46th day of the year. All date and time components can be extracted relative to the most common parent components e.g. month can be calculated relative to the quarter or the year. Similarly second can be calculated relative to the minute, the hour or the day. All date and time components can be extracted relative to the Epoch where Epoch is defined as the date: '01-01-1970 00:00'. If the date attribute has no time information then all calculations on time components will result to 0. All these things can be understood easily by studying the attached Example Process.

Input Ports

example set (exa) This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of
6.2. Type Conversion

other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in the meta data. The Generate Data operator provides meta data along-with the data. The ExampleSet should have at least one date attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

example set (exa) The ExampleSet with selected date attribute converted to numeric type is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute name (string) This parameter specifies the attribute of the input ExampleSet that should be converted from date to numerical form.
time unit (selection) This parameter specifies the unit in which the time is measured. In other words, this parameter specifies the component of the date that should be extracted. The following components can be extracted: millisecond, second, minute, hour, day, week, month, quarter, half year, and year.
millisecond relative to (selection) This parameter is only available when the time unit parameter is set to 'millisecond'. This parameter specifies the component relative to which the milliseconds should be extracted. The following options are available: second, epoch.
second relative to (selection) This parameter is only available when the time unit parameter is set to 'second'. This parameter specifies the component relative to which the seconds should be extracted. The following options are available: minute, hour, day, epoch.
minute relative to (selection) This parameter is only available when the time unit parameter is set to 'minute'. This parameter specifies the component relative
6. Data Transformation

to which the minutes should be extracted. The following options are available: hour, day, epoch.

**hour relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'hour'. This parameter specifies the component relative to which the hours should be extracted. The following options are available: day, epoch.

**day relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'day'. This parameter specifies the component relative to which the days should be extracted. The following options are available: week, month, year, epoch.

**week relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'week'. This parameter specifies the component relative to which the weeks should be extracted. The following options are available: month, year, epoch.

**month relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'month'. This parameter specifies the component relative to which the months should be extracted. The following options are available: quarter, year, epoch.

**quarter relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'quarter'. This parameter specifies the component relative to which the quarters should be extracted. The following options are available: year, epoch.

**half year relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'half year'. This parameter specifies the component relative to which the half years should be extracted. The following options are available: year, epoch.

**year relative to** *(selection)* This parameter is only available when the *time unit* parameter is set to 'year'. This parameter specifies the component relative to which the years should be extracted. The following options are available: epoch, era.

**keep old attribute** *(selection)* This is an expert parameter. This parameter indicates if the original date attribute of the input ExampleSet should be kept. This parameter is set to false by default thus the original date attribute is removed from the input ExampleSet.
Tutorial Processes

Introduction to the Date to Numerical operator

The Generate Data by User Specification operator is used in this Example Process to create a date type attribute. The attribute is named 'Date' and it is defined by the expression 'date.parse("04/21/2012")'. Thus an attribute named 'Date' is created with just a single example. The value of the date is 21/April/2012. Please note that no information about time is given. The Date to Numerical operator is applied on this ExampleSet. The 'Date' attribute is selected in the attribute name parameter.

- If the time unit parameter is set to 'year' and the year relative to parameter is set to 'era' then the result is 2012. This is so because this is the 2012th year relative to the era.

- If the time unit parameter is set to 'year' and the year relative to parameter is set to 'epoch' then the result is 42. This is so because the year of epoch date is 1970 and difference between 2012 and 1970 is 42.

- If the time unit parameter is set to 'half year' and the half year relative to parameter is set to 'year' then the result is 1. This is so because April is in the first half of the year.

- If the time unit parameter is set to 'quarter' and the quarter relative to parameter is set to 'year' then the result is 2. This is so because April is the 4th month of the year and it comes in the second quarter of the year.

- If the time unit parameter is set to 'month' and the month relative to parameter is set to 'year' then the result is 4. This is so because April is the fourth month of the year.

- If the time unit parameter is set to 'month' and the month relative to parameter is set to 'quarter' then the result is 1. This is so because April is the first month of the second quarter of the year.
6. Data Transformation

- If the time unit parameter is set to 'week' and the week relative to parameter is set to 'year' then the result is 16. This is so because 21st April comes in the 16th week of the year.

- If the time unit parameter is set to 'week' and the week relative to parameter is set to 'month' then the result is 3. This is so because 21st day of month comes in the 3rd week of the month.

- If the time unit parameter is set to 'day' and the day relative to parameter is set to 'year' then the result is 112. This is so because 21st April is the 112th day of the year.

- If the time unit parameter is set to 'day' and the day relative to parameter is set to 'month' then the result is 21. This is so because 21st April is the 21st day of the month.

- If the time unit parameter is set to 'day' and the day relative to parameter is set to 'week' then the result is 7. This is so because 21st April 2012 is on Saturday. Saturday is the seventh day of the week. Sunday is the first day of the week.

- If the time unit parameter is set to 'hour' and the hour relative to parameter is set to 'day' then the result is 0. This is so because no time information was provided for this date attribute and all time information was assumed to be 00 by default.
Date to Nominal

This operator parses the date values of the specified date attribute with respect to the given date format string and transforms the values into nominal values.

Description

The Date to Nominal operator transforms the specified date attribute and writes a new nominal attribute in a user specified format. This conversion is done with respect to the specified date format string that is specified by the date format parameter. This operator might be useful for time base OLAP to change the granularity of the time stamps from day to week or month. The date attribute is selected by the attribute name parameter. The old date attribute will be removed and replaced by a new nominal attribute if the keep old attribute parameter is not set to true. The understanding of Date and Time patterns is very important for using this operator properly.

Date and Time Patterns

This section explains the date and time patterns. Understanding of date and time patterns is necessary especially for specifying the date format string in the date format parameter. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters that represent the components of a date or time. Text can be quoted using single quotes (’) to avoid interpretation as date or time components. All other characters are not interpreted as date or time components; they are simply matched against the input string during parsing.

Here is a brief description of the defined pattern letters. The format types like 'Text', 'Number', 'Year', 'Month' etc are described in detail after this section.
6. Data Transformation

- G: This pattern letter is the era designator. For example: AD, BC etc. It follows the rules of 'Text' format type.

- y: This pattern letter represents year. yy represents year in two digits e.g. 96 and yyyy represents year in four digits e.g. 1996. This pattern letter follows the rules of the 'Year' format type.

- M: This pattern letter represents the month of the year. It follows the rules of the 'Month' format type. Month can be represented as; for example; March, Mar or 03 etc.

- w: This pattern letter represents the week number of the year. It follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the last week of December can be represented as 52.

- W: This pattern letter represents the week number of the month. It follows the rules of the 'Number' format type. For example, the first week of January can be represented as 01 and the forth week of December can be represented as 04.

- D: This pattern letter represents the day number of the year. It follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and last day of December can be represented as 365 (or 366 in case of a leap year).

- d: This pattern letter represents the day number of the month. It follows the rules of the 'Number' format type. For example, the first day of January can be represented as 01 and the last day of December can be represented as 31.

- F: This pattern letter represents the day number of the week. It follows the rules of the 'Number' format type.

- E: This pattern letter represents the name of the day of the week. It follows the rules of the 'Text' format type. For example, Tuesday or Tue etc.

- a: This pattern letter represents the AM/PM portion of the 12-hour clock.
6.2. Type Conversion

It follows the rules of the 'Text' format type.

- H: This pattern letter represents the hour of the day (from 0 to 23). It follows the rules of the 'Number' format type.

- k: This pattern letter represents the hour of the day (from 1 to 24). It follows the rules of the 'Number' format type.

- K: This pattern letter represents the hour of the day for 12-hour clock (from 0 to 11). It follows the rules of the 'Number' format type.

- h: This pattern letter represents the hour of the day for 12-hour clock (from 1 to 12). It follows the rules of the 'Number' format type.

- m: This pattern letter represents the minutes of the hour (from 0 to 59). It follows the rules of the 'Number' format type.

- s: This pattern letter represents the seconds of the minute (from 0 to 59). It follows the rules of the 'Number' format type.

- S: This pattern letter represents the milliseconds of the second (from 0 to 999). It follows the rules of the 'Number' format type.

- z: This pattern letter represents the time zone. It follows the rules of the 'General Time Zone' format type. Examples include Pacific Standard Time, PST, GMT-08:00 etc.

- Z: This pattern letter represents the time zone. It follows the rules of the 'RFC 822 Time Zone' format type. Examples include -08:00 etc.

Please note that all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved. Pattern letters are usually repeated, as their number determines the exact presentation. Here is the explanation of various format types:

- Text: For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used (if available). For parsing, both forms are acceptable independent of the number of pattern letters.
6. Data Transformation

- **Number**: For formatting, the number of pattern letters is the minimum number of digits. The numbers that are shorter than this minimum number of digits are zero-padded to this amount. For example if the minimum number of digits is 3 then the number 5 will be changed to 005. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.

- **Year**: If the underlying calendar is the Gregorian calendar, the following rules are applied:
  
  - For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a 'Number' format type.
  
  - For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern 'MM/dd/yyyy', the string '01/11/12' parses to 'Jan 11, 12 A.D'.

  - For parsing with the abbreviated year pattern ('y' or 'yy'), this operator must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the operator is created. For example, using a pattern of 'MM/dd/yy' and the operator created on Jan 1, 1997, the string '01/11/12' would be interpreted as Jan 11, 2012 while the string '05/04/64' would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that is not all digits (for example, '-1'), is interpreted literally. So '01/02/3' or '01/02/003' are parsed, using the same pattern, as 'Jan 2, 3 AD'. Likewise, '01/02/-3' is parsed as 'Jan 2, 4 BC'.

Otherwise, if the underlying calendar is not the Gregorian calendar, calendar system specific forms are applied. If the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar short
or abbreviated form is used.

- **Month:** If the number of pattern letters is 3 or more, the month is interpreted as 'Text' format type otherwise, it is interpreted as a 'Number' format type.

- **General time zone:** Time zones are interpreted as 'Text' format type if they have names. It is possible to define time zones by representing a GMT offset value. RFC 822 time zones are also acceptable.

- **RFC 822 time zone:** For formatting, the RFC 822 4-digit time zone format is used. General time zones are also acceptable.

This operator also supports localized date and time pattern strings by defining the `locale` parameter. In these strings, the pattern letters described above may be replaced with other, locale-dependent pattern letters.

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

- `'yyyy.MM.dd G 'at' HH:mm:ss z'`: 2001.07.04 AD at 12:08:56 PDT
- `'EEE, MMM d, yy'`: Wed, Jul 4, '01
- `'h:mm a'`: 12:08 PM
- `'hh 'oclock' a, zzzz'`: 12 oclock PM, Pacific Daylight Time
- `'K:mm a, z'`: 0:08 PM, PDT
- `'yyyy.MMMMM.dd GGG hh:mm aaa'`: 2001.July.04 AD 12:08 PM
- `'EEE, d MMM yyyy HH:mm:ss Z'`: Wed, 4 Jul 2001 12:08:56 -0700
- `'yyMMddHHmmsZ'`: 010704120856-0700
6. Data Transformation

Input Ports

**example set input** (*exa*) This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one date attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

**example set output** (*exa*) The selected date attribute is converted to a nominal attribute according to the specified date format string and the resultant ExampleSet is delivered through this port.

**original** (*ori*) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**attribute name** (*string*) The name of the date attribute is specified here. The attribute name can be selected from the drop down box of the **attribute name** parameter if the meta data is known.

**date format** This is the most important parameter of this operator. It specifies the date time format of the desired nominal attribute. This date format string specifies what portion of the date attribute should be stored in the nominal attribute. Date format strings are discussed in detail in the description of this operator.

**locale** (*selection*) This is an expert parameter. A long list of locales is provided; users can select any of them.

**keep old attribute** (*boolean*) This parameter indicates if the original date attribute should be kept or it should be discarded.
6.2. Type Conversion

Tutorial Processes

Introduction to the Date to Nominal operator

This Example Process starts with a Subprocess operator. The subprocess delivers an ExampleSet with just a single attribute. The name of the attribute is 'deadline_date'. The type of the attribute is date. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, all the examples of this attribute have both date and time information. The Date to Nominal operator is applied on this ExampleSet to change the type of the 'deadline_date' attribute from date to nominal type. Have a look at the parameters of the Date to Nominal operator. The attribute name parameter is set to 'deadline_date'. The date format parameter is set to 'EEEE', here is an explanation of this date format string:

'E' is the pattern letter used for the representation of the name of the day of the week. As explained in the description, if the number of pattern letters is 4 or more, the full form is used. Thus 'EEEE' is used for representing the day of the week in full form e.g. Monday, Tuesday etc. Thus the date attribute is changed to a nominal attribute which has only name of days as possible values. Please note that this date format string is used for specifying the format of the nominal values of the new nominal attribute of the input ExampleSet.
Parse Numbers

This operator changes the type of selected nominal attributes to a numeric type. It also maps all values of these attributes to numeric values by parsing the numbers if possible.

Description

The Parse Numbers operator is used for changing the type of nominal attributes to a numeric type. This operator not only changes the type of selected attributes but it also maps all values of these attributes to numeric values by parsing the numbers if possible. In contrast to the Nominal to Numerical operator, this operator directly parses numbers from the afore wrongly encoded as nominal values. The Nominal to Numeric operator is used when the values are actually nominal but you want to change them to numerical values. On the other hand the Parse Numbers operator is used when the values should actually be numerical but they are wrongly stored as nominal values. Please note that this operator will first check the stored nominal mappings for all attributes. If (old) mappings are still stored which actually are nominal (without the corresponding data being part of the ExampleSet), the attribute will not be converted. Please use the Guess Types operator in these cases.

Differentiation

Nominal to Numerical The Nominal to Numerical operator provides various coding types to convert nominal attributes to numerical attributes. On the other hand the Parse Numbers operator is used when the values should actually be numerical but they are wrongly stored as nominal values. See page 311 for details.
6.2. Type Conversion

Input Ports

example set input \((\text{exa})\) This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

example set output \((\text{exa})\) The ExampleSet with selected nominal attributes converted to numeric types is output of this port.
original \((\text{ori})\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type \((\text{selection})\) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- all This option simply selects all the attributes of the ExampleSet. This is the default option.
- single This option allows selection of a single attribute. When this option is selected another parameter \((\text{attribute})\) becomes visible in the Parameter View.
- subset This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not
6. Data Transformation

known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

*attribute* (*string*) The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

*attributes* (*string*) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will
6.2. Type Conversion

remain unchanged.

**regular expression (string)** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition (string)** The numeric condition for testing examples of nu-
6. Data Transformation

numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' > ', '=' and '< ' e.g. '< 5' will not work, so use '< 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

decimal character (char) This character is used as the decimal character.

grouped digits (boolean) This option decides whether grouped digits should be parsed or not. If this option is set to true, grouping character parameter should be specified.

grouping character (char) This character is used as the grouping character. If this character is found between numbers, the numbers are combined and this character is ignored. For example if "22-14"is present in the nominal attribute and is set as grouping character, then "2214"will be stored in the corresponding numerical attribute.

Related Documents

Nominal to Numerical (311)
6.2. Type Conversion

Tutorial Processes

Nominal to Numeric conversion by the Parse Numbers operator

This Example Process starts with a Subprocess operator. The Subprocess operator provides an ExampleSet as its output. The ExampleSet has some nominal attributes. But these nominal attributes actually wrongly store numerical values as nominal values. A breakpoint is inserted here so that you can have a look at the ExampleSet. The type of these attributes should be numerical. To convert these nominal attributes to numerical attributes the Parse Numbers operator is applied. All parameters are used with default values. The resultant ExampleSet can be seen in the Results Workspace. You can see that the type of all attributes has been changed from nominal to numerical type.

Format Numbers

This operator reformats the selected numerical attributes according to the specified format and changes the attributes to nominal.
6. Data Transformation

Description

This operator parses numerical values and formats them into the specified format. The format is specified by the format type parameter. It supports different kinds of number formats including integers (e.g. 123), fixed-point numbers (e.g. 123.4), scientific notation (e.g. 1.23E4), percentages (e.g. 12%), and currency amounts (e.g. $123). Please note that this operator only works on numerical attributes and the result will be in any case a nominal attribute even if the resulting format is a number which can be parsed again.

If the format type parameter is set to 'pattern', the pattern parameter is used for defining the format. If two different formats for positive and negative numbers should be used, those formats can be defined by separating them by a semi-colon ';'. The pattern parameter provides a lot of flexibility for defining the pattern. Important structures that can be used for defining a pattern are listed below. The structures in brackets are optional.

- pattern := subpattern{:subpattern}
- subpattern := {prefix}integer{.fraction}{suffix}
- prefix := any character combination including whitespace
- suffix := any character combination including whitespace
- integer := #* 0* 0
- fraction := 0* #*

0* and #* stand for multiple 0 or # respectively. 0 and # perform similar functions but 0 ensures that length of all numbers is same i.e. if a digit is missing it is replaced by 0. For example 54 will be formatted to 0054 with pattern '0000' and it will be formatted to 54 with pattern '#####'.

The following placeholders can be used within the pattern parameter:

- . placeholder for decimal separator.
6.2. Type Conversion

- , placeholder for grouping separator.
- E separates mantissa and exponent for exponential formats.
- - default negative prefix.
- % multiply by 100 and show as percentage.
- ' used to quote special characters in a prefix or suffix.

The locale parameter is ignored when the format type parameter is set to 'pattern'. In other cases it plays its role e.g. if the format type parameter is set to 'currency' then the locale parameter specifies the notation for that currency (i.e. dollar, euro etc).

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one numerical attribute because if there is no such attribute, the use of this operator does not make sense.

Output Ports

example set output (exa) The selected numerical attributes are reformatted and converted to nominal and the resultant ExampleSet is delivered through this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.
6.2. Type Conversion

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression (string)** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.
6. Data Transformation

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (**except value type**) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. **value type** parameter’s value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (**except block type**) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=!' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**format type (selection)** This parameter specifies the type of formatting to per-
6.2. Type Conversion

form on the selected numerical attributes.

**pattern (string)** This parameter is only available when the *format type* parameter is set to 'pattern'. This parameter specifies the pattern for formatting the numbers. Various structures and replacement patterns for this parameter have been discussed in the description of this operator.

**locale (selection)** This is an expert parameter. A long list of locales is provided; users can select any of them.

**use grouping (boolean)** This parameter indicates if a grouping character should be used for larger numbers.

---

**Tutorial Processes**

**Changing numeric values to currency format**

This process starts with the Generate Data operator which generates a random ExampleSet with a numeric attribute named 'att1'. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. The Format Numbers operator is applied on it to change the format of this attribute to a currency format. The *attribute filter type* parameter is set to 'single' and the *attribute* parameter is set to 'att1' to select the required attribute. The *format type* parameter is set to 'currency'. Run the process and switch to the Results Workspace. You can see that the 'att1' attribute has been changed from numeric to nominal type and its values have a '$' sign in the beginning because they have been converted to a currency format. The *locale* parameter specifies the required currency. In this process the *locale* parameter was set to 'English (United States)' therefore the numeric values were converted to the currency of United States (i.e. dollar).
6. Data Transformation

Guess Types

This operator (re-)guesses the value types of all attributes of the input ExampleSet and changes them accordingly.

Description

The Guess Types operator can be used to (re-)guess the value types of the attributes of the input ExampleSet. This might be useful after some preprocessing transformations and purification of some of the attributes. This operator can be useful especially if nominal attributes can be handled as numerical attributes after some preprocessing. It is not necessary to (re-)guess the type of all the attributes with this operator. You can select the attributes whose type is to be (re-)guessed. Please study the attached Example Process for more information. Please note that this operator has no impact on the values of the ExampleSet.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.
6.2. Type Conversion

Output Ports

**example set output** *(exa)* The type of the selected attributes of the input ExampleSet is (re-)guessed and the resultant ExampleSet is delivered through this output port.

**original** *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**attribute filter type** *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter *(attribute)* becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters *(regular expression, use except expression)* become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular
6. Data Transformation

type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

**regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.
6.2. Type Conversion

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '!=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster,
weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

first character index (integer) This parameter specifies the index of the first character of the substring which should be kept. Please note that the counting starts with 1.

last character index (integer) This parameter specifies the index of the last character of the substring which should be kept. Please note that the counting starts with 1.

decimal point character (char) The character specified by this parameter is used as the decimal character.

number grouping character (char) The character specified by this parameter is used as the grouping character. This character is used for grouping the numbers. If this character is found between numbers, the numbers are combined and this character is ignored. For example if "22-14" is present in the ExampleSet and "is set as the number grouping character, then the number will be considered to be "2214".

Tutorial Processes

Guessing the type of an attribute after preprocessing

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. Please note the 'id' attribute. The 'id' attribute is of nominal type and it has the values of the for-
mat 'id_1', 'id_2' and so on. The Cut operator is applied on the ExampleSet to remove the substring 'id.' from the start of the 'id' attribute values. A breakpoint is inserted after the Cut operator. You can see that now the values in the 'id' attribute are of the form '1', '2', '3' and so on but the type of this attribute is still nominal. The Guess Types operator is applied on this ExampleSet. The attribute filter type parameter is set to 'single', the attribute parameter is set to 'id' and the include special attributes parameter is also set to 'true'. Thus the Guess Types operator will re-guess the type of the 'id' attribute. A breakpoint is inserted after the Guess Type operator. You can see that the type of the 'id' attribute has now been changed to integer.

Discretize by Size

This operator converts the selected numerical attributes into nominal attributes by discretizing the numerical attribute into bins of user-specified size. Thus each bin contains a user-defined number of examples.
6. Data Transformation

Description

This operator discretizes the selected numerical attributes to nominal attributes. The *size of bins* parameter is used for specifying the required size of bins. This discretization is performed by binning examples into bins containing the same, user-specified number of examples. Each bin range is named automatically. The naming format of the range can be changed by using the *range name type* parameter. The values falling in the range of a bin are named according to the name of that range.

It should be noted that if the number of examples is not evenly divisible by the requested number of examples per bin, the actual result may slightly differ from the requested bin size. Similarly, if a range of examples cannot be split, because the numerical values are identical within this set, only all or none can be assigned to a bin. This may lead to further deviations from the requested bin size.

This operator is closely related to the Discretize By Frequency operator. There you have to specify the number of bins you need (say $x$) and the operator automatically creates it with an almost equal number of values (i.e. $n/x$ values where $n$ is the total number of values). In the Discretize by Size operator you have to specify the number of values you need in each bin (say $y$) and the operator automatically creates $n/y$ bins with $y$ values.

Differentiation

**Discretize by Binning** The Discretize By Binning operator creates bins so their range is (almost) equal. See page 368 for details.

**Discretize by Frequency** The Discretize By Frequency operator creates bins so the number of unique values in all bins are (almost) equal. See page 375 for details.

**Discretize by Entropy** The discretization is performed by selecting bin boundaries so the entropy is minimized in the induced partitions. See page 389 for details.

**Discretize by User Specification** This operator discretizes the selected nu-
6.2. Type Conversion

merical attributes into user-specified classes. See page 382 for details.

Input Ports

deset input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. Please note that there should be at least one numerical attribute in the input ExampleSet, otherwise the use of this operator does not make sense.

Output Ports

deset output (exa) The selected numerical attributes are converted into nominal attributes by discretization and the resultant ExampleSet is delivered through this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

create view (boolean) It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.
attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required
attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (**attribute**) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (**regular expression**, **use except expression**) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example **real** and **integer** types both belong to the **numeric** type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (**value type**, **use value type exception**) become visible in the Parameter View.

- **block_type** This option is similar in working to the **value type** option. It allows selection of all the attributes of a particular block type. When this option is selected some other parameters (**block type**, **use block type exception**) become visible in the Parameter View.

- **no_missing_values** This option simply selects all attributes of the ExampleSet which don’t contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (**nu-
6.2. Type Conversion

A numeric condition becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

attribute (string) The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

attributes (string) The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

regular expression (string) The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

use except expression (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except regular expression (string) This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

value type (selection) The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binomial, polynominal, file_path.

use value type exception (boolean) If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will be removed from the final output even if they matched the previously mentioned type.
6. Data Transformation

i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

block type (selection) The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

use block type exception (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

except block type (selection) The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

numeric condition (string) The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 | | < 0'. But & & and | | cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) | | (> 10 & & < 12)' are not allowed because they use both & & and | |. Use a blank space after ' > ', '=' and '< '. e.g. '< 5' will not work, so use '< 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

size of bins (integer) This parameter specifies the required size of bins i.e. number of examples contained in a bin.

sorting direction (selection) This parameter indicates if the values should be sorted in increasing or decreasing order.

range name type (selection) This parameter is used for changing the naming format for range. 'long', 'short' and 'interval' formats are available.
6.2. Type Conversion

**automatic number of digits (boolean)** This is an expert parameter. It is only available when the range name type parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits (integer)** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

**Related Documents**

- Discretize by Binning (368)
- Discretize by Frequency (375)
- Discretize by Entropy (389)
- Discretize by User Specification (382)

**Tutorial Processes**

**Discretizing the Temperature attribute of the 'Golf' data set**

The focus of this Example Process is the discretization procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can give a look at the ExampleSet. You can see that the 'Temperature' attribute is a numerical attribute. The Discretize by Size operator is applied on it. The 'Temperature' attribute is selected for discretization. The *size of bins* parameter is set to 5. Run the process and switch to the Results Workspace. You can see that the 'Temperature' attribute has been changed from numerical to nominal form. The values of the 'Temperature' attribute have been divided into three ranges. Each range has an equal number of unique values. You can see that 'range1' and 'range3' have 4 examples while the 'range2' has 6
examples. All bins do not have exactly equal values because 14 examples cannot be grouped by 5 examples per bin. But in 'range2' the 'Temperature' values 72 and 75 occur twice. Thus essentially 4 unique numerical values are present in 'range2'.

6. Data Transformation

Discretize by Binning

This operator discretizes the selected numerical attributes into user-specified number of bins. Bins of equal range are automatically generated, the number of the values in different bins may vary.

Description

This operator discretizes the selected numerical attributes to nominal attributes. The number of bins parameter is used to specify the required number of bins. This discretization is performed by simple binning. The range of numerical values is partitioned into segments of equal size. Each segment represents a bin. Numerical values are assigned to the bin representing the segment covering the numerical value. Each range is named automatically. The naming format for range can be changed using the range name type parameter. Values falling in the range of a bin are named according to the name of that range. This operator also allows
you to apply binning only on a range of values. This can be enabled by using the \textit{define boundaries} parameter. The \textit{min value} and \textit{max value} parameter are used for defining the boundaries of the range. If there are any values that are less than the \textit{min value} parameter, a separate range is created for them. Similarly if there are any values that are greater than the \textit{max value} parameter, a separate range is created for them. Then, the discretization by binning is performed only on the values that are within the specified boundaries.

\section*{Differentiation}

\textbf{Discretize by Frequency} The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. See page 375 for details.

\textbf{Discretize by Size} The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 361 for details.

\textbf{Discretize by Entropy} The discretization is performed by selecting bin boundaries such that the entropy is minimized in the induced partitions. See page 389 for details.

\textbf{Discretize by User Specification} This operator discretizes the selected numerical attributes into user-specified classes. See page 382 for details.

\section*{Input Ports}

\textbf{example set (exa)} This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. the output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with the data. Note that there should be at least one numerical attribute in the input ExampleSet, otherwise the use of this operator does not make sense.
6. Data Transformation

Output Ports

**example set** (*exa*) The selected numerical attributes are converted into nominal attributes by binning and the resultant ExampleSet is delivered through this port.

**original** (*ori*) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model** (*pre*) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

**create view** (*boolean*) It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type** (*selection*) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter
6.2. Type Conversion

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.
6. Data Transformation

**regular expression** *(string)* The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression** *(boolean)* If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter *(except regular expression)* becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression *(regular expression that was specified in the regular expression parameter)*.

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition `' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: `' > 6 & & < 11' or ' <= 5 | | < 0'. But & & and | | cannot be used together in one numeric
condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '> ', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**number of bins** *(integer)* This parameter specifies the number of bins which should be used for each attribute.

**define boundaries:** *(boolean)* The Discretize by Binning operator allows you to apply binning only on a range of values. This can be enabled by using the *define boundaries* parameter. If this is set to true, discretization by binning is performed only on the values that are within the specified boundaries. The lower and upper limit of the boundary is specified by the *min value* and *max value* parameters respectively.

**min value** *(real)* This parameter is only available when the *define boundaries* parameter is set to true. It is used to specify the lower limit value for the binning range.

**max value** *(real)* This parameter is only available when the *define boundaries* parameter is set to true. It is used to specify the upper limit value for the binning range.

**range name type** *(selection)* This parameter is used to change the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits** *(boolean)* This is an expert parameter. It is only available when the *range name type* parameter is set to 'interval'. It indicates if
the number of digits should be automatically determined for the range names. 

**number of digits (integer)** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

**Related Documents**

- Discretize by Frequency (375)
- Discretize by Size (361)
- Discretize by Entropy (389)
- Discretize by User Specification (382)

**Tutorial Processes**

**Discretizing numerical attributes of the 'Golf' data set by Binning**

The focus of this Example Process is the binning procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by Binning operator is applied on it. The 'Temperature' and 'Humidity' attributes are selected for discretization. The **number of bins** parameter is set to 2. The **define boundaries** parameter is set to true. The **min value** and **max value** parameters are set to 70 and 80 respectively. Thus binning will be performed only in the range from 70 to 80. As the **number of bins** parameter is set to 2, the range will be divided into two equal segments. Approximately speaking, the first segment of the range will be from 70 to 75 and the second segment of the range will be from 76 to 80. These are not exact values, but they are good enough for the explanation of this process. There will be a separate range for all those values that are less than the **min value** parameter i.e. less than 70. This range is au-
6.2. Type Conversion

tomatically named 'range1'. The first and second segment of the binning range are named 'range2' and 'range3' respectively. There will be a separate range for all those values that are greater than the \textit{max value} parameter i.e. greater than 80. This range is automatically named 'range4'. Run the process and compare the original data set with the discretized one. You can see that the values less than or equal to 70 in the original data set are named 'range1' in the discretized data set. The values greater than 70 and less than or equal to 75 in the original data set are named 'range2' in the discretized data set. The values greater than 75 and less than or equal to 80 in the original data set are named 'range3' in the discretized data set. The values greater than 80 in the original data set are named 'range4' in the discretized data set.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diagram.png}
\caption{Main Process Diagram}
\end{figure}

\section*{Discretize by Frequency}

This operator converts the selected numerical attributes into nominal attributes by discretizing the numerical attribute into a user-specified number of bins. Bins of equal frequency are automatically generated, the range of different bins may vary.
6. Data Transformation

Description

This operator discretizes the selected numerical attributes to nominal attributes. The number of bins parameter is used to specify the required number of bins. The number of bins can also be specified by using the use sqrt of examples parameter. If the use sqrt of examples parameter is set to true, then the number of bins is calculated as the square root of the number of examples with non-missing values (calculated for every single attribute). This discretization is performed by equal frequency binning i.e. the thresholds of all bins is selected in a way that all bins contain the same number of numerical values. Numerical values are assigned to the bin representing the range segment covering the numerical value. Each range is named automatically. The naming format for the range can be changed using the range name type parameter. Values falling in the range of a bin are named according to the name of that range.

Other discretization operators are also available in RapidMiner. The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. In contrast, the Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal.

Differentiation

Discretize by Binning The Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal. See page 368 for details.
Discretize by Size The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 361 for details.
Discretize by Entropy The discretization is performed by selecting bin boundaries such that the entropy is minimized in the induced partitions. See page 389 for details.
Discretize by User Specification This operator discretizes the selected numerical attributes into user-specified classes. See page 382 for details.
6.2. Type Conversion

Input Ports

**example set** *(exa)* This input port expects an ExampleSet. It is the output of the Retrieve operator in attached Example Process. The output of other operators can also be used as input. Please note that there should be at least one numerical attribute in the input ExampleSet, otherwise use of this operator does not make sense.

Output Ports

**example set** *(exa)* The selected numerical attributes are converted into nominal attributes by discretization (frequency) and the resultant ExampleSet is delivered through this port.

**original** *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model** *(pre)* This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

**create view** *(boolean)* It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type** *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is
6. Data Transformation

- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example, *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example, *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric at-
6.2. Type Conversion

attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*ex-
except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. block type parameter's value.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 && < 11' or ' <= 5 || < 0'. But && and || cannot be used together in one numeric condition. Conditions like '( > 0 && < 2) || ( > 10 && < 12)' are not allowed because they use both && and ||. Use a blank space after ' >', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**use sqrt of examples (boolean)** If set to true, the number of bins is determined by the square root of the number of non-missing values instead of using the number of bins parameter.

**number of bins (integer)** This parameter is available only when the use sqrt of examples parameter is not set to true. This parameter specifies the number of bins which should be used for each attribute.

**range name type (selection)** This parameter is used for changing the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits (boolean)** This is an expert parameter. It is only
available when the range name type parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits** *(integer)* This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

### Related Documents

- Discretize by Binning (368)
- Discretize by Size (361)
- Discretize by Entropy (389)
- Discretize by User Specification (382)

### Tutorial Processes

**Discretizing the Temperature attribute of the 'Golf' data set by Frequency**

The focus of this Example Process is the discretization (by frequency) procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by Frequency operator is applied on it. The 'Temperature' attribute is selected for discretization. The *number of bins* parameter is set to 3. Run the process and switch to the Results Workspace. You can see that the 'Temperature' attribute has been changed from numerical to nominal form. The values of the 'Temperature' attribute have been divided into three ranges. Each range has an equal number of unique values. You can see that 'range1' and 'range3' have 4 examples while the 'range2' has 6 examples. But in 'range2' the 'Temperature' values 72 and 75 occur twice. Thus essentially 4 unique numerical values are present in 'range2'.
6. Data Transformation

Discretize by User Specification

This operator discretizes the selected numerical attributes into user-specified classes. The selected numerical attributes will be changed to either nominal or ordinal attributes.

Description

This operator discretizes the selected numerical attributes to either nominal or ordinal attributes. The numerical values are mapped to the classes according to the thresholds specified by the user in the classes parameter. The user can define the classes by specifying the upper limit of each class. The lower limit of every class is automatically defined as the upper limit of the previous class. The lower limit of the first class is assumed to be negative infinity. 'Infinity' can be used to specify positive infinity as upper limit in the classes parameter. This is usually done in the last class. If a class is named as '?', the numerical values falling in this class will be replaced by unknown values in the resulting attributes.
6.2. Type Conversion

Differentiation

Discretize by Binning The Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal. See page 368 for details.

Discretize by Frequency The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. See page 375 for details.

Discretize by Size The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 361 for details.

Discretize by Entropy The discretization is performed by selecting bin boundaries such that the entropy is minimized in the induced partitions. See page 389 for details.

Input Ports

element set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data. Note that there should be at least one numerical attribute in the input ExampleSet, otherwise use of this operator does not make sense.

Output Ports

element set (exa) The selected numerical attributes are converted into nominal or ordinal attributes according to the user specified classes and the resultant ExampleSet is delivered through this port.

original (ori) ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet
6. Data Transformation

in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model** *(pre)* This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

**Parameters**

**create view** *(boolean)* It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type** *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter *(attribute)* becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters *(regular expression, use except expression)* become visible in the Parameter View.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real*
and integer types both belong to the numeric type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

- **attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the parameter attribute if the meta data is known.

- **attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

- **regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

- **use except expression (boolean)** If enabled, an exception to the first regular
expression can be specified. When this option is selected another parameter *except regular expression* becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter *except value type* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter’s value.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *except block type* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. *block type* parameter’s value.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ’> 6’ will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ’> 6 & & < 11’ or ’< = 5 || < 0’. But & & and || cannot be used together in one numeric condition. Conditions like ’(> 0 & & < 2) || (> 10 & & < 12)’ are not allowed because they use both & & and ||. Use a blank space after ’>’, ’=’ and ’< ’ e.g. ’< 5’ will not work, so use ’< 5’ instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles. Special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special at-
attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**attribute type** The type of the discretized attributes is specified here. They can be either nominal or numerical

- **nominal** If this option is selected the discretized attributes will be of nominal type in the resultant ExampleSet.

- **ordinal** If this option is selected the discretized attributes will be of ordinal type in the resultant ExampleSet.

**classes** This is the most important parameter of this operator. It is used to specify the classes into which the numerical values will be mapped. The names and upper limits of the classes are specified here. The numerical values are mapped to the classes according to the thresholds specified by the user in this parameter. The user can define the classes by specifying the upper limit of each class. The lower limit of every class is automatically defined as the upper limit of the previous class. The lower limit of the first class is assumed to be negative infinity. 'Infinity' can be used to specify positive infinity as upper limit in the classes parameter. This is usually done in the last class. If a class is named as '?', the numerical values falling in this class will be replaced by unknown values in the resulting attributes.
Related Documents

Discretize by Binning (368)
Discretize by Frequency (375)
Discretize by Size (361)
Discretize by Entropy (389)

Tutorial Processes

Discretizing numerical attributes of the Golf data set

The focus of this Example Process is the classes parameter. Almost all parameters other than the classes parameter are for selection of attributes on which discretization is to be performed. For understanding these parameters please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by User Specification operator is applied on it. The 'Temperature' and 'Humidity' attributes are selected for discretization. As you can see in the classes parameter, four classes have been specified. The values from negative infinity to 70 will be mapped to 'low' class. The values above 70 to 80 will be mapped to 'average' class. The values above 80 to 90 will be mapped to 'high' class. The values above 90 will be considered as unknown (missing) values. This can be verified by running the process and viewing the results in the Results Workspace. Note that value of the 'Humidity' attribute was 96 and 95 in Row No. 4 and 8 respectively. In the discretized attributes these values are replaced by unknown values because of the last class defined in the classes parameter.
6.2. Type Conversion

Discretize by Entropy

This operator converts the selected numerical attributes into nominal attributes. The boundaries of the bins are chosen so that the entropy is minimized in the induced partitions.

Description

This operator discretizes the selected numerical attributes to nominal attributes. The discretization is performed by selecting a bin boundary that minimizes the entropy in the induced partitions. Each bin range is named automatically. The naming format of the range can be changed using the range name type parameter. The values falling in the range of a bin are named according to the name of that range.

The discretization is performed by selecting a bin boundary that minimizes the entropy in the induced partitions. The method is then applied recursively for both new partitions until the stopping criterion is reached. For more information please study:

- Multi-interval discretization of continued-values attributes for classification learning (Fayyad,Irani)
6. Data Transformation

- Supervised and Unsupervised Discretization (Dougherty, Kohavi, Sahami).

This operator can automatically remove all attributes with only one range i.e. those attributes which are not actually discretized since the entropy criterion is not fulfilled. This behavior can be controlled by the *remove useless* parameter.

**Differentiation**

**Discretize by Binning** The Discretize By Binning operator creates bins in such a way that the range of all bins is (almost) equal. See page 368 for details.

**Discretize by Frequency** The Discretize By Frequency operator creates bins in such a way that the number of unique values in all bins are (almost) equal. See page 375 for details.

**Discretize by Size** The Discretize By Size operator creates bins in such a way that each bin has user-specified size (i.e. number of examples). See page 361 for details.

**Discretize by User Specification** This operator discretizes the selected numerical attributes into user-specified classes. See page 382 for details.

**Input Ports**

example set input *(exa)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. Please note that there should be at least one numerical attribute in the input ExampleSet, otherwise the use of this operator does not make sense.

**Output Ports**

example set output *(exa)* The selected numerical attributes are converted into nominal attributes by discretization and the resultant ExampleSet is delivered.
6.2. Type Conversion

through this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

create view (boolean) It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- all This option simply selects all the attributes of the ExampleSet. This is the default option.
- single This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.
- subset This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- regular_expression This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.
6. Data Transformation

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example, *real* and *integer* types both belong to the *numeric* type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type*, *use value type exception*) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value type* option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (*block type*, *use block type exception*) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** *(string)* The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good
6.2. Type Conversion

idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '-> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '-> 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed
because they use both \& \& and \|. Use a blank space after '>' , '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**remove useless (boolean)** This parameter indicates if the useless attributes, i.e. attributes containing only a single range, should be removed. If this parameter is set to true then all those attributes that are not actually discretized since the entropy criterion is not fulfilled are removed.

**range name type (selection)** This parameter is used for changing the naming format for range. 'long', 'short' and 'interval' formats are available.

**automatic number of digits (boolean)** This is an expert parameter. It is only available when the **range name type** parameter is set to 'interval'. It indicates if the number of digits should be automatically determined for the range names.

**number of digits (integer)** This is an expert parameter. It is used to specify the minimum number of digits used for the interval names.

**Related Documents**

Discretize by Binning (368)
Discretize by Frequency (375)
Discretize by Size (361)
Discretize by User Specification (382)
Tutorial Processes

Discretizing the 'Sonar' data set by entropy

The focus of this Example Process is the discretization procedure. For understanding the parameters related to attribute selection please study the Example Process of the Select Attributes operator.

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can give a look at the ExampleSet. You can see that this data set has 60 regular attributes (all of real type). The Discretize by Entropy operator is applied on it. The attribute filter type parameter is set to 'all', thus all the numerical attributes will be discretized. The remove useless parameter is set to true, thus attributes with only one range are removed from the ExampleSet. Run the process and switch to the Results Workspace. You can see that the 'Sonar' data set has been reduced to just 22 regular attributes. These numerical attributes have been discretized to nominal attributes.
6. Data Transformation

Generate ID

This operator adds a new attribute with \( id \) role in the input \( \text{ExampleSet} \). Each example in the input \( \text{ExampleSet} \) is tagged with an incremented \( id \). If an attribute with \( id \) role already exists, it is overridden by the new \( id \) attribute.

Description

This operator adds a new attribute with \( id \) role in the input \( \text{ExampleSet} \). It assigns a unique \( id \) to each example. This operator is usually used to uniquely identify each example. Each example in the input \( \text{ExampleSet} \) is tagged with an incremented \( id \). The number from where the \( ids \) start can be controlled by the \( offset \) parameter. Numerical and integer \( ids \) can be assigned. If an attribute with \( id \) role already exists in the input \( \text{ExampleSet} \), it is overridden by the new \( id \) attribute.

Input Ports

\text{example set input (exa)} This input port expects an \( \text{ExampleSet} \). It is output of the Retrieve operator in the attached Example Process.

Output Ports

\text{example set output (exa)} The \( \text{ExampleSet} \) with an \( id \) attribute is output of this port.

\text{original (ori)} The \( \text{ExampleSet} \) that was given as input is passed without changing to the output through this port. This is usually used to reuse the same \( \text{ExampleSet} \) in further operators or to view the \( \text{ExampleSet} \) in the Results Workspace.
6.3. Attribute Set Reduction and Transformation

Parameters

create nominal ids (boolean) This parameter indicates if nominal ids should be created instead of integer ids. By default this parameter is not checked, thus integer ids are created by default. Nominal ids are of the format id_1, id_2, id_3 and so on.

offset (integer) This is an expert parameter. It is used if you want to start id from a number other than 1. This parameter is used to set the offset value. It is 0 by default, thus ids start from 1 by default.

Tutorial Processes

Overriding the id attribute of the 'Iris' data set

The 'Iris' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it. All parameters are used with default values. The 'Iris' data set already has an id attribute. The old id attribute is overridden when the Generate ID operator is applied on it. Run the process and you can see the ExampleSet with the new id attribute. The type of this new attribute is integer. Set the create nominal ids parameter to true and run the process again, you will see that the ids are in nominal form now (i.e. id_1, id_2 and so on). The offset parameter is set to 0 that is why the ids start from 1. Now set the offset parameter to 10 and run the process again. Now you can see that ids start from 11.
Generate Empty Attribute

This operator adds a new attribute of specified name and type to the input ExampleSet.

Description

The Generate Empty Attribute operator creates an empty attribute of specified name and type which are specified by the name and the value type parameter respectively. One of the following types can be selected: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time. Please note that all values are missing right after creation of the attribute. The operators like the Set Data operator can be used to fill values of this attribute. Please note that the name of the attribute can be changed later by the Rename operator and many type conversion operators are also available for changing the type of the attribute. Please note that this operator creates an empty attribute independent of the input ExampleSet, if you want to generate an attribute from the existing attributes of the input ExampleSet you can use the Generate Attributes operator.

Input Ports

example set input \((exa)\) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set output \((exa)\) An empty attribute of the specified name and type is added to the input ExampleSet and the resultant ExampleSet is delivered through this output port.
The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

- **name (string)** This parameter specifies the name of the new attribute. Please note that the names of attributes should be unique. Please make sure that the input ExampleSet does not have an attribute with the same name.
- **value type (selection)** The type of the new attribute is specified by this parameter. One of the following types can be selected: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**Tutorial Processes**

**Adding an empty attribute to the 'Golf' data set**

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the input ExampleSet. As you can see that the 'Golf' data set has 5 attributes: Play, Outlook, Temperature, Humidity and Wind. The Generate Empty Attribute operator is applied on the 'Golf' data set. The name parameter is set to 'name' and the value type parameter is set to 'nominal'. When the process execution is complete, you can see the ExampleSet in the Results Workspace. This ExampleSet has one attribute more than the 'Golf' data set. The name and type of the attribute are the same as specified in the parameters of the Generate Empty Attribute operator. Please note that all values of this new attribute are missing. These values can be filled by using operators like the Set Data operator. Please note that the created empty attribute is independent of the input ExampleSet, if you want to generate an attribute from the existing attributes of the input ExampleSet you can use the Generate
Attributes operator.

**Generate Copy**

This operator generates the copy of an attribute. The original attribute remains unchanged.

**Description**

The Generate Copy operator adds a copy of the selected attribute to the input ExampleSet. Please note that the original attribute remains unchanged, just a new attribute is added to the ExampleSet. The attribute whose copy is required is specified by the *attribute name* parameter. The name of the new attribute is specified through the *new name* parameter. Please note that the names of attributes of an ExampleSet should be unique. Please note that only the view on the data column is copied, not the data itself.

**Input Ports**

example set input *(exa)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.
6.3. Attribute Set Reduction and Transformation

Output Ports

example set output (exa) The ExampleSet with the new attribute that is a copy of the specified attribute is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute name (string) The attribute whose copy is required is specified by the attribute name parameter.
new name (string) The name of the new attribute is specified through the new name parameter. Please note that the names of attributes of an ExampleSet should be unique.

Tutorial Processes

Generating a copy of the Temperature attribute of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Generate Copy operator is applied on it. The attribute name parameter is set to 'Temperature'. The new name parameter is set to 'New Temperature'. Run the process. You will see that an attribute named 'New Temperature' has been added to the 'Golf' data set. The new attribute has the same values as the 'Temperature' attribute. The 'Temperature' attribute remains unchanged.
6. Data Transformation

Generate Attributes

This operator constructs new user defined attributes using mathematical expressions.

Description

The Generate Attributes operator constructs new attributes from the attributes of the input ExampleSet and arbitrary constants using mathematical expressions. The attribute names of the input ExampleSet might be used as variables in the mathematical expressions for new attributes. During the application of this operator these expressions are evaluated on each example, these variables are then filled with the example's attribute values. Thus this operator not only creates new columns for new attributes, but also fills those columns with corresponding values of those attributes. If a variable is undefined in an expression, the entire expression becomes undefined and '?' is stored at its location.

Please note that there are some restrictions for the attribute names in order to let this operator work properly:

- Attribute names containing parentheses are not allowed.
- Attribute names having dashes '-' are not allowed. You can use underscores '_' instead.
- Attribute names containing blanks are not allowed.
6.3. Attribute Set Reduction and Transformation

- Attribute names with function names or operator names are not allowed.

- If the standard constants are usable i.e. the use standard constants parameter is checked, attribute names with names of standard constants are not allowed e.g. 'e' or 'pi' are not allowed.

If you want to apply this operator but the attributes of your ExampleSet do not fulfill above mentioned conditions you can rename attributes with the Rename operator before application of the Generate Attributes operator. When replacing several attributes following a certain schema, the Rename by Replacing operator might prove useful.

A large number of operations is supported, which allows you to write rich expressions. Here is a list of supported expressions along with its explanation and examples. In these examples A, B and C are used as attribute names. These are just simple examples; more complicated expressions can be created by using multiple operations. Parenthesis can be used to nest operations. The description of all operations follows this format:

Name of operation (syntax of operation): brief description; examples: example1,example2

The following basic operations are supported:

- Addition (+): Calculates the addition of the two terms surrounding this operator; examples: A+7, A+B, A+B+9

- Subtraction(-): Calculates the subtraction of the first term from the second one; examples: 15-A, A-B, (A-B)-C

- Multiplication (*): Calculates the multiplication of the two terms surrounding this operator; examples: 5 * B, A*B, A*B*C

- Division(/): Calculates the division of the first term by the second one; examples: B/4, A/B, 4/2

- Power(^): Calculates the first term to the power of the second one; examples: 2^3, A^2, A^B
6. Data Transformation

- Modulus(\%): Calculates the modulus of the first term by the second one; examples: 11\%2, A\%5, B\%C

- Less Than(< ): Delivers true if the first term is less than the second; examples: A< 4, B< A

- Greater Than(> ): Delivers true if the first term is greater than the second; examples: A> 3, B> C

- Less or Equal(<=): Delivers true if the first term is less than or equal to the second; examples: A<= 5, C<= B

- More or Equal(>=): Delivers true if the first term is greater than or equal to the second; examples: A>= 4, 78>= B

- Equal(==): Delivers true if the first term is equal to the second; examples: A==B, A==12

- Not Equal(!=): Delivers true if the first term is not equal to the second; examples: A!=B, A!=25

- Boolean Not(!): Delivers true if the following term is false or vice versa; examples: !(A> 2), !(B< =25)

- Boolean And(& & ): Delivers true if both surrounding terms are true; examples: (A> 2) & & (B< =C)

- Boolean Or(||): Delivers true if at least one of the surrounding terms is true; examples: (A> 2)|| (B< =C)

The following log and exponential functions are supported:

- Natural Logarithm(ln(x)): Calculates the logarithm of the argument to the base e; examples: ln(5), ln(A)

- Logarithm Base 10 (log(x)): Calculates the logarithm of the argument to the base 10; examples: log(25), log(A)

- Logarithm Base 2 (ld(x)): Calculates the logarithm of the argument to the base 2; examples: ld(8), ld(A)
6.3. Attribute Set Reduction and Transformation

base 2, it is also called logarithm dualis; examples: \( \log(24) \), \( \log(A) \)

- **Exponential**(exp(x)): Calculates the value of the constant \( e \) to the power of the argument, it is equivalent to \( e^x \); examples: exp(12), exp(A)

- **Power**(pow(x,y)): Calculates the first term to the power of the second one; examples: pow(A, B), pow(A,3)

The following *trigonometric* functions are supported:

- **Sine** (sin(x)): Calculates the sine of the given argument; examples: sin(A), sin(45)

- **Cosine** (cos(x)): Calculates the cosine of the given argument; examples: cos(A), cos(45)

- **Tangent** (tan(x)): Calculates the tangent of the given argument; examples: tan(A), tan(45)

- **Arc Sine** (asin(x)): Calculates the inverse sine of the given argument; examples: asin(A), asin(0.50)

- **Arc Cosine** (acos(x)): Calculates the inverse cosine of the given argument; examples: acos(A), acos(0.50)

- **Arc Tangent** (atan(x)): Calculates the inverse tangent of the given argument; examples: atan(A), atan(1)

- **Arc Tangent with 2 parameters** (atan2(x,y)): Calculates the inverse tangent based on the two given arguments; examples: atan(A, 0.5)

- **Hyperbolic Sine** (sinh(x)): Calculates the hyperbolic sine of the given argument; examples: sinh(A)

- **Hyperbolic Cosine** (cosh(x)): Calculates the hyperbolic cosine of the given argument; examples: cosh(A)

- **Hyperbolic Tangent** (tanh(x)): Calculates the hyperbolic tangent of the given argument; examples: tanh(A)
6. Data Transformation

- Inverse Hyperbolic Sine (asinh(x)): Calculates the inverse hyperbolic sine of the given argument; examples: asinh(A)

- Inverse Hyperbolic Cosine (acosh(x)): Calculates the inverse hyperbolic cosine of the given argument; examples: acosh(A)

- Inverse Hyperbolic Tangent (atanh(x)): Calculates the inverse hyperbolic tangent of the given argument; examples: atanh(A)

The following statistical functions are supported:

- Round (round(x)): Rounds the given number to the next integer. If two arguments are given, the first one is rounded to the number of digits indicated by the second argument; examples: round(A), round(A, 3). round(A, 3) rounds attribute 'A' to 3 decimal places.

- Floor (floor(x)): Calculates the first integer less than the given argument, e.g. floor(4.7) would be 4; examples: floor(A), floor(23.34)

- Ceiling (ceil(x)): Calculates the next integer greater than the given argument e.g. ceil(23.4) would be 24; examples: ceil(A), ceil(23.34)

- Average (avg(x,y,z...)): Calculates the average of the given arguments; examples: avg(A,B), avg(A,B,C), avg(A,B,25)

- Minimum (min(x,y,z...)): Calculates the minimum of the given arguments; examples: min(A,B), min(A,B,C), min(A,0,B)

- Maximum (max(x,y,z...)): Calculates the maximum of the given arguments; examples: max(A,B), max(A,B,C), max(A,B,45)

The following text functions are supported:

- To String (str(a)): Transforms the given number into a string (i.e. nominal value); example: str(17)

- To Number (parse(a)): Transforms the given string (nominal value) into a number by parsing it; example: parse(A)
6.3. Attribute Set Reduction and Transformation

- **Cut** (cut(attribute or string, start index, length)): Cuts the substring of given length from the given start index from a string. Index starts from 0.; examples: cut(TText“, 1, 2) delivers 'ex' , cut(SSString“, 0, 3) delivers 'Str', cut(A,0,4) delivers first four characters of attribute A's values.

- **Concatenation** (concat(a, b)): Concatenates the given arguments (the + operator can also be used for this); example: both concat(Ät“, öm“) and Ät“+ öm“deliver Ätom“. concat(Ä“, "B“) delivers concatenation of attribute A's value and attribute B's value.

- **Replace** (replace(string or attribute ,string, replacement)): Replaces the first occurrence of a string by the defined replacement; example: replace(A, äm“, ppm“) replaces the first äm¨in each value of attribute 'A' by ppm¨.

- **Replace All** (replaceAll(a , string, replacement)): Replaces all occurrences of a string by the defined replacement; example: replaceAll(A, äm“, ppm“) replaces all äm¨in each value of attribute 'A' by ppm¨.

- **Lower** (lower(a)): Transforms the given argument into lower case characters; example: lower(A) transforms all values of attribute 'A' to lower case.

- **Upper** (upper(a)): Transforms the given argument into upper case characters; example: upper(B) transforms all values of attribute 'B' to upper case.

- **Index** (index(a, b)): Delivers the first position of the given search string in the text. If string is not found '-1' is delivered; examples: index(TText“, ö“) delivers 1, index(TText“, pp“) delivers -1.

- **Length** (length(a)): Delivers the length of the given argument; example: length(A)

- **Character At** (char(a, x)): Delivers the character at the specified position; examples: char(A, 3) delivers the character at fourth place in attribute 'A' values. char(ttext“, 2) delivers 'x'

- **Compare** (compare(a, b)): Compares the two arguments and delivers a negative value, if the first argument is lexicographically smaller; example:
6. Data Transformation

compare(A, B), compare(A, “yes“)

- Contains (contains(a, b)): Delivers true if the second argument is part of the first one; example: contains(A, ppa“) delivers true whenever attribute 'A' value has string 'pa' in it. 'finds(a,b)' function also performs similar task. In comparison to the 'contains (a, b)' function, 'finds(a, b)' allows you to specify regular expression patterns which are more powerful.

- Equals (equals(a, b)): Delivers true if the two arguments are lexicographically equal to each other; examples: equals(A, B), equals(“yes“, B)

- Starts With (starts(a, b)): Delivers true if the first argument starts with the second. The comparison is case-sensitive; examples: starts(A, “GE“), starts(A, B)

- Ends With (ends(a, b)): Delivers true if the first argument ends with the second. The comparison is case-sensitive; examples: ends(B, ÖN“), ends(B, A)

- Matches (matches(a, b)): Delivers true if the first argument matches the regular expression defined by the second argument; example: matches(A, “.*mm.*“). You should have a good understanding of regular expressions in order to apply this operation. Documentation of the Filter Examples operator may provide a good introduction to regular expressions.

- Finds (finds(a, b)): Delivers true if, and only if, a subsequence of the first matches the regular expression defined by the second argument; example: finds(A, “.*OM.*“). In comparison to the 'contains (a, b)' function, 'finds(a, b)' allows you to specify regular expression patterns which are more powerful.

- Suffix (suffix(a, length)): Delivers the suffix of the specified length; example: suffix(A, 2) delivers last two characters of attribute 'A's' values.

- Prefix (prefix(a, length)): Delivers the prefix of the specified length; example: prefix(B, 3) delivers first three characters of attribute 'B's' values.

- Trim (trim(a)): Removes all leading and trailing white space characters;
6.3. Attribute Set Reduction and Transformation

- Example: `trim(A)`

- Escape HTML (escape_html(a)): Escapes the given string with HTML entities; example: `escape_html(A)` where 'A' is an attribute.

The following `date` functions are supported:

- Parse Date (date_parse(a)): Parses the given string or double to a date. The type of the new attribute will be date time instead of string; examples: `date_parse("12/01/88")` is parsed as 'Dec 01, 1988'.

- Parse Date with Locale (date_parse_loc(a, llocale")): Parses the given string or double to a date with the given locale which should be specified via lowercase two-letter ISO-639 code; example: `date_parse_loc(A, "en")`

- Parse Custom Date (date_parse_custom(date1, fformat", llocale")): Parses the given date string to a date using a custom pattern and the given locale which should be specified via lowercase two-letter ISO-639 code; examples: `date_parse_custom("01|09|88", "dd|MM|yy", "en")`, `date_parse_custom(A, "dd-MM-yy", "de")` where attribute 'A' has dates in same format as 'dd-MM-yy'

- Date Before (date_before(date1, date2)): Determines if the first date is strictly earlier than the second date; examples: `date_before(A, B), date_before(A, "Dec 1, 2011")`

- Date After (date_after(date1, date2)): Determines if the first date is strictly later than the second date; example examples: `date_after(A, B), date_after(A, "Dec 1, 2011")`

- Date to String (date_str(date1, DATE_FORMAT_CONSTANT, DATE_SHOW_CONSTANT)): Changes a date to a string using the specified format; example: `date_str(A, DATE_FULL, DATE_SHOW_DATE_AND_TIME)"

- Date to String with Locale (date_str_loc(A, DATE_FORMAT_CONSTANT, DATE_SHOW_CONSTANT, llocale")): Changes a date to a string using the specified format and the given locale which should be specified via lowercase two-letter ISO-639 code; example: `date_str_loc(A, DATE_MEDIUM, DATE_SHOW_TIME_ONLY, "us")"
6. Data Transformation

- **Date to String with custom pattern** (`date_str_custom(date1, fformat="", ll-locale")`): Changes a date to a string using the specified custom format pattern and the given locale which is an optional argument here. Locale should be specified via lowercase two-letter ISO-639 code; example: `date_str_custom(A, "dd|MM|yy", "us")`

- **Create Date** (`date_now()`) Creates the current date; example: `date_now()`

- **Date Difference** (`date_diff(date1, date2, llocale", ttimezone")`): Calculates the elapsed time between two dates. Locale and time zone arguments are optional; example: `date_diff(timeStart, timeEnd)` where `timeStart` and `timeEnd` are date type attributes.

- **Add Time** (`date_add(date1, value, DATE_UNIT_CONSTANT, llocale", ttimezone")`): Adds a custom amount of time to a given date. Note that only the integer portion of a given value will be used. Locale and timezone arguments are optional; example: `date_add(date1, 15, DATE_UNIT_DAY)` adds 15 days to the 'date1' attribute values.

- **Set Time** (`date_set(date1, value, DATE_UNIT_CONSTANT, llocale", ttimezone")`): Allows to set a custom value for a portion of a given date, e.g. set the day to 23. Note that only the integer portion of a given value will be used. Locale and timezone arguments are optional; example: `date_set(date1, 12, DATE_UNIT_DAY)` sets day of every 'date1' attribute value to 12.

- **Get Time** (`date_get(date1, DATE_UNIT_CONSTANT, llocale", ttimezone")`): Allows to get a portion of a given date, e.g. get the day of a month only or get month of the year only. Locale and timezone arguments are optional; example: `date_get(date1, DATE_UNIT_DAY)` gets days of 'date1' attribute values.

The following process functions are supported:

- **Parameter** (`param(öperator="", pparameter")`): Delivers the specified parameter of the specified operator; example: `param(RRead Excel="", ffilee")`
6.3. Attribute Set Reduction and Transformation

The following _miscellaneous_ functions are supported:

- **If-Then-Else** (if(condition,true-evaluation, false-evaluation)): The first argument specifies the condition. If the condition is evaluated as true then the result of the second argument is delivered otherwise the result of the third argument is delivered; example: if(A > 5, 7*A, B/2)

- **Constant** (const(a)): Delivers the argument as numerical constant value; example: const(A)

- **Square Root** (sqrt(x)): Delivers the square root of the given argument; examples: sqrt(16), sqrt(A)

- **Signum** (sgn(x)): Delivers -1 or +1 depending on the sign of the argument; example: sgn(-5) delivers -1.

- **Random** (rand()): Delivers a random number between 0 and 1. It can be used to generate random numbers in any range when used with other functions like multiplication, addition, floor etc; example: floor((rand()*10) + 15) produces random integers between 15 and 25.

- **Modulus** (mod(x, y)): Calculates the modulus of the first term by the second one; example: 11%2, A%3

- **Sum** (sum(x, y, z, ...)): Calculates the sum of all arguments; example: sum(A, B, 42)

- **Binomial** (binom(x, y)): Calculates the binomial coefficients; example: binom(5, 2)

- **Missing** (missing(a)): Checks if the given value is missing. This function will return true whenever there is a missing value in an attribute; example: missing(A) returns true if attribute 'A' has missing values.

The following _CONSTANTS_ are supported:

- **Full Date Format** (DATE_FULL): This constant is used to specify the most detailed date format. Example: Tuesday, January 3, 2012 1:46:33 AM PKT
6. Data Transformation

- Long Date Format (DATE_LONG): This constant is used to specify a very detailed date format. Example: January 3, 2012 1:48:51 AM PKT

- Medium Date Format (DATE_MEDIUM): This constant is used to specify a medium detailed date format. Example: Jan 3, 2012 1:49:52 AM

- Short Date Format (DATE_SHORT): This constant is used to specify a date format with little detail. Example: 1/3/12 1:51 AM

- Show Date and Time (DATE_SHOW_DATE_AND_TIME): This constant is used to specify that the date should contain date and time. Example: Tuesday, January 3, 2012 1:52:13 AM PKT

- Show Date Only (DATE_SHOW_DATE_ONLY): This constant is used to specify that the date should contain the date but not time. Example: Tuesday, January 3, 2012

- Show Time Only (DATE_SHOW_TIME_ONLY): This constant is used to specify that the date should only contain the time. Example: 1:54:42 AM PKT

- Date Unit Year (DATE_UNIT_YEAR): This constant is used to specify the date unit as year. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Month (DATE_UNIT_MONTH): This constant is used to specify the date unit as month. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Week (DATE_UNIT_WEEK): This constant is used to specify the date unit as week. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Day (DATE_UNIT_DAY): This constant is used to specify the date unit as day of the week. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Hour (DATE_UNIT_HOUR): This constant is used to specify
6.3. Attribute Set Reduction and Transformation

the date unit as hour of the day. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Minute (DATE_UNIT_MINUTE): This constant is used to specify the date unit as minute. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Second (DATE_UNIT_SECOND): This constant is used to specify the date unit as second. It is used in operations like 'add time', 'set time' and 'get time'.

- Date Unit Millisecond (DATE_UNIT_MILLISECOND): This constant is used to specify the date unit as millisecond. It is used in operations like 'add time', 'set time' and 'get time'.

This operator also supports the constants 'pi' and 'e' if the use standard constants parameter is set to true. You can also use strings in operations but the string values should be enclosed in double quotes ("). 

Input Ports

example set (exa) This input port expects an ExampleSet. It is the output of the Rename operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set (exa) The ExampleSet with new attributes is output of this port. original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the results workspace.
Parameters

**function descriptions** The list of functions for generating new attributes is provided here.

**use standard constants** *(boolean)* This parameter indicates if standard constants like 'e' or 'pi' should be available. If checked, these constants can be used in expressions for generating new attributes.

**keep all** *(boolean)* If set to true, all the original attributes are kept, otherwise they are removed from the output ExampleSet.

Tutorial Processes

Generating attributes through different function descriptions

The 'Labor-Negotiations' data set is loaded using the Retrieve operator. The Rename operator is used to rename attribute names of the 'Labor-Negotiations' data set. This is done to ensure that all attribute names satisfy the conditions of the Generate Attributes operator. Attributes are renamed as:

- The 'wage-inc-1st' attribute is renamed to 'w1'
- The 'wage-inc-2nd' attribute is renamed to 'w2'
- The 'wage-inc-3rd' attribute is renamed to 'w3'
- The 'working-hours' attribute is renamed to 'hours'
- The 'education-allowance' attribute is renamed to 'allowance'
- The 'shift-differential' attribute is renamed to 'shift'
- The 'standby-pay' attribute is renamed to 'standby'
6.3. Attribute Set Reduction and Transformation

- The 'statutory-holidays' attribute is renamed to 'holidays'

You can see that all new names satisfy the naming convention conditions of the Generate Attributes operator. Now have a look at the Generate Attributes operator's parameters. The use standard constants parameter is checked, which allows us to use constants like 'e'. The keep all parameter is also checked, thus all attributes of the 'Labor-Negotiations' data set are also kept along with attributes generated by the Generate Attributes operator.

Click on the Edit List button of the function descriptions parameter to have a look at descriptions of functions defined for generating new attributes. 14 new attributes are generated, there might be better ways of generating these attributes but here they are written to explain the usage of the different type of functions available in the Generate Attributes operator. Please read the function description of each attribute and then see the values of the corresponding attribute in the Results Workspace to understand it completely. Here is a description of attributes created by this operator:

- The 'average wage-inc' attribute takes sum of the w1, w2 and w3 attribute values and divides the sum by 3. This gives an average of wage-increments. There are better ways of doing this, but this example was just shown to clarify the use of some basic functions.

- The 'neglected worker bool' attribute is a boolean attribute i.e. it has only two possible values '0' and '1'. This attribute was created here to show usage of logical operations like 'AND' and 'OR' in the Generate Attributes operator. This attribute assumes value '1' if three conditions are satisfied. First, the hours attribute has value 35 or more. Second, the allowance attribute is not equal to 'yes'. Third, the vacation attribute has value 'average' OR 'below-average'. If any of these conditions is not satisfied, the new attribute gets value '0'.

- The 'logarithmic attribute' attribute shows the usage of logarithm base 10 and natural logarithm functions.

- The 'trigno attribute' attribute shows the usage of various trigonometric
functions like sine and cosine.

- The 'rounded average wage-inc' attribute uses the \textit{avg} function to take average of wage-increments and then uses the \textit{round} function to round the resultant values.

- The 'vacations' attribute uses the \textit{replaceAll} function to replace all occurrences of value 'generous' with 'above-average' in the 'vacation' attribute.

- The 'deadline' attribute shows usage of the \textit{If-then-Else} and Date functions. This attribute assumes value of current date plus 25 days if class attribute has value 'good'. Otherwise it stores the date of the current date plus 10 days.

- The 'shift complete' attribute shows the usage of the \textit{If-then-Else, random, floor} and \textit{missing} functions. This attribute has values of the shift attribute but it does not have missing values. Missing values are replaced with a random number between 0 and 25.

- The 'remaining_holidays' attribute stores the difference of the holidays attribute value from 15.

- The 'remaining_holidays_percentage' attribute uses the 'remaining_holidays' attribute to find the percentage of remaining holidays. This attribute was created to show that attributes created in this Generate Attribute operator can be used to generate new attributes in the same Generate Attributes operator.

- The 'constants' attribute was created to show the usage of constants like 'e' and 'pi'.

- The 'cut' attribute shows the usage of \textit{cut} function. If you want to specify a string, you should place it in double quotes (') as in the last term of this attribute's expression. If you want to specify name of an attribute you should not place it in the quotes. First term of expression cuts first two characters of the 'class' attribute values. This is because name of attribute is not placed in quotes. Last term of the expression selects first two characters
of the string 'class'. As first two characters of string 'class' are 'cl', thus cl is appended at the end of this attribute's values. The middle term is used to concatenate a blank space between first and last term's results.

- The 'index' attribute shows usage of the index function. If the 'class' attribute has value 'no', 1 is stored because 'o' is at first index. If the 'class' attribute has value 'yes', -1 is stored because 'o' is not present in this value.

- The 'date constants' attribute shows the usage of the date constants. It shows the date of the 'deadline' attribute in full format, but only time is selected for display.

---

**Generate Concatenation**

This operator merges two attributes into a single new attribute by concatenating their values. The new attribute is of nominal type. The original attributes remain unchanged.

**Description**

The Generate Concatenation operator merges two attributes of the input ExampleSet into a single new nominal attribute by concatenating the values of the two attributes. If the resultant attribute is actually of numerical type, it can be
6. Data Transformation

converted from nominal to numerical type by using the Nominal to Numeric operator. The original attributes remain unchanged, just a new attribute is added to the ExampleSet. The two attributes to be concatenated are specified by the first attribute and second attribute parameters.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

Output Ports

example set output (exa) The ExampleSet with the new attribute that has concatenated values of the specified attributes is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

first attribute (string) This parameter specifies the first attribute to be concatenated.
second attribute (string) This parameter specifies the second attribute to be concatenated.
separator (string) This parameter specifies the string which is used as separation of values of the first and second attribute i.e. the string that is concatenated between the two values.
trim values (boolean) This parameter indicates if the values of the first and second attribute should be trimmed i.e. leading and trailing whitespaces should be removed before the concatenation is performed.
6.3. Attribute Set Reduction and Transformation

Tutorial Processes

Generating a concatenated attribute in the Labor-Negotiations data set

The 'Labor-Negotiations' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The 'vacation' and 'statutory-holidays' attributes will be concatenated to form a new attribute. The Generate Concatenation operator is applied on the Labor-Negotiations data set. The first attribute and second attribute parameters are set to 'vacation' and 'statutory-holidays' respectively. The separator parameter is set to '. '. Thus the values of the 'vacation' and 'statutory-holidays' attributes will be merged with a '.' between them. You can verify this by seeing the resultant ExampleSet in the Results Workspace. The 'vacation' and 'statutory-holidays' attributes remain unchanged. A new attribute named 'vacation_statutory-holidays' is created. The type of the new attribute is nominal.

Generate Aggregation

This operator generates a new attribute by performing the specified aggregation function on every example of the selected attributes.
6. Data Transformation

Description

This operator can be considered to be a blend of the Generate Attributes operator and the Aggregate operator. This operator generates a new attribute which consists of a function of several other attributes. These 'other' attributes can be selected by the attribute filter type parameter and other associated parameters. The aggregation function is selected through the aggregation function parameter. Several aggregation functions are available e.g. count, minimum, maximum, average, mode etc. The attribute name parameter specifies the name of the new attribute. If you think this operator is close to your requirement but not exactly what you need, have a look at the Aggregate and the Generate Attributes operators because they perform similar tasks.

Differentiation

Aggregate This operator performs the aggregation functions known from SQL. It provides a lot of functionalities in the same format as provided by the SQL aggregation functions. SQL aggregation functions and GROUP BY and HAVING clauses can be imitated using this operator. See page 615 for details.

Generate Attributes It is a very powerful operator for generating new attributes from existing attributes. It even supports regular expressions and conditional statements for specifying the new attributes See page 402 for details.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.
Output Ports

example set output (exa) The ExampleSet with the additional attribute generated after applying the specified aggregation function is output of this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute name (string) The name of the resulting attribute is specified through this parameter.

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- all This option simply selects all the attributes of the ExampleSet. This is the default option.
- single This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.
- subset This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- regular_expression This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.
6. Data Transformation

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example, real and integer types both belong to the numeric type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression (string)** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular
6.3. Attribute Set Reduction and Transformation

expressions.

**use except expression** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '>= 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '>= 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.
6. Data Transformation

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**aggregation function** *(selection)* This parameter specifies the function for aggregating the values of the selected attribute. Numerous options are available e.g. average, variance, standard deviation, count, minimum, maximum, sum, mode, median and product.

**keep all** *(boolean)* This parameter indicates if all old attributes should be kept. If this parameter is set to false then all the selected attributes (i.e. attributes that are used for aggregation) are removed.

**ignore missings** *(boolean)* This parameter indicates if missing values should be ignored and if the aggregation function should be only applied on existing values. If this parameter is not set to true the aggregated value will be a missing value in the presence of missing values in the selected attribute.

### Related Documents

- Aggregate (615)
- Generate Attributes (402)

### Tutorial Processes

Generating an attribute having average of integer attributes of Golf data set
6.3. Attribute Set Reduction and Transformation

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has three nominal and two integer attributes. The Generate Aggregation operator is applied on this ExampleSet to generate a new attribute from the integer attributes of the ExampleSet i.e. Temperature and Humidity attributes. The attribute name parameter is set to 'Avg' thus the new attribute will be named 'Avg'. The attribute filter type parameter is set to 'value type' and the value type parameter is set to 'integer', thus the new attribute will be created from integer attributes of the ExampleSet. The aggregation function parameter is set to 'average', thus the new attribute will be average of the selected attributes. The resultant ExampleSet can be seen in the Results Workspace. You can see that there is a new attribute named 'Avg' in the ExampleSet that has the average value of the Temperature and Humidity attributes.

Generate Function Set

This is an attribute generation operator which generates new attributes by applying a set of selected functions on all attributes.
6. Data Transformation

Description

This operator applies a set of selected functions on all attributes of the input ExampleSet for generating new attributes. Numerous functions are available including summation, difference, multiplication, division, reciprocal, square root, power, sine, cosine, tangent, arc tangent, absolute, minimum, maximum, ceiling, floor and round. It is important to note that the functions with two arguments will be applied on all possible pairs. For example suppose an ExampleSet with three numerical attributes A, B and C. If the summation function is applied on this ExampleSet then three new attributes will be generated with values A+B, A+C and B+C. Similarly non-commutative functions will be applied on all possible permutations. This is a useful attribute generation operator but if it does not meet your requirements please try the Generate Attributes operator which is a very powerful attribute generation operator.

Input Ports

example set input \((\text{exa})\) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set output \((\text{exa})\) New attributes are created by application of the selected functions and the resultant ExampleSet is delivered through this port. original \((\text{ori})\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

**keep all** *(boolean)* This parameter indicates if the original attributes should be kept.

**use plus** *(boolean)* This parameter indicates if the summation function should be applied for generation of new attributes.

**use diff** *(boolean)* This parameter indicates if the difference function should be applied for generation of new attributes.

**use mult** *(boolean)* This parameter indicates if the multiplication function should be applied for generation of new attributes.

**use div** *(boolean)* This parameter indicates if the division function should be applied for generation of new attributes.

**use reciprocals** *(boolean)* This parameter indicates if the reciprocal function should be applied for generation of new attributes.

**use square roots** *(boolean)* This parameter indicates if the square roots function should be applied for generation of new attributes.

**use power functions** *(boolean)* This parameter indicates if the power function should be applied for generation of new attributes.

**use sin** *(boolean)* This parameter indicates if the sine function should be applied for generation of new attributes.

**use cos** *(boolean)* This parameter indicates if the cosine function should be applied for generation of new attributes.

**use tan** *(boolean)* This parameter indicates if the tangent function should be applied for generation of new attributes.

**use atan** *(boolean)* This parameter indicates if the arc tangent function should be applied for generation of new attributes.

**use exp** *(boolean)* This parameter indicates if the exponential function should be applied for generation of new attributes.

**use log** *(boolean)* This parameter indicates if the logarithmic function should be applied for generation of new attributes.

**use absolute values** *(boolean)* This parameter indicates if the absolute values function should be applied for generation of new attributes.

**use min** *(boolean)* This parameter indicates if the minimum values function
6. Data Transformation

should be applied for generation of new attributes.

**use max (boolean)** This parameter indicates if the maximum values function should be applied for generation of new attributes.

**use ceil (boolean)** This parameter indicates if the ceiling function should be applied for generation of new attributes.

**use floor (boolean)** This parameter indicates if the floor function should be applied for generation of new attributes.

**use rounded (boolean)** This parameter indicates if the round function should be applied for generation of new attributes.

## Tutorial Processes

### Using the power function for attribute generation

The 'Iris' data set is loaded using the Retrieve operator. A **breakpoint** is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 4 real attributes. The Generate Function Set operator is applied on this ExampleSet for generation of new attributes, only the Power function is used. It is not a commutative function e.g. 2 raised to power 3 is not the same as 3 raised to power 2. The non-commutative functions are applied for all possible permutations. As there are 4 original attributes, there are 16 (i.e. 4 x 4) possible permutations. Thus 16 new attributes are created as a result of this operator. The resultant ExampleSet can be seen in the Results Workspace. As the **keep all** parameter was set to true, the original attributes of the ExampleSet are not discarded.
6.3. Attribute Set Reduction and Transformation

Optimize by Generation (YAGGA)

This operator may select some attributes from the original attribute set and it may also generate new attributes from the original attribute set. YAGGA (Yet Another Generating Genetic Algorithm) does not change the original number of attributes unless adding or removing (or both) attributes prove to have a better fitness.

Description

Sometimes the selection of features alone is not sufficient. In these cases other transformations of the feature space must be performed. The generation of new attributes from the given attributes extends the feature space. Maybe a hypothesis can be easily found in the extended feature space. This operator can be considered to be a blend of attribute selection and attribute generation procedures. It may select some attributes from original set of attributes and it may also generate new attributes from the original attributes. The (generating) mutation can do one of the following things with different probabilities:

- Probability $p/4$: Add a newly generated attribute to the feature vector.
- Probability $p/4$: Add a randomly chosen original attribute to the feature vector.
- Probability $p/2$: Remove a randomly chosen attribute from the feature vector.

Thus it is guaranteed that the length of the feature vector can both grow and shrink. On average it will keep its original length, unless longer or shorter individuals prove to have a better fitness.
A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. For studying basic algorithm of a genetic algorithm please study the description of the Optimize Selection (Evolutionary) operator.

This operator is a nested operator i.e. it has a subprocess. The subprocess must return a performance vector. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

**Differentiation**

**Optimize by Generation (YAGGA2)** The YAGGA2 operator is an improved version of the usual YAGGA operator, this operator allows more feature generators and provides several techniques for redundancy prevention. This leads to smaller ExampleSets containing less redundant features. See page ?? for details.

**Input Ports**

`example set in (exa)` This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**Output Ports**

`example set out (exa)` The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet is delivered through this port.

`attribute weights out (att)` The attribute weights are delivered through this
6.3. Attribute Set Reduction and Transformation

port.

**performance out** *(per)* This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

**Parameters**

**limit max total number of attributes** *(boolean)* This parameter indicates if the total number of attributes in all generations should be limited. If set to true, the maximum number is specified by the **max total number of attributes** parameter.

**max total number of attributes** *(integer)* This parameter is only available when the **limit max total number of attributes** parameter is set to true. This parameter specifies the maximum total number of attributes in all generations.

**use local random seed** *(boolean)* This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is available only if the **use local random seed** parameter is set to true.

**show stop dialog** *(boolean)* This parameter determines if a dialog with a stop button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**maximal fitness** *(real)* This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**population size** *(integer)* This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations** *(integer)* This parameter specifies the number of generations after which the algorithm should be terminated.

**use plus** *(boolean)* This parameter indicates if the summation function should be applied for generation of new attributes.

**use diff** *(boolean)* This parameter indicates if the difference function should be applied for generation of new attributes.
6. Data Transformation

**use mult** *(boolean)* This parameter indicates if the multiplication function should be applied for generation of new attributes.

**use div** *(boolean)* This parameter indicates if the division function should be applied for generation of new attributes.

**use reciprocals** *(boolean)* This parameter indicates if the reciprocal function should be applied for generation of new attributes.

**use early stopping** *(boolean)* This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.

**generations without improvement** *(integer)* This parameter is only available when the **use early stopping** parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after \( n \) generations without improvement in the performance. \( n \) is specified by this parameter.

**tournament size** *(real)* This parameter specifies the fraction of the current population which should be used as tournament members.

**start temperature** *(real)* This parameter specifies the scaling temperature.

**dynamic selection pressure** *(boolean)* If this parameter is set to true, the selection pressure is increased to maximum during the complete optimization run.

**keep best individual** *(boolean)* If set to true, the best individual of each generation is guaranteed to be selected for the next generation.

**p initialize** *(real)* The initial probability for an attribute to be switched on is specified by this parameter.

**p crossover** *(real)* The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type** *(selection)* The type of the crossover can be selected by this parameter.

**use heuristic mutation probability** *(boolean)* If this parameter is set to true, the probability for mutations will be chosen as \( 1/n \) where \( n \) is the number of attributes. Otherwise the probability for mutations should be specified through the **p mutation** parameter.

**p mutation** *(real)* The probability for an attribute to be changed is specified by this parameter. If set to -1, the probability will be set to \( 1/n \) where \( n \) is the total number of attributes.
Related Documents

Optimize by Generation (YAGGA2) (??)

Tutorial Processes

Applying YAGGA on the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other than the label attribute. The Optimize by Generation (YAGGA) operator is applied on the ExampleSet. Optimize by Generation (YAGGA) is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator is used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The Linear Regression operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance (Regression) operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes now has 6 attributes. The attributes 'a1' and 'a2' were selected from the original attribute set and the attributes 'gensym2', 'gensym35', 'gensym63' and 'gensym72' were generated. The number of resultant attributes is not less than the number of original attributes because YAGGA is not an attribute reduction operator. It may (or may not) increase or decrease the number of attributes depending on what proves to have a better fitness.
Principal Component Analysis

This operator performs a Principal Component Analysis (PCA) using the covariance matrix. The user can specify the amount of variance to cover in the original data while retaining the best number of principal components. The user can also specify manually the number of principal components.

Description

Principal component analysis (PCA) is an attribute reduction procedure. It is useful when you have obtained data on a number of attributes (possibly a large number of attributes), and believe that there is some redundancy in those attributes. In this case, redundancy means that some of the attributes are correlated with one another, possibly because they are measuring the same construct. Because of this redundancy, you believe that it should be possible to reduce the observed attributes into a smaller number of principal components (artificial attributes) that will account for most of the variance in the observed attributes.

Principal Component Analysis is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number
of original attributes. This transformation is defined in such a way that the first principal component’s variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

Please note that PCA is sensitive to the relative scaling of the original attributes. This means that whenever different attributes have different units (like temperature and mass); PCA is a somewhat arbitrary method of analysis. Different results would be obtained if one used Fahrenheit rather than Celsius for example.

**Input Ports**

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

**Output Ports**

example set (exa) The Principal Component Analysis is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.
6. Data Transformation

Parameters

dimensionality reduction (selection) This parameter indicates which type of dimensionality reduction (reduction in number of attributes) should be applied.

- **none** if this option is selected, no component is removed from the ExampleSet.
- **keep_variance** if this option is selected, all the components with a cumulative variance greater than the given threshold are removed from the ExampleSet. The threshold is specified by the variance threshold parameter.
- **fixed_number** if this option is selected, only a fixed number of components are kept. The number of components to keep is specified by the number of components parameter.

variance threshold (real) This parameter is available only when the dimensionality reduction parameter is set to 'keep variance'. All the components with a cumulative variance greater than the variance threshold are removed from the ExampleSet.

number of components (integer) This parameter is only available when the dimensionality reduction parameter is set to 'fixed number'. The number of components to keep is specified by the number of components parameter.

Tutorial Processes

Dimensionality reduction of the Polynomial data set using the Principal Component Analysis operator

The 'Polynomial' data set is loaded using the Retrieve operator. The Covariance Matrix operator is applied on it. A breakpoint is inserted here so that you can have a look at the ExampleSet and its covariance matrix. For this purpose
the Covariance Matrix operator is applied otherwise it is not required here. The Principal Component Analysis operator is applied on the 'Polynomial' data set. The dimensionality reduction parameter is set to 'fixed number' and the number of components parameter is set to 4. Thus the resultant ExampleSet will be composed of 4 principal components. As mentioned in the description, the principal components are uncorrelated with each other thus their covariance should be zero. The Covariance Matrix operator is applied on the output of the Principal Component Analysis operator. You can see the covariance matrix of the resultant ExampleSet in the Results Workspace. As you can see that the covariance of the components is zero.

Principal Component Analysis (Kernel)

This operator performs Kernel Principal Component Analysis (PCA) which is a non-linear extension of PCA.
6. Data Transformation

Description

Kernel principal component analysis (kernel PCA) is an extension of principal component analysis (PCA) using techniques of kernel methods. Using a kernel, the originally linear operations of PCA are done in a reproducing kernel Hilbert space with a non-linear mapping. By the use of integral operator kernel functions, one can efficiently compute principal components in high-dimensional feature spaces, related to input space by some nonlinear map. The result will be the set of data points in a non-linearly transformed space. Please note that in contrast to the usual linear PCA the kernel variant also works for large numbers of attributes but will become slow for large number of examples.

RapidMiner provides the Principal Component Analysis operator for applying linear PCA. Principal Component Analysis is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

Differentiation

Principal Component Analysis Kernel principal component analysis (kernel PCA) is an extension of principal component analysis (PCA) using techniques of kernel methods. In contrast to the usual linear PCA the kernel variant also works for large numbers of attributes but will become slow for large number of examples. See page 434 for details.
6.3. Attribute Set Reduction and Transformation

Input Ports

default_text

example set input *(exa)* This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

Output Ports

default_text

example set output *(exa)* The kernel-based Principal Component Analysis is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

original *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

preprocessing model *(pre)* This port delivers the preprocessing model, which has the information regarding the parameters of this operator in the current process.

Parameters

kernel type *(selection)* The type of the kernel function is selected through this parameter. Following kernel types are supported: dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric

- **dot** The dot kernel is defined by $k(x,y)=x^*y$ i.e. it is inner product of $x$ and $y$.
- **radial** The radial kernel is defined by $\exp(-g \|x-y\|^2)$ where $g$ is the gamma,
it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by \( k(x,y) = (x^*y + 1)^d \) where \( d \) is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net \( \tanh(a \cdot x^*y + b) \) where \( a \) is *alpha* and \( b \) is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is \( 1/N \), where \( N \) is the data dimension. Note that not all choices of \( a \) and \( b \) lead to a valid kernel function.

- **anova** The anova kernel is defined by raised to power \( d \) of summation of \( \exp(-g \cdot (x-y)) \) where \( g \) is *gamma* and \( d \) is *degree*. *gamma* and *degree* are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.

- **epachnenikov** The epachnenikov kernel is this function \( (3/4)(1-u^2) \) for \( u \) between -1 and 1 and zero for \( u \) outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.

- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1*, *kernel sigma2* and *kernel sigma3*.

- **multiquadric** The multiquadric kernel is defined by the square root of \( ||x-y||^2 + c^2 \). It has adjustable parameters *kernel sigma1* and *kernel sigma shift*.

**kernel gamma** (*real*) This is the kernel parameter gamma. This is only available when the *kernel type* parameter is set to radial or anova.

**kernel sigma1** (*real*) This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** (*real*) This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to gaussian combination.
6.3. Attribute Set Reduction and Transformation

**kernel sigma3** *(real)* This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** *(real)* This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree** *(real)* This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** *(real)* This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *neural*.

**kernel b** *(real)* This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *neural*.

**Related Documents**

Principal Component Analysis (434)

**Tutorial Processes**

Introduction to the Principal Component Analysis (Kernel) operator

The 'Polynomial' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes. The Principal Component Analysis (Kernel) operator is applied on this ExampleSet with default values of all parameters. The *kernel type* parameter is set to 'radial' and the *kernel gamma* parameter is set to 1.0. The resultant ExampleSet can be seen in the Results Workspace. You can see that this ExampleSet has a different set of attributes.
Independent Component Analysis

This operator performs the Independent Component Analysis (ICA) of the given ExampleSet using the FastICA-algorithm of Hyvärinen and Oja.

Description

Independent component analysis (ICA) is a very general-purpose statistical technique in which observed random data are linearly transformed into components that are maximally independent from each other, and simultaneously have interesting distributions. Such a representation seems to capture the essential structure of the data in many applications, including feature extraction. ICA is used for revealing hidden factors that underlie sets of random variables or measurements. ICA is superficially related to principal component analysis (PCA) and factor analysis. ICA is a much more powerful technique, however, capable of finding the underlying factors or sources when these classic methods fail completely. This operator implements the FastICA-algorithm of A. Hyvärinen and E. Oja. The FastICA-algorithm has most of the advantages of neural algorithms: It is parallel, distributed, computationally simple, and requires little memory space.
6.3. Attribute Set Reduction and Transformation

Input Ports

dexample set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

Output Ports

dexample set output (exa) The Independent Component Analysis is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

don original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

dpreprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

ddimensionality reduction (selection) This parameter indicates which type of dimensionality reduction (reduction in number of attributes) should be applied.

• none if this option is selected, dimensionality reduction is not performed.

• fixed_number if this option is selected, only a fixed number of components are kept. The number of components to keep is specified by the number of components parameter.
number of components \textit{(integer)} This parameter is only available when the \textit{dimensionality reduction} parameter is set to 'fixed number'. The number of components to keep is specified by the \textit{number of components} parameter.

\textbf{algorithm type} \textit{(selection)} This parameter specifies the type of algorithm to be used.

- \textbf{parallel} If parallel option is selected, the components are extracted simultaneously.

- \textbf{deflation} If deflation option is selected, the components are extracted one at a time.

\textbf{function} \textit{(selection)} This parameter specifies the functional form of the G function to be used in the approximation to neg-entropy.

\textbf{alpha} \textit{(real)} This parameter specifies the alpha constant in range [1, 2] which is used in approximation to neg-entropy.

\textbf{row norm} \textit{(boolean)} This parameter indicates whether rows of the data matrix should be standardized beforehand.

\textbf{max iteration} \textit{(integer)} This parameter specifies the maximum number of iterations to perform.

\textbf{tolerance} \textit{(real)} This parameter specifies a positive scalar giving the tolerance at which the un-mixing matrix is considered to have converged.

\textbf{use local random seed} \textit{(boolean)} This parameter indicates if a \textit{local random seed} should be used for randomization. Using the same value of \textit{local random seed} will produce the same randomization.

\textbf{local random seed} \textit{(integer)} This parameter specifies the \textit{local random seed}. This parameter is only available if the \textit{use local random seed} parameter is set to true.

\section*{Tutorial Processes}

Dimensionality reduction of the Sonar data set using the Independent Component Analysis operator
The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 60 attributes. The Independent Component Analysis operator is applied on the 'Sonar' data set. The dimensionality reduction parameter is set to 'fixed number' and the number_of_components parameter is set to 10. Thus the resultant ExampleSet will be composed of 10 components (artificial attributes). You can see the resultant ExampleSet in the Results Workspace and verify that it has only 10 attributes. Please note that these attributes are not original attributes of the 'Sonar' data set. These attributes were created using the ICA procedure.

**Generalized Hebbian Algorithm**

This operator is an implementation of Generalized Hebbian Algorithm (GHA) which is an iterative method for computing principal components. The user can specify manually the required number of principal components.
6. Data Transformation

Description

The Generalized Hebbian Algorithm (GHA) is a linear feedforward neural network model for unsupervised learning with applications primarily in principal components analysis. From a computational point of view, it can be advantageous to solve the eigenvalue problem by iterative methods which do not need to compute the covariance matrix directly. This is useful when the ExampleSet contains many attributes (hundreds or even thousands).

Principal Component Analysis (PCA) is an attribute reduction procedure. It is useful when you have obtained data on a number of attributes (possibly a large number of attributes), and believe that there is some redundancy in those attributes. In this case, redundancy means that some of the attributes are correlated with one another, possibly because they are measuring the same construct. Because of this redundancy, you believe that it should be possible to reduce the observed attributes into a smaller number of principal components (artificial attributes) that will account for most of the variance in the observed attributes. Principal Component Analysis is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their
6.3. Attribute Set Reduction and Transformation

meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

Output Ports

description example set (exa) The Generalized Hebbian Algorithm is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.
description original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
description preprocessing model (pre) This port delivers the GHA model.

Parameters

number of components (integer) The number of components to keep is specified by the number of components parameter. If set to -1; number of principal components in the resultant ExampleSet it equal to the number of attributes in the original ExampleSet.
description number of iterations (integer) This parameter specifies the number of iterations to apply the update rule.
description learning rate (real) This parameter specifies the learning rate of the GHA.
description use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization.
description local random seed (integer) This parameter specifies the local random seed. It is available only if the use local random seed parameter is set to true.
6. Data Transformation

Tutorial Processes

Dimensionality reduction of the Polynomial data set using the GHA operator

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes. The Generalized Hebbian Algorithm operator is applied on the 'Polynomial' data set. The number of components parameter is set to 3. Thus the resultant ExampleSet will be composed of 3 principal components. All other parameters are used with default values. Run the process, you will see that the ExampleSet that had 5 attributes has been reduced to an ExampleSet with 3 principal components.

Singular Value Decomposition

This operator performs a dimensionality reduction of the given ExampleSet based on Singular Value Decomposition (SVD). The user can specify the required number of dimensions or specify the cumulative variance threshold. In the latter case all components having cumula-
6.3. Attribute Set Reduction and Transformation

tive variance above this threshold are discarded.

Description

Singular Value Decomposition (SVD) can be used to better understand an ExampleSet by showing the number of important dimensions. It can also be used to simplify the ExampleSet by reducing the number of attributes of the ExampleSet. This reduction removes unnecessary attributes that are linearly dependent in the point of view of Linear Algebra. It is useful when you have obtained data on a number of attributes (possibly a large number of attributes), and believe that there is some redundancy in those attributes. In this case, redundancy means that some of the attributes are correlated with one another, possibly because they are measuring the same construct. Because of this redundancy, you believe that it should be possible to reduce the observed attributes into a smaller number of components (artificial attributes) that will account for most of the variance in the observed attributes. For example, imagine an ExampleSet which contains an attribute that stores the water's temperature on several samples and another that stores its state (solid, liquid or gas). It is easy to see that the second attribute is dependent on the first attribute and, therefore, SVD could easily show us that it is not important for the analysis.

RapidMiner provides various dimensionality reduction operators e.g. the Principal Component Analysis operator. The Principal Component Analysis technique is a specific case of SVD. It is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.
6. Data Transformation

Differentiation

Principal Component Analysis PCA is a dimensionality reduction procedure. PCA is a specific case of SVD. See page 434 for details.

Input Ports

eexample set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

Output Ports

eexample set output (exa) The Singular Value Decomposition is performed on the input ExampleSet and the resultant ExampleSet is delivered through this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

dimensionality reduction (selection) This parameter indicates which type of dimensionality reduction (reduction in number of attributes) should be applied.
6.3. Attribute Set Reduction and Transformation

- **none** if this option is selected, dimensionality reduction is not performed.
- **keep_percentage** if this option is selected, all the components with a cumulative variance greater than the given threshold are removed from the ExampleSet. The threshold is specified by the *percentage threshold* parameter.
- **fixed_number** if this option is selected, only a fixed number of components are kept. The number of components to keep is specified by the *dimensions* parameter.

*percentage threshold* (*real*) This parameter is only available when the *dimensionality reduction* parameter is set to 'keep percentage'. All the components with a cumulative variance greater than the *percentage threshold* are removed from the ExampleSet.

*dimensions* (*integer*) This parameter is only available when the *dimensionality reduction* parameter is set to 'fixed number'. The number of components to keep is specified by the *dimensions* parameter.

Related Documents

Principal Component Analysis (434)

Tutorial Processes

Dimensionality reduction of the Sonar data set using the Singular Value Decomposition operator

The 'Sonar' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 60 attributes. The Singular Value Decomposition operator is applied on the 'Sonar' data set. The *dimensionality reduction* parameter is set
6. Data Transformation

to 'fixed number' and the \textit{dimensions} parameter is set to 10. Thus the resultant ExampleSet will be composed of 10 dimensions (artificial attributes). You can see the resultant ExampleSet in the Results Workspace and verify that it has only 10 attributes. Please note that these attributes are not original attributes of the 'Sonar' data set. These attributes were created using the SVD procedure.

Self-Organizing Map

This operator performs a dimensionality reduction of the given ExampleSet based on a self-organizing map (SOM). The user can specify the required number of dimensions.

Description

A self-organizing map (SOM) or self-organizing feature map (SOFM) is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map. Self-organizing maps are different from other artificial neural networks in the sense that they use a neighborhood function to preserve the topological properties of the input space. This makes SOMs useful for visualizing low-dimensional views of high-dimensional data, akin to multidimensional scaling. The model was first described as an artificial neural network by Teuvo Kohonen, and is sometimes called a Kohonen map.

Like most artificial neural networks, SOMs operate in two modes: training and
6.3. Attribute Set Reduction and Transformation

mapping. Training builds the map using input examples. Mapping automatically classifies a new input vector. A self-organizing map consists of components called nodes or neurons. Associated with each node is a weight vector of the same dimension as the input data vectors and a position in the map space. The usual arrangement of nodes is a regular spacing in a hexagonal or rectangular grid. The self-organizing map describes a mapping from a higher dimensional input space to a lower dimensional map space. The procedure for placing a vector from data space onto the map is to first find the node with the closest weight vector to the vector taken from data space. Once the closest node is located it is assigned the values from the vector taken from the data space.

While it is typical to consider this type of network structure as related to feed-forward networks where the nodes are visualized as being attached, this type of architecture is fundamentally different in arrangement and motivation.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along with the data. Please note that this operator cannot handle nominal attributes; it works on numerical attributes.

Output Ports

example set output (exa) The dimensionality reduction of the given ExampleSet is performed based on a self-organizing map and the resultant ExampleSet is delivered through this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same Exam-
pleSet in further operators or to view the ExampleSet in the Results Workspace. **preprocessing model** *(pre)* This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

**Parameters**

**return preprocessing model** *(boolean)* This parameter indicates if the preprocessing model should be returned.

**number of dimensions** *(integer)* This parameter specifies the number of dimensions to keep i.e. the number of attributes of the resultant ExampleSet.

**net size** *(integer)* This parameter specifies the size of the SOM net, by setting the length of every edge of the net.

**training rounds** *(integer)* This parameter specifies the number of training rounds.

**learning rate start** *(real)* This parameter specifies the strength of an adaption in the first round. The strength will decrease every round until it reaches the **learning rate end** in the last round.

**learning rate end** *(real)* This parameter specifies the strength of an adaption in the last round. The strength will decrease to this value in last round, beginning with **learning rate start** in the first round.

**adaption radius start** *(real)* This parameter specifies the radius of the sphere around a stimulus in the first round. This radius decreases every round, starting by **adaption radius start** in the first round, to **adaption radius end** in the last round.

**adaption radius end** *(real)* This parameter specifies the radius of the sphere around a stimulus in the last round. This radius decreases every round, starting by **adaption radius start** in the first round, to **adaption radius end** in the last round.
Tutorial Processes

Dimensionality reduction of the Sonar data set using the Self-Organizing Map operator

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 60 attributes. The Self-Organizing Map operator is applied on the 'Sonar' data set. The number of dimensions parameter is set to 10. Thus the resultant ExampleSet will be composed of 10 dimensions (artificial attributes). You can see the resultant ExampleSet in the Results Workspace and verify that it has only 10 attributes. Please note that these attributes are not original attributes of the 'Sonar' data set. These attributes were created using the SOM procedure.

Select Attributes

This operator selects which attributes of an ExampleSet should be kept and which attributes should be removed. This is used in cases when not all attributes of an ExampleSet are required; it helps you to select required attributes.
6. Data Transformation

Description

Often need arises for selecting attributes before applying some operators. This is especially true for large and complex data sets. The Select Attributes operator lets you select required attributes conveniently. Different filter types are provided to make attribute selection easy. Only the selected attributes will be delivered from the output port and the rest will be removed from the ExampleSet.

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

Output Ports

example set (exa) The ExampleSet with selected attributes is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- all This option simply selects all the attributes of the ExampleSet, no
attributes are removed. This is the default option.

- **single** This option allows the selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows the selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for the attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows the selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric_value_filter** When this option is selected another parameter (nu-
6. Data Transformation

...meric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the *parameter* attribute if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes that will make it to the output port; all other attributes will be removed.

**regular expression** *(string)* The attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** *(boolean)* If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in regular expression parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. the *value type* parameter's value. One of the following types can be selected
6.3. Attribute Set Reduction and Transformation

Here: nominal, numeric, integer, real, text, binominal, polynominal, file_path, date_time, date, time.

**block type (selection)** The Block type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type. One of the following block types can be selected here: single_value, value_series, value_series_start, value_series_end, value_matrix, value_matrix_start, value_matrix_end, value_matrix_row_start.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is mentioned here. For example, the numeric condition '

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

greater than 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '!=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are delivered to the output port irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are removed and previously removed attributes are selected. For example, if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After
6. Data Transformation

selection of this parameter 'att1' will be removed and 'att2' will be selected.

Tutorial Processes

Selecting attributes by specifying regular expressions matching their names

In the given Example process the Labor-Negotiations ExampleSet is loaded using the Retrieve operator. Then Select Attribute operator is applied on it. Have a look at the Parameter View of the Select Attributes operator. Here is a stepwise explanation of this process.

- See that at the bottom of Parameters View the *include special attributes* parameter is set to true. This means that all special attributes will also be checked against all the given conditions, they will appear in the output only if they pass all the conditions. The only special attribute is the 'class' attribute in this ExampleSet. Though 'class' is a special attribute; it will make to the output port only if it passes the conditions because the *include special attributes* parameter is set to true.

- The regular expression specified is = w.*|.*y.*
  - w.* means all attribute names with starting alphabet 'w'.
  - wage-inc-1st, wage-inc-2nd, wage-inc-3rd, working-hours satisfy this condition
  - .*y.* means all attributes that have a 'y' in their name.
  - standby-pay, statutory-holidays, longterm-disability-assistance satisfy this condition.
  - || means logical OR operator. So if any attribute whose name starts with 'w' or its name contains a 'y', it satisfies this expression and is selected.
6.3. Attribute Set Reduction and Transformation


- The use except expression parameter is also set to true which means attributes that satisfy the condition in the except regular expression parameter would be removed.

  - The regular expression for except regular expression is = .*[0-9].*

  - This expression means any attribute whose name contains a digit.

  - Three attributes satisfy this condition: wage-inc-1st, wage-inc-2nd, wage-inc-3rd. Thus these three attributes do not make it to the output port even though they satisfied the regular expression of the regular expression parameter.

- Finally we are left with the following four attributes: working-hours, standby-pay, statutory-holidays, longterm-disability-assistance. These four attributes make it to the output port.

- Notice that the invert selection parameter was not set to true. If it was set to true, all attributes other than these four attributes would have made it to the output port.
6. Data Transformation

Select by Weights

This operator selects only those attributes of an input ExampleSet whose weights satisfy the specified criterion with respect to the input weights.

Description

This operator selects only those attributes of an input ExampleSet whose weights satisfy the specified criterion with respect to the input weights. Input weights are provided through the weights input port. The criterion for attribute selection by weights is specified by the weight relation parameter.

Input Ports

example set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data

weights (wei) This port expects the attribute weights. There are numerous operators that provide the attribute weights. The Weight by Correlation operator is used in the Example Process.

Output Ports

example set (exa) The ExampleSet with selected attributes is output of this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same Exam-
Attribute Set Reduction and Transformation

pleSet in further operators or to view the ExampleSet in the Results Workspace.

weights (wei) The Attributes weights that were provided at the weights input port are delivered through this output port.

Parameters

weight relation Only those attributes are selected whose weights satisfy this relation.

- **greater** Attributes whose weights are greater than the weight parameter are selected.
- **greater_equal** Attributes whose weights are equal or greater than the weight parameter are selected.
- **equal** Attributes whose weights are equal to the weight parameter are selected.
- **less_equal** Attributes whose weights are equal or less than the weight parameter are selected.
- **less** Attributes whose weights are less than the weight parameter are selected.
- **top_k** The k attributes with highest weights are selected. k is specified by the k parameter.
- **bottom_k** The k attributes with lowest weights are selected. k is specified by the k parameter.
- **all_but_top_k** All attributes other than the k attributes with highest weights are selected. k is specified by the k parameter.
- **all_but_bottom_k** All attributes other than k attributes with lowest weights are selected. k is specified by the k parameter.
- **top_p** The top p percent attributes with highest weights are selected. p
is specified by the \( p \) parameter.

- **bottom \( p\% \)** The bottom \( p \) percent attributes with lowest weights are selected. \( p \) is specified by the \( p \) parameter.

**weight** This parameter is available only when the weight relation parameter is set to 'greater', 'greater equals', 'equals', 'less equals' or 'less'. This parameter is used to compare weights.

**k** This parameter is available only when the weight relation parameter is set to 'top k', 'bottom k', 'all but top k' or 'all but bottom k'. It is used to count the number of attributes to select.

**p** This parameter is available only when the weight relation parameter is set to 'top \( p\% \)' or 'bottom \( p\% \)'. It is used to specify the percentage of attributes to select.

**deselect unknown** This is an expert parameter. This parameter indicates if attributes whose weight is unknown should be removed from the ExampleSet.

**use absolute weights** This is an expert parameter. This parameter indicates if the absolute values of the weights should be used for comparison.

### Tutorial Processes

#### Selecting attributes from Sonar data set

The 'Sonar' data set is loaded using the Retrieve operator. The Weight by Correlation operator is applied on it to generate attribute weights. A breakpoint is inserted here. You can see the attributes with their weights here. The Select by Weights operator is applied next. The 'Sonar' data set is provided at the exampleset port and weights calculated by the Weight by Correlation operator are provided at the weights input port. The weight relation parameter is set to 'bottom k' and the \( k \) parameter is set to 4. Thus 4 attributes with minimum weights are selected. As you can see the 'attribute_57', 'attribute_17', 'attribute_30' and 'attribute_16' have lowest weights, thus these four attributes are selected. Also note that the label attribute 'class' is also selected. This is because the attributes
with special roles are selected irrespective of weights condition.

### Remove Attribute Range

This operator removes a range of attributes from the given Example-Set.

**Description**

The Remove Attribute Range operator removes the attributes within the specified range. The first and last attribute of the range are specified by the *first attribute* and *last attribute* parameters. All attributes in this range (including first and last attribute) will be removed from the ExampleSet. It is important to note that the attribute range starts from 1. This is a little different from the way attributes are counted in the Table Index where counting starts from 0. So, first and last attributes should be specified carefully.

**Differentiation**

- **Select Attributes** Provides a lot of options for selecting desired attributes e.g. on the basis of type, block, numerical value even regular expressions. See page 455 for details.
- **Remove Correlated Attributes** Selects attributes on the basis of correlations
of the attributes. See page 472 for details.

**Remove Useless Attributes** Selects attributes on the basis of usefulness. Different usefulness measures are available e.g. numerical attributes with minimum deviation etc. See page 468 for details.

### Input Ports

**example set input (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output (exa)** The ExampleSet with selected attributes removed from the original ExampleSet is output of this port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**first attribute (integer)** The first attribute of the attribute range which should be removed is specified through this parameter. The counting of attributes starts from 1.

**last attribute (integer)** The last attribute of the attribute range which should be removed is specified through this parameter. The counting of attributes starts from 1.
6.3. Attribute Set Reduction and Transformation

Related Documents

Select Attributes (455)
Remove Correlated Attributes (472)
Remove Useless Attributes (468)

Tutorial Processes

Removing the first two attributes of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the Table Index of the Outlook attribute is 0. The Table Index column can be seen if the Show column 'Table Index' option is selected in the Meta Data View tab. The Table Index of the Temperature attribute is 1. The Remove Attribute Range operator is applied on the 'Golf' data set to remove the first two attributes. The first attribute and second attribute parameters are set to 1 and 2 respectively to remove the first two attributes. The first attribute and second attribute parameters were not set to 0 and 1 respectively because here attribute counting starts from 1 (instead of 0). The resultant ExampleSet can be seen in the Results Workspace. You can see that the Outlook and Temperature attributes have been removed from the ExampleSet.
Remove Useless Attributes

This operator removes useless attributes from an ExampleSet. The thresholds for useless attributes are specified by the user.

Description

The Remove Useless Attributes operator removes four kinds of useless attributes:

1. Such nominal attributes where the most frequent value is contained in more than the specified ratio of all examples. The ratio is specified by the nominal useless above parameter. This ratio is defined as the number of examples with most frequent attribute value divided by the total number of examples. This property can be used for removing such nominal attributes where one value dominates all other values.

2. Such nominal attributes where the most frequent value is contained in less than the specified ratio of all examples. The ratio is specified by the nominal useless below parameter. This ratio is defined as the number of examples with most frequent attribute value divided by the total number of examples. This property can be used for removing nominal attributes with too many possible values.

3. Such numerical attributes where the Standard Deviation is less than or equal to a given deviation threshold. The numerical min deviation parameter specifies the deviation threshold. The Standard Deviation is a measure of how spread out values are. Standard Deviation is the square root of the Variance which is defined as the average of the squared differences from the Mean.

4. Such nominal attributes where the value of all examples is unique. This property can be used to remove id-like attributes.

Please note that this is not an intelligent operator i.e. it cannot figure out at its own whether an attribute is useless or not. It simply removes those attributes.
that satisfy the criteria for uselessness defined by the user.

Input Ports

.example set input (exa) This input port expects an ExampleSet. It is the output of the Filter Examples operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

.example set output (exa) The attributes that satisfy the user-defined criteria for useless attributes are removed from the ExampleSet and this ExampleSet is delivered through this output port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

.numerical min deviation (real) The numerical min deviation parameter specifies the deviation threshold. Such numerical attributes where Standard Deviation is less than or equal to this deviation threshold are removed from the input ExampleSet. The Standard Deviation is a measure of how spread out values are. Standard Deviation is the square root of the Variance which is defined as the average of the squared differences from the Mean.

.nominal useless above (real) The nominal useless above parameter specifies the ratio of the number of examples with most frequent value to the total number of examples. Such nominal attributes where the ratio of the number of examples with most frequent value to the total number of examples is more than this ratio are removed from the input ExampleSet. This property can be used to remove
such nominal attributes where one value dominates all other values.

**nominal remove id like (boolean)** If this parameter is set to true, all such nominal attributes where the value of all examples is unique are removed from the input ExampleSet. This property can be used to remove id-like attributes.

**nominal useless below (real)** The nominal useless below parameter specifies the ratio of the number of examples with most frequent value to the total number of examples. Such nominal attributes where the ratio of the number of examples with most frequent value to the total number of examples is less than this ratio are removed from the input ExampleSet. This property can be used to remove nominal attributes with too many possible values.

### Tutorial Processes

**Removing useless nominal attributes from an ExampleSet**

This Example Process explains how the **nominal useless above** and **nominal useless below** parameters can be used to remove useless nominal attributes. Please keep in mind that the Remove Useless Attributes operator removes those attributes that satisfy the user-defined criteria for useless attributes.

The 'Golf' data set is loaded using the Retrieve operator. The Filter Examples operator is applied on it to filter the first 10 examples. This is done to just simplify the calculations for understanding this process. A **breakpoint** is inserted after the Filter Examples operator so that you can see the ExampleSet before application of the Remove Useless Attributes operator. You can see that the ExampleSet has 10 examples. There are 2 regular nominal attributes: 'Outlook' and 'Wind'. The most frequent values in the 'Outlook' attribute are 'rain' and 'sunny', they occur in 4 out of 10 examples. Thus their ratio is 0.4. The most frequent value in the 'Wind' attribute is 'false', it occurs in 7 out of 10 examples. Thus its ratio is 0.7.

The Remove Useless Attributes operator is applied on the ExampleSet. The **nominal useless above** parameter is set to 0.6. Thus attributes where the ratio of
6.3. Attribute Set Reduction and Transformation

most frequent value to total number of examples is above 0.6 are removed from the ExampleSet. As the ratio of most frequent value in the Wind attribute is greater than 0.6, it is removed from the ExampleSet.

The *nominal useless below* parameter is set to 0.5. Thus attributes where the ratio of most frequent value to total number of examples is below 0.5 are removed from the ExampleSet. As the ratio of most frequent value in the Outlook attribute is below 0.5, it is removed from the ExampleSet.

This can be verified by seeing the results in the Results Workspace.

**Removing useless numerical attributes from an ExampleSet**

This Example Process explains how the *numerical min deviation* parameter can be used to remove useless numerical attributes. The *numerical min deviation* parameter specifies the deviation threshold. Such numerical attributes where the Standard Deviation is less than or equal to this deviation threshold are removed from the input ExampleSet. The Standard Deviation is a measure of how spread out values are. Standard Deviation is the square root of the Variance which is defined as the average of the squared differences from the Mean. Please keep in mind that the Remove Useless Attributes operator removes those attributes that satisfy the user-defined criteria for useless attributes.

The 'Golf' data set is loaded using the Retrieve operator. The Filter Examples operator is applied on it to filter the first 10 examples. This is done to just simplify
6. Data Transformation

the calculations for understanding this process. A breakpoint is inserted after the Filter Examples operator so that you see the ExampleSet before application of the Remove Useless Attributes operator. You can see that it has 10 examples. There are 2 regular numerical attributes: 'Temperature' and 'Humidity'. The Aggregate operator is applied on the ExampleSet to calculate and display the Standard Deviations of both numerical attributes. This operator is inserted here so that you can see that Standard Deviations without actually calculating them, otherwise this operator is not required here. You can see that the Standard Deviation of the 'Temperature' and 'Humidity' attributes is 7.400 and 10.682 respectively.

The Remove Useless Attributes operator is applied on the original ExampleSet (the ExampleSet with the first 10 examples of the 'Golf' data set). The numerical min deviation parameter is set to 9.0. Thus the numerical attributes where the Standard Deviation is less than 9.0 are removed from the ExampleSet. As the Standard Deviation of the Temperature attribute is less than 9.0, it is removed from the ExampleSet.

This can be verified by seeing the results in the Results Workspace.

Remove Correlated Attributes

This operator removes correlated attributes from an ExampleSet. The
correlation threshold is specified by the user. Correlation is a statistical technique that can show whether and how strongly pairs of attributes are related.

Description

A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

Suppose we have two attributes X and Y, with means $X'$ and $Y'$ respectively and standard deviations $S(X)$ and $S(Y)$ respectively. The correlation is computed as summation from 1 to n of the product $(X(i)-X')(Y(i)-Y')$ and then dividing this summation by the product $(n-1).S(X).S(Y)$ where $n$ is the total number of examples and $i$ is the increment variable of summation. There can be other formulas and definitions but let us stick to this one for simplicity.

As discussed earlier a positive value for the correlation implies a positive association. Suppose that an X value was above average, and that the associated Y value was also above average. Then the product $(X(i)-X')(Y(i)-Y')$ would be the product of two positive numbers which would be positive. If the X value and the Y value were both below average, then the product above would be of two negative numbers, which would also be positive. Therefore, a positive correlation is evidence of a general tendency that large values of X are associated with large values of Y and small values of X are associated with small values of Y.

As discussed earlier a negative value for the correlation implies a negative or inverse association. Suppose that an X value was above average, and that the associated Y value was instead below average. Then the product $(X(i)-X')(Y(i)-Y')$ would be the product of a positive and a negative number which would make the product negative. If the X value was below average and the Y value was above
6. Data Transformation

average, then the product above would also be negative. Therefore, a negative correlation is evidence of a general tendency that large values of X are associated with small values of Y and small values of X are associated with large values of Y.

This operator can be used for removing correlated or uncorrelated attributes depending on the setting of parameters specially the filter relation parameter. The procedure is quadratic in number of attributes i.e. for m attributes an m x m matrix of correlations is calculated. Please note that this operator might fail in some cases when the attributes should be filtered out. For example, it might not be able to remove for example all negative correlated attributes because for the complete m x m - matrix of correlation the correlations will not be recalculated and hence not checked if one of the attributes of the current pair was already marked for removal. This means that for three attributes X, Y, and Z that it might be that Y was already ruled out by the negative correlation with X and is now not able to rule out Z any longer. The used correlation function in this operator is the Pearson correlation. In order to get more stable results the original, random, and reverse order of attributes is available.

Correlated attributes are usually removed because they are similar in behavior and will have similar impact in prediction calculations, so keeping attributes with similar impacts is redundant. Removing correlated attributes saves space and time of calculation of complex algorithms. Moreover, it also makes processes easier to design, analyze, understand and comprehend.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Filter Examples operator in the attached Example Process. The output of other operators can also be used as input.
Output Ports

**example set output** *(exa)* The (un-)correlated attributes are removed from the ExampleSet and this ExampleSet is delivered through this output port.

**original** *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**correlation** *(real)* This parameter specifies the correlation for filtering attributes. A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

**filter relation** *(selection)* Correlations of two attributes are compared at a time. One of the two attributes is removed if their correlation fulfills the relation specified by this parameter.

**attribute order** *(selection)* The algorithm takes this attribute order to calculate correlations and for filtering the attributes.

**use absolute correlation** *(boolean)* This parameter indicates if the absolute value of the correlations should be used for comparison.

Tutorial Processes

Removing correlated attributes from the Sonar data set
6. Data Transformation

The 'Sonar' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can view the ExampleSet before further operators are applied on it. You can see that the 'Sonar' data set has 60 numerical attributes. The Correlation Matrix operator is applied on it. This operator is applied so that you can view the correlation matrix of the 'Sonar' data set otherwise this operator was not required here. The Remove Correlated Attributes operator is applied on the 'Sonar' data set. The *correlation* parameter is set to 0.8. The *filter relation* parameter is set to 'greater' and the *attribute order* parameter is set to 'original'. Run the process and you will see in the Results Workspace that 19 out of 60 numerical attributes of the 'Sonar' data set have been removed. Now have a look at the correlation matrix generated by the Correlation Matrix operator. You can see that most of the attributes with correlations above 0.8 have been removed from the data set. Some such attributes are not removed because this operator might fail in some cases when the attributes should be filtered out. It might not be able to remove all correlated attributes because for the complete $m \times m$ matrix of correlation the correlations will not be recalculated and hence not checked if one of the attributes of the current pair was already marked for removal. Change the value of the *attribute order* parameter to 'random' and run the process again. Compare these results with the previous ones. This time a different set of attributes is removed from the data set. So, the order in which correlation operator is applied may change the output.
6.3. Attribute Set Reduction and Transformation

Work on Subset

This operator selects a subset (one or more attributes) of the input ExampleSet and applies the operators in its subprocess on the selected subset.

Description

The Work on the Subset operator can be considered as the blend of the Select Attributes and Subprocess operator to some extent. The attributes are selected in the same way as selected by the Select Attributes operator and the subprocess of this operator works in the same way as the Subprocess operator works. A subprocess can be considered as a small unit of a process where all operators and a combination of operators can be applied in a subprocess. That is why a subprocess can also be defined as a chain of operators that is subsequently applied. For more information about subprocess please study the Subprocess operator. Although the Work on Subset operator has similarities with the Select Attributes and Subprocess operators however, this operator provides some functionality that cannot be performed by the combination of the Select Attributes and Subprocess operator. Most importantly, this operator can merge the results of its subprocess with the input ExampleSet such that the original subset is overwritten by the subset received after processing of the subset in the subprocess. This merging can be controlled by the keep subset only parameter. This parameter is set to false by default. Thus merging is done by default. If this parameter is set to true, then only the result of the subprocess is returned by this operator and no merging is done. In such a case this operator behaves very similar to the combination of the Select Attributes and Subprocess operator. This can be understood easily by studying the attached Example Process.

This operator can also deliver the additional results of the subprocess if desired. This can be controlled by the deliver inner results parameter. Please note that this is a very powerful operator. It can be used to create new preprocessing schemes by combining it with other preprocessing operators. However, there are
6. Data Transformation

two major restrictions:

- Since the result of the subprocess will be combined with the rest of the input ExampleSet, the number of examples is not allowed to be changed inside the subprocess.

- The changes in the role of an attribute will not be delivered outside the subprocess.

Input Ports

d**example set (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

d**example set (exa)** The result of the subprocess will be combined with the rest of the input ExampleSet and delivered through this port. However if the *keep subset only* parameter is set to true then only the result of the subprocess will be delivered.

**through (thr)** This operator can also deliver the additional results of the subprocess if desired. This can be controlled by the *deliver inner results* parameter. This port is used for delivering the additional results of the subprocess. The Work on Subset operator can have multiple *through* ports. When one *through* port is connected, another *through* port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The object passed at the first *through* port inside the subprocess of the Work on Subset operator is delivered at the first *through* port of the operator.
Parameters

attribute filter type *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter *(attribute)* becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters *(regular expression, use except expression)* become visible in the Parameter View.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters *(value type, use value type exception)* become visible in the Parameter View.
- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters *(block type, *
6. Data Transformation

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list, which is the list of selected attributes.

**regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the *regular expression* parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except
6.3. Attribute Set Reduction and Transformation

(value type) becomes visible in the Parameter View.

except value type (selection) The attributes matching this type will not be selected even if they match the previously mentioned type i.e. value type parameter's value.

block type (selection) The block type of attributes to be selected can be chosen from a drop down list.

use block type exception (boolean) If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

except block type (selection) The attributes matching this block type will not be selected even if they match the previously mentioned block type i.e. block type parameter's value.

numeric condition (string) The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles. Special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.
6. Data Transformation

**keep subset only (boolean)** The Work on Subset operator can merge the results of its subprocess with the input ExampleSet such that the original subset is overwritten by the subset received after processing of the subset in the subprocess. This merging can be controlled by the *keep subset only* parameter. This parameter is set to false by default. Thus merging is done by default. If this parameter is set to true, then only the result of the subprocess is returned by this operator and no merging is done.

**deliver inner results (boolean)** This parameter indicates if the additional results (other than the input ExampleSet) of the subprocess should also be returned. If this parameter is set to true then the additional results are delivered through the *through* ports.

### Tutorial Processes

#### Working on a subset of Golf data set

The 'Golf' data set is loaded using the Retrieve operator. Then the Work on Subset operator is applied on it. The *attribute filter type* parameter is set to *subset*. The *attributes* parameter is used for selecting the 'Temperature' and 'Humidity' attributes. Double-click on the Work on Subset operator to see its subprocess. All the operations in the subprocess will be performed only on the selected attributes i.e. the 'Temperature' and 'Humidity' attributes. The Normalize operator is applied in the subprocess. The *attribute filter type* parameter of the Normalize operator is set to *all*. Please note that the Normalize operator will not be applied on 'all' the attributes of the input ExampleSet rather it would be applied on 'all' selected attributes of the input ExampleSet i.e. the 'Temperature' and 'Humidity' attributes. Run the process. You will see that the normalized 'Humidity' and 'Temperature' attribute are combined with the rest of the input ExampleSet. Now set the *keep subset only* parameter to true and run the process again. Now you will see that only the results of the subprocess are delivered by the Work on Subset operator. This Example Process just explains the basic usage of this operator. This operator can be used for creating new
preprocessing schemes by combining it with other preprocessing operators.

Forward Selection

This operator selects the most relevant attributes of the given ExampleSet through a highly efficient implementation of the forward selection scheme.

Description

The Forward Selection operator is a nested operator i.e. it has a subprocess. The subprocess of the Forward Selection operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator.

The Forward Selection operator starts with an empty selection of attributes and, in each round, it adds each unused attribute of the given ExampleSet. For each added attribute, the performance is estimated using the inner operators, e.g. a cross-validation. Only the attribute giving the highest increase of performance is added to the selection. Then a new round is started with the modified selec-
6. Data Transformation

tion. This implementation avoids any additional memory consumption besides the memory used originally for storing the data and the memory which might be needed for applying the inner operators. The stopping behavior parameter specifies when the iteration should be aborted. There are three different options:

- without increase: The iteration runs as long as there is any increase in performance.
- without increase of at least: The iteration runs as long as the increase is at least as high as specified, either relative or absolute. The minimal relative increase parameter is used for specifying the minimal relative increase if the use relative increase parameter is set to true. Otherwise, the minimal absolute increase parameter is used for specifying the minimal absolute increase.
- without significant increase: The iteration stops as soon as the increase is not significant to the level specified by the alpha parameter.

The speculative rounds parameter defines how many rounds will be performed in a row, after the first time the stopping criterion is fulfilled. If the performance increases again during the speculative rounds, the selection will be continued. Otherwise all additionally selected attributes will be removed, as if no speculative rounds had executed. This might help avoiding getting stuck in local optima.

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

Differentiation

Backward Elimination The Backward Elimination operator starts with the full set of attributes and, in each round, it removes each remaining attribute of
the given ExampleSet. For each removed attribute, the performance is estimated
using the inner operators, e.g. a cross-validation. Only the attribute giving the
least decrease of performance is finally removed from the selection. Then a new
round is started with the modified selection. See page 488 for details.

Input Ports

example set (exa) This input port expects an ExampleSet. This ExampleSet is
available at the first port of the nested chain (inside the subprocess) for processing
in the subprocess.

Output Ports

example set (exa) The feature selection algorithm is applied on the input Exam-
pleSet. The resultant ExampleSet with reduced attributes is delivered through
this port.
attribute weights (att) The attribute weights are delivered through this port.
performance (per) This port delivers the Performance Vector for the selected
attributes. A Performance Vector is a list of performance criteria values.

Parameters

maximal number of attributes (integer) This parameter specifies the maximal
number of attributes to be selected through Forward Selections.
speculative rounds (integer) This parameter specifies the number of times,
the stopping criterion might be consecutively ignored before the elimination is
actually stopped. A number higher than one might help avoiding getting stuck
in local optima.
stopping behavior (selection) The stopping behavior parameter specifies when
the iteration should be aborted. There are three different options:
6. Data Transformation

- **without_increase** The iteration runs as long as there is any increase in performance.

- **without_increase_of_at_least** The iteration runs as long as the increase is at least as high as specified, either relative or absolute. The *minimal relative increase* parameter is used for specifying the minimal relative increase if the *use relative increase* parameter is set to true. Otherwise, the *minimal absolute increase* parameter is used for specifying the minimal absolute increase.

- **without_significant_increase** The iteration stops as soon as the increase is not significant to the level specified by the *alpha* parameter.

**use relative increase** *(boolean)* This parameter is available only when the *stopping behavior* parameter is set to 'without increase of at least'. If the *use relative increase* parameter is set to true the *minimal relative increase* parameter will be used otherwise the *minimal absolute increase* parameter will be used.

**minimal absolute increase** *(real)* This parameter is available only when the *stopping behavior* parameter is set to 'without increase of at least' and the *use relative increase* parameter is set to false. If the absolute performance increase to the last step drops below this threshold, the selection will be stopped.

**minimal relative increase** *(real)* This parameter is available only when the *stopping behavior* parameter is set to 'without increase of at least' and the *use relative increase* parameter is set to true. If the relative performance increase to the last step drops below this threshold, the selection will be stopped.

**alpha** *(real)* This parameter is available only when the *stopping behavior* parameter is set to 'without significant increase'. This parameter specifies the probability threshold which determines if differences are considered as significant.

Related Documents

Backward Elimination (488)
6.3. Attribute Set Reduction and Transformation

**Tutorial Processes**

**Feature reduction of the Polynomial data set through Forward Selection**

The 'Polynomial' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other than the *label* attribute. The Forward Selection operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature reduction algorithm. Have a look at the subprocess of this operator. The X-Validation operator is used there which itself is a nested operator. Have a look at the subprocesses of the X-Validation operator. The K-NN operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 3 attributes.

![Diagram of the process](image)
Backward Elimination

This operator selects the most relevant attributes of the given ExampleSet through an efficient implementation of the backward elimination scheme.

Description

The Backward Elimination operator is a nested operator i.e. it has a subprocess. The subprocess of the Backward Elimination operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator.

The Backward Elimination operator starts with the full set of attributes and, in each round, it removes each remaining attribute of the given ExampleSet. For each removed attribute, the performance is estimated using the inner operators, e.g. a cross-validation. Only the attribute giving the least decrease of performance is finally removed from the selection. Then a new round is started with the modified selection. This implementation avoids any additional memory consumption besides the memory used originally for storing the data and the memory which might be needed for applying the inner operators. The stopping behavior parameter specifies when the iteration should be aborted. There are three different options:

- with decrease: The iteration runs as long as there is any increase in performance.
- with decrease of more than: The iteration runs as long as the decrease is less than the specified threshold, either relative or absolute. The maximal relative decrease parameter is used for specifying the maximal relative decrease if the use relative decrease parameter is set to true. Otherwise, the maximal absolute decrease parameter is used for specifying the maximal absolute decrease.
6.3. Attribute Set Reduction and Transformation

- with significant decrease: The iteration stops as soon as the decrease is significant to the level specified by the alpha parameter.

The speculative rounds parameter defines how many rounds will be performed in a row, after the first time the stopping criterion is fulfilled. If the performance increases again during the speculative rounds, the elimination will be continued. Otherwise all additionally eliminated attributes will be restored, as if no speculative rounds had executed. This might help avoiding getting stuck in local optima.

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

Differentiation

Forward Selection The Forward Selection operator starts with an empty selection of attributes and, in each round, it adds each unused attribute of the given ExampleSet. For each added attribute, the performance is estimated using the inner operators, e.g. a cross-validation. Only the attribute giving the highest increase of performance is added to the selection. Then a new round is started with the modified selection. See page 483 for details.

Input Ports

example set (exa) This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.
Output Ports

**example set (exa)** The feature selection algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**attribute weights (att)** The attribute weights are delivered through this port.

**performance (per)** This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

Parameters

**maximal number of eliminations (integer)** This parameter specifies the maximal number of backward eliminations.

**speculative rounds (integer)** This parameter specifies the number of times, the stopping criterion might be consecutively ignored before the elimination is actually stopped. A number higher than one might help avoiding getting stuck in local optima.

**stopping behavior (selection)** The stopping behavior parameter specifies when the iteration should be aborted. There are three different options:

- **with_decrease** The iteration runs as long as there is any increase in performance.

- **with_decrease_of_more_than** The iteration runs as long as the decrease is less than the specified threshold, either relative or absolute. The *maximal relative decrease* parameter is used for specifying the maximal relative decrease if the *use relative decrease* parameter is set to true. Otherwise, the *maximal absolute decrease* parameter is used for specifying the maximal absolute decrease.

- **with_significant_decrease** The iteration stops as soon as the decrease is significant to the level specified by the *alpha* parameter.
6.3. Attribute Set Reduction and Transformation

**use relative decrease (boolean)** This parameter is only available when the *stopping behavior* parameter is set to 'with decrease of more than'. If the *use relative decrease* parameter is set to true the *maximal relative decrease* parameter will be used otherwise the *maximal absolute decrease* parameter.

**maximal absolute decrease (real)** This parameter is only available when the *stopping behavior* parameter is set to 'with decrease of more than' and the *use relative decrease* parameter is set to false. If the absolute performance decrease to the last step exceeds this threshold, the elimination will be stopped.

**maximal relative decrease (real)** This parameter is only available when the *stopping behavior* parameter is set to 'with decrease of more than' and the *use relative decrease* parameter is set to true. If the relative performance decrease to the last step exceeds this threshold, the elimination will be stopped.

**alpha (real)** This parameter is only available when the *stopping behavior* parameter is set to 'with significant decrease'. This parameter specifies the probability threshold which determines if differences are considered as significant.

**Related Documents**

Forward Selection (483)

**Tutorial Processes**

**Feature reduction of the Polynomial data set**

The 'Polynomial' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the *label* attribute. The Backward Elimination operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature
6. Data Transformation

reduction algorithm. Have a look at the subprocess of this operator. The X-
Validation operator is used there which itself is a nested operator. Have a look
at the subprocesses of the X-Validation operator. The K-NN operator is used in
the 'Training' subprocess to train a model. The trained model is applied using
the Apply Model operator in the 'Testing' subprocess. The performance is mea-
sured through the Performance operator and the resultant performance vector
is used by the underlying algorithm. Run the process and switch to the Results
Workspace. You can see that the ExampleSet that had 5 attributes has now been
reduced to 3 attributes.

![Diagram of the process](image)

Optimize Selection

This operator selects the most relevant attributes of the given ExampleSet. Two deterministic greedy feature selection algorithms 'forward
selection' and 'backward elimination' are used for feature selection.

Description

Feature selection i.e. the question for the most relevant features for classification
or regression problems, is one of the main data mining tasks. A wide range
of search methods have been integrated into RapidMiner including evolutionary
algorithms. For all search methods we need a performance measurement which
indicates how well a search point (a feature subset) will probably perform on the
given data set.
A deterministic algorithm is an algorithm which, in informal terms, behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states.

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. On some problems, a greedy strategy may not produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution.

This operator realizes the two deterministic greedy feature selection algorithms 'forward selection' and 'backward elimination'. However, we have added some enhancements to the standard algorithms which are described below:

**Forward Selection**

1. Create an initial population with \( n \) individuals where \( n \) is the number of attributes in the input ExampleSet. Each individual will use exactly one of the features.

2. Evaluate the attribute sets and select only the best \( k \).

3. For each of the \( k \) attribute sets do: If there are \( j \) unused attributes, make \( j \) copies of the attribute set and add exactly one of the previously unused attributes to the attribute set.

4. As long as the performance improved in the last \( p \) iterations go to step 2

**Backward Elimination**

1. Start with an attribute set which uses all features.
6. Data Transformation

2. Evaluate all attribute sets and select the best $k$.

3. For each of the $k$ attribute sets do: If there are $j$ attributes used, make $j$ copies of the attribute set and remove exactly one of the previously used attributes from the attribute set.

4. As long as the performance improved in the last $p$ iterations go to step 2.

The parameter $k$ can be specified by the `keep best` parameter, the parameter $p$ can be specified by the `generations without improv` parameter. These parameters have default values 1 which means that the standard selection algorithms are used. Using other values increases the runtime but might help to avoid local extrema in the search for the global optimum.

Another unusual parameter is the `maximum number of generations` parameter. This parameter bounds the number of iterations to this maximum of feature selections / de-selections. In combination with the `generations without improv` parameter, this allows several different selection schemes (which are described for forward selection, backward elimination works analogous):

- **maximum number of generations = m and generations without improv = p:**

  Selects maximal $m$ features. The selection stops if no performance improvement was measured in the last $p$ generations.

- **maximum number of generations = -1 and generations without improv = p:**

  Tries to selects new features until no performance improvement was measured in the last $p$ generations.
maximum number of generations = m and generations without improvement = -1:

Selects maximal $m$ features. The selection stops is not stopped until all combinations with maximal $m$ were tried. However, the result might contain fewer features than these.

maximum number of generations = -1 and generations without improvement = -1:

Test all combinations of attributes (brute force, this might take a very long time and should only be applied to small attribute sets).

**Differentiation**

*Optimize Selection (Evolutionary)* This is also an attribute set reduction operator but it uses a genetic algorithm for this purpose. See page 499 for details.

**Input Ports**

*example set in (exa)* This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

*through (thr)* This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the
first through port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected the right number of ports at the subprocess level.

Output Ports

example set out (exa) The feature selection algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.
weights (wei) The attribute weights are delivered through this port.
performance (per) This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

Parameters

selection direction (selection) This parameter specifies which of the 'forward selection' and 'backward elimination' algorithms should be used.
limit generations without improvement (boolean) This parameter indicates if the optimization should be aborted if this number of generations showed no improvement. If unchecked, always the maximal number of generations will be used.
generations without improvement (integer) This parameter is only available when the limit generations without improvement parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after \( n \) generations without improvement in the performance. \( n \) is specified by this parameter.
limit number of generations (boolean) This parameter indicates if the number of generations should be limited to a specific number.
keep best (integer) The best \( n \) individuals are kept in each generation where \( n \) is the value of this parameter.
maximum number of generations (integer) This parameter is only available when the limit number of generations parameter is set to true. This parameter specifies the number of generations after which the algorithm should be terminated.
6.3. Attribute Set Reduction and Transformation

**normalize weights (boolean)** This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed (boolean)** This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

**local random seed (integer)** This parameter specifies the local random seed and is only available if the use local random seed parameter is set to true.

**show stop dialog (boolean)** This parameter determines if a dialog with a stop button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**user result individual selection (boolean)** If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter (boolean)** This parameter determines if the current population should be displayed in the performance space.

**plot generations (integer)** This parameter is only available when the show population plotter parameter is set to true. The population plotter is updated in these generations.

**constraint draw range (boolean)** This parameter is only available when the show population plotter parameter is set to true. This parameter determines if the draw range of the population plotter should be constrained between 0 and 1.

**draw dominated points (boolean)** This parameter is only available when the show population plotter parameter is set to true. This parameter determines if only points which are not Pareto dominated should be drawn on the population plotter.

**population criteria data file (filename)** This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness (real)** This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.
6. Data Transformation

Related Documents

Optimize Selection (Evolutionary) (499)

Tutorial Processes

Feature reduction of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other then the \textit{label} attribute. The Optimize Selection operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying feature reduction algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 2 attributes.
Optimize Selection (Evolutionary)

This operator selects the most relevant attributes of the given ExampleSet. A Genetic Algorithm is used for feature selection.

Description

Feature selection i.e. the question for the most relevant features for classification or regression problems, is one of the main data mining tasks. A wide range of search methods have been integrated into RapidMiner including evolutionary algorithms. For all search methods we need a performance measurement which indicates how well a search point (a feature subset) will probably perform on the given data set.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

In genetic algorithm for feature selection 'mutation' means switching features on and off and 'crossover' means interchanging used features. Selection is done by the specified selection scheme which is selected by the selection scheme parameter. A genetic algorithm works as follows:

Generate an initial population consisting of $p$ individuals. Each attribute is switched on with probability $p_i$. The numbers $p$ and $p_i$ can be adjusted by the population size and $p$ initialize parameters respectively.

For all individuals in the population

1. Perform mutation, i.e. set used attributes to unused with probability $p_m$ and vice versa. The probability $p_m$ can be adjusted by the $p$ mutation parameter.
6. Data Transformation

2. Choose two individuals from the population and perform crossover with probability \( p_c \). The probability \( p_c \) can be adjusted by the \( p \) crossover parameter. The type of crossover can be selected by the crossover type parameter.

3. Perform selection, map all individuals according to their fitness and draw \( p \) individuals at random according to their probability where \( p \) is the population size which can be adjusted by the population size parameter.

4. As long as the fitness improves, go to step number 2.

If the ExampleSet contains value series attributes with block numbers, the whole block will be switched on and off. Exact, minimum or maximum number of attributes in combinations to be tested can be specified by the appropriate parameters. Many other options are also available for this operator. Please study the parameters section for more information.

Input Ports

**example set in (exa)** This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**attribute weights in (att)** This port expects attribute weights. It is not compulsory to use this port.

**through (thr)** This operator can have multiple through ports. When one input is connected with the through port, another through port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first through port of this operator is available at the first through port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected right number of ports at subprocess level.
Output Ports

example set out (exa) The genetic algorithm is applied on the input Example-Set. The resultant ExampleSet with reduced attributes is delivered through this port.
weights (wei) The attribute weights are delivered through this port.
performance (per) This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

Parameters

use exact number of attributes (boolean) This parameter determines if only combinations containing exact numbers of attributes should be tested. The exact number is specified by the exact number of attributes parameter.
exact number of attributes (integer) This parameter is only available when the use exact number of attributes parameter is set to true. Only combinations containing this numbers of attributes would be generated and tested.
restrict maximum (boolean) If set to true, the maximum number of attributes whose combinations will be generated and tested can be restricted. Otherwise all combinations of all attributes are generated and tested. This parameter is only available when the use exact number of attributes parameter is set to true.
min of attributes (integer) This parameter determines the minimum number of features used for the combinations to be generated and tested.
max number of attributes (integer) This parameter determines the maximum number of features used for the combinations to be generated and tested. This parameter is only available when the restrict maximum parameter is set to true.
population size (integer) This parameter specifies the population size i.e. the number of individuals per generation.
maximum number of generations (integer) This parameter specifies the number of generations after which the algorithm should be terminated.
use early stopping (boolean) This parameter enables early stopping. If not set to true, always the maximum number of generations are performed.
6. Data Transformation

generations without improvement \((\text{integer})\) This parameter is only available when the use early stopping parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after \(n\) generations without improvement in the performance. \(n\) is specified by this parameter.

normalize weights \((\text{boolean})\) This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

use local random seed \((\text{boolean})\) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

local random seed \((\text{integer})\) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

show stop dialog \((\text{boolean})\) This parameter determines if a dialog with a stop button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

user result individual selection \((\text{boolean})\) If this parameter is set to true, it allows the user to select the final result individual from the last population.

show population plotter \((\text{boolean})\) This parameter determines if the current population should be displayed in performance space.

plot generations \((\text{integer})\) This parameter is only available when the show population plotter parameter is set to true. The population plotter is updated in these generations.

constraint draw range \((\text{boolean})\) This parameter is only available when the show population plotter parameter is set to true. This parameter determines if the draw range of the population plotter should be constrained between 0 and 1.

draw dominated points \((\text{boolean})\) This parameter is only available when the show population plotter parameter is set to true. This parameter determines if only points which are not Pareto dominated should be drawn on the population plotter.

population criteria data file \((\text{filename})\) This parameter specifies the path to the file in which the criteria data of the final population should be saved.

maximal fitness \((\text{real})\) This parameter specifies the maximal fitness. The opti-
6.3. Attribute Set Reduction and Transformation

Optimization will stop if the fitness reaches this value.

**selection scheme** *(selection)* This parameter specifies the selection scheme of this evolutionary algorithms.

**tournament size** *(real)* This parameter is only available when the *selection scheme* parameter is set to 'tournament'. It specifies the fraction of the current population which should be used as tournament members.

**start temperature** *(real)* This parameter is only available when the *selection scheme* parameter is set to 'Boltzmann'. It specifies the scaling temperature.

**dynamic selection pressure** *(boolean)* This parameter is only available when the *selection scheme* parameter is set to 'Boltzmann' or 'tournament'. If set to true the selection pressure is increased to maximum during the complete optimization run.

**keep best individual** *(boolean)* If set to true, the best individual of each generations is guaranteed to be selected for the next generation.

**save intermediate weights** *(boolean)* This parameter determines if the intermediate best results should be saved.

**intermediate weights generations** *(integer)* This parameter is only available when the *save intermediate weights* parameter is set to true. The intermediate best results would be saved every $k$ generations where $k$ is specified by this parameter.

**intermediate weights file** *(filename)* This parameter specifies the file into which the intermediate weights should be saved.

**p initialize** *(real)* The initial probability for an attribute to be switched on is specified by this parameter.

**p mutation** *(real)* The probability for an attribute to be changed is specified by this parameter. If set to -1, the probability will be set to $1/n$ where $n$ is the total number of attributes.

**p crossover** *(real)* The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type** *(selection)* The type of the crossover can be selected by this parameter.
6. Data Transformation

Tutorial Processes

Feature reduction of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other than the label attribute. The Optimize Selection (Evolutionary) operator is applied on the ExampleSet it is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 3 attributes.
This operator sets the value of one or more attributes of the specified example.

Description

The Set Data operator sets the value of one or more attributes of the specified example of the input ExampleSet. The example is specified by the example index parameter. The attribute name parameter specifies the attribute whose value is to be set. The value parameter specifies the new value. Values of other attributes of the same example can be set by the additional values parameter. Please note that the values should be consistent with the type of the attribute e.g. specifying a string value is not allowed for an integer type attribute.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

Output Ports

example set output (exa) The ExampleSet with new values of the selected example's attributes is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

example index (integer) This parameter specifies the index of the example whose value should be set. Please note that counting starts from 1.
6. Data Transformation

attribute name (string) This parameter specifies the name of the attribute whose value should be set.

count backwards (boolean) If set to true, the counting order is reversed. The last example is addressed by index 1, the second last example is addressed by index 2 and so on.

value (string) This parameter specifies the new value of the selected attribute (selected by the attribute name parameter) of the specified example (specified by the example index parameter).

additional values The values of other attributes of the same example can be set by this parameter.

Tutorial Processes

Introduction to the Set Data operator

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the data set before application of the Set Data operator. You can see that the value of the Temperature and Wind attributes is '85' and 'false' respectively in the first example. The Set Data operator is applied on the 'Golf' data set. The example index parameter is set to 1, the attribute name parameter is set to 'Temperature' and the value parameter is set to 50. Thus the value of the Temperature attribute will be set to 50 in the first example. Similarly, the value of the Wind attribute in the first example is set to 'fast' using the additional values parameter. You can verify this by running the process and seeing the results in the Results Workspace. Please note that a string value cannot be set for the Temperature attribute because it is of integer type. An integer value can be specified for the Wind attribute (nominal type) but it will be stored as a nominal value.
6.4. Value Modification

Declare Missing Value

This operator declares the specified values of the selected attributes as missing values.

Description

The Declare Missing Value operator replaces the specified values of the selected attributes by Double.NaN, thus these values will become missing values. These values will be treated as missing values by the subsequent operators. The desired values can be selected through nominal, numeric or regular expression mode. This behavior can be controlled by the mode parameter.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.
6. Data Transformation

Output Ports

element set output \((exa)\) The specified values of the selected attributes are replaced by missing values and the resultant ExampleSet is delivered through this port.

original \((ori)\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type \((selection)\) This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter \((attribute)\) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters \((regular\ expression,\ use\ except\ expression)\) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular
6.4. Value Modification

type. It should be noted that types are hierarchical. For example real and integer types both belong to the numeric type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression (string)** The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.
6. Data Transformation

**use except expression (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' >', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** The special attributes are attributes with
6.4. Value Modification

special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

\textbf{invert selection (boolean)} If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

\textbf{mode (selection)} This parameter specifies the type of the value that should be set to missing value. The type can be nominal or numeric or it can be specified through a regular expression.

\textbf{numeric value (real)} This parameter specifies the numerical value that should be declared as missing value.

\textbf{nominal value (string)} This parameter specifies the nominal value that should be declared as missing value.

\textbf{expression value (string)} This parameter specifies the value that should be declared as missing value through a regular expression.

\section*{Tutorial Processes}

\subsection*{Declaring a nominal value as missing value}

The 'Golf' data set is loaded using the Retrieve operator. A \textit{breakpoint} is inserted here so that you can have a look at the ExampleSet. You can see that the 'Outlook' attribute has three possible values i.e. 'sunny', 'rain' and 'overcast'. The Declare Missing Value operator is applied on this ExampleSet to change the 'overcast' value of the 'Outlook' attribute to missing value. The \textit{attribute filter type} parameter is set to 'single' and the \textit{attribute} parameter is set to 'Outlook'. The \textit{mode} parameter is set to 'nominal' and the \textit{nominal value} parameter is set to 'overcast'. Run the process and compare the resultant ExampleSet with the original ExampleSet. You can clearly see that the value 'overcast' has been replaced.
6. Data Transformation

by missing values.

Normalize

This operator normalizes the attribute values of the selected attributes.

Description

Normalization is a preprocessing technique used to rescale attribute values to fit in a specific range. Normalization of the data is very important when dealing with attributes of different units and scales. For example, some data mining techniques use the Euclidean distance. Therefore, all attributes should have the same scale for a fair comparison between them. In other words normalization is a technique used to level the playing field when looking at attributes that widely vary in size as a result of the units selected for representation. This operator performs normalization of selected attributes. Four normalization methods are provided. These methods are explained in the parameters.
6.4. Value Modification

Input Ports

**example set (exa)** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for input because the attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

Output Ports

**example set (exa)** The ExampleSet with selected attributes in normalized form is output of this port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**preprocessing model (pre)** This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

**create view (boolean)** It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

**attribute filter type (selection)** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes that you want to normalize. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
6. Data Transformation

- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to the value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespec-
6.4. Value Modification

tive of the given numerical condition.

**attribute** *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the parameter attribute if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression** *(string)* Attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** *(boolean)* If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in regular expression parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from drop down list.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type** *(selection)* Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.

**block type** *(selection)* The Block type of the attributes to be selected can be chosen from a drop down list.

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.
6. Data Transformation

**except block type** *(selection)* Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes** *(boolean)* Special attributes are attributes with special roles. Special attributes are those attributes which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**method** *(selection)* Four methods are provided here for normalizing data. These methods are also explained in the attached Example Process.

- **z_transformation** This is also called Statistical normalization. The purpose of statistical normalization is to convert a data into Normal distribution with mean = 0 and variance = 1. The formula of statistical normalization is \( Z = \frac{(X-u)}{s} \). You have your attribute values as vector \( X \) then you subtract the mean of the attribute values, \( u \), and divide this difference by the standard deviation, you will get another vector \( Z \) that has normal distribution with zero mean and unit variance. It is also called Standard Normal distribution, \( N(0,1) \). However, the range of the standard Normal
distribution is not between [0,1] but about -3 to +3 (actually infinity to infinity but by using -3 to +3 you already capture 99.9% of your data).

- **range_transformation** When this method is selected, two other parameters (min, max) appear in the Parameter View. Range transformation normalizes all attribute values in the specified range [min,max]. min and max are specified using min and max parameters respectively.

- **proportion_transformation** Each attribute value is normalized as proportion of the total sum of the respective attribute i.e. each attribute value is divided by the total sum of that attribute values.

- **interquartile_range** Normalization is performed using interquartile range. The range is the difference between the largest and the smallest value in the data set. Since the range only takes into account two values from the entire data set, it may be heavily influenced by outliers in the data. Therefore, another criterion - the interquartile range - is commonly used. It is the distance between the 25th and 75th percentiles (Q3 - Q1). The interquartile range is essentially the range of the middle 50% of the data. Because it uses the middle 50%, the interquartile range is not affected by outliers or extreme values.

**min** (real) This parameter is available only when the method parameter is set to 'range transformation'. It is used to specify the minimum point of the range.

**max** (real) This parameter is available only when the method parameter is set to 'range transformation'. It is used to specify the maximum point of the range.

**Tutorial Processes**

**Different methods of normalization**

The focus of this process is to show different methods available for normalization. All parameters other than the method parameter are for selection of attributes
6. Data Transformation

on which normalization is to be applied. To understand these parameters please study the Example Process of the Select Attributes operator.

In this process the Retrieve operator is used to load the 'golf' data set from the Repository. The Filter Examples operator is applied on it to select just four examples of the 'golf' data set. This is done to just simplify the calculations. The breakpoint is inserted after this operator so that you can have a look at the examples. There are four examples with 'Humidity' attribute values 65, 70, 70 and 70. The 'Humidity' attribute is selected for normalization in the Normalize operator.

The method parameter is set to 'proportion transformation'. All values of the 'Humidity' attribute are divided by the sum of all values of the 'Humidity' attribute. The sum is 275 (65+70+70+70). Thus the values after normalization are 0.236 (65/275) and 0.255 (70/275).

Now run the process again with the method parameter set to 'z-transformation'. The mean of the four 'Humidity' attribute values (65, 70, 70, 70,) is 68.75. The Standard deviation of these values is calculated to be 2.5. Now for each attribute value, subtract the mean from the attribute value and divide the result by the standard deviation. You will see that results are the same as in the Results Workspace.

Select the 'Temperature' attribute and set the method parameter to 'range transformation'. Use 0 and 1 for min and max parameters. Run the process. You will see that all values of the 'Temperature' attribute are in range [0,1].
6.4. Value Modification

Scale by Weights

This operator scales the input ExampleSet according to the given weights. This operator deselects attributes with weight 0 and calculates new values for numeric attributes according to the given weights.

Description

The Scale by Weights operator selects attributes with non zero weight. The values of the remaining numeric attributes are recalculated based on the weights delivered at the weights input port. The new values of numeric attributes are calculated by multiplying the original values by the weight of that attribute. This operator can hardly be used for selecting a subset of attributes according to weights determined by a former weighting scheme. For this purpose the Select by Weights operator should be used which selects only those attributes that fulfill a specified weight relation.

Input Ports

example set (exa) This input port expects an ExampleSet. It is the output of the Weight by Chi Squared Statistic operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

weights (wei) This port expects the attribute weights. There are numerous operators that provide the attribute weights. The Weight by Chi Squared Statistic operator is used in the Example Process.
6. Data Transformation

Output Ports

example set \( (exa) \) The attributes with weight 0 are removed from the input ExampleSet. The values of the remaining numeric attributes are recalculated based on the weights provided at the weights input port. The resultant ExampleSet is delivered through this port.

Tutorial Processes

Applying the Scale by Weights operator on the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Chi Squared Statistic operator is applied on it to generate attribute weights. A breakpoint is inserted here. You can see the attributes with their weights here. You can see that the Wind, Humidity, Outlook and Temperature attributes have weights 0, 0.438, 0.450 and 1 respectively. The Scale by Weights operator is applied next. The 'Golf' data set is provided at the example set input port and weights calculated by the Weight by Chi Squared Statistic operator are provided at the weights input port. The Scale by Weights operator removes the attributes with weight 0 i.e. the Wind attribute is removed. The values of the remaining numeric attributes (i.e. the Temperature and Humidity attribute) are recalculated based on their weights. The weight of the Temperature attribute is 1 thus its values remain unchanged. The weight of the Humidity attribute is 0.438 thus its new values are calculated by multiplying the original values by 0.438. This can be verified by viewing the results in the Results Workspace.
6.4. Value Modification

Map

This operator maps specified values of selected attributes to new values. This operator can be applied on both numerical and nominal attributes.

Description

This operator can be used to replace nominal values (e.g. replace the value 'green' by the value 'green_color') as well as numerical values (e.g. replace all values '3' by '-1'). But, one use of this operator can do mappings for attributes of only one type. A single mapping can be specified using the parameters replace what and replace by as in Replace operator. Multiple mappings can be specified through the value mappings parameter. Additionally, the operator allows defining a default mapping. This operator allows you to select attributes to make mappings in. This operator allows you to specify a regular expression. Attribute values of selected attributes that match this regular expression are mapped by the specified value mapping. Please go through the parameters and the Example Process to develop a better understanding of this operator.
6. Data Transformation

Input Ports

**example set (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

Output Ports

**example set (exa)** The ExampleSet with value mappings is output of this port. **original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**attribute filter type (selection)** This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes on which you want to apply mappings. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in...
6.4. Value Modification

the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for the attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. The user should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.
6. Data Transformation

**regular expression (string)** Attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.

**block type (selection)** Block type of attributes to be selected can be chosen from drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition (string)** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition.
6.4. Value Modification

Conditions like 

\((> 0 \& \& < 2) \| (> 10 \& \& < 12)\) are not allowed because they use both \& \& and \| |. Use a blank space after '>', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes** (boolean) Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**value mappings** Multiple mappings can be specified through this parameter. If only a single mapping is required. It can be done using the parameters replace what and replace by as in the Replace operator. Old values and new values can be easily specified through this parameter. Multiple mappings can be defined for the same old value but only the new value corresponding to the first mapping is taken as replacement. Regular expressions can also be used here if the consider regular expressions parameter is set to true.

**replace what** (string) This parameter specifies what is to be replaced. This can be specified using regular expressions. This parameter is useful only if single mapping is to be done. For multiple mappings use the value mappings parameter.

**replace by** (string) Regions matching regular expression of the replace what parameter are replaced by the value of the replace by parameter. This parameter is useful only if single mapping is to be done. For multiple mappings use the value mappings parameter.

**consider regular expressions** (boolean) This parameter enables matching based on regular expressions; old values(old values are original values, old values and 'replace what' represent the same thing) may be specified as regular expressions. If the parameter consider regular expressions is enabled, old values are replaced by the new values if the old values match the given regular expressions. The
6. Data Transformation

value corresponding to the first matching regular expression in the mappings list is taken as a replacement.

add default mapping (boolean) If set to true, all values that occur in the selected attributes of the ExampleSet but are not listed in the value mappings list are mapped to the value of the default value parameter.

default value (string) This parameter is only available if the add default mapping parameter is checked. If add default mapping is set to true and the default value is properly set, all values that occur in the selected attributes of the ExampleSet but are not listed in the value mappings list are replaced by the default value. This may be helpful in cases where only some values should be mapped explicitly and many unimportant values should be mapped to a default value (e.g. 'other').

Tutorial Processes

Mapping multiple values

Focus of this Example Process is the use of the value mappings parameter and the default value parameter. Use of the replace what and replace by parameter can be seen in the Example Process of the Replace operator. Almost all other parameters of the Map operator are also part of the Select Attributes operator, their use can be better understood by studying the Attributes operator and its Example Process.

The 'Golf' data set is loaded using the Retrieve operator. The Map operator is applied on it. 'Wind' and 'Outlook' attributes are selected for mapping. Thus, the effect of the Map operator will be limited to just these two attributes. Four value mappings are specified in the value mappings parameter. 'true', 'false', 'overcast' and 'sunny' are replaced by 'yes', 'no', 'bad' and 'good' respectively. The add default mappings parameter is set to true and 'other' is specified in the default value parameter. 'Wind' attribute has only two possible values i.e. 'true' and 'false'. Both of them were mapped in the mappings list. 'Outlook'
attribute has three possible values i.e. 'sunny', 'overcast' and 'rain'. 'sunny' and 'overcast' were mapped in the mappings list but 'rain' was not mapped. As add default mappings parameter is set to true, 'rain' will be mapped to the default value i.e. 'other'.

Replace

This operator replaces parts of the values of selected nominal attributes matching a specified regular expression by a specified replacement.

Description

This operator allows you to select attributes to make replacements in and to specify a regular expression. Attribute values of selected attributes that match this regular expression are replaced by the specified replacement. The replacement can be empty and can contain capturing groups. Please keep in mind that although regular expressions are much more powerful than simple strings, you might simply enter characters to search for.

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. The output of other
6. Data Transformation

operators can also be used as input. It is essential that meta data should be
attached with the data for the input because attributes are specified in their
meta data. The Retrieve operator provides meta data along-with data.

Output Ports

data set (exa) An ExampleSet with replacements is output of this port.
data original (ori) The ExampleSet that was given as input is passed without chang-
ing to the output through this port. This is usually used to reuse the same Exam-
pleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type (selection) This parameter allows you to select the at-
tribute selection filter; the method you want to use for selecting attributes in
which you want to make replacements. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is
  the default option.

- **single** This option allows selection of a single attribute. When this option
  is selected another parameter (attribute) becomes visible in the Parameter
  View.

- **subset** This option allows selection of multiple attributes through a list. All
  attributes of ExampleSet are present in the list; required attributes can be
  easily selected. This option will not work if meta data is not known. When
  this option is selected another parameter becomes visible in the Parameter
  View.

- **regular_expression** This option allows you to specify a regular expression
  for attribute selection. When this option is selected some other parameters
  (regular expression, use except expression) become visible in the Parameter

528
6.4. Value Modification

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example, `real` and `integer` types both belong to the `numeric` type. User should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the `value_type` option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example, `value_series_start` and `value_series_end` block types both belong to the `value_series` block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the `parameter` attribute if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression** *(string)* Attributes whose name match this expression will be selected. Regular expression is very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. It gives a good idea of regular ex-
6. Data Transformation

expressions and also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.

**block type (selection)** The Block type of attributes to be selected can be chosen from drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition (string)** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition '\> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '\> 6 \& \& < 11' or '< = 5 || < 0'. But \& \& and || cannot be used together in one numeric condition. Conditions like '(> 0 \& \& < 2) || (> 10 \& \& < 12)' are not allowed because they use both \& \& and ||. Use a blank space after '\>', '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes (boolean)** Special attributes are attributes with spe-
cial roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**replace what** *(string)* This parameter specifies what is to be replaced. This can be specified using regular expressions. The *edit regular expression* menu can assist you in specifying the right regular expression.

**replace by** *(string)* The regions matching regular expression of the *replace what* parameter are replaced by the value of the *replace by* parameter.

### Tutorial Processes

#### Use of replace what and replace by parameters

The focus of this process is to show the use of the *replace what* and *replace by* parameters. All other parameters are for the selection of attributes on which the replacement is to be made. For understanding these parameters please study the Example Process of the Select Attributes operator.

The 'Golf' data set is loaded using the Retrieve operator. The *attribute filter type* parameter is set to 'all' and the *include special attributes* parameter is also checked. Thus, replacements are made on all attributes including special attributes. The *replace what* parameter is provided with the regular expression '*.e.*' which means any attribute value that has character 'e' in it. The *replace by* parameter is given the value 'E'. Run the process. You will see that 'E' is
placed in place of 'yes', 'overcast', 'true' and 'false'. This is because all the values have an 'e' in it. You can see the power of this operator. Now set the regular expression of replace what operator to 'e'. Run the process again. This time you will see that the entire values are not replaced by 'E', instead only the character 'e' is replaced by 'E'. Thus new values of 'yes', 'overcast', 'true' and 'false' are 'yEs', 'ovErcast', 'truE' and 'falsE' respectively. You can see the power of this operator and regular expressions. Thus it should be made sure that the correct regular expression is provided. If you leave the replace by parameter empty or write '?' in it, the null value is used as replacement.

![Diagram of the process](image)

**Cut**

This operator cuts the nominal values of the specified regular attributes. The resultant attributes have values that are substrings of the original attribute values.

**Description**

The Cut operator creates new attributes from nominal attributes where the new attributes contain only substrings of the original values. The range of characters to be cut is specified by the first character index and last character index parameters. The first character index parameter specifies the index of the first character and the last character index parameter specifies the index of the last character to be included. All characters of the attribute values that are at index equal to or greater than the first character index and less than or equal to the
last character index are included in the resulting substring. Please note that the counting starts with 1 and that the first and the last character will be included in the resulting substring. For example, if the value is RRapidMinerând the first index is set to 6 and the last index is set to 9 the result will be MMineâ. If the last index is larger than the length of the word, the resulting substrings will end with the last character.

**Input Ports**

**example set input** (*exa*) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

**Output Ports**

**example set output** (*exa*) The ExampleSet with new attributes that have values that are substrings of the original attributes is output of this port.  
**original** (*ori*) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**attribute filter type** (*selection*) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameter
6. Data Transformation

View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.
6.4. Value Modification

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and shifted to the right list, which is the list of selected attributes.

**regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (except value type) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. value type parameter's value.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be not be selected even if they match the previously mentioned block type i.e. block
6. Data Transformation

$type$ parameter's value.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' $> 6$' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' $> 6 \& \& < 11$' or ' $< = 5 \| < 0$'. But $\& \&$ and $\| \|$ cannot be used together in one numeric condition. Conditions like ' $(> 0 \& \& < 2) \| (> 10 \& \& < 12)$' are not allowed because they use both $\& \&$ and $\| \|$. Use a blank space after ' $>$', ' $=$' and ' $<$' e.g. ' $< 5$' will not work, so use ' $< 5$' instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**first character index** *(integer)* This parameter specifies the index of the first character of the substring which should be kept. Please note that the counting starts with 1.

**last character index** *(integer)* This parameter specifies the index of the last character of the substring which should be kept. Please note that the counting starts with 1.

**Tutorial Processes**

Applying the Cut operator on label of the Iris data set
6.4. ValueModification

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the data set before application of the Cut operator. You can see that the label attribute has three possible values: 'Iris-setosa', 'Iris-versicolor' and 'Iris-virginica'. If we want to remove the 'Iris-' substring from the start of all the label values we can use the Cut operator. The Cut operator is applied on the Iris data set. The first character index parameter is set to 6 because we want to remove first 5 characters ('Iris-'). The last character index parameter can be set to any value greater than the length of longest possible value. Thus the last character index parameter can be safely set to 20 because if the last index is larger than the length of the word, the resulting substrings will end with the last character. Run the process and you can see that the substring 'Iris-' has been removed from the start of all possible values of the label attribute.

Split

This operator creates new attributes from the selected nominal attributes by splitting the nominal values into parts according to the specified split criterion.

Description

The Split operator creates new attributes from the selected nominal attributes by splitting the nominal values into parts according to the split criterion which is specified through the split pattern parameter in form of a regular expression.
6. Data Transformation

This operator provides two different modes for splitting; the desired mode can be selected by the *split mode* parameter. The two splitting modes are explained with an imaginary ExampleSet with a nominal attribute named 'att' assuming that the *split pattern* parameter is set to ',' (comma). Suppose the ExampleSet has three examples:

1. value1
2. value2, value3
3. value3

### Ordered Splits

In case of ordered split the resulting attributes get the name of the original attribute together with a number indicating the order. In our example scenario there will be two attributes named 'att_1' and 'att_2' respectively. After splitting the three examples will have the following values for 'att_1' and 'att_2' (described in form of tuples):

1. (value1,?)
2. (value2, value3)
3. (value3,?)

This mode is useful if the original values indicated some order like, for example, a preference.

### Unordered Splits

In case of unordered split the resulting attributes get the name of the original attribute together with the value for each of the occurring values. In our exam-
ple scenario there will be three attributes named 'att_value1', 'att_value2' and 'att_value3' respectively. All these new attributes are boolean. After splitting the three examples will have the following values for 'att_value1', 'att_value2' and 'att_value3' (described in form of tuples):

1. (true, false, false)
2. (false, true, true)
3. (false, false, true)

This mode is useful if the order is not important but the goal is a basket like data set containing all occurring values.

**Input Ports**

*example set input (exa)* This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. The ExampleSet should have at least one nominal attribute because if there is no such attribute, the use of this operator does not make sense.

**Output Ports**

*example set output (exa)* The selected nominal attributes are split into new attributes and the resultant ExampleSet is delivered through this port.

*original (ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
6. Data Transformation

Parameters

**attribute filter type (selection)** This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.

- **single** This option allows selection of a single attribute. When this option is selected another parameter (**attribute**) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (**regular expression**, **use except expression**) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example **real** and **integer** types both belong to the **numeric** type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When it is selected some other parameters (**value type**, **use value type exception**) become visible in the Parameter View.

- **block_type** This option is similar in working to the **value type** option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (**block type**, **use block type exception**) become visible in the Parameter View.
6.4. Value Modification

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute** *(string)* The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of *attribute* parameter if the meta data is known.

**attributes** *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes on which the conversion from nominal to numeric will take place; all other attributes will remain unchanged.

**regular expression** *(string)* The attributes whose name matches this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions. This menu also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter (*except value type*) becomes visible in the Parameter View.

**except regular expression** *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in the *regular expression* parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list. One of the following types can be chosen: nominal, text, binominal, polynominal, file_path.
6. Data Transformation

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is selected another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value. One of the following types can be selected here: nominal, text, binominal, polynominal, file_path.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list. The only possible value here is 'single_value'

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like ' ( > 0 & & < 2) || ( > 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>' , '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**split pattern** *(string)* This parameter specifies the pattern which is used for di-
viding the nominal values into different parts. It is specified in form of a regular expression. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions.

**split mode (selection)** This parameter specifies the split mode for splitting. The two options of this parameter are explained in the description of this operator.

### Tutorial Processes

#### Ordered and unordered splits

This Example Process starts with a Subprocess operator. The operator chain inside the Subprocess operator generates an ExampleSet for this process. The explanation of this inner chain of operators is not relevant here. A breakpoint is inserted here so that you can have a look at the ExampleSet before the application of the Split operator. You can see that this ExampleSet is the same ExampleSet that is described in the description of this operator. The Split operator is applied on it with default values of all parameters. The split mode parameter is set to 'ordered split' by default. Run the process and compare the results with the explanation of ordered split in the description section of this document. Now change the split mode parameter to 'unordered split' and run the process again. You can understand the results by studying the description of unordered split in the description of this operator.
6. Data Transformation

Merge

This operator merges two nominal values of the specified regular attribute.

Description

The Merge operator is used for merging two nominal values of the specified attribute of the input ExampleSet. Please note that this operator can merge only the values of regular attributes. The required regular attribute is specified using the attribute name parameter. The first value parameter is used for specifying the first value to be merged. The second value parameter is used for specifying the second value to be merged. The two values are merged in 'first_second' format where first is the value of the first value parameter and second is the value of the second value parameter. It is not compulsory for the first value and second value parameters to have values from the range of possible values of the selected attribute. However, at least one of the first value and second value parameters should have a value from the range of possible values of the selected attribute. Otherwise this operator will have no affect on the input ExampleSet.
6.4. Value Modification

Input Ports

**example set input (exa)** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

**example set output (exa)** The ExampleSet with the merged attribute values is output of this port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**attribute name (string)** The required nominal attribute whose values are to be merged is selected through this parameter. This operator can be applied only on regular attributes.

**first value (string)** This parameter is used for specifying the first value to be merged. It is not compulsory for the *first value* parameter to have a value from the range of possible values of the selected attribute.

**second value (string)** This parameter is used for specifying the second value to be merged. It is not compulsory for the *second value* parameter to have a value from the range of possible values of the selected attribute.

Tutorial Processes

**Introduction to the Merge operator**
6. Data Transformation

The Golf data set is loaded using the Retrieve operator. The Merge operator is applied on it. The attribute name parameter is set to 'Outlook'. The first value parameter is set to 'sunny' and the second value parameter is set to 'hot'. All the occurrences of value 'sunny' are replaced by 'sunny_hot' in the Outlook attribute of the resultant ExampleSet. Now set the value of the second value parameter to 'rain' and run the process again. As 'rain' is also a possible value of the Outlook attribute, all occurrences of 'sunny' and 'rain' in the Outlook attribute are replaced by 'sunny_rain' in the resultant ExampleSet. This Example Process is just to explain basic working of the Merge operator.

Remap Binominals

This operator modifies the internal value mapping of binominal attributes according to the specified negative and positive values.

Description

The Remap Binominals operator modifies the internal mapping of binominal attributes according to the specified positive and negative values. The positive and negative values are specified by the positive value and negative value parameters respectively. If the internal mapping differs from the specified values then the
internal mapping is switched. If the internal mapping contains other values than
the specified ones the mapping is not changed and the attribute is simply skipped.
Please note that this operator changes the internal mapping so the changes are
not explicitly visible in the ExampleSet. This operator can be applied only on
binominal attributes. Please note that if there is a nominal attribute in the Ex-
ampleSet with only two possible values, this operator will still not be applicable
on it. This operator requires the attribute to be explicitly defined as binominal
in the meta data.

Input Ports

example set input (exa) This input port expects an ExampleSet. Please note
that there should be at least one binominal attribute in the input ExampleSet.

Output Ports

example set output (exa) The resultant ExampleSet is output of this port.
Externally this data set is the same as the input ExampleSet, only the internal
mappings may be changed.
original (ori) The ExampleSet that was given as input is passed without chang-
ing to the output through this port. This is usually used to reuse the same Exam-
pleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute filter type (selection) This parameter allows you to select the at-
tribute selection filter; the method you want to use for selecting attributes. It
has the following options:

• all This option simply selects all the attributes of the ExampleSet This is
the default option.
6. Data Transformation

- **single** This option allows selection of a single attribute. When this option is selected another parameter (*attribute*) becomes visible in the Parameter View.

- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (*regular expression, use except expression*) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (*value type, use value type exception*) become visible in the Parameter View.

- **block_type** This option is similar in working to the *value_type* option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example *value_series_start* and *value_series_end* block types both belong to the *value_series* block type. When this option is selected some other parameters (*block type, use block type exception*) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (*numeric condition*) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespec-
6.4. Value Modification

tive of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

**regular expression (string)** The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the *edit and preview regular expression* menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (*except regular expression*) becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the regular expression parameter).

**value type (selection)** The type of attributes to be selected can be chosen from a drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter (*except value type*) becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will not be selected even if they match the previously mentioned type i.e. value type parameter's value.

**block type (selection)** The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (*except block type*) becomes visible in the Parameter View.
except block type (selection) The attributes matching this block type will not be selected even if they match the previously mentioned block type i.e. block type parameter's value.

numeric condition (string) The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 & & < 11' or '<= 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>' , '=' and '<' e.g. '< 5' will not work, so use '< 5' instead.

include special attributes (boolean) The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes are selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

invert selection (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

negative value (string) This parameter specifies the internal mapping for the negative or false value of the selected binominal attributes.

positive value (string) This parameter specifies the internal mapping for the positive or true value of the selected binominal attributes.

Tutorial Processes

Changing mapping of the Wind attribute of the Golf data set
The 'Golf' data set is loaded using the Retrieve operator. In this Example Process we shall change the internal mapping of the 'Wind' attribute of the 'Golf' data set. A breakpoint is inserted after the Retrieve operator so that you can view the 'Golf' data set. As you can see the 'Wind' attribute of the 'Golf' data set is nominal but it has only two possible values. The Remap Binominals operator cannot be applied on such an attribute; it requires that the attribute should be explicitly declared as binomial in the meta data. To accomplish this, the Nominal to Binominal operator is applied on the 'Golf' data set to convert the 'Wind' attribute to binominal type. A breakpoint is inserted here so that you can view the ExampleSet. Now that the 'Wind' attribute has been converted to binominal type, the Remap Binominals operator can be applied on it. The 'Wind' attribute is selected in the Remap Binominals operator. The negative value and positive value parameter are set to 'true' and 'false' respectively. Run the process and the internal mapping is changed. This change is an internal one so it will not be visible explicitly in the Results Workspace. Now change the value of the positive value and negative value parameters to 'a' and 'b' respectively and run the complete process. Have a look at the log. You will see the following message: "WARNING: Remap Binominals: specified values do not match values of attribute Wind, attribute is skipped." This log shows that as the values 'a' and 'b' are not values of the 'Wind' attribute so no change in mapping is done.
Replace Missing Values

This operator replaces missing values in examples of selected attributes by a specified replacement.

Description

This operator replaces missing values in examples of selected attributes by a specified replacement. Missing values can be replaced by the minimum, maximum or average value of that attribute. Zero can also be placed in place of missing values. Any replenishment value can also be specified as a replacement of missing values.

Input Ports

example set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

Output Ports

example set (exa) The ExampleSet with missing values replaced by specified replacement is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
preprocessing model (pre) This port delivers the preprocessing model, which has information regarding the parameters of this operator in the current process.

Parameters

create view (boolean) It is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data.

attribute filter type (selection) This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes in which you want to replace missing values. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter (attribute) becomes visible in Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in Parameter View.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in Parameter View.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. User should have basic understanding of type hierarchy when selecting attributes through this option.
6. Data Transformation

When this option is selected some other parameters (value type, use value type exception) become visible in Parameter View.

- **block_type** This option is similar in working to value_type option. This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example value_series_start and value_series_end block types both belong to value_series block type. When this option is selected some other parameters (block type, use block type exception) become visible in Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don’t contain a missing value in any example. Attributes that have even a single missing value are not selected.

- **numeric_value_filter** When this option is selected another parameter (numeric condition) becomes visible in Parameter View. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The required attribute can be selected from this option. The attribute name can be selected from the drop down box of the parameter attribute if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list and can be shifted to the right list which is the list of selected attributes.

**regular expression (string)** Attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through edit and preview regular expression menu. This menu gives a good idea of regular expressions. It also allows you to try different expressions and preview the results simultaneously. This will enhance your concept of regular expressions.

**use except expression (boolean)** If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (except regular expression) becomes visible in Parameter View.

**except regular expression (string)** This option allows you to specify a regular
expression. Attributes matching this expression will be filtered out even if they match the first expression (expression that was specified in *regular expression* parameter).

**value type (selection)** Type of attributes to be selected can be chosen from drop down list.

**use value type exception (boolean)** If enabled, an exception to the selected type can be specified. When this option is selected another parameter (except value type) becomes visible in Parameter View.

**except value type (selection)** Attributes matching this type will be removed from the final output even if they matched the previously mentioned type i.e. value type parameter's value.

**block type (selection)** Block type of attributes to be selected can be chosen from drop down list.

**use block type exception (boolean)** If enabled, an exception to the selected block type can be specified. When this option is selected another parameter (except block type) becomes visible in Parameter View.

**except block type (selection)** Attributes matching this block type will be removed from the final output even if they matched the previously mentioned block type.

**numeric condition (string)** Numeric condition for testing examples of numeric attributes is mention here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ', ' and '< ' e.g. '< 5' will not work, so use '< 5' instead.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is removed prior to selection of this parameter. After selection of this parameter 'att1' will be removed and 'att2' will be selected.

**include special attributes (boolean)** Special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply de-
6. Data Transformation

scribe the examples. Special attributes are: id, label, prediction, cluster, weight
and batch. By default all special attributes are delivered to the output port ir-
respective of the conditions in the Select Attribute operator. If this parameter is
set to true, Special attributes are also tested against conditions specified in the
Select Attribute operator and only those attributes are selected that satisfy the
conditions.

default (selection) Function to apply to all columns that are not explicitly spec-
ified by the columns parameter.

- **none** If this option is selected, no function is applied by default i.e. missing
  values are not replaced by default.

- **minimum** If this option is selected, by default missing values are replaced
  by the minimum value of that attribute.

- **maximum** If this option is selected, by default missing values are replaced
  by the maximum value of that attribute.

- **average** If this option is selected, by default missing values are replaced
  by the average value of that attribute.

- **zero** If this option is selected, by default missing values are replaced by
  zero.

- **value** If this option is selected, by default missing values are replaced by
  the value specified in the replenishment value parameter.

columns (list) Different attributes can be provided with a different type of re-
placements through this parameter. The default function selected by the de-
fault parameter is applied on attributes that are not explicitly mentioned in the
columns parameter

replenishment value (string) This parameter is available for replacing missing
values by a specified value.
6.5. Data Cleansing

Tutorial Processes

Replacing missing values of the Labor Negotiations data set

The focus of this process is to show the use of the default and columns parameters. All other parameters are for selection of attributes on which replacement is to be applied. For understanding these parameters please study the Example Process of the Select Attributes operator.

The 'Labor Negotiations' data set is loaded using the Retrieve operator. A breakpoint is inserted at this point so that you can view the data before the application of the Replace Missing Values operator. The Replace Missing Values operator is applied on it. The attribute filter type parameter is set to 'no missing values' and the invert selection parameter is also checked, thus all attributes with missing values are selected. In the columns parameter the 'wage-inc-1st', 'wage-inc-2nd', 'wage-inc-3rd' and 'working hours' attributes are set to 'minimum', 'maximum', 'zero' and 'value' respectively. The minimum value of the 'wage-inc-1st' attribute is 2.000, thus missing values are replaced with 2.000. The maximum value of the 'wage-inc-2nd' attribute is 7.000, thus missing values are replaced with 7.000. Missing values of wage-inc-3rd are replaced by 0. The replenishment value parameter is set to 35, thus missing values of the 'working hours' operator are set to 35. The default parameter is set to 'average', thus missing values of all other attributes are replaced by the average value of that attribute.
6. Data Transformation

Fill Data Gaps

This operator fills the gaps (based on the ID attribute) in the given ExampleSet by adding new examples in the gaps. The new example will have null values.

Description

The Fill Data Gaps operator fills the gaps (based on gaps in the ID attribute) in the given ExampleSet by adding new examples in the gaps. The new examples will have null values for all attributes (except the id attribute) which can be replenished by operators like the Replace Missing Values operator. It is ideal that the ID attribute should be of integer type. This operator performs the following steps:

- The data is sorted according to the ID attribute
- All occurring distances between consecutive ID values are calculated.
- The greatest common divisor (GCD) of all distances is calculated.
- All rows which would have an ID value which is a multiple of the GCD but are missing are added to the data set.
6.5. Data Cleansing

Input Ports

example set input *(exa)* This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

Output Ports

example set output *(exa)* The gaps in the ExampleSet are filled with new examples and the resulting ExampleSet is output of this port.
original *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

use gcd for step size *(boolean)* This parameter indicates if the greatest common divisor (GCD) should be calculated and used as the underlying distance between all data points.
step size *(integer)* This parameter is only available when the use gcd for step size parameter is set to false. This parameter specifies the step size to be used for filling the gaps.
start *(integer)* This parameter can be used for filling the gaps at the beginning (if they occur) before the first data point. For example, if the ID attribute of the given ExampleSet starts with 3 and the start parameter is set to 1. Then this operator will fill the gaps in the beginning by adding rows with ids 1 and 2.
end *(integer)* This parameter can be used for filling the gaps at the end (if they occur) after the last data point. For example, if the ID attribute of the given ExampleSet ends with 100 and the end parameter is set to 105. Then this
6. Data Transformation

operator will fill the gaps at the end by adding rows with ids 101 to 105.

Tutorial Processes

Introduction to the Fill Data Gaps operator

This Example Process starts with the Subprocess operator which delivers an
ExampleSet. A *breakpoint* is inserted here so that you can have a look at the
ExampleSet. You can see that the ExampleSet has 10 examples. Have a look
at the id attribute of the ExampleSet. You will see that certain ids are missing
i.e. ids 3, 6, 8 and 10 are missing. The Fill Data Gaps operator is applied on
this ExampleSet to fill these data gaps with examples that have the appropriate
ids. You can see the resultant ExampleSet in the Results Workspace. You can
see that this ExampleSet has 14 examples. New examples with ids 3, 6, 8 and
10 have been added. But these examples have missing values for all attributes
(except id attribute) which can be replenished by using operators like the Replace
Missing Values operator.

Detect Outlier (Distances)
This operator identifies $n$ outliers in the given ExampleSet based on the distance to their $k$ nearest neighbors. The variables $n$ and $k$ can be specified through parameters.

**Description**

This operator performs outlier search according to the outlier detection approach recommended by Ramaswamy, Rastogi and Shim in Efficient Algorithms for Mining Outliers from Large Data Sets. In their paper, a formulation for distance-based outliers is proposed that is based on the distance of a point from its $k$-th nearest neighbor. Each point is ranked on the basis of its distance to its $k$-th nearest neighbor and the top $n$ points in this ranking are declared to be outliers. The values of $k$ and $n$ can be specified by the number of neighbors and number of outliers parameters respectively. This search is based on simple and intuitive distance-based definitions for outliers by Knorr and Ng which in simple words is: 'A point $p$ in a data set is an outlier with respect two parameters $k$ and $d$ if no more than $k$ points in the data set are at a distance of $d$ or less from $p$'.

This operator adds a new boolean attribute named 'outlier' to the given ExampleSet. If the value of this attribute is true that example is an outlier and vice versa. $n$ examples will have the value true in the 'outlier' attribute (where $n$ is the value specified in the number of outliers parameter). Different distance functions are supported by this operator. The desired distance function can be selected by the distance function parameter.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.
6. Data Transformation

Input Ports

*example set input (exa)* This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

*example set output (exa)* A new boolean attribute 'outlier' is added to the given ExampleSet and the ExampleSet is delivered through this output port. *original (ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

*number of neighbors (integer)* This parameter specifies the k value for the k-th nearest neighbors to be the analyzed. The minimum and maximum values for this parameter are 1 and 1 million respectively.

*number of outliers (integer)* This parameter specifies the number of top-n outliers to be looked for. The resultant ExampleSet will have n number of examples that are considered outliers. The minimum and maximum values for this parameter are 2 and 1 million respectively.

*distance function (selection)* This parameter specifies the distance function that will be used for calculating the distance between two examples.
6.5. Data Cleansing

Tutorial Processes

Detecting outliers from an ExampleSet

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (Distances) operator is applied on this ExampleSet. The number of neighbors and number of outliers parameters are set to 4 and 12 respectively. Thus 12 examples of the resultant ExampleSet will have true value in the 'outlier' attribute. This can be verified by viewing the ExampleSet in the Results Workspace. For better understanding switch to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'outlier' to view the scatter plot of the ExampleSet (the outliers are marked red).

Detect Outlier (Densities)

This operator identifies outliers in the given ExampleSet based on the data density. All objects that have at least $p$ proportion of all objects
6. Data Transformation

farther away than distance $D$ are considered outliers.

Description

The Detect Outlier (Densities) operator is an outlier detection algorithm that calculates the $DB(p,D)$-outliers for the given ExampleSet. A $DB(p,D)$-outlier is an object which is at least $D$ distance away from at least $p$ proportion of all objects. The two real-valued parameters $p$ and $D$ can be specified through the proportion and distance parameters respectively. The $DB(p,D)$-outliers are distance-based outliers according to Knorr and Ng. This operator implements a global homogenous outlier search.

This operator adds a new boolean attribute named 'outlier' to the given ExampleSet. If the value of this attribute is true, that example is an outlier and vice versa. Different distance functions are supported by this operator. The desired distance function can be selected by the distance function parameter.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set output (exa) A new boolean attribute 'outlier' is added to the given ExampleSet and the ExampleSet is delivered through this output port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

distance (real) This parameter specifies the distance \( D \) parameter for calculation of the DB\((p,D)\)-outliers.

proportion (real) This parameter specifies the proportion \( p \) parameter for calculation of the DB\((p,D)\)-outliers.

distance function (selection) This parameter specifies the distance function that will be used for calculating the distance between two examples.

Tutorial Processes

Detecting outliers from an ExampleSet

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (Densities) operator is applied on the ExampleSet. The distance and proportion parameters are set to 4.0 and 0.8 respectively. The resultant ExampleSet can be viewed in the Results Workspace. For better understanding switch to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'outlier' to view the scatter plot of the ExampleSet (the outliers are marked red). The number of outliers may differ depending on
6. Data Transformation

the randomization, if the *random seed* parameter of the process is set to 1997, you will see 5 outliers.

![Diagram](image)

Detect Outlier (LOF)

This operator identifies outliers in the given ExampleSet based on local outlier factors (LOF). The LOF is based on a concept of a local density, where locality is given by the $k$ nearest neighbors, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbors, one can identify regions of similar density, and points that have a substantially lower density than their neighbors. These are considered to be outliers.

Description

This operator performs a LOF outlier search. LOF outliers or outliers with a local outlier factor per object are density based outliers according to Breunig, Kriegel, et al. As indicated by the name, the local outlier factor is based on a concept of a local density, where locality is given by $k$ nearest neighbors, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbors, one can identify regions of similar density, and points that have a substantially lower density than their neighbors. These are considered to be outliers. The local density is estimated by the typical distance at which a point can be 'reached' from its neighbors. The definition of 'reachability distance' used in LOF is an additional measure to produce more
stable results within clusters.

The approach to find the outliers is based on measuring the density of objects and its relation to each other (referred to as local reachability density). Based on the average ratio of the local reachability density of an object and its k-nearest neighbors (i.e. the objects in its k-distance neighborhood), a local outlier factor (LOF) is computed. The approach takes a parameter MinPts (actually specifying the 'k') and it uses the maximum LOFs for objects in a MinPts range (lower bound and upper bound to MinPts).

This operator supports cosine, inverted cosine, angle and squared distance in addition to the usual euclidian distance which can be specified by the distance function parameter. In the first step, the objects are grouped into containers. For each object, using a radius screening of all other objects, all the available distances between that object and another object (or group of objects) on the same radius given by the distance are associated with a container. That container then has the distance information as well as the list of objects within that distance (usually only a few) and the information about how many objects are in the container.

In the second step, three things are done:

1. The containers for each object are counted in ascending order according to the cardinality of the object list within the container (= that distance) to find the k-distances for each object and the objects in that k-distance (all objects in all the subsequent containers with a smaller distance).

2. Using this information, the local reachability densities are computed by using the maximum of the actual distance and the k-distance for each object pair (object and objects in k-distance) and averaging it by the cardinality of the k-neighborhood and then taking the reciprocal value.

3. The LOF is computed for each MinPts value in the range (actually for all up to upper bound) by averaging the ratio between the MinPts-local reachability-density of all objects in the k-neighborhood and the object itself. The maximum LOF in the MinPts range is passed as final LOF to each object.
6. Data Transformation

Afterwards LOFs are added as values for a special real-valued outlier attribute in the ExampleSet which the operator will return.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.

Input Ports

document, 6.133

example set input (exa) This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set output (exa) A new attribute 'outlier' is added to the given ExampleSet which is then delivered through this output port.

document, 6.134

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

minimal points lower bound (integer) This parameter specifies the lower bound for MinPts for the Outlier test.

minimal points upper bound (integer) This parameter specifies the upper bound for MinPts for the Outlier test.

document, 6.135

distance function (selection) This parameter specifies the distance function that will be used for calculating the distance between two objects.
6.5. Data Cleansing

Tutorial Processes

Detecting outliers from an ExampleSet

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (LOF) operator is applied on this ExampleSet with default values for all parameters. The minimal points lower bound and minimal points upper bound parameters are set to 10 and 20 respectively. The resultant ExampleSet can be seen in the Results Workspace. For better understanding switch to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'outlier' to view the scatter plot of the ExampleSet.

Detect Outlier (COF)

This operator identifies outliers in the given ExampleSet based on the Class Outlier Factors (COF).
6. Data Transformation

Description

The main concept of ECODB (Enhanced Class Outlier - Distance Based) algorithm is to rank each instance in the ExampleSet given the parameters $N$ (top $N$ class outliers), and $K$ (the number of nearest neighbors). The rank of each instance is found using the formula:

$$COF = PCL(T,K) - \text{norm}(\text{deviation}(T)) + \text{norm}(\text{kDist}(T))$$

- $PCL(T,K)$ is the Probability of the Class Label of the instance $T$ with respect to the class labels of its $K$ nearest neighbors.
- $\text{norm}(\text{Deviation}(T))$ and $\text{norm}(\text{KDist}(T))$ are the normalized values of $\text{Deviation}(T)$ and $\text{KDist}(T)$ respectively and their values fall in the range [0 - 1].
- $\text{Deviation}(T)$ is how much the instance $T$ deviates from instances of the same class. It is computed by summing the distances between the instance $T$ and every instance belonging to the same class.
- $\text{KDist}(T)$ is the summation of distance between the instance $T$ and its $K$ nearest neighbors.

This operator adds a new boolean attribute named 'outlier' to the given ExampleSet. If the value of this attribute is true, that example is an outlier and vice versa. Another special attribute 'COF Factor' is also added to the ExampleSet. This attribute measures the degree of being Class Outlier for an example.

An outlier is an example that is numerically distant from the rest of the examples of the ExampleSet. An outlying example is one that appears to deviate markedly from other examples of the ExampleSet. Outliers are often (not always) indicative of measurement error. In this case such examples should be discarded.
6.5. Data Cleansing

Input Ports

**example set input (exa)** This input port expects an ExampleSet. It is the output of the Generate Data operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

**example set output (exa)** A new boolean attribute 'outlier' and a real attribute 'COF Factor' is added to the given ExampleSet and the ExampleSet is delivered through this output port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**number of neighbors** *(integer)* This parameter specifies the \( k \) value for the \( k \) nearest neighbors to be the analyzed. The minimum and maximum values for this parameter are 1 and 1 million respectively.

**number of class outliers** *(integer)* This parameter specifies the number of top-\( n \) Class Outliers to be looked for. The resultant ExampleSet will have \( n \) number of examples that are considered outliers. The minimum and maximum values for this parameter are 2 and 1 million respectively.

**measure types** *(selection)* This parameter is used for selecting the type of measure to be used for measuring the distance between points. The following options are available: *mixed measures, nominal measures, numerical measures* and *Bregman divergences*.

**mixed measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'
6. Data Transformation

**nominal measure (selection)** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure (selection)** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence (selection)** This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type (selection)** This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by $k(x,y) = x^*y$ i.e. it is inner product of $x$ and $y$.

- **radial** The radial kernel is defined by $\exp(-g \|x-y\|^2)$ where $g$ is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter $gamma$ plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x,y) = (x^*y+1)^d$ where $d$ is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net $\tanh(\alpha x^*y + b)$ where $\alpha$ is *alpha* and $b$ is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for $\alpha$ is $1/N$, where $N$ is the data dimension. Note that not all choices of $\alpha$ and $b$ lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has adjustable parameters $gamma$ and
6.5. Data Cleansing

The Epanechnikov kernel is this function \((3/4)(1-u^2)\) for \(u\) between -1 and 1 and zero for \(u\) outside that range. It has two adjustable parameters \(kernel\ sigma1\) and \(kernel\ degree\).

- **epachnenikov** The Epanechnikov kernel is this function \((3/4)(1-u^2)\) for \(u\) between -1 and 1 and zero for \(u\) outside that range. It has two adjustable parameters \(kernel\ sigma1\) and \(kernel\ degree\).

- **gaussian\_combination** This is the gaussian combination kernel. It has adjustable parameters \(kernel\ sigma1, kernel\ sigma2\) and \(kernel\ sigma3\).

- **multiquadric** The multiquadric kernel is defined by the square root of \(||x-y||^2 + c^2\). It has adjustable parameters \(kernel\ sigma1\) and \(kernel\ sigma_3\).

**kernel\ gamma** *(real)* This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to radial or anova.

**kernel\ sigma1** *(real)* This is the SVM kernel parameter sigma1. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel\ sigma2** *(real)* This is the SVM kernel parameter sigma2. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to gaussian combination.

**kernel\ sigma3** *(real)* This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to gaussian combination.

**kernel\ shift** *(real)* This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to multiquadric.

**kernel\ degree** *(real)* This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to polynomial, anova or epachnenikov.

**kernel\ a** *(real)* This is the SVM kernel parameter a. This parameter is only
6. Data Transformation

available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to neural.

**kernel b (real)** This is the SVM kernel parameter b. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to neural.

Tutorial Processes

Detecting outliers from an ExampleSet

The Generate Data operator is used for generating an ExampleSet. The target function parameter is set to 'gaussian mixture clusters'. The number examples and number of attributes parameters are set to 200 and 2 respectively. A breakpoint is inserted here so that you can view the ExampleSet in the Results Workspace. A good plot of the ExampleSet can be seen by switching to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1' and y-Axis to 'att2' to view the scatter plot of the ExampleSet.

The Detect Outlier (COF) operator is applied on the ExampleSet. The number of neighbors and number of class outliers parameters are set to 7. The resultant ExampleSet can be viewed in the Results Workspace. For better understanding, switch to the 'Plot View' tab. Set Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'outlier' to view the scatter plot of the ExampleSet (the outliers are marked red).
Filter Examples

This operator selects which examples (i.e. rows) of an ExampleSet should be kept and which examples should be removed. Examples satisfying the given condition are kept, remaining examples are removed.

Description

This operator takes an ExampleSet as input and returns a new ExampleSet including only those examples that satisfy the specified condition. Several predefined conditions are provided; users can select any of them. Users can also define their own conditions to filter examples. This operator may reduce the number of examples in an ExampleSet but it has no effect on the number of attributes. Select Attributes operator is used to select required attributes.

Filter Examples operator is frequently used to filter examples that have (or do not have) missing values. It is also frequently used to filter examples with correct or wrong predictions (usually after testing a learnt model).

Input Ports

example set input \((exa)\) This input port expects an ExampleSet. It is output of Retrieve operator in the attached Example Process.

Output Ports

example set output \((exa)\) New ExampleSet including only the examples that satisfied the specified condition is output of this port.
6. Data Transformation

original (ori) ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

condition class (selection) Various predefined conditions are available for filtering examples. Users can select any of them. Examples satisfying the selected condition are passed to the output port, others are removed. Following conditions are available:

- **all** if this option is selected, no examples are removed.
- **correct_predictions** if this option is selected, only those examples make to the output port that have correct predictions i.e. value of label attribute and prediction attribute are same.
- **wrong_predictions** if this option is selected, only those examples make to the output port that have wrong predictions i.e. value of label attribute and prediction attribute are not same.
- **no_missing_attributes** if this option is selected, only those examples make to the output port that have no missing values in their attribute values. Missing values or null values are usually shown by '?' in RapidMiner.
- **missing_attributes** if this option is selected, only those examples make to the output port that have some missing values in their attribute values.
- **no_missing_labels** if this option is selected, only those examples make to the output port that do not have any missing values in their label attribute values. Missing values or null values are usually shown by '?' in RapidMiner.
- **missing_label** if this option is selected, only those examples make to the output port that have some missing values in their label attribute values.
6.6. Filtering

- **attribute_value_filter** if this option is selected, another parameter (parameter string) is enabled in the Parameter View.

**string (string) parameter string(Range: string):** Instead of using one of the predefined conditions users can define their own conditions here. It is important to understand how to specify conditions here because true power of this operator lies in using it with defining own conditions according to requirements. For numerical attributes conditions can be specified easily using `attribute op value` format. Where 'attribute' is name of attribute, 'value' is a value that attribute can take and 'op' represents binary logical operators like `>`, `<`, `=>`, `<=`, `=` and `!=`. For nominal attributes conditions can be specified easily using `attribute op exp` format. Where 'attribute' is name of the attribute, 'op' can be either `=` or `!=' operator, 'exp' stands for regular expression. Users should have a good understanding of regular expressions. You can have a good idea of regular expressions if you use Select Attributes operator with `attribute filter type` parameter set to `regular_expression` and then using the `edit and preview regular expression` menu.

Multiple conditions can be linked by using logical AND (written as `& &`) or logical OR (written as `||`) operators. Instead of writing multiple AND conditions you can use multiple Filter Examples operators in a row to reduce complexity.

Missing values or null values can be written as `'?'` for numerical attributes and as `'\?'` for nominal attributes. `'\?'` is used instead of `'?'` in nominal attributes because this is the way missing values are specified in regular expressions.

For 'unknown attributes' the parameter string must be empty. This filter removes all examples containing attributes that have missing or illegal values. For 'unknown label' the parameter string must also be empty. This filter removes all examples with an unknown label value.

**invert filter (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected examples are removed and previously removed examples are selected. In other words it inverts the condition. For example if missing_attributes option is selected in condition class parameter and invert filter parameter is also set to true. Then output port will deliver an ExampleSet with no missing values.
6. Data Transformation

Tutorial Processes

Filtering correctly predicted examples

Golf dataset is loaded using Retrieve operator and k-NN operator is applied on it to generate a classification model. That model is then applied on Golf-Testset data set using Apply Model operator. Apply Model operator applies the model learnt by k-NN operator on the Golf-Testset data set and records the predicted values in a new attribute named 'prediction(Play)'. Labeled data from Apply Model serves as input to the Filter Examples operator. Correct predictions option is selected in condition class parameter which ensures that only those examples make to the output port that have correct predictions. Correct prediction means value of label and prediction attributes are same in that example. But, as invert filter parameter is set to true, it reverses the selection and instead of correct predictions, wrong predictions are delivered through the output port. It can be seen in the Results Workspace that label attribute (Play) and prediction attribute (prediction(Play)) have opposite values in all the resultant examples. Breakpoint is inserted before Filter Examples operator to have a look at the examples before application of Filter Examples operator. Press the green-colored Run button to continue with the process.
Filtering examples according to their values

Golf data set is loaded using Retrieve operator and Filter Examples is applied on it with parameter string: "Outlook = .n.* & & Temperature > 70". Outlook attribute is a nominal attribute thus regular expression is used to describe it. Regular expression Outlook = .n.*mmeans all examples that have alphabet 'n' in its Outlook attribute value. 10 examples qualify, all have 'Outlook = rain' or 'Outlook = sunny'. Temperature attribute is a numerical attribute so attribute op valueßyntax is used to select rows. 9 examples satisfy the condition where Temperature attribute has a value greater than 70. As these two conditions are joined using logical AND (& & ), finally selected examples are those that meet both the conditions. Only 6 such rows are present that have an 'n' in Outlook attribute value and their Temperature attribute value is also greater than 70. This can be seen clearly in the Results Workspace.

Filtering examples according to their values with or condition

Labor-Negotiations data set is loaded using the Retrieve operator and Filter Examples is applied on it with parameter string: "duration=? || pension !=\?". Duration attribute is a numerical attribute so attribute op valueßyntax is used to select rows. 1 example satisfies the condition where Duration attribute has a missing value. Pension attribute is a nominal attribute thus regular expression is used to describe it. Regular expression ppension !=\?mmeans all examples that do not have missing values in its Pension attribute value. 18 examples qualify; all have no missing values in their Pension attribute. Note that '?' is used for
missing values of numerical attributes and "\?" is used for missing values of nominal attributes. Note that for nominal values the question mark must be escaped ("\?") because, as noted above, a regular expression is expected in this case. As these two conditions are joined using logical OR (||), finally selected examples are those that meet both the conditions. 18 such rows are present that have no missing values in Pension attribute values or have missing values in Duration attribute values. This can be seen clearly in the Results Workspace.

Remove Duplicates

This operator removes duplicate examples from an ExampleSet by comparing all examples with each other on the basis of the specified attributes. Two examples are considered duplicate if the selected attributes have the same values in them.

Description

The Remove Duplicates operator removes duplicate examples from an ExampleSet by comparing all examples with each other on the basis of the specified attributes. This operator removes duplicate examples such that only one of all the duplicate examples is kept. Two examples are considered duplicate if the selected attributes have the same values in them. Attributes can be selected from the attribute filter type parameter and other associated parameters. Suppose two attributes 'att1' and 'att2' are selected and 'att1' and 'att2' have three and two possible values respectively. Thus there are total 6 (i.e. 3 x 2) unique combina-
tions of these two attribute. Thus the resultant ExampleSet can have 6 examples at most. This operator works on all attribute types.

### Input Ports

**example set input** *(exa)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**example set output** *(exa)* The duplicate examples are removed from the given ExampleSet and the resultant ExampleSet is delivered through this port.

**original** *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**attribute filter type** *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting the required attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter *(attribute)* becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list.
6. Data Transformation

All attributes of the ExampleSet are present in the list; required attributes can be easily selected. This option will not work if the meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.

- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters (regular expression, use except expression) become visible in the Parameter View.

- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example real and integer types both belong to numeric type. Users should have a basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters (value type, use value type exception) become visible in the Parameter View.

- **block_type** This option is similar in working to the value type option. This option allows selection of all the attributes of a particular block type. When this option is selected some other parameters (block type, use block type exception) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (numeric condition) becomes visible in the Parameter View. All numeric attributes whose examples all satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

**attribute (string)** The desired attribute can be selected from this option. The attribute name can be selected from the drop down box of attribute parameter if the meta data is known.

**attributes (string)** The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list.
6.6. Filtering

and can be shifted to the right list which is the list of selected attributes on which
the conversion from nominal to numeric will take place; all other attributes will
remain unchanged.

**regular expression (string)** The attributes whose name matches this expression
will be selected. Regular expression is a very powerful tool but needs a detailed
explanation to beginners. It is always good to specify the regular expression
through the *edit and preview regular expression* menu. This menu gives a good
idea of regular expressions. This menu also allows you to try different expressions
and preview the results simultaneously. This will enhance your concept of regular
expressions.

**use except expression** *(boolean)* If enabled, an exception to the selected type
can be specified. When this option is selected another parameter *(except value
type)* becomes visible in the Parameter View.

**except regular expression (string)** This option allows you to specify a regular
expression. Attributes matching this expression will be filtered out even if they
match the first expression *(expression that was specified in the regular expression
parameter).*

**value type (selection)** The type of attributes to be selected can be chosen from
a drop down list. One of the following types can be chosen: nominal, text, bi-
nominal, polynominal, file_path.

**use value type exception** *(boolean)* If enabled, an exception to the selected
type can be specified. When this option is selected another parameter *(except
value type)* becomes visible in the Parameter View.

**except value type (selection)** The attributes matching this type will be re-
moved from the final output even if they matched the previously mentioned type
i.e. *value type* parameter's value. One of the following types can be selected here:
nominal, text, binominal, polynominal, file_path.

**block type (selection)** The block type of attributes to be selected can be chosen
from a drop down list. The only possible value here is 'single_value'

**use block type exception** *(boolean)* If enabled, an exception to the selected
block type can be specified. When this option is selected another parameter *(ex-
cept block type)* becomes visible in the Parameter View.

**except block type (selection)** The attributes matching this block type will be
removed from the final output even if they matched the previously mentioned
6. Data Transformation

block type.

**numeric condition (string)** The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition ' > 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: ' > 6 & & < 11' or ' < = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like ' (> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after ' > ', ' = ' and ' < ' e.g. ' < 5' will not work, so use ' < 5' instead.

**include special attributes (boolean)** The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch.

**invert selection (boolean)** If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

**treat missing values as duplicates (boolean)** This parameter specifies if missing values should be treated as duplicates or not. If set to true, missing values are considered as duplicate values.

**Tutorial Processes**

Removing duplicate values from the Golf data set on the basis of the Outlook and Wind attributes

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the Outlook attribute has three possible values i.e. 'sunny', 'rain' and 'overcast'. The Wind attribute has two possible values i.e. 'true' and 'false'. The Remove
Duplicates operator is applied on this ExampleSet to remove duplicate examples on the basis of the Outlook and Wind attributes. The \textit{attribute filter type} parameter is set to 'value type' and the \textit{value type} parameter is set to 'nominal', thus two examples that have same values in their Outlook and Wind attributes are considered as duplicate. Note that the Play attribute is not selected although its value type is nominal because it is a special attribute (because it has label role). To select attributes with special roles the \textit{include special attributes} parameter should be set to true. The Outlook and Wind attributes have 3 and 2 possible values respectively. Thus the resultant ExampleSet will have 6 examples at most i.e. one example for each possible combination of attribute values. You can see the resultant ExampleSet in the Results Workspace. You can see that it has 6 examples and all examples have a different combination of the Outlook and Wind attribute values.

\begin{center}
\includegraphics[width=0.8\textwidth]{filtering_diagram.png}
\end{center}

\section*{Filter Example Range}

This operator selects which examples (i.e. rows) of an ExampleSet should be kept and which examples should be removed. Examples within the specified index range are kept, remaining examples are removed.
6. Data Transformation

Description

This operator takes an ExampleSet as input and returns a new ExampleSet including only those examples that are within the specified index range. Lower and upper bound of index range are specified using first example and last example parameters. This operator may reduce the number of examples in an ExampleSet but it has no effect on the number of attributes. The Select Attributes operator is used to select required attributes.

If you want to filter examples by options other than index range, you may use the Filter Examples operator. It takes an ExampleSet as input and returns a new ExampleSet including only those examples that satisfy the specified condition. Several predefined conditions are provided; users can select any of them. Users can also define their own conditions to filter examples. The Filter Examples operator is frequently used to filter examples that have (or do not have) missing values. It is also frequently used to filter examples with correct or wrong predictions (usually after testing a learnt model).

Input Ports

descriptor set input (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

example set output (exa) A new ExampleSet including only the examples that are within the specified index range is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

first example (integer) This parameter is used to set the lower bound of the index range. The last example parameter is used to set the upper bound of the index range. Examples within this index range are delivered to the output port. Examples outside this index range are discarded.

last example (integer) This parameter is used to set the upper bound of the index range. The first example parameter is used to set the lower bound of the index range. Examples within this index range are delivered to the output port. Examples outside this index range are discarded.

invert filter (boolean) If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected examples are removed and previously removed examples are selected. In other words it inverts the index range. For example if the first example parameter is set to 1 and the last example parameter is set to 10. Then the output port will deliver an ExampleSet with all examples other than the first ten examples.

Tutorial Processes

Filtering examples using the invert filter parameter

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with offset set to 0. Thus all examples are assigned unique ids from 1 to 14. This is done so that examples can be distinguished easily. A breakpoint is inserted here so that you can have a look at the data set before application of the Filter Example Range operator. In the Filter Example Range operator the first example parameter is set to 5 and the last example parameter is set to 10. The invert filter parameter is also set to true. Thus all examples other than examples in index range 5 to 10 are delivered through the output port. You can clearly identify rows through their ids. Rows with IDs from 1 to 4 and from 11 to 14 make it to the output port.
Sample

This operator creates a sample from an ExampleSet by selecting examples randomly. The size of a sample can be specified on absolute, relative and probability basis.

Description

This operator is similar to the Filter Examples operator in principle that it takes an ExampleSet as input and delivers a subset of the ExampleSet as output. The difference is this that the Filter Examples operator filters examples on the basis of specified conditions. But the Sample operator focuses on the number of examples and class distribution in the resultant sample. Moreover, the samples are generated randomly. The number of examples in the sample can be specified on absolute, relative or probability basis depending on the setting of the sample parameter. The class distribution of the sample can be controlled by the balance data parameter.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.
Output Ports

example set output (exa) A randomized sample of the input ExampleSet is output of this port.
original (ori) ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

sample (selection) This parameter determines how the amount of data is specified.

- **absolute** If the sample parameter is set to 'absolute' the sample is created of an exactly specified number of examples. The required number of examples is specified in the sample size parameter.

- **relative** If the sample parameter is set to 'relative' the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the sample ratio parameter.

- **probability** If the sample parameter is set to 'probability' the sample is created of probability basis. The required probability is specified in the sample probability parameter.

balance data (boolean) You can set this parameter to true if you need to sample differently for examples of a certain class. If this parameter is set to true, sample size, sample ratio and sample probability parameters are replaced by sample size per class, sample ratio per class and sample probability per class parameters respectively. These parameters allow you to specify different sample sizes for different values of the label attribute.

sample size (integer) This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the sample parameter is set to 'absolute' and the balance data parameter is not set to true.
6. Data Transformation

**sample ratio** *(real)* This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative' and the *balance data* parameter is not set to true.

**sample probability** *(real)* This parameter specifies the sample probability for each example. This parameter is only available when the *sample* parameter is set to 'probability' and the *balance data* parameter is not set to true.

**sample size per class** This parameter specifies the absolute sample size per class. This parameter is only available when the *sample* parameter is set to 'absolute' and the *balance data* parameter is set to true.

**sample ratio per class** This parameter specifies the fraction of examples per class. This parameter is only available when the *sample* parameter is set to 'relative' and the *balance data* parameter is set to true.

**sample probability per class** This parameter specifies the probability of examples per class. This parameter is only available when the *sample* parameter is set to 'probability' and the *balance data* parameter is set to true.

**use local random seed** *(boolean)* This parameter indicates if a local random seed should be used for randomizing examples of the sample. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way the examples are randomized, thus the sample will have a different set of examples.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is only available if the *use local random seed* parameter is set to true.

### Tutorial Processes

#### Sampling the Ripley-Set data set

The 'Ripley-Set' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it so that the examples can be identified uniquely. A breakpoint is inserted at this stage so that you can see the ExampleSet before the Sample operator is applied. You can see that there are 250 examples with...
two possible classes: 0 and 1. 125 examples have class 0 and 125 examples have class 1. Now, the Sample operator is applied on the ExampleSet. The sample parameter is set to 'relative'. The balance data parameter is set to true. The sample ratio per class parameter specifies two ratios.

1. Class 0 is assigned ratio 0.2. Thus, of all the examples where label attribute is 0 only 20 percent will be selected. There were 125 examples with class 0, so 25 (i.e. 20% of 125) examples will be selected.

2. Class 1 is assigned ratio 1. Thus, of all the examples where label attribute is 1, 100 percent will be selected. There were 125 examples with class 1, so all 125 (i.e. 100% of 125) examples will be selected.

Run the process and you can verify the results. Also note that the examples are taken randomly. The randomization can be changed by changing the local random seed parameter.

Sample (Stratified)

This operator creates a stratified sample from an ExampleSet. Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. This
6. Data Transformation

operator cannot be applied on data sets without a label or with a numerical label. The size of the sample can be specified on absolute and relative basis.

Description

The stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class labels.

When there are different classes in an ExampleSet, it is sometimes advantageous to sample each class independently. Stratification is the process of dividing examples of the ExampleSet into homogeneous subgroups (i.e. classes) before sampling. The subgroups should be mutually exclusive i.e. every examples in the ExampleSet must be assigned to only one subgroup (or class). The subgroups should also be collectively exhaustive i.e. no example can be excluded. Then random sampling is applied within each subgroup. This often improves the representativeness of the sample by reducing the sampling error.

A real-world example of using stratified sampling would be for a political survey. If the respondents needed to reflect the diversity of the population, the researcher would specifically seek to include participants of various minority groups such as race or religion, based on their proportionality to the total population as mentioned above. A stratified survey could thus claim to be more representative of the population than a survey of simple random sampling or systematic sampling.

In contrast to the simple sampling operator (the Sample operator), this operator performs a stratified sampling of the data sets with nominal label attributes, i.e. the class distributions remains (almost) the same after sampling. Hence, this operator cannot be applied on data sets without a label or with a numerical label. In these cases a simple sampling without stratification should be performed through the Sample operator.
This operator is similar to the Filter Examples operator in principle that it takes an ExampleSet as input and delivers a subset of the ExampleSet as output. The difference is this that the Filter Examples operator filters examples on the basis of specified conditions. But the Sample operator focuses on the number of examples and class distribution in the resultant sample. Moreover, the samples are generated randomly. The number of examples in the sample can be specified on absolute and relative basis depending on the setting of the sample parameter.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is output of the Filter Examples operator in the attached Example Process.

Output Ports

example set output (exa) A randomized sample of the input ExampleSet is output of this port. The class distributions of the sample is (almost) the same as the class distribution of the complete ExampleSet.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

sample (selection) This parameter determines how the amount of data is specified.

- absolute If the sample parameter is set to 'absolute' then the sample is created of an exactly specified number of examples. The required number of examples is specified in the sample size parameter.
6. Data Transformation

- **relative** If the *sample* parameter is set to 'relative' then the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the *sample ratio* parameter.

  *sample size* (*integer*) This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'absolute'.

  *sample ratio* (*real*) This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative'.

  *use local random seed* (*boolean*) This parameter indicates if a *local random seed* should be used for randomizing examples of the sample. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way the examples are randomized, thus sample will have a different set of examples.

  *local random seed* (*integer*) This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**Tutorial Processes**

**Stratified Sampling of the Golf data set**

The 'Golf' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it to select the first 10 examples. This is done to simplify the Example Process otherwise the filtering was not necessary here. A *breakpoint* is inserted here so that you can view the ExampleSet before the application of the Sample (Stratified) operator. As you can see, the ExampleSet has 10 examples. 6 examples (i.e. 60%) belong to class 'yes' and 4 examples (i.e. 40%) belong to class 'no'. The Sample (Stratified) operator is applied on the ExampleSet. The *sample* parameter is set to 'absolute' and the *sample size* parameter is set to 5. Thus the resultant sample will have only 5 examples. The
sample will have the same class distribution as the class distribution of the input ExampleSet i.e. 60% examples with class 'yes' and 40% examples with class 'no'. You can verify this by viewing the results of this process. 3 out of 5 examples (i.e. 60%) have class 'yes' and 2 out of 5 examples (i.e. 40%) have class 'no'.

Sample (Bootstrapping)

This operator creates a bootstrapped sample from an ExampleSet. Bootstrapped sampling uses sampling with replacement, thus the sample may not have all unique examples. The size of the sample can be specified on absolute and relative basis.

Description

This operator is different from other sampling operators because it uses sampling with replacement. In sampling with replacement, at every step all examples have equal probability of being selected. Once an example has been selected for the sample, it remains candidate for selection and it can be selected again in any other coming steps. Thus a sample with replacement can have the same example multiple number of times. More importantly, a sample with replacement can be used to generate a sample that is greater in size than the original ExampleSet.
6. Data Transformation

The number of examples in the sample can be specified on absolute or relative basis depending on the setting of the sample parameter.

Input Ports

example set input \((\text{exa})\) This input port expects an ExampleSet. It is output of the Generate ID operator in the attached Example Process.

Output Ports

example set output \((\text{exa})\) A bootstrapped sample of the input ExampleSet is output of this port.
original \((\text{ori})\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

sample \((\text{selection})\) This parameter determines how the amount of data is specified.

- absolute If the sample parameter is set to 'absolute' the sample is created of the exactly specified number of examples. The required number of examples is specified in the sample size parameter.

- relative If the sample parameter is set to 'relative' the sample is created as a fraction of the total number of examples in the input ExampleSet. The required ratio of examples is specified in the sample ratio parameter.

sample size \((\text{integer})\) This parameter specifies the exact number of examples which should be sampled. This parameter is only available when the sample
6.6. Filtering

Parameter is set to 'absolute'.

**Sample ratio** *(real)* This parameter specifies the fraction of examples which should be sampled. This parameter is only available when the *sample* parameter is set to 'relative'.

**Use weights** *(boolean)* If set to true, example weights will be considered during the bootstrapping if such weights are present.

**Use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomizing examples of the sample. Using the same value of the *local random seed* will produce the same sample. Changing the value of this parameter changes the way the examples are randomized, thus the sample will have a different set of examples.

**Local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**Tutorial Processes**

**Bootstrapped Sampling of the Golf data set**

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to create an *id* attribute with ids starting from 1. This is done so that the examples can be identified uniquely, otherwise the *id* attribute was not necessary here. A *breakpoint* is inserted here so that you can view the ExampleSet before the application of the Sample (Bootstrapping) operator. As you can see, the ExampleSet has 14 examples. The Sample (Bootstrapping) operator is applied on the ExampleSet. The *sample* parameter is set to 'absolute' and the *sample size* parameter is set to 140. Thus a sample 10 times in size of the original ExampleSet is generated. Instead of repeating each example of the input ExampleSet 10 times, examples are selected randomly. You can verify this by seeing the results of this process in the Results Workspace.
6. Data Transformation

Split Data

This operator produces the desired number of subsets of the given ExampleSet. The ExampleSet is partitioned into subsets according to the specified relative sizes.

Description

The Split Data operator takes an ExampleSet as its input and delivers the subsets of that ExampleSet through its output ports. The number of subsets (or partitions) and the relative size of each partition are specified through the partitions parameter. The sum of the ratio of all partitions should be 1. The sampling type parameter decides how the examples should be shuffled in the resultant partitions. For more information about this operator please study the parameters section of this description. This operator is different from other sampling and filtering operators in the sense that it is capable of delivering multiple partitions of the given ExampleSet.
Input Ports

dexample set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

Output Ports

partition (par) This operator can have multiple number of partition ports. The number of useful partition ports depends on the number of partitions (or subsets) this operator is configured to produce. The partitions parameter is used for specifying the desired number of partitions.

Parameters

partitions (enumeration) This is the most important parameter of this operator. It specifies the number of partitions and the relative ratio of each partition. The user just requires to specify the ratio of all partitions. The number of required partitions is not explicitly specified by the user because it is calculated automatically by the number of ratios specified in this parameter. The ratios should be between 0 and 1. The sum of all ratios should be 1. For better understanding of this parameter please study the attached Example Process.

dampling type (selection) The Split Data operator can use several types of sampling for building the subsets. Following options are available:

- **Linear sampling** Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.

- **Shuffled sampling** Shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.

- **Stratified sampling** Stratified sampling builds random subsets and en-
6. Data Transformation

...sures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of the class labels.

**use local random seed** (boolean) Indicates if a local random seed should be used for randomizing examples of a subset. Using the same value of local random seed will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed** (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

### Tutorial Processes

#### Creating partitions of the Golf data set using the Split Data operator

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it so the examples can be identified uniquely. A breakpoint is inserted here so the ExampleSet can be seen before the application of the Split Data operator. It can be seen that the ExampleSet has 14 examples which can be uniquely identified by the id attribute. The examples have ids from 1 to 14. The Split Data operator is applied next. The *sampling type* parameter is set to 'linear sampling'. The *partitions* parameter is configured to produce two partitions with ratios 0.8 and 0.2 respectively. The partitions can be seen in the Results Workspace. The number of examples in each partition is calculated by this formula:

\[
\frac{(\text{Total number of examples})}{(\text{sum of ratios})} \times \text{ratio of this partition}
\]
6.7. Sorting

If the answer is a decimal number it is rounded off. The number of examples in each partition turns out to be:

- \( \frac{14}{0.8 + 0.2} \times 0.8 = 11.2 \) which is rounded off to 11
- \( \frac{14}{0.8 + 0.2} \times 0.2 = 2.8 \) which is rounded off to 3

It is a good practice to adjust ratios such that the sum of ratios is 1. But this operator also works if the sum of ratios is lower than or greater than 1. For example if two partitions are created with ratios 1.0 and 0.4. The resultant partitions would be calculated as follows:

- \( \frac{14}{1.0 + 0.4} \times 1.0 = 10 \)
- \( \frac{14}{1.0 + 0.4} \times 0.4 = 4 \)

Sort

This operator sorts the input ExampleSet in ascending or descending order according to a single attribute.
6. Data Transformation

Description

This operator sorts the ExampleSet provided at the input port. The complete data set is sorted according to a single attribute. This attribute is specified using the attribute name parameter. Sorting is done in increasing or decreasing direction depending on the setting of the sorting direction parameter.

Input Ports

example set input (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

example set output (exa) The sorted ExampleSet is output of this port.
original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

attribute name (string) This parameter is used to specify the attribute which should be used for sorting the ExampleSet.

sorting direction This parameter indicates the direction of the sorting. The ExampleSet can be sorted in increasing(ascending) or decreasing(descending) order.
6.7. Sorting

Tutorial Processes

Sorting the Golf data set according to Temperature

The 'Golf' data set is loaded using the Retrieve operator. The Sort operator is applied on it. The attribute name parameter is set to 'Temperature'. The sort direction parameter is set to 'increasing'. Thus the 'Golf' data set is sorted in ascending order of the 'Temperature' attribute. The example with the smallest value of the 'Temperature' attribute becomes the first example and the example with the largest value of the 'Temperature' attribute becomes the last example of the ExampleSet.

Sorting on multiple attributes

This Example Process shows how two Sort operators can be used to sort an ExampleSet on two attributes. The 'Golf' data set is loaded using the Retrieve operator. The Sort operator is applied on it. The attribute name parameter is set to 'Temperature'. The sort direction parameter is set to 'increasing'. Then another Sort operator is applied on it. The attribute name parameter is set to 'Humidity' this time. The sort direction parameter is set to 'increasing'. Thus the 'Golf' data set is sorted in ascending order of the 'Humidity' attribute. The example with smallest value of the 'Humidity' attribute becomes the first exam-
6. Data Transformation

ple and the example with the largest value of the 'Humidity' attribute becomes the last example of the ExampleSet. If some examples have the same value of the 'Humidity' attribute, they are sorted using the 'Temperature' attribute. Where examples have same value of the 'Humidity' attribute then the examples with smaller value of the 'Temperature' attribute precede the examples with higher value of the 'Temperature' attribute. This can be seen in the Results Workspace.

Pivot

This operator rotates an ExampleSet by grouping multiple examples of same groups to single examples.

Description

The Pivot operator rotates the given ExampleSet by grouping multiple examples of same groups to single examples. The group attribute parameter specifies the grouping attribute (i.e. the attribute which identifies examples belonging to the groups). The resultant ExampleSet has \( n \) examples where \( n \) is the number of unique values of the group attribute. The index attribute parameter specifies the attribute whose values are used to identify the examples inside the groups. The
values of this attribute are used to name the group attributes which are created during the pivoting. Typically the values of such an attribute capture subgroups or dates. The resultant ExampleSet has $m$ regular attributes in addition to the group attribute where $m$ is the number of unique values of the index attribute. If the given ExampleSet contains example weights (i.e. an attribute with weight role), these weights may be aggregated in each group to maintain the weightings among groups. This description can be easily understood by studying the attached Example Process.

**Differentiation**

**Transpose** The Transpose operator simply rotates the given ExampleSet (i.e. interchanges rows and columns) but the Pivot operator provides additional options like grouping and handling weights. See page 612 for details.

**Input Ports**

**example set input** *(exa)* This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process.

**Output Ports**

**example set output** *(exa)* The ExampleSet produced after pivoting is the output of this port.

**original** *(ori)* The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
6. Data Transformation

Parameters

group attribute \textit{(string)} This parameter specifies the grouping attribute (i.e. the attribute which identifies examples belonging to the groups). The resultant ExampleSet has \( n \) examples where \( n \) is the number of unique values of the group attribute.

index attribute \textit{(string)} This parameter specifies the attribute whose values are used to identify the examples inside the groups. The values of this attribute are used to name the group attributes which are created during the pivoting. Typically the values of such an attribute capture subgroups or dates. The resultant ExampleSet has \( m \) regular attributes in addition to the group attribute where \( m \) is the number of unique values of the index attribute.

consider weights \textit{(boolean)} This parameter specifies whether attribute weights (if any) should be kept and aggregated or ignored.

weight aggregation \textit{(selection)} This parameter is only available when the \textit{consider weights} parameter is set to true. It specifies how example weights should be aggregated in the groups. It has the following options: average, variance, standard deviation, count, minimum, maximum, sum, mode, median, product.

skip constant attributes \textit{(boolean)} This parameter specifies if the attributes should be skipped if their value never changes within a group.

data management \textit{(selection)} This is an expert parameter. There are different options, users can choose any of them

Related Documents

Transpose (612)

Tutorial Processes

Introduction to the Pivot operator
This Example Process starts with the Subprocess operator. There is a sequence of operators in this Subprocess operator that produces an ExampleSet that is easy to understand. A breakpoint is inserted after the Subprocess operator to show this ExampleSet. The Pivot operator is applied on this ExampleSet. The group attribute and index attribute parameters are set to 'group_attribute' and 'index_attribute' respectively. The consider weights parameter is set to true and the weight aggregation parameter is set to 'sum'. The group attribute has 5 possible values therefore the pivoted ExampleSet has 5 examples i.e. one for each possible value of the group attribute. The index attribute has 5 possible values therefore the pivoted ExampleSet has 5 regular attributes (in addition to the group attribute). Here is an explanation of values of the first example of the pivoted ExampleSet. The remaining examples also follow the same idea.

The value of the group attribute of the first example of the pivoted ExampleSet is 'group0', therefore all values of this example will be derived from all examples of the input ExampleSet where the group attribute had the value 'group0'. The ids of examples with 'group0' in the input ExampleSet are 12, 16, 19 and 20. In the coming explanation these examples will be called group0 examples for simplicity.

- The value of the weight_attribute attribute of the pivoted ExampleSet is 11. It is the sum of weights of group0 examples i.e. 4 + 4 + 0 + 3 = 11. The weights were added because the weight aggregation parameter is set to 'sum'.

- The value of the value_attribute_index0 attribute of the pivoted ExampleSet is 4. Only two examples (id 12 and 16) of the group0 examples had 'index0' in index_attribute. The value of the latter of these examples (id 16) is selected i.e. 4 is selected.

- The value of the value_attribute_index1 attribute of the pivoted ExampleSet is 1. Only one example (id 19) of the group0 examples had 'index1' in index_attribute. Therefore its value (i.e. 1) is selected.

- The value of the value_attribute_index2 attribute of the pivoted ExampleSet is undefined because no example of the group0 examples had 'index2' in
6. Data Transformation

index attribute. Therefore its value is missing in the pivoted ExampleSet.

- The value of the value_attribute_index3 attribute of the pivoted ExampleSet is 3. Only one example (id 20) of the group0 examples had 'index3' in index_attribute. Therefore its value (i.e. 3) is selected.

- The value of the value_attribute_index4 attribute of the pivoted ExampleSet is undefined because no example of the group0 examples had 'index4' in index_attribute. Therefore its value is missing in the pivoted ExampleSet.

De-Pivot

This operator transforms the ExampleSet by converting the examples of the selected attributes (usually attributes that measure the same characteristic) into examples of a single attribute.

Description

This operator is usually used when your ExampleSet has multiple attributes that measure the same characteristic (may be at different time intervals) and you want to merge these observations into a single attribute without loss of information. If the original ExampleSet has \( n \) examples and \( k \) attributes that measure the same characteristic, after application of this operator the ExampleSet will have \( k \times n \)
examples. The $k$ attributes will be combined into one attribute. This attribute will have $n$ examples of each of the $k$ attributes. This can be easily understood by studying the attached Example Process.

In other words, this operator converts an ExampleSet by dividing examples which consist of multiple observations (at different times) into multiple examples, where each example covers one point in time. An index attribute is added in the ExampleSet, which denotes the actual point in time the example belongs to after the transformation.

The keep missings parameter specifies whether an example should be kept, even if it has missing values for all series at a certain point in time. The create nominal index parameter is only applicable if only one time series per example exists. Instead of using a numeric index, then the names of the attributes representing the single time points are used as index attribute values.

**Input Ports**

**example set input (exa)** This input port expects an ExampleSet. It is the output of the Subprocess operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**example set output (exa)** The selected attributes are converted into examples of a new attribute and the resultant ExampleSet is output of this port.

**original (ori)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
6. Data Transformation

Parameters

attribute name (list) This parameter maps a number of source attributes onto result attributes. The attribute name parameter is used for specifying the group of attributes that you want to combine and the name of the new attribute. The attributes of a group are selected through a regular expression. There can be numerous groups with each group having multiple attributes.

index attribute (string) This parameter specifies the name of the newly created index attribute. The index attribute is used for differentiating between examples of different attributes of a group after the transformation.

create nominal index (boolean) The create nominal index parameter is only applicable if only one time series per example exists. Instead of using a numeric index, then the names of the attributes representing the single time points are used as index attribute values.

keep missings (boolean) The keep missings parameter specifies whether an example should be kept, even if it has missing values for all series at a certain point in time.

Tutorial Processes

Merging multiple attributes that measure the same characteristic into a single attribute

This process starts with the Subprocess operator which delivers an Example-Set. The subprocess is used for creating a sample ExampleSet therefore it is not important to understand what is going on inside the subprocess. A breakpoint is inserted after the subprocess so that you can have a look at the ExampleSet. You can see that the ExampleSet has 14 examples and it has two attributes i.e. 'Morning' and 'Evening'. These attributes measure the temperature of an area in morning and evening respectively. We want to convert these attributes into a single attribute but we still want to be able to differentiate between morning and
evening temperatures.

This process starts with the Subprocess operator which delivers an ExampleSet. The subprocess is used for creating a sample ExampleSet therefore it is not important to understand what is going on inside the subprocess. A breakpoint is inserted after the subprocess so that you can have a look at the ExampleSet. You can see that the ExampleSet has 14 examples and it has two attributes i.e. 'Morning' and 'Evening'. These attributes measure the temperature of an area in morning and evening respectively. We want to convert these attributes into a single attribute but we still want to be able to differentiate between morning and evening temperatures.

The De-Pivot operator is applied on this ExampleSet to perform this task. The attribute name parameter is used for specifying the group of attributes that you want to combine and the name of the new attribute. The attributes of a group are selected through a regular expression. There can be numerous groups with each group having multiple attributes. In our case, there is only one group which has all the attributes of the ExampleSet (i.e. both 'Morning' and 'Evening' attributes). The new attribute is named 'Temperatures' and the regular expression: '.*' is used for selecting all the attributes of the ExampleSet. The index attribute is used for differentiating between examples of different attributes of a group after transformation. The name of the index attribute is set to 'Time'. The create nominal index parameter is also set to true so that the resultant ExampleSet is more self-explanatory.

Execute the process and have a look at the resultant ExampleSet. You can see that there are 28 examples in this ExampleSet. The original ExampleSet had 14 examples, and 2 attributes were grouped, therefore the resultant ExampleSet has 28 (i.e. 14 x 2) examples. There are 14 examples from the Morning attribute and 14 examples of the Evening attribute in the 'Temperatures' attribute. The 'Time' attribute explains whether an example measures morning or evening temperature.
6. Data Transformation

Transposte

This operator transposes the input ExampleSet i.e. the current rows become columns of the output ExampleSet and current columns become rows of the output ExampleSet. This operator works very similar to the well known transpose operation for matrices.

Description

This operator transposes the input ExampleSet i.e. the current rows become the columns of the output ExampleSet and the current columns become the rows of the output ExampleSet. In other words every example or row becomes a column with attribute values and each attribute column becomes an example row. This operator works very similar to the well known transpose operation for matrices. The transpose of a transpose of a matrix is same as the original matrix, but the same rule cannot be applied here because the types of the original ExampleSet and the transpose of the transpose of an ExampleSet may be different.

If an id attribute is part of the input ExampleSet, the ids will become the names of the new attributes. The names of the old attributes will be transformed into the id values of a new id attribute. All other new attributes will have regular role after the transformation. You can use the Set Role operator after the transpose operator to assign roles to new attributes.

If all old attributes have the same value type, all new attributes will have the same value type. If at least one nominal attribute is part of the input ExampleSet,
the type of all new attributes will be nominal. If the old attribute values were all mixed numbers, the type of all new attributes will be real. This operator produces a copy of the data in the main memory. Therefore, it should not be used on very large data sets.

Input Ports

example set input *(exa)* This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

example set output *(exa)* The transpose of the input ExampleSet is output of this port.
original *(ori)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Tutorial Processes

Different scenarios of Transpose

There are four different cases in this Example Process:

*Case 1:* The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. You can see that the 'Golf' data set has no *id* attribute. The types of attributes are different including attributes of nominal type. Press the *Run* button to continue. Now the Transpose operator is applied on the 'Golf'
6. Data Transformation

data set. A breakpoint is inserted here so that you can see the ExampleSet after the application of the Transpose operator. Here you can see that a new attribute with id role has been created. The values of the new id attribute are the names of the old attributes. New attributes are named in a general format like 'att_1', 'att_2' etc because the input ExampleSet had no id attribute. The type of all new attributes is nominal because there were attributes with different types including at least one nominal attribute in the input ExampleSet.

Case 2: The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. You can see that the 'Iris' data set has an id attribute. The types of attributes are different including attributes of nominal type. Press the Run button to continue. Now the Transpose operator is applied on the 'Iris' data set. A breakpoint is inserted here so that you can see the ExampleSet after the application of the Transpose operator. Here you can see that a new attribute with id role has been created. The values of the new id attribute are the names of the old attributes. The ids of the old ExampleSet become names of the new attributes. The type of all new attributes is nominal because there were attributes with different types including at least one nominal attribute in the input ExampleSet.

Case 3:The 'Market-Data' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. You can see that the 'Market-Data' data set has no special attributes. The type of all attributes is integer. Press the Run button to continue. Now the Transpose operator is applied on the 'Market-Data' data set. A breakpoint is inserted here so that you can see the ExampleSet after the application of the Transpose operator. Here you can see that a new attribute with id role has been created. Values of the new id attribute are the names of the old attributes. New attributes are named in a general format like 'att_1', 'att_2' etc because the input ExampleSet had no id attribute. The Type of all new attributes is real because there were attributes with mixed numbers type in the input ExampleSet.
6.9. Aggregation

Case 4: The 'Golf-Testset' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Transpose operator. The Transpose operator is applied on the 'Golf-Testset' data set. Then the Transpose operator is applied on the output of the first Transpose operator. Note that the types of the attributes of the original ExampleSet and the Transpose of the Transpose of the original data set are different.

Aggregate

This operator performs the aggregation functions known from SQL. This operator provides a lot of functionalities in the same format as provided by the SQL aggregation functions. SQL aggregation functions and GROUP BY and HAVING clauses can be imitated using this
6. Data Transformation

operator.

Description

The Aggregate operator creates a new ExampleSet from the input ExampleSet showing the results of the selected aggregation functions. Many aggregation functions are supported including SUM, COUNT, MIN, MAX, AVERAGE and many other similar functions known from SQL. The functionality of the GROUP BY clause of SQL can be imitated by using the group by attributes parameter. You need to have a basic understanding of the GROUP BY clause of SQL for understanding the use of this parameter because it works exactly the same way. If you want to imitate the known HAVING clause from SQL, you can do that by applying the Filter Examples operator after the Aggregation operator. This operator imitates aggregation functions of SQL. It focuses on obtaining summary information, such as averages and counts etc. It can group examples in an ExampleSet into smaller sets and apply aggregation functions on those sets. Please study the attached Example Process for better understanding of this operator.

Input Ports

description

description

example set (exa) This input port expects an ExampleSet. It is output of the Filter Examples operator in the attached Example Process. Output of other operators can also be used as input.

Output Ports

description

description

example set (exa) The ExampleSet generated after applying the specified aggregation functions is output of this port.

original (ori) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

**use default aggregation** *(boolean)* This parameter allows you to define the default aggregation for selected attributes. A number of parameters become available if this parameter is set to true. These parameters allow you to select the attributes and corresponding default aggregation function.

**attribute filter type** *(selection)* This parameter allows you to select the attribute selection filter; the method you want to use for selecting attributes. It has the following options:

- **all** This option simply selects all the attributes of the ExampleSet. This is the default option.
- **single** This option allows selection of a single attribute. When this option is selected another parameter *(attribute)* becomes visible in the Parameter View.
- **subset** This option allows selection of multiple attributes through a list. All attributes of ExampleSet are present in the list; required attributes can be easily selected. This option will not work if meta data is not known. When this option is selected another parameter becomes visible in the Parameter View.
- **regular_expression** This option allows you to specify a regular expression for attribute selection. When this option is selected some other parameters *(regular expression, use except expression)* become visible in the Parameter View.
- **value_type** This option allows selection of all the attributes of a particular type. It should be noted that types are hierarchical. For example *real* and *integer* types both belong to the *numeric* type. Users should have basic understanding of type hierarchy when selecting attributes through this option. When this option is selected some other parameters *(value type, use value type exception)* become visible in the Parameter View.
- **block_type** This option is similar in working to the *value_type* option.
This option allows selection of all the attributes of a particular block type. It should be noted that block types may be hierarchical. For example, `value_series_start` and `value_series_end` block types both belong to the `value_series` block type. When this option is selected some other parameters (`block type`, `use block type exception`) become visible in the Parameter View.

- **no_missing_values** This option simply selects all the attributes of the ExampleSet which don't contain a missing value in any example. Attributes that have even a single missing value are removed.

- **numeric value filter** When this option is selected another parameter (`numeric condition`) becomes visible in the Parameter View. All numeric attributes whose all examples satisfy the mentioned numeric condition are selected. Please note that all nominal attributes are also selected irrespective of the given numerical condition.

attribute *(string)* The required attribute can be selected from this option. The attribute name can be selected from the drop down box of parameter attribute if the meta data is known.

attributes *(string)* The required attributes can be selected from this option. This opens a new window with two lists. All attributes are present in the left list. Attributes can be shifted to the right list, which is the list of selected attributes.

regular expression *(string)* The attributes whose name match this expression will be selected. Regular expression is a very powerful tool but needs a detailed explanation to beginners. It is always good to specify the regular expression through the edit and preview regular expression menu. This menu gives a good idea of regular expressions and it also allows you to try different expressions and preview the results simultaneously.

use except expression *(boolean)* If enabled, an exception to the first regular expression can be specified. When this option is selected another parameter (`except regular expression`) becomes visible in the Parameter View.

except regular expression *(string)* This option allows you to specify a regular expression. Attributes matching this expression will be filtered out even if they match the first regular expression (regular expression that was specified in the
regular expression parameter).

**value type** *(selection)* The type of attributes to be selected can be chosen from a drop down list.

**use value type exception** *(boolean)* If enabled, an exception to the selected type can be specified. When this option is enabled, another parameter *(except value type)* becomes visible in the Parameter View.

**except value type** *(selection)* The attributes matching this type will not be selected even if they match the previously mentioned type i.e. *value type* parameter's value.

**block type** *(selection)* The block type of attributes to be selected can be chosen from a drop down list.

**use block type exception** *(boolean)* If enabled, an exception to the selected block type can be specified. When this option is selected another parameter *(except block type)* becomes visible in the Parameter View.

**except block type** *(selection)* The attributes matching this block type will not be selected even if they match the previously mentioned block type i.e. *block type* parameter's value.

**numeric condition** *(string)* The numeric condition for testing examples of numeric attributes is specified here. For example the numeric condition '> 6' will keep all nominal attributes and all numeric attributes having a value of greater than 6 in every example. A combination of conditions is possible: '> 6 & & < 11' or '< = 5 || < 0'. But & & and || cannot be used together in one numeric condition. Conditions like '(> 0 & & < 2) || (> 10 & & < 12)' are not allowed because they use both & & and ||. Use a blank space after '>' , '=' and '< ' e.g. '< 5' will not work, so use '< 5' instead.

**include special attributes** *(boolean)* The special attributes are attributes with special roles which identify the examples. In contrast regular attributes simply describe the examples. Special attributes are: id, label, prediction, cluster, weight and batch. By default all special attributes selected irrespective of the conditions in the Select Attribute operator. If this parameter is set to true, Special attributes are also tested against conditions specified in the Select Attribute operator and only those attributes are selected that satisfy the conditions.

**invert selection** *(boolean)* If this parameter is set to true, it acts as a NOT gate, it reverses the selection. In that case all the selected attributes are unselected.
6. Data Transformation

and previously unselected attributes are selected. For example if attribute 'att1' is selected and attribute 'att2' is unselected prior to checking of this parameter. After checking of this parameter 'att1' will be unselected and 'att2' will be selected.

default aggregation function This parameter is only available when the use default aggregation parameter is set to true. It is used for specifying the default aggregation function for the selected attributes.

aggregation attributes This parameter is one of the most important parameters of the operator. It allows you to select attributes and the aggregation function to apply on them. Many aggregation functions are available including count, average, minimum, maximum variance and many more.

group by attributes This operator can group examples of the input ExampleSet into smaller groups using this parameter. The aggregation functions are applied on these groups. This parameter allows the Aggregate operator to replicate the functionality of the known GROUP BY clause of SQL.

count all combinations (boolean) This parameter indicates if all possible combinations of the values of the group by attributes are counted, even if they don't occur. All possible combinations may result in a huge number so handle this parameter carefully.

only distinct (boolean) This parameter indicates if only examples with distinct values for the aggregation attribute should be used for the calculation of the aggregation function.

ignore missings (boolean) This parameter indicates if missing values should be ignored and aggregation functions should be applied only on existing values. If this parameter is not set to true then the aggregated value will be a missing value in the presence of missing values in the selected attribute.

Tutorial Processes

Imitating an SQL aggregation query using the Aggregate operator
6.9. Aggregation

This Example Process discusses an arbitrary scenario. Then describes how this scenario could be handled using SQL aggregation functions. Then the SQL's solution is imitated in RapidMiner. The Aggregate operator plays a key role in this process.

Let us assume a scenario where we want to apply certain aggregation functions on the Golf data set. We don't want to include examples where the Outlook attribute has the value 'overcast'. We group the remaining examples of the 'Golf' data set by values of the Play and Wind attributes. We wish to find the average Temperature and average Humidity for these groups. Once these averages have been calculated, we want to see only those examples where the average Temperature is above 71. Lastly, we want to see the results in ascending order of the average Temperature.

This problem can be solved by the following SQL query:

```sql
SELECT Play, Wind, AVG (Temperature), AVG (Humidity)
FROM Golf
WHERE Outlook NOT LIKE 'overcast'
GROUP BY Play, Wind
HAVING AVG (Temperature) > 71
ORDER BY AVG (Temperature)
```

The SELECT clause selects the attributes to be displayed. The FROM clause specifies the data set. The WHERE clause pre-excludes the examples where the Outlook attribute has value 'overcast'. The GROUP BY clause groups the data set according to the specified attributes. The HAVING clause filters the results after the aggregation functions have been applied. Finally the ORDER BY clause sorts the results in ascending order of the Temperature averages.

Here is how this scenario can be tackled using RapidMiner. First of all the Retrieve operator is used for loading the 'Golf' data set. This is similar to the FROM clause. Then the Select Attributes operator is applied on it to select the
required attributes. This works a little different from the SQL query. If we select only the Play and Wind attributes as in the query, then the coming operators cannot be applied. Thus we select all attributes for now. You will see later that the attribute set will be reduced automatically, thus the Select Attributes operator is not really required here. Then the Filter Examples operator is applied to pre-exclude examples where the Outlook attribute has the value 'overcast'. This is similar to the WHERE clause of SQL. Then the Aggregate operator is applied on the remaining examples. The Aggregate operator performs a number of tasks here. Firstly, it specifies the aggregation functions using the aggregation attributes parameter. We need averages of the Temperature and Humidity attribute; this is specified using the aggregation attributes parameter. Secondly, we do not want the averages of the entire data set. We want the averages by groups, grouped by the Play and Wind attribute values. These groups are specified using the group by attributes parameter of the Aggregate operator. Thirdly, required attributes are automatically filtered by this operator. Only those attributes appear in the resultant data set that have been specified in the Aggregate operator. Next, we are interested only in those examples where the average Temperature is greater than 71. This condition can be applied using the Filter Examples operator. This step is similar to the HAVING clause. Lastly we want the results to be sorted. The Sort operator is used to do the required sorting. This step is very similar to the ORDER BY clause. Breakpoints are inserted after every operator in the Example Process so that you can understand the part played by each operator.
Append

This operator builds a merged ExampleSet from two or more compatible ExampleSets by adding all examples into a combined set.

Description

This operator builds a merged ExampleSet from two or more compatible ExampleSets by adding all examples into a combined set. All input ExampleSets must have the same attribute signature. This means that all ExampleSets must have the same number of attributes. Names and roles of attributes should be the same in all input ExampleSets. Please note that the merged ExampleSet is built in memory and this operator might therefore not be applicable for merging huge data set tables from database. In that case other preprocessing tools should be used that aggregate, join, and merge tables into one table which is then used by RapidMiner.

Input Ports

element set (exa) The Append operator can have multiple inputs. When one input port is connected, another input port becomes available which is ready to accept another input (if any). This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process. Output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data. The Retrieve operator provides meta data along-with data.

Output Ports

merged set (mer) The merged ExampleSet is delivered through this port.
6. Data Transformation

Parameters

data management \((selection)\) This is an expert parameter. A long list is provided; users can select any option from this list.

Tutorial Processes

Merging Golf and Golf-Testset data sets

In this process the 'Golf' data set and 'Golf-Testset' data set are loaded using the Retrieve operators. \textit{Breakpoints} are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. When you continue the process, you will see the 'Golf-Testset' data set. It also has 14 examples. The Append operator is applied to merge these two ExampleSets into a single ExampleSet. The merged ExampleSet has all examples from all input ExampleSets, thus it has 28 examples. You can see that both input ExampleSets had the same number of attributes, same names and roles of attributes. This is why the Append operator could produce a merged ExampleSet.
Join

This operator joins two ExampleSets using specified key attribute(s) of the two ExampleSets.

Description

The Join operator joins two ExampleSets using one or more attributes of the input ExampleSets as key attributes. Identical values of the key attributes indicate matching examples. An attribute with id role is selected as key by default but an arbitrary set of one or more attributes can be chosen as key. Four types of joins are possible: inner, left, right and outer join. All these types of joins are explained in the parameters section.

Input Ports

left (left) The left input port expects an ExampleSet. This ExampleSet will be used as the left ExampleSet for the join.
right (rig) The right input port expects an ExampleSet. This ExampleSet will be used as the right ExampleSet for the join.

Output Ports

join (joi) The join of the left and right ExampleSets is delivered through this port.
6. Data Transformation

Parameters

**remove double attributes** *(boolean)* This parameter indicates if double attributes should be removed or renamed. Double attributes are those attributes that are present in both ExampleSets. If this parameter is checked, from attributes which are present in both ExampleSets only the one from the left ExampleSet will be taken and the one from the right ExampleSet would be discarded. The key attributes will always be taken from the left ExampleSet. Please note that this check for double attributes will only be applied for regular attributes. Special attributes of the right ExampleSet which do not exist in the left ExampleSet will simply be added. If they already exist they are simply skipped.

**join type** *(selection)* This parameter specifies which Join should be performed. You can easily understand these joins by studying the Example Process. Four types of joins are supported:

- **inner** The resultant ExampleSet will contain only those examples where the *key attributes* of both input ExampleSets match i.e. have the same value.

- **left** This is also called left outer join. The resultant ExampleSet will contain all records from the left ExampleSet. If no matching records were found in the right ExampleSet, then its fields will contain the null value i.e. '?' will be stored there. The left join will always contain the results of the inner join; however it can contain some examples that have no matching examples in the right ExampleSet.

- **right** This is also called right outer join. The resultant ExampleSet will contain all records from the right ExampleSet. If no matching records were found in the left ExampleSet, then its fields will contain the null values i.e. '?' will be stored there. The right join will always contain the results of the inner join; however it can contain some examples that have no matching examples in the left ExampleSet.

- **outer** This is also called full outer join. This type of join combines the results of the left and the right join. All examples from both ExampleSets
will be part of the resultant ExampleSet, whether the matching key attribute value exists in the other ExampleSet or not. If no matching key attribute value was found then the corresponding resultant fields will have a null value. The outer join will always contain the results of the inner join; however it can contain some examples that have no matching examples in the other ExampleSet.

**use id attribute as key (boolean)** This parameter indicates if the attribute with the id role should be used as the key attribute. This option is checked by default. If unchecked, then you have to specify the key attributes for both left and right ExampleSets. Identical values of the key attributes indicate matching examples.

**key attributes** This parameter specifies attribute(s) which are used as the key attributes. Identical values of the key attributes indicate matching examples. For each key attribute from the left ExampleSet a corresponding one from the right ExampleSet has to be chosen. Choosing appropriate key attributes is critical for obtaining the desired results. This parameter is available only when the use id attribute as key parameter is unchecked.

**keep both join attributes** If checked, both columns of a join pair will be kept. Usually this is unnecessary since both attributes are identical. It may be useful to keep such a column if there are missing values on one side.

**Tutorial Processes**

**Different types of join**

The last operator of this process is the Join operator. The sequence of operators leading to the left input port of the Join operator is used to generate the left ExampleSet. Similarly, the sequence of operators leading to the right input port of the Join operator is used to generate the right ExampleSet. The sequence of operators leading to the left and right input ports of the Join operator are pretty similar.

In both cases the Retrieve operator is used to load the 'Golf' data set. Then the
Generate Attribute operator is applied on it to generate a dummy attribute. All attributes of the 'Golf' data set other than the 'Play' attribute and the newly generated attribute are discarded because the keep all parameter is unchecked. Then the Generate ID operator is applied to generate an attribute with the id role. This attribute will later be used as the key attribute for joining.

The only difference is that for the left ExampleSet, the name of the attribute generated by the Generate Attribute operator is 'Golf 1 attribute' and for the right ExampleSet the name of this attribute is 'Golf 2 attribute'. The other major difference is in the value of the offset parameter of the Generate ID operator. For the left ExampleSet the offset parameter of the Generate ID operator is set to 0 and for the right ExampleSet it is set to 7. Thus the left ExampleSet has id from 1 to 14 and the right ExampleSet has id from 8 to 21. The breakpoints are inserted after the Generate ID operator so that you can have a look at the left and right ExampleSets before application of the Join operator.

The use id attribute as key parameter of the Join operator is set to true. Thus attributes with id role will be used to join the left and right ExampleSets. The remove double attributes parameter is also checked. Thus regular attributes common in both input ExampleSets would appear just once in the resultant ExampleSet. Only the 'Play' and 'id' attributes are common in both the ExampleSets, but as they are not regular attributes so the remove double attributes parameter has no effect on them. As mentioned earlier the key attributes will always be taken from the left ExampleSet. Please note that this check for double attributes will only be applied for regular attributes. Special attributes of the right ExampleSet which do not exist in the left ExampleSet will simply be added. If they already exist they are simply skipped.

In this example process the join type is set as inner join. You can change it to other values and run the process again. Here is an explanation of results that are obtained on each value of the join type parameter.

If inner join is selected as join type the resultant ExampleSet has examples with ids from 8 to 14. This is because the inner join delivers only those examples where the key attribute of both input ExampleSets have the same values. In this example process, the left ExampleSet has ids from 1 to 14 and the right
ExampleSet has ids from 8 to 21. Thus examples with ids from 8 to 14 have equal value of the **key attribute** (i.e. the id attribute).

If *left join* is selected as *join type* the resultant ExampleSet has examples with ids from 1 to 14. This is because the left join delivers all examples of the left ExampleSet with corresponding values of the right ExampleSet. If there is no match present in the right ExampleSet, the null value is placed at its place. This is why you can see that the 'Golf 2 attribute' has null values for ids 1 to 7.

If *right join* is selected as *join type* the resultant ExampleSet has examples with ids from 8 to 21. This is because the right join delivers all examples of the right ExampleSet with corresponding values of the left ExampleSet. If there is no match present in the left ExampleSet, a null value is placed at its place. This is why you can see that the 'Golf 1 attribute' has null values for ids 15 to 21.

If *outer join* is selected as *join type* the resultant ExampleSet has examples with ids from 1 to 21. This is because the outer join combines the results of the left and right join. All examples from both ExampleSets will be part of the resultant ExampleSet, whether the matching **key attribute** value exists in the other ExampleSet or not. If no matching **key attribute** value was found then the corresponding resultant fields will have a null value. In this example process the left ExampleSet has ids from 1 to 14 and the right ExampleSet has ids from 8 to 21. Thus examples with ids from 1 to 21 are part of the resultant ExampleSet. The 'Golf 2 attribute' has null values in examples with ids from 1 to 7. Similarly, the 'Golf 1 attribute' has null values in examples with ids from 15 to 21. There are no null values in examples with ids 8 to 14. The 'Play' attribute has null values in examples with id from 15 to 21. This is because special attributes are taken from the left ExampleSet which in this example process has no values of the 'Play' attribute corresponding to ids 15 to 21.
6. Data Transformation

Set Minus

This operator returns those examples of the ExampleSet (given at the example set input port) whose IDs are not contained within the other ExampleSet (given at the subtrahend port). It is necessary that both ExampleSets should have the ID attribute. The ID attribute of both ExampleSets should be of the same type.

Description

This operator performs a set minus on two ExampleSets on the basis of the ID attribute i.e. the resulting ExampleSet contains all the examples of the minuend ExampleSet (given at the example set input port) whose IDs do not appear in the subtrahend ExampleSet (given at the subtrahend port). It is important to note that the ExampleSets do not need to have the same number of columns or the same data types. The operation only depends on the ID attributes of the ExampleSets. It should be made sure that the ID attributes of both ExampleSets are of the same type i.e. either both are nominal or both are numerical.
Differentiation

**Intersect** The Set Minus and Intersect operators can be considered as opposite of each other. The Intersect operator performs a set intersect on two ExampleSets on the basis of the ID attribute i.e. the resulting ExampleSet contains all the examples of the first ExampleSet whose IDs appear in the second ExampleSet. See page ?? for details.

**Input Ports**

*example set input (exa)* This input port expects an ExampleSet. It is the output of the Generate ID operator in the attached Example Process because this operator only works if the ExampleSets have the ID attribute.

*subtrahend (sub)* This input port expects an ExampleSet. It is the output of the Generate ID operator in the attached Example Process because this operator only works if the ExampleSets have the ID attribute.

**Output Ports**

*example set output (exa)* The ExampleSet with remaining examples (i.e. examples remaining after the set minus) of the minuend ExampleSet is output of this port.

*original (ori)* The ExampleSet that was given as input (at *example set input* port) is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Related Documents**

*Intersect (??)*
6. Data Transformation

Tutorial Processes

Introduction to the Set Minus operator

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with the *offset* parameter set to 0. Thus the ids of the 'Golf' data set are from 1 to 14. A *breakpoint* is inserted here so you can have a look at the 'Golf' data set. The 'Polynomial' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it with the *offset* parameter set to 10. Thus the ids of the 'Polynomial' data set are from 11 to 210. A *breakpoint* is inserted here so you can have a look at the 'Polynomial' data set.

The Set Minus operator is applied next. The 'Golf' data set is provided at the *example set input* port and the 'Polynomial' data set is provided at the *subtrahend* port. The order of ExampleSets is very important. The Set Minus operator compares the ids of the 'Golf' data set with the ids of the 'Polynomial' data set and then returns only those examples of the 'Golf' data set whose id is not present in the 'Polynomial' data set. The 'Golf' data set ids are from 1 to 14 and the 'Polynomial' data set ids are from 11 to 210. Thus 'Golf' data set examples with ids 1 to 10 are returned by the Set Minus operator. It is important to note that the meta data of both ExampleSets is very different but it does not matter because the Set Minus operator only depends on the ID attribute.

If the ExampleSets are switched at the input ports of the Set Minus operator the results will be very different. In this case the Set Minus operator returns only those examples of the 'Polynomial' data set whose id is not present in the 'Golf' data set. The 'Golf' data set ids are from 1 to 14 and the 'Polynomial' data set ids are from 11 to 210. Thus the 'Polynomial' data set examples with ids 15 to 210 are returned by the Set Minus operator.
6.10. Set Operations

Union

This operator builds the union of the input ExampleSets. The input ExampleSets are combined in such a way that attributes and examples of both input ExampleSets are part of the resultant union ExampleSet.

Description

The Union operator builds the superset of features of both input ExampleSets such that all regular attributes of both ExampleSets are part of the superset. The attributes that are common in both ExampleSets are not repeated in the superset twice, a single attribute is created that holds data of both ExampleSets. If the special attributes of both input ExampleSets are compatible with each other then only one special attribute is created in the superset which has examples of both the input ExampleSets. If special attributes of ExampleSets are not compatible, the special attributes of the first ExampleSet are kept. If both ExampleSets have any attributes with the same name, they should be compatible with each other; otherwise you will get an error message. This can be understood by studying the
attached Example Process.

### Input Ports

**example set 1 (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

**example set 2 (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

### Output Ports

**union (uni)** The union of the input ExampleSets is delivered through this port.

### Tutorial Processes

**Union of the Golf and Golf-Testset data sets**

In this process the 'Golf' data set and 'Golf-Testset' data set are loaded using the Retrieve operators. Breakpoints are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. When you continue the process, you will see the 'Golf-Testset' data set. It also has 14 examples. Note that the meta data of both ExampleSets is almost the same. The Union
operator is applied to combine these two ExampleSets into a single ExampleSet. The combined ExampleSet has all attributes and examples from the input ExampleSets, thus it has 28 examples. You can see that both input ExampleSets had the same number of attributes, same names and roles of attributes. This is why the Union ExampleSet also has the same number of attributes with the same names and roles. Here the Union operator behaves like the Append operator i.e. it simply combines examples of two ExampleSets with compatible meta data.

Union of the Golf and Iris data sets

In this process the 'Golf' data set and the 'Iris' data set are loaded using the Retrieve operators. Breakpoints are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. When you continue the process, you will see the 'Iris' data set. It has 4 regular and 2 special attributes with 150 examples. Note that the meta data of both ExampleSets is very different. The Union operator is applied to combine these two ExampleSets into a single ExampleSet. The combined ExampleSet has all attributes and examples from the input ExampleSets, thus it has 164 (14+150) examples. Note that the 'Golf' data set has an attribute with label role: the 'Play' attribute. The 'Iris' data set also has an attribute with label role: the 'label' attribute.
6. Data Transformation

As these two label attributes are not compatible, only the label attribute of the first ExampleSet is kept. The examples of 'Iris' data set have null values in this attribute of the resultant Union ExampleSet.

Union of the Golf(with id attribute) and Iris data sets

In this process the 'Golf' data set and 'Iris' data set are loaded using the Retrieve operators. The Generate ID operator is applied on the Golf data set to generate nominal ids starting from id_1. Breakpoints are inserted before the Union operator so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. As you can see, it has 14 examples. It has two special attributes. When you continue the process, you will see the 'Iris' data set. It has 4 regular and 2 special attributes with 150 examples. Note that the meta data of both ExampleSets is very different. The Union operator is applied to combine these two ExampleSets into a single ExampleSet. The combined ExampleSet has all attributes and examples from the input ExampleSets, thus it has 164 (14+150) examples. Note that the 'Golf' data set has an attribute with label role: the 'Play' attribute. The 'Iris' data set also has an attribute with label role: the 'label' attribute. As these two label attributes are not compatible, only the label attribute of the first ExampleSet is kept. The examples of the 'Iris' data set have null values in this attribute of the union ExampleSet. Also note
that both input ExampleSets have *id* attributes. The names of these attributes are the same and they both have nominal values, thus these two attributes are compatible with each other. Thus a single *id* attribute is created in the resultant Union ExampleSet. Also note that the values of ids are not unique in the resultant ExampleSet.

**Superset**

This operator takes two ExampleSets as input and adds new features of the first ExampleSet to the second ExampleSet and vice versa to generate two supersets. The resultant supersets have the same set of attributes but the examples may be different.

**Description**

The Superset operator generates supersets of the given ExampleSets by adding new features of one ExampleSet to the other ExampleSet. The values of the new features are set to missing values in the supersets. This operator delivers two supersets as output:
6. Data Transformation

1. The first has all attributes and examples of the first ExampleSet + all attributes of the second ExampleSet (with missing values)

2. The second has all attributes and examples of the second ExampleSet + all attributes of the first ExampleSet (with missing values)

Thus both supersets have the same set of regular attributes but the examples may be different. It is important to note that the supersets can have only one special attribute of a kind. By default this operator adds only new 'regular' attributes to the other ExampleSet for generating supersets. For example, if both input ExampleSets have a label attribute then the first superset will have all attributes of the first ExampleSet (including label) + all regular attributes of the second ExampleSet. The second superset will behave correspondingly. The include special attributes parameter can be used for changing this behavior. But it should be used carefully because even if this parameter is set to true, the resultant supersets can have only one special attribute of a kind. Please study the attached Example Process for better understanding.

Input Ports

data set 1 (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.

data set 2 (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input. It is essential that meta data should be attached with the data for the input because attributes are specified in their meta data.
Output Ports

**superset 1** (*sup*) The first superset of the input ExampleSets is delivered through this port.

**superset 2** (*sup*) The second superset of the input ExampleSets is delivered through this port.

Parameters

**include special attributes** (*boolean*) This parameter indicates if the special attributes should be included for generation of the supersets. This operator should be used carefully especially if both ExampleSets have the same special attributes because the resultant supersets can have only one special attribute of a kind.

Tutorial Processes

Generating supersets of the Golf and Iris data sets

In this process the 'Golf' and 'Iris' data sets are loaded using the Retrieve operators. *Breakpoints* are inserted after the Retrieve operators so that you can have a look at the input ExampleSets. When you run the process, first you see the 'Golf' data set. It has four regular and one special attribute with 14 examples each. When you continue the process, you will see the 'Iris' data set. It has four regular and two special attributes with 150 examples each. Note that the meta data of both ExampleSets is very different. The Superset operator is applied for generating supersets of these two ExampleSets. The resultant supersets can be seen in the Results Workspace. You can see that one superset has all attributes and examples of the 'Iris' data set + 4 regular attributes of the 'Golf' data set (with missing values). The other superset has all attributes and examples of the
6. Data Transformation

'Golf' data set + 4 regular attributes of the 'Iris' data set (with missing values).
Default Model

This operator generates a model that provides the specified default value as prediction.

Description

The Default Model operator generates a model that predicts the specified default value for the label in all examples. The method to use for generating a default value can be selected through the method parameter. The default value can be median, mode or average of label values. For numeric label, a constant default value can be specified through the constant parameter. For nominal label, values of an attribute can be used as predictions; the attribute can be selected through the attribute parameter. This operator should not be used for 'actual' prediction tasks, but it can be used for comparing the results of 'actual' learning schemes with guessing.
Input Ports

training set (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

model (mod) The default model is delivered from this output port. This model can now be applied on unseen data sets for the prediction of the label attribute. This model should not be used for 'actual' prediction tasks, but it can be used for comparing the results of 'actual' learning schemes with guessing.

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

method (selection) This parameter specifies the method for computing the default values. The default value can be median, mode or average of the label values. For a numeric label, a constant default value can be specified through the constant parameter. For a nominal label, values of an attribute can be used as predictions; the attribute can be selected through the attribute parameter.

constant (real) This parameter is only available when the method parameter is set to 'constant'. This parameter specifies a constant default value for a numeric label.

attribute (string) This parameter is only available when the method parameter is set to 'attribute'. This parameter specifies the attribute to get the predicted values from. This option is only applicable on nominal labels. It should be made sure that the selected attribute has the same set of possible values as the label.
7.1. Classification and Regression

Tutorial Processes

Using the Default Model operator with 'mode' method

The 'Sonar' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that there are two possible label values i.e. 'Rock' and 'Mine'. The most frequently occurring label value is 'Mine'. The Split Validation operator is applied on this ExampleSet for training and testing a classification model. The Default Model operator is applied in the training subprocess of the Split Validation operator. The method parameter of the Default Model operator is set to 'mode', thus the most frequently occurring label value (i.e. 'Mine') will be used as prediction in all examples. The Apply Model operator is used in the testing subprocess for applying the model generated by the Default Model operator. A breakpoint is inserted here so that you can have a look at the labeled ExampleSet. You can see that all examples have been predicted as 'Mine'. This labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The default model and its performance vector are connected to the output and they can be seen in the Results Workspace.
K-NN

This operator generates a k-Nearest Neighbor model from the input ExampleSet. This model can be a classification or regression model depending on the input ExampleSet.

Description

The k-Nearest Neighbor algorithm is based on learning by analogy, that is, by comparing a given test example with training examples that are similar to it. The training examples are described by n attributes. Each example represents a point in an n-dimensional space. In this way, all of the training examples are stored in an n-dimensional pattern space. When given an unknown example, a k-nearest neighbor algorithm searches the pattern space for the k training examples that are closest to the unknown example. These k training examples are the k nearest neighbors of the unknown example. Closeness is defined in terms of a distance metric, such as the Euclidean distance.

The k-nearest neighbor algorithm is amongst the simplest of all machine learning algorithms: an example is classified by a majority vote of its neighbors, with the example being assigned to the class most common amongst its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the example is simply assigned to the class of its nearest neighbor. The same method can be used for regression, by simply assigning the label value for the example to be the average of the values of its k nearest neighbors. It can be useful to weight the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones.

The neighbors are taken from a set of examples for which the correct classification (or, in the case of regression, the value of the label) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

The basic k-Nearest Neighbor algorithm is composed of two steps:
1. Find the k training examples that are closest to the unseen example.

2. Take the most commonly occurring classification for these k examples (or, in the case of regression, take the average of these k label values).

Input Ports

training set (tra) This input port expects an ExampleSet. It is output of the Select Attributes operator in the attached Example Processes. Output of other operators can also be used as input.

Output Ports

model (mod) The k-Nearest Neighbor model is delivered from this output port. This model can now be applied on unseen data sets for prediction of the label attribute.
example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

k (integer) Finding the k training examples that are closest to the unseen example is the first step of the k-NN algorithm. If k = 1, the example is simply assigned to the class of its nearest neighbor. k is a typically small positive integer. Mostly k is provided with an odd integer value.
weighted vote (boolean) If this parameter is set, the weight of examples is also taken into account. It can be useful to weight the contributions of the neighbors, so that the nearer neighbors contribute more than the more distant ones.
measure types (selection) This parameter is used for selecting the type of mea-
7. Modeling

sure to be used for finding the nearest neighbors. The following options are available: *mixed measures*, *nominal measures*, *numerical measures* and *Bregman divergences*.

**mixed measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'.

**nominal measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. Otherwise the 'numerical measure' option should be selected.

**numerical measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. Otherwise the 'nominal measure' option should be selected.

**divergence** *(selection)* This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type** *(selection)* This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by \( k(x, y) = x^*y \) i.e. it is the inner product of \( x \) and \( y \).

- **radial** The radial kernel is defined by \( \exp(-g \|x-y\|^2) \) where \( g \) is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter \( gamma \) plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by \( k(x, y) = (x^*y+1)^d \) where \( d \) is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net \( \tanh(a \cdot x^*y+b) \) where \( a \) is *alpha* and \( b \) is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common
7.1. Classification and Regression

value for alpha is $1/N$, where N is the data dimension. Note that not all choices of $a$ and $b$ lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that it is not valid under some parameters.

- **anova** The anova kernel is defined by the raised to the power $d$ of summation of $\exp(-g (x-y))$ where $g$ is gamma and $d$ is degree. The two are adjusted by the kernel gamma and kernel degree parameters respectively.

- **epachnenikov** The Epanechnikov kernel is this function $(3/4)(1-u^2)$ for $u$ between -1 and 1 and zero for $u$ outside that range. It has the two adjustable parameters kernel sigma1 and kernel degree.

- **gaussian combination** This is the gaussian combination kernel. It has the adjustable parameters kernel sigma1, kernel sigma2 and kernel sigma3.

- **multiquadric** The multiquadric kernel is defined by the square root of $||x-y||^2 + c^2$. It has the adjustable parameters kernel sigma1 and kernel sigma shift.

**kernel gamma** *(real)* This is the SVM kernel parameter gamma. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to radial or anova.

**kernel sigma1** *(real)* This is the SVM kernel parameter sigma1. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** *(real)* This is the SVM kernel parameter sigma2. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to gaussian combination.

**kernel sigma3** *(real)* This is the SVM kernel parameter sigma3. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to gaussian combination.
7. Modeling

**kernel shift** *(real)* This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric.*

**kernel degree** *(real)* This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial, anova* or *epachnenikov.*

**kernel a** *(real)* This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural.*

**kernel b** *(real)* This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural.*

**Tutorial Processes**

**Classification of the 'Golf-Testset' data set using the K-NN operator**

The 'Golf' data set is loaded using the Retrieve operator. The Select Attributes operator is applied on it to select just the 'Play' and 'Temperature' attributes to simplify this example process. Then the K-NN operator is applied on it. All the parameters of the K-NN operator are used with default values. The resultant classification model is applied on the 'Golf-Testset' data set using the Apply Model operator. Note that the same two attributes of the 'Golf-Testset' data set were selected before the application of the classification model on it.

Run the process. You can see the 'Golf' data set and the labeled 'Golf-Testset' data set in the Results Workspace. As the *k* parameter was set to 1, each example of the 'Golf-Testset' data set is simply assigned the class of its nearest neighbor in the 'Golf' data set. To understand how examples were classified, simply select an example in the 'Golf-Testset' data set and note the value of the 'Temperature'
attribute of that example, we call it $t1$ here. Now, have a look at the 'Golf' data set and find the example where the 'Temperature' value is closest to $t1$. The example in the 'Golf-Testset' data set is assigned the same class as the class of this example in the 'Golf' data set. For example let us consider the first example of the 'Golf-Testset' data set. The value of the 'Temperature' attribute is 85 in this example. Now find the example in the 'Golf' data set where 'Temperature' value is closest to 85. As you can see that the first example of the 'Golf' data set has a 'Temperature' value equal to 85. This example is labeled 'no', thus the corresponding example in the 'Golf-Testset' data set is also predicted as 'no'.

Regression of the Polynomial data set using the K-NN operator

The 'Polynomial' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it to select just the first 100 examples. Then the Select Attributes operator is applied on it to select just the 'label' and 'a1' attributes to simplify this example process. Then the K-NN operator is applied on it. The $k$ parameter is set to 3, the measure types parameter is set to 'Numerical Measures' and the numerical measure parameter is set to 'Euclidean
7. Modeling

Distance'. The resultant regression model is applied on the last 100 examples of the 'Polynomial' data set using the Apply Model operator. Note that the same two attributes of the 'Polynomial' data set were selected before the application of the regression model on it.

Run the process. You can see the 'Polynomial' data set (first 100 examples) and the labeled 'Polynomial' data set (last 100 examples) in the Results Workspace. For convenience we call these data sets as 'Polynomial' first and 'Polynomial' last data sets respectively. As the $k$ parameter was set to 3, each example of the 'Polynomial' last data set is simply assigned the average label value of its 3 nearest neighbors in the 'Polynomial' first data set. To understand how regression was performed, simply select an example in the 'Polynomial' last data set and note the value of the 'a1' attribute of that example, we call it $a1_{val}$ here. Now, have a look at the 'Polynomial' first data set and find 3 examples where the 'a1' attribute value is closest to $a1_{val}$. The 'label' attribute of the example of the 'Polynomial' last data set is assigned the average of these three label values of the 'Polynomial' first data set. For example let us consider the last example of the 'Polynomial' last data set. The value of the 'a1' attribute is 4.788 in this example. Now find 3 examples in the 'Polynomial' first data set where the value of the 'a1' attribute is closest to 4.788. These 3 examples are at Row No. 65, 71 and 86. The value of the 'label' attribute of these examples is 41.798, 124.250 and 371.814 respectively. The average of these three 'label' values is 179.288. Thus the value of the 'label' attribute in the last example of the 'Polynomial' last data set is predicted to be 179.288.
Naive Bayes

This operator generates a Naive Bayes classification model.

Description

A Naive Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics) with strong (naive) independence assumptions. A more descriptive term for the underlying probability model would be 'independent feature model'. In simple terms, a Naive Bayes classifier assumes that the presence (or absence) of a particular feature of a class (i.e. attribute) is unrelated to the presence (or absence) of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 4 inches in diameter. Even if these features depend on each other or upon the existence of the other features, a Naive Bayes classifier considers all of these properties to
Modeling

Independently contribute to the probability that this fruit is an apple.

The advantage of the Naive Bayes classifier is that it only requires a small amount of training data to estimate the means and variances of the variables necessary for classification. Because independent variables are assumed, only the variances of the variables for each label need to be determined and not the entire covariance matrix.

Input Ports

training set (tra) The input port expects an ExampleSet. It is the output of the Select Attributes operator in our example process. The output of other operators can also be used as input.

Output Ports

model (mod) The Naive Bayes classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the label attribute.

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

laplace correction (boolean) This is an expert parameter. This parameter indicates if Laplace correction should be used to prevent high influence of zero probabilities. There is a simple trick to avoid zero probabilities. We can assume that our training set is so large that adding one to each count that we need would only make a negligible difference in the estimated probabilities, yet would
avoid the case of zero probability values. This technique is known as Laplace correction.

**Tutorial Processes**

**Working of Naive Bayes**

The Retrieve operator is used to load the 'Golf' data set. The Select Attributes operator is applied on it to select just *Outlook* and *Wind* attributes. This is done to simplify the understanding of this Example Process. The Naive Bayes operator is applied on it and the resulting model is applied on the 'Golf-testset' data set. The Same two attributes of the 'Golf-testset' data set were selected before application of the Naive Bayes model. A breakpoint is inserted after the Naive Bayes operator. Run the process and see the distribution table in the Results Workspace. We will use this distribution table to explain how Naive Bayes works. Hit the Run button again to continue with the process. Let us see how the first and last examples of the 'Golf-testset' data set were predicted by Naive Bayes. Note that 9 out of 14 examples of the training set had *label = yes*, thus the posterior probability of the *label = yes* is 9/14. Similarly the posterior probability of the *label = no* is 5/14.

Note that in the testing set, the attributes of the first example are *Outlook = sunny* and *Wind = false*. Naive Bayes does calculation for all possible label values and selects the *label* value that has maximum calculated probability.

Calculation for *label = yes*

Find product of following:

- Posterior probability of *label = yes* (i.e. 9/14)
- value from distribution table when *Outlook = sunny* and *label = yes* (i.e. 0.223)
7. Modeling

- value from distribution table when Wind = false and label = yes (i.e. 0.659)

Thus the answer = \( \frac{9}{14} \times 0.223 \times 0.659 = 0.094 \)

Calculation for label = no

Find product of following:

- posterior probability of label = no (i.e. 5/14)
- value from distribution table when Outlook = sunny and label = no (i.e. 0.581)
- value from distribution table when Wind = false and label = no (i.e. 0.397)

Thus the answer = \( \frac{5}{14} \times 0.581 \times 0.397 = 0.082 \)

As the value for label = yes is the maximum of all possible label values, label is predicted to be yes.

Similarly let us have a look at the last example of the 'Golf-testset' data set. Note that in the testing set, in first example Outlook = rain and Wind = true. Naive Bayes does calculation for all possible label values and selects the label value that has maximum calculated probability.

Calculation for label = yes

Find product of following:

- posterior probability of label = yes (i.e. 9/14)
- value from distribution table when Outlook = rain and label = yes (i.e. 0.331)
- value from distribution table when Wind = true and label = yes (i.e. 0.333)

Thus the answer = \( \frac{9}{14} \times 0.331 \times 0.333 = 0.071 \)

Calculation for label = no

Find product of following:
7.1. Classification and Regression

- posterior probability of label = no (i.e. 5/14)
- value from distribution table when Outlook = rain and label = no (i.e. 0.392)
- value from distribution table when Wind = true and label = no (i.e. 0.589)

Thus the answer = 5/14 * 0.392 * 0.589 = 0.082

As the value for label = no is the maximum of all possible label values, label is predicted to be no.

Now run the process again, but this time uncheck the laplace correction parameter. Now you can see that as laplace correction is not used for avoiding zero probability, there are numerous zeroes in the distribution table.

Naive Bayes (Kernel)

This operator generates a Kernel Naive Bayes classification model using estimated kernel densities.
Description

A Naive Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics) with strong (naive) independence assumptions. A more descriptive term for the underlying probability model would be the 'independent feature model'. In simple terms, a Naive Bayes classifier assumes that the presence (or absence) of a particular feature of a class (i.e. attribute) is unrelated to the presence (or absence) of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 4 inches in diameter. Even if these features depend on each other or upon the existence of the other features, a Naive Bayes classifier considers all of these properties to independently contribute to the probability that this fruit is an apple. The Naive Bayes classifier performs reasonably well even if the underlying assumption is not true.

The advantage of the Naive Bayes classifier is that it only requires a small amount of training data to estimate the means and variances of the variables necessary for classification. Because independent variables are assumed, only the variances of the variables for each label need to be determined and not the entire covariance matrix. In contrast to the Naive Bayes operator, the Naive Bayes (Kernel) operator can be applied on numerical attributes.

A kernel is a weighting function used in non-parametric estimation techniques. Kernels are used in kernel density estimation to estimate random variables' density functions, or in kernel regression to estimate the conditional expectation of a random variable.

Kernel density estimators belong to a class of estimators called non-parametric density estimators. In comparison to parametric estimators where the estimator has a fixed functional form (structure) and the parameters of this function are the only information we need to store, Non-parametric estimators have no fixed structure and depend upon all the data points to reach an estimate.
7.1. Classification and Regression

Input Ports

**training set** *(tra)* The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

**model** *(mod)* The Kernel Naive Bayes classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the *label* attribute.

**example set** *(exa)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**laplace correction** *(boolean)* This parameter indicates if Laplace correction should be used to prevent high influence of zero probabilities. There is a simple trick to avoid zero probabilities. We can assume that our training set is so large that adding one to each count that we need would only make a negligible difference in the estimated probabilities, yet would avoid the case of zero probability values. This technique is known as Laplace correction.

**estimation mode** *(selection)* This parameter specifies the kernel density estimation mode. Two options are available.

- **full** If this option is selected, you can select a bandwidth through heuristic or a fix bandwidth can be specified.

- **greedy** If this option is selected, you have to specify the minimum bandwidth and the number of kernels.
bandwidth selection (selection) This parameter is only available when the estimation mode parameter is set to 'full'. This parameter specifies the method to set the kernel bandwidth. The bandwidth can be selected through heuristic or a fix bandwidth can be specified. Please note that the bandwidth of the kernel is a free parameter which exhibits a strong influence on the resulting estimate. It is important to choose the most appropriate bandwidth as a value that is too small or too large is not useful.

bandwidth (real) This parameter is only available when the estimation mode parameter is set to 'full' and the bandwidth selection parameter is set to 'fix'. This parameter specifies the kernel bandwidth.

minimum bandwidth (real) This parameter is only available when the estimation mode parameter is set to 'greedy'. This parameter specifies the minimum kernel bandwidth.

number of kernels (integer) This parameter is only available when the estimation mode parameter is set to 'greedy'. This parameter specifies the number of kernels.

use application grid (boolean) This parameter indicates if the kernel density function grid should be used in the model application. It speeds up model application at the expense of the density function precision.

application grid size (integer) This parameter is only available when the use application grid parameter is set to true. This parameter specifies the size of the application grid.

Tutorial Processes

Introduction to the Naive Bayes (Kernel) operator

The 'Golf' data set is loaded using the Retrieve operator. The Naive Bayes (Kernel) operator is applied on it. All parameters of the Naive Bayes (Kernel) operator are used with default values. The model generated by the Naive Bayes (Kernel) operator is applied on the 'Golf-Testset' data set using the Apply Model operator. The results of the process can be seen in the Results Workspace. Please
7.1. Classification and Regression

note that parameters should be carefully chosen for this operator to obtain better performance. Specially the bandwidth should be selected carefully.

Decision Tree

Generates a Decision Tree for classification of both nominal and numerical data.

Description

A decision tree is a tree-like graph or model. It is more like an inverted tree because it has its root at the top and it grows downwards. This representation of the data has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of a target attribute (often called class or label) based on several input attributes of the ExampleSet. In RapidMiner an attribute with label role is predicted by the Decision Tree operator. Each interior node of tree corresponds
7. Modeling

to one of the input attributes. The number of edges of a nominal interior node is equal to the number of possible values of the corresponding input attribute. Outgoing edges of numerical attributes are labeled with disjoint ranges. Each leaf node represents a value of the label attribute given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the attached Example Process.

Decision Trees are generated by recursive partitioning. Recursive partitioning means repeatedly splitting on the values of attributes. In every recursion the algorithm follows the following steps:

- An attribute $A$ is selected to split on. Making a good choice of attributes to split on each stage is crucial to generation of a useful tree. The attribute is selected depending upon a selection criterion which can be selected by the criterion parameter.

- Examples in the ExampleSet are sorted into subsets, one for each value of the attribute $A$ in case of a nominal attribute. In case of numerical attributes, subsets are formed for disjoint ranges of attribute values.

- A tree is returned with one edge or branch for each subset. Each branch has a descendant subtree or a label value produced by applying the same algorithm recursively.

In general, the recursion stops when all the examples or instances have the same label value, i.e. the subset is pure. Or recursion may stop if most of the examples are of the same label value. This is a generalization of the first approach; with some error threshold. However there are other halting conditions such as:

- There are less than a certain number of instances or examples in the current subtree. This can be adjusted by using the minimal size for split parameter.

- No attribute reaches a certain threshold. This can be adjusted by using the minimum gain parameter.

- The maximal depth is reached. This can be adjusted by using the maximal depth parameter.
7.1. Classification and Regression

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.

Differentiation

CHAID The CHAID operator works exactly like the Decision Tree operator with one exception: it uses a chi-squared based criterion instead of the information gain or gain ratio criteria. Moreover this operator cannot be applied on ExampleSets with numerical attributes. See page 669 for details.

Input Ports

training set (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

model (mod) The Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the label attribute.

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

criterion (selection) Selects the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.

- **gain ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.

- **gini index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.

- **accuracy** Such an attribute is selected for split that maximizes the accuracy of the whole Tree.

minimal size for split (integer) The size of a node is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the minimal size for split parameter.

minimal leaf size (integer) The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the minimal leaf size number of instances.

minimal gain (real) The gain of a node is calculated before splitting it. The node is split if its Gain is greater than the minimal gain. Higher value of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

maximal depth (integer) The depth of a tree varies depending upon size and nature of the ExampleSet. This parameter is used to restrict the size of the Decision Tree. The tree generation process is not continued when the tree depth is equal to the maximal depth. If its value is set to '-1', the maximal depth pa-
rameter puts no bound on the depth of the tree, a tree of maximum depth is
generated. If its value is set to '1', a Tree with a single node is generated.

**confidence (real)** This parameter specifies the confidence level used for the pes-
simistic error calculation of pruning.

**number of prepruning alternatives (integer)** As prepruning runs parallel to
the tree generation process, it may prevent splitting at certain nodes when split-
ting at that node does not add to the discriminative power of the entire tree.
In such a case alternative nodes are tried for splitting. This parameter adjusts
the number of alternative nodes tried for splitting when split is prevented by
prepruning at a certain node.

**no prepruning (boolean)** By default the Decision Tree is generated with preprun-
ing. Setting this parameter to true disables the prepruning and delivers a tree
without any prepruning.

**no pruning (boolean)** By default the Decision Tree is generated with pruning.
Setting this parameter to true disables the pruning and delivers an unpruned
Tree.

### Related Documents

**CHAID (669)**

### Tutorial Processes

**Getting started with Decision Trees**

The 'Golf' data set is retrieved using the Retrieve operator. Then the Deci-
sion Tree operator is applied on it. Click on the Run button to go to the Results
Workspace. First take a look at the resultant tree. First of all, basic terminology
of trees is explained here using this resultant tree. The first node of the tree is
called the *root* of the tree; 'Outlook' is the *root* in this case. As you can see from
this tree, trees grow downwards. This example clearly shows why interpretation of data is easy through trees. Just take a glance at this tree and you will come to know that whenever the 'Outlook' attribute value is 'overcast', the 'Play' attribute will have the value 'yes'. Similarly whenever the 'Outlook' attribute value is 'rain' and the 'Wind' attribute has value 'false', then the 'Play' attribute will have the value 'yes'. The Decision Tree operator predicts the value of an attribute with the label role, in this example the 'Play' attribute is predicted. The nodes that do not have child nodes are called the leaf nodes. All leaf non-leaf nodes correspond to one of the input attributes. In this example the 'Outlook', 'Wind' and 'Humidity' attributes are non-leaf nodes. The number of edges of a nominal interior node is equal to the number of possible values of the corresponding input attribute. The 'Outlook' attribute has three possible values: 'overcast', 'rain' and 'sunny' thus it has three outgoing edges. Similarly the 'Wind' attribute has two outgoing edges. As 'Humidity' is a numerical attribute, its outgoing edges are labeled with disjoint ranges. Each leaf node represents a value of the label given the values of the input attributes represented by the path from the root to the leaf. That is why all leaf nodes assume possible values of the label attribute i.e. 'yes' or 'no'.

In this Example Process the 'Gain ratio' is used as the selection criterion. However using any other criterion on this ExampleSet produces the same tree. This is because this is a very simple data set. On large and complex data sets different selection criterion may produce different trees.

When the tree is split on the 'Outlook' attribute, it is divided into three subtrees, one with each value of the 'Outlook' attribute. The 'overcast' subtree is pure i.e. all its label values are same ('yes'), thus it is not split again. The 'rain' subtree and the 'sunny' subtree are split again in the next iteration. In this Example Process the minimal size of split parameter is set to 4. Set it to 10 and run the process again. You will see that you get a tree with a single node. This is because this time the nodes with size less than 10 cannot be split. As the size of all subtrees of the 'Outlook' attribute is less than 10, thus no splitting takes place, and we get a tree with just a single node.

The minimal gain parameter is set to 0.1. In general if you want to reduce the
7.1. Classification and Regression

size of tree, you can increase the *minimal gain*. Similarly if you want to increase
the size of your tree, you can reduce the value of the *minimal gain* parameter.
The *maximal depth* parameter is set to 20. The actual depth of the resultant tree
is 3. You can set an upper bound to the depth of tree using the *maximal depth*
parameter. Prepruning is enabled in this Example Process. To disable it, check
the *no prepruning* parameter. Now click the Run button. The resultant tree is
much more complex than the previous one. The previous tree was more useful
and comprehendible than this one.

ID3

This operator learns an unpruned Decision Tree from nominal data
for classification. This decision tree learner works similar to Quinlan's
ID3.

Description

ID3 (Iterative Dichotomiser 3) is an algorithm used to generate a decision tree
invented by Ross Quinlan. ID3 is the precursor to the C4.5 algorithm. Very
simply, ID3 builds a decision tree from a fixed set of examples. The resulting
tree is used to classify future samples. The examples of the given ExampleSet
have several attributes and every example belongs to a class (like yes or no). The
leaf nodes of the decision tree contain the class name whereas a non-leaf node
is a decision node. The decision node is an attribute test with each branch (to
another decision tree) being a possible value of the attribute. ID3 uses feature selection heuristic to help it decide which attribute goes into a decision node. The required heuristic can be selected by the criterion parameter.

The ID3 algorithm can be summarized as follows:

1. Take all unused attributes and calculate their selection criterion (e.g. information gain)
2. Choose the attribute for which the selection criterion has the best value (e.g. minimum entropy or maximum information gain)
3. Make node containing that attribute

ID3 searches through the attributes of the training instances and extracts the attribute that best separates the given examples. If the attribute perfectly classifies the training sets then ID3 stops; otherwise it recursively operates on the n (where n = number of possible values of an attribute) partitioned subsets to get their best attribute. The algorithm uses a greedy search, meaning it picks the best attribute and never looks back to reconsider earlier choices.

Some major benefits of ID3 are:

- Understandable prediction rules are created from the training data.
- Builds a short tree in relatively small time.
- It only needs to test enough attributes until all data is classified.
- Finding leaf nodes enables test data to be pruned, reducing the number of tests.

ID3 may have some disadvantages in some cases e.g.

- Data may be over-fitted or over-classified, if a small sample is tested.
- Only one attribute at a time is tested for making a decision.
7.1. Classification and Regression

Input Ports

**training set (tra)** This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. This operator cannot handle numerical attributes. The output of other operators can also be used as input.

Output Ports

**model (mod)** The Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the label attribute.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**criterion (selection)** This parameter specifies the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.

- **gain ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.

- **gini index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting
subsets.

- **accuracy** Such an attribute is selected for a split that maximizes the accuracy of the whole Tree.

**minimal size for split (integer)** The size of a node is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the **minimal size for split** parameter.

**minimal leaf size (integer)** The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the **minimal leaf size** number of instances.

**minimal gain (real)** The gain of a node is calculated before splitting it. The node is split if its Gain is greater than the **minimal gain**. Higher value of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

### Tutorial Processes

#### Getting started with ID3

To understand the basic terminology of trees, please study the Example Process of the Decision Tree operator.

The Generate Nominal Data operator is used for generating an ExampleSet with nominal attributes. It should be kept in mind that the ID3 operator cannot handle numerical attributes. A **breakpoint** is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has three attributes and each attribute has three possible values. The ID3 operator is applied on this ExampleSet with default values of all parameters. The resultant Decision Tree model is delivered to the **result** port of the process and it can be seen in the Results Workspace.
7.1. Classification and Regression

CHAID

This operator generates a pruned decision tree based on the chi-squared attribute relevance test. This operator can be applied only on ExampleSets with nominal data.

Description

The CHAID decision tree operator works exactly like the Decision Tree operator with one exception: it uses a chi-squared based criterion instead of the information gain or gain ratio criteria. Moreover this operator cannot be applied on ExampleSets with numerical attributes. It is recommended that you study the documentation of the Decision Tree operator for basic understanding of decision trees.

CHAID stands for CHi-squared Automatic Interaction Detection. The chi-square statistic is a nonparametric statistical technique used to determine if a distribution of observed frequencies differs from the theoretical expected frequencies. Chi-square statistics use nominal data, thus instead of using means and variances, this test uses frequencies. CHAID's advantages are that its output is highly visual and easy to interpret. Because it uses multiway splits by default, it needs rather large sample sizes to work effectively, since with small sample sizes
the respondent groups can quickly become too small for reliable analysis.

This representation of the data has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of the label based on several input attributes of the ExampleSet. Each interior node of the tree corresponds to one of the input attributes. The number of edges of an interior node is equal to the number of possible values of the corresponding input attribute. Each leaf node represents a value of the label given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the Example Process of the Decision Tree operator.

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.

### Differentiation

**Decision Tree** The CHAID operator works exactly like the Decision Tree operator with one exception: it uses a chi-squared based criterion instead of the information gain or gain ratio criteria. Moreover this operator cannot be applied on ExampleSets with numerical attributes. See page 659 for details.

**Decision Tree (Weight-Based)** If the Weight by Chi Squared Statistic operator is applied for attribute weighting in the subprocess of the Decision Tree (Weight-Based) operator, it works exactly like the CHAID operator. See page ?? for details.
7.1. Classification and Regression

Input Ports

**training set (tra)** This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. The output of other operators can also be used as input. This operator cannot handle numerical data, therefore the ExampleSet should not have numerical attributes.

Output Ports

**model (mod)** The CHAID Decision Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the label attribute.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**minimal size for split (integer)** The size of a node is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the **minimal size for split** parameter.

**minimal leaf size (integer)** The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the **minimal leaf size** number of instances.

**minimal gain (real)** The gain of a node is calculated before splitting it. The node is split if its gain is greater than the **minimal gain**. Higher values of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

**maximal depth (integer)** The depth of a tree varies depending upon size and
nature of the ExampleSet. This parameter is used to restrict the size of the Decision Tree. The tree generation process is not continued when the tree depth is equal to the maximal depth. If its value is set to '-1', the maximal depth parameter puts no bound on the depth of the tree, a tree of maximum depth is generated. If its value is set to '1', a Tree with a single node is generated.

**confidence (real)** This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**number of prepruning alternatives (integer)** As prepruning runs parallel to the tree generation process, it may prevent splitting at certain nodes when splitting at that node does not add to the discriminative power of the entire tree. In such a case alternative nodes are tried for splitting. This parameter adjusts the number of alternative nodes tried for splitting when it is prevented by prepruning at a certain node.

**no prepruning (boolean)** By default the Decision Tree is generated with prepruning. Setting this parameter to true disables the prepruning and delivers a tree without any prepruning.

**no pruning (boolean)** By default the Decision Tree is generated with pruning. Setting this parameter to true disables the pruning and delivers an unpruned Tree.

### Related Documents

- Decision Tree (659)
- Decision Tree (Weight-Based) (??)

### Tutorial Processes

**Introduction to the CHAID operator**

The Generate Nominal Data operator is used for generating an ExampleSet with
7.1. Classification and Regression

100 examples. There are three nominal attributes in the ExampleSet and every attribute has three possible values. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. The CHAID operator is applied on this ExampleSet with default values of all parameters. The resultant model is connected to the *result* port of the process and it can be seen in the Results Workspace.

**Decision Stump**

This operator learns a Decision Tree with only one single split. This operator can be applied on both nominal and numerical data sets.

**Description**

The Decision Stump operator is used for generating a decision tree with only one single split. The resulting tree can be used for classifying unseen examples. This operator can be very efficient when boosted with operators like the AdaBoost operator. The examples of the given ExampleSet have several attributes and every example belongs to a class (like yes or no). The leaf nodes of a decision tree contain the class name whereas a non-leaf node is a decision node. The decision node is an attribute test with each branch (to another decision tree) being a possible value of the attribute. For more information about decision trees, please study the Decision Tree operator.
7. Modeling

Input Ports

training set (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

model (mod) The Decision Tree with just a single split is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the label attribute.

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

criterion (selection) This parameter specifies the criterion on which attributes will be selected for splitting. It can have one of the following values:

- information_gain The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.

- gain_ratio It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.

- gini_index This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.
• **accuracy** Such an attribute is selected for split that maximizes the accuracy of the whole Tree.

**minimal leaf size** (*integer*) The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the *minimal leaf size* number of instances.

### Tutorial Processes

#### Introduction to the Decision Stump operator

To understand the basic terminology of trees, please study the Example Process of the Decision Tree operator.

The 'Golf' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at the ExampleSet. The Decision Stump operator is applied on this ExampleSet. The *criterion* parameter is set to 'information gain' and the *minimal leaf size* parameter is set to 1. The resultant decision tree model is connected to the *result* port of the process and it can be seen in the Results Workspace. You can see that this decision tree has just a single split.
Random Tree

This operator learns a decision tree. This operator uses only a random subset of attributes for each split.

Description

The Random Tree operator works exactly like the Decision Tree operator with one exception: for each split only a random subset of attributes is available. It is recommended that you study the documentation of the Decision Tree operator for basic understanding of decision trees.

This operator learns decision trees from both nominal and numerical data. Decision trees are powerful classification methods which can be easily understood. The Random Tree operator works similar to Quinlan's C4.5 or CART but it selects a random subset of attributes before it is applied. The size of the subset is specified by the subset ratio parameter.

Representation of the data as Tree has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of the label based on several input attributes of the ExampleSet. Each interior node of tree corresponds to one of the input attributes. The number of edges of an interior node is equal to the number of possible values of the corresponding input attribute. Each leaf node represents a value of the label given the values of the input attributes represented by the path from the root to the leaf. This description can be easily understood by studying the Example Process of the Decision Tree operator.

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.
7.1. Classification and Regression

Differentiation

**Decision Tree** The Random Tree operator works exactly like the Decision Tree operator with one exception: for each split only a random subset of attributes is available. See page 659 for details.

Input Ports

*training set (tra)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

*model (mod)* The Random Tree is delivered from this output port. This classification model can now be applied on unseen data sets for the prediction of the label attribute.

*example set (exa)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

*criterion (selection)* This parameter selects the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information.gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.
7. Modeling

- **gain_ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.

- **gini_index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.

- **accuracy** Such an attribute is selected for split that maximizes the accuracy of the whole Tree.

**minimal size for split** *(integer)* The size of a node in a Tree is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the **minimal size for split** parameter.

**minimal leaf size** *(integer)* The size of a leaf node in a Tree is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the **minimal leaf size** number of instances.

**minimal gain** *(real)* The gain of a node is calculated before splitting it. The node is split if its Gain is greater than the **minimal gain**. Higher value of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

**maximal depth** *(integer)* The depth of a tree varies depending upon size and nature of the ExampleSet. This parameter is used to restrict the size of the Tree. The tree generation process is not continued when the tree depth is equal to the **maximal depth**. If its value is set to '-1', the **maximal depth** parameter puts no bound on the depth of the tree, a tree of maximum depth is generated. If its value is set to '1', a Tree with a single node is generated.

**confidence** *(real)* This parameter specifies the confidence level used for the pessimistic error calculation of pruning.

**number of prepruning alternatives** *(integer)* As prepruning runs parallel to the tree generation process, it may prevent splitting at certain nodes when splitting at that node does not add to the discriminative power of the entire tree. In such a case alternative nodes are tried for splitting. This parameter adjusts the number of alternative nodes tried for splitting when split is prevented by
7.1. Classification and Regression

prepruning at a certain node.

no prepruning (boolean) By default the Tree is generated with prepruning. Setting this parameter to true disables the prepruning and delivers a tree without any prepruning.

no pruning (boolean) By default the Tree is generated with pruning. Setting this parameter to true disables the pruning and delivers an unpruned Tree.

guess subset ratio (boolean) This parameter specifies if the subset ratio should be guessed or not. If set to true, \( \log(m) + 1 \) features are used as subset, otherwise a ratio has to be specified through the subset ratio parameter.

subset ratio (real) This parameter specifies the subset ratio of randomly chosen attributes.

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of the local random seed will produce the same randomization.

local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

Related Documents

Decision Tree (659)

Tutorial Processes

Introduction to the Random Tree operator

The 'Iris' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. The Random Tree operator is applied on this ExampleSet with default values of all parameters. The resultant tree is connected to the result port of the process and it can be seen in
Random Forest

This operator generates a set of a specified number of random trees i.e. it generates a random forest. The resulting model is a voting model of all the trees.

Description

The Random Forest operator generates a set of random trees. The random trees are generated in exactly the same way as the Random Tree operator generates a tree. The resulting forest model contains a specified number of random tree models. The number of trees parameter specifies the required number of trees. The resulting model is a voting model of all the random trees. For more information about random trees please study the Random Tree operator.

The representation of the data in form of a tree has the advantage compared with other approaches of being meaningful and easy to interpret. The goal is to create a classification model that predicts the value of a target attribute (often called class or label) based on several input attributes of the ExampleSet. Each interior node of the tree corresponds to one of the input attributes. The number of edges of a nominal interior node is equal to the number of possible values of the corresponding input attribute. Outgoing edges of numerical attributes are labeled with disjoint ranges. Each leaf node represents a value of the label attribute given the values of the input attributes represented by the path from...
the root to the leaf. For better understanding of the structure of a tree please study the Example Process of the Decision Tree operator.

Pruning is a technique in which leaf nodes that do not add to the discriminative power of the tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. Pre-pruning is a type of pruning performed parallel to the tree creation process. Post-pruning, on the other hand, is done after the tree creation process is complete.

**Input Ports**

**training set** *(tra)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**model** *(mod)* The Random Forest model is delivered from this output port. This model can be applied on unseen data sets for the prediction of the *label* attribute. This model is a voting model of all the random trees

**example set** *(exa)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**number of trees** *(integer)* This parameter specifies the number of random trees to generate.
criterion (selection) Selects the criterion on which attributes will be selected for splitting. It can have one of the following values:

- **information gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.

- **gain ratio** It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values.

- **gini index** This is a measure of impurity of an ExampleSet. Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets.

- **accuracy** Such an attribute is selected for a split that maximizes the accuracy of the whole Tree.

**minimal size for split (integer)** The size of a node is the number of examples in its subset. The size of the root node is equal to the total number of examples in the ExampleSet. Only those nodes are split whose size is greater than or equal to the minimal size for split parameter.

**minimal leaf size (integer)** The size of a leaf node is the number of examples in its subset. The tree is generated in such a way that every leaf node subset has at least the minimal leaf size number of instances.

**minimal gain (real)** The gain of a node is calculated before splitting it. The node is split if its Gain is greater than the minimal gain. Higher value of minimal gain results in fewer splits and thus a smaller tree. A too high value will completely prevent splitting and a tree with a single node is generated.

**maximal depth (integer)** The depth of a tree varies depending upon size and nature of the ExampleSet. This parameter is used to restrict the size of the trees. The tree generation process is not continued when the tree depth is equal to the maximal depth. If its value is set to '-1', the maximal depth parameter puts no bound on the depth of the tree, a tree of maximum depth is generated. If its value is set to '1', a tree with a single node is generated.

**confidence (real)** This parameter specifies the confidence level used for the pes-
7.1. Classification and Regression

simistic error calculation of pruning.

**number of prepruning alternatives** *(integer)* As prepruning runs parallel to the tree generation process, it may prevent splitting at certain nodes when splitting at that node does not add to the discriminative power of the entire tree. In such a case alternative nodes are tried for splitting. This parameter adjusts the number of alternative nodes tried for splitting when a split is prevented by prepruning at a certain node.

**no prepruning** *(boolean)* By default the trees are generated with prepruning. Setting this parameter to true disables the prepruning and generates trees without any prepruning.

**no pruning** *(boolean)* By default the tree is generated with pruning. Setting this parameter to true disables the pruning and generates unpruned trees.

**guess subset ratio** *(boolean)* If this parameter is set to true then \( \log(m) + 1 \) attributes are used, otherwise a ratio should be specified by the **subset ratio** parameter.

**subset ratio** *(real)* This parameter specifies the ratio of randomly chosen attributes to test.

**use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the **use local random seed** parameter is set to true.

### Tutorial Processes

#### Generating a set of random trees using the Random Forest operator

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Random Forest operator is applied in the training subprocess of the Split Vali-
Rule Induction

This operator learns a pruned set of rules with respect to the information gain from the given ExampleSet.

Description

The Rule Induction operator works similar to the propositional rule learner named 'Repeated Incremental Pruning to Produce Error Reduction' (RIPPER, Cohen 1995). Starting with the less prevalent classes, the algorithm iteratively
grows and prunes rules until there are no positive examples left or the error rate is greater than 50%.

In the growing phase, for each rule greedily conditions are added to the rule until it is perfect (i.e. 100% accurate). The procedure tries every possible value of each attribute and selects the condition with highest information gain.

In the prune phase, for each rule any final sequences of the antecedents is pruned with the pruning metric \( p/(p+n) \).

Rule Set learners are often compared to Decision Tree learners. Rule Sets have the advantage that they are easy to understand, representable in first order logic (easy to implement in languages like Prolog) and prior knowledge can be added to them easily. The major disadvantages of Rule Sets were that they scaled poorly with training set size and had problems with noisy data. The RIPPER algorithm (which this operator implements) pretty much overcomes these disadvantages. The major problem with Decision Trees is overfitting i.e. the model works very well on the training set but does not perform well on the validation set. Reduced Error Pruning (REP) is a technique that tries to overcome overfitting. After various improvements and enhancements over the period of time REP changed to IREP, IREP* and RIPPER.

Pruning in decision trees is a technique in which leaf nodes that do not add to the discriminative power of the decision tree are removed. This is done to convert an over-specific or over-fitted tree to a more general form in order to enhance its predictive power on unseen datasets. A similar concept of pruning implies on Rule Sets.

**Input Ports**

*training set (tra)* This input port expects an ExampleSet. It is the output of the Discretize by Frequency operator in the attached Example Process. The output of other operators can also be used as input.
7. Modeling

Output Ports

**model** *(mod)* The Rule Model is delivered from this output port. This model can now be applied on unseen data sets.

**example set** *(exa)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**criterion** *(selection)* This parameter specifies the criterion for selecting attributes and numerical splits. It can have one of the following values:

- **information gain** The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values.

- **accuracy** Such an attribute is selected for a split that maximizes the accuracy of the Rule Set.

**sample ratio** *(real)* This parameter specifies the sample ratio of training data used for growing and pruning.

**pureness** *(real)* This parameter specifies the desired pureness, i.e. the minimum ratio of the major class in a covered subset in order to consider the subset pure.

**minimal prune benefit** *(real)* This parameter specifies the minimum amount of benefit which must be exceeded over unpruned benefit in order to be pruned.

**use local random seed** *(boolean)* Indicates if a *local random seed* should be used for randomization.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.
Tutorial Processes

Introduction to the Rule Induction operator

The 'Golf' data set is loaded using the Retrieve operator. The Discretize by Frequency operator is applied on it to convert the numerical attributes to nominal attributes. This is done because the Rule Learners usually perform well on nominal attributes. The number of bins parameter of the Discretize by Frequency operator is set to 3. All other parameters are used with default values. A breakpoint is inserted here so that you can have a look at the ExampleSet before application of the Rule Induction operator. The Rule Induction operator is applied next. All parameters are used with default values. The resulting model is connected to the result port of the process. The Rule Set (RuleModel) can be seen in the Results Workspace after execution of the process.

Subgroup Discovery

This operator performs an exhaustive subgroup discovery. The goal of subgroup discovery is to find rules describing subsets of the population that are sufficiently large and statistically unusual.
7. Modeling

Description

This operator discovers subgroups (or induces a rule set) by generating hypotheses exhaustively. Generation is done by stepwise refining the empty hypothesis (which contains no literals). The loop for this task hence iterates over the depth of the search space, i.e. the number of literals of the generated hypotheses. The maximum depth of the search can be specified by the \textit{max depth} parameter. Furthermore the search space can be pruned by specifying a minimum coverage (by the \textit{min coverage} parameter) of the hypothesis or by using only a given amount of hypotheses which have the highest coverage. From the hypotheses, rules are derived according to the user's preference. This operator allows the derivation of positive rules and negative rules separately or the combination by deriving both rules or only the one which is the most probable due to the examples covered by the hypothesis (hence: the actual prediction for that subset). This behavior can be controlled by the \textit{rule generation} parameter. All generated rules are evaluated on the ExampleSet by a user specified utility function (which is specified by the \textit{utility function} parameter) and stored in the final rule set if:

- They exceed a minimum utility threshold (which is specified by the \textit{min utility} parameter) or
- They are among the k best rules (where k is specified by the \textit{k best rules} parameter).

The desired behavior can be specified by the \textit{mode} parameter.

The problem of subgroup discovery has been defined as follows: Given a population of individuals and a property of those individuals we are interested in finding population subgroups that are statistically most interesting, e.g. are as large as possible and have the most unusual statistical (distributional) characteristics with respect to the property of interest. In subgroup discovery, rules have the form \textit{Class} - \textit{Cond}, where the property of interest for subgroup discovery is the class value \textit{Class} which appears in the rule consequent, and the rule antecedent \textit{Cond} is a conjunction of features (attribute-value pairs) selected from the features describing the training instances. As rules are induced from labeled
training instances (labeled positive if the property of interest holds, and negative otherwise), the process of subgroup discovery is targeted at uncovering properties of a selected target population of individuals with the given property of interest. In this sense, subgroup discovery is a form of supervised learning. However, in many respects subgroup discovery is a form of descriptive induction as the task is to uncover individual interesting patterns in data.

Rule learning is most frequently used in the context of classification rule learning and association rule learning. While classification rule learning is an approach to predictive induction (or supervised learning), aimed at constructing a set of rules to be used for classification and/or prediction, association rule learning is a form of descriptive induction (non-classification induction or unsupervised learning), aimed at the discovery of individual rules which define interesting patterns in data.

Let us emphasize the difference between subgroup discovery (as a task at the intersection of predictive and descriptive induction) and classification rule learning (as a form of predictive induction). The goal of standard rule learning is to generate models, one for each class, consisting of rule sets describing class characteristics in terms of properties occurring in the descriptions of training examples. In contrast, subgroup discovery aims at discovering individual rules or 'patterns' of interest, which must be represented in explicit symbolic form and which must be relatively simple in order to be recognized as actionable by potential users. Moreover, standard classification rule learning algorithms cannot appropriately address the task of subgroup discovery as they use the covering algorithm for rule set construction which hinders the applicability of classification rule induction approaches in subgroup discovery. Subgroup discovery is usually seen as different from classification, as it addresses different goals (discovery of interesting population subgroups instead of maximizing classification accuracy of the induced rule set).
7. Modeling

Input Ports

\textbf{training set (tra)} This input port expects an ExampleSet. It is the output of the Generate Nominal Data operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

\textbf{model (mod)} The Rule Set is delivered from this output port.
\textbf{example set (exa)} The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

\textbf{mode (selection)} This parameter specifies the discovery mode.

- \textbf{minimum utility} If this option is selected the rules are stored in the final rule set if they exceed the minimum utility threshold specified by the \textit{min utility} parameter

- \textbf{k best rules} If this option is selected the rules are stored in the final rule set if they are among the k best rules (where k is specified by the \textit{k best rules} parameter).

\textbf{utility function (selection)} This parameter specifies the desired utility function.
\textbf{min utility (real)} This parameter specifies the minimum utility. This parameter is useful when the \textit{mode} parameter is set to 'minimum utility'. The rules are stored in the final rule set if they exceed the minimum utility threshold specified by this parameter.
\textbf{k best rules (integer)} This parameter specifies the number of required best rules.
This parameter is useful when the \textit{mode} parameter is set to 'k best rules'. The rules are stored in the final rule set if they are among the \textit{k} best rules where \textit{k} is specified by this parameter.

\textbf{rule generation (selection)} This parameter determines which rules should be generated. This operator allows the derivation of positive rules and negative rules separately or the combination by deriving both rules or only the one which is the most probable due to the examples covered by the hypothesis (hence: the actual prediction for that subset).

\textbf{max depth (integer)} This parameter specifies the maximum depth of breadth-first search. The loop for this task iterates over the depth of the search space, i.e. the number of literals of the generated hypotheses. The maximum depth of the search can be specified by this parameter

\textbf{min coverage (real)} This parameter specifies the minimum coverage. Only the rules which exceed this coverage threshold are considered.

\textbf{max cache (integer)} This parameter bounds the number of rules which are evaluated (only the most supported rules are used).

\section*{Tutorial Processes}

\subsection*{Introduction to the Subgroup Discovery operator}

The Generate Nominal Data operator is used for generating an ExampleSet. The ExampleSet has two binominal attributes with 100 examples. The Subgroup Discovery operator is applied on this ExampleSet with default values of all parameters. The \textit{mode} parameter is set to 'k best rules' and the \textit{k best rules} parameter is set to 10. Moreover the \textit{utility function} parameter is set to 'WRAcc'. Thus the Rule Set will be composed of 10 best rules where rules are evaluated by the WRAcc function. The resultant Rule Set can be seen in the Results Workspace. You can see that there are 10 rules and they are sorted in order of their WRAcc values.
Tree to Rules

This operator is a meta learner. It uses an inner tree learner for creating a rule model.

Description

The Tree to Rules operator determines a set of rules from the given decision tree model. This operator is a nested operator i.e. it has a subprocess. The subprocess must have a tree learner i.e. an operator that expects an ExampleSet and generates a tree model. This operator builds a rule model using the tree learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Decision tree is a predictive model which maps observations about an item to conclusions about the item's target value. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making.
7.1. Classification and Regression

Input Ports

**training set** (*tra*) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

**model** (*mod*) The rule model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set** (*exa*) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Tutorial Processes

Generating rules from a Decision Tree

The 'Sonar' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can have a look at this ExampleSet. The Tree to Rules operator is applied on this ExampleSet. The Decision Tree operator is applied in the subprocess of the Tree to Rules operator. A *breakpoint* is inserted after the Decision Tree operator so that you can have a look at the Decision Tree. The Tree to Rules operator generates a rule model from this Tree. The resultant rule model can be seen in the Results Workspace.
7. Modeling

Perceptron

This operator learns a linear classifier called Single Perceptron which finds separating hyperplane (if existent). This operator cannot handle polynominal attributes.

Description

The perceptron is a type of artificial neural network invented in 1957 by Frank Rosenblatt. It can be seen as the simplest kind of feed-forward neural network: a linear classifier. Beside all biological analogies, the single layer perceptron is simply a linear classifier which is efficiently trained by a simple update rule: for all wrongly classified data points, the weight vector is either increased or decreased by the corresponding example values. The coming paragraphs explain the basic ideas about neural networks and feed-forward neural networks.

An artificial neural network (ANN), usually called neural network (NN), is a mathematical model or computational model that is inspired by the structure and functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation (the central connectionist principle is that mental phenomena can be described by interconnected networks of simple and often uniform units). In most cases an ANN is an adaptive system that changes
its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are usually used to model complex relationships between inputs and outputs or to find patterns in data.

A feed-forward neural network is an artificial neural network where connections between the units do not form a directed cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) to the output nodes. There are no cycles or loops in the network. If you want to use a more sophisticated neural net, please use the Neural Net operator.

### Input Ports

**training set (tra)** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**model (mod)** The Hyperplane model is delivered from this output port. This model can now be applied on unseen data sets for the prediction of the label attribute.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
7. Modeling

Parameters

**rounds (integer)** This parameter specifies the number of datascans to use to adapt the hyperplane.

**learning rate (real)** This parameter determines how much the weights should be changed at each step. It should not be 0. The hyperplane will adapt to each example with this rate.

Tutorial Processes

Introduction to Perceptron operator

The 'Ripley' data set is loaded using the Retrieve operator. A breakpoint is inserted here so you can see the data set before the application of the Perceptron operator. You can see that this data set has two regular attributes: att1 and att2. The label attribute has two possible values: 1 and 0. The Perceptron operator is applied on this ExampleSet. All parameters are used with default values. The rounds parameter is set to 3 and the learning rate parameter is set to 0.05. After running the process, you can see the resultant hyperplane model in the Results Workspace.
7.1. Classification and Regression

**Neural Net**

This operator learns a model by means of a feed-forward neural network trained by a back propagation algorithm (multi-layer perceptron). This operator cannot handle polynominal attributes.

**Description**

This operator learns a model by means of a feed-forward neural network trained by a back propagation algorithm (multi-layer perceptron). The coming paragraphs explain the basic ideas about neural networks, feed-forward neural networks, back-propagation and multi-layer perceptron.

An artificial neural network (ANN), usually called neural network (NN), is a mathematical model or computational model that is inspired by the structure and functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation (the central connectionist principle is that mental phenomena can be described by interconnected networks of simple and often uniform units). In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are usually used to model complex relationships between inputs and outputs or to find patterns in data.

A feed-forward neural network is an artificial neural network where connections between the units do not form a directed cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) to the output nodes. There are no cycles or loops in the network.

Back propagation algorithm is a supervised learning method which can be divided into two phases: propagation and weight update. The two phases are repeated until the performance of the network is good enough. In back propagation algorithms, the output values are compared with the correct answer to compute the
value of some predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small. In this case, one would say that the network has learned a certain target function.

A multilayer perceptron (MLP) is a feed-forward artificial neural network model that maps sets of input data onto a set of appropriate output. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes back propagation for training the network. This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. In many applications the units of these networks apply a sigmoid function as an activation function.

In this operator usual sigmoid function is used as the activation function. Therefore, the values ranges of the attributes should be scaled to -1 and +1. This can be done through the normalize parameter. The type of the output node is sigmoid if the learning data describes a classification task and linear if the learning data describes a numerical regression task.

Input Ports

**training set** *(tra)* The input port expects an ExampleSet. It is output of the Retrieve operator in our example process. The output of other operators can also be used as input.
Output Ports

model \((mod)\) The Neural Net model is delivered from this output port. This model can now be applied on unseen data sets for prediction of the \(label\) attribute.

example set \((exa)\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

hidden layers This parameter describes the name and the size of all hidden layers. The user can define the structure of the neural network with this parameter. Each list entry describes a new hidden layer. Each entry requires name and size of the hidden layer. The layer name can be chosen arbitrarily. It is only used for displaying the model. Note that the actual number of nodes will be one more than the value specified as hidden layer size because an additional constant node will be added to each layer. This node will not be connected to the preceding layer. If the hidden layer size value is set to -1 the layer size would be calculated from the number of attributes of the input example set. In this case, the layer size will be set to \((\text{number of attributes} + \text{number of classes}) / 2 + 1\). If the user does not specify any hidden layers, a default hidden layer with sigmoid type and size equal to \((\text{number of attributes} + \text{number of classes}) / 2 + 1\) will be created and added to the net. If only a single layer without nodes is specified, the input nodes are directly connected to the output nodes and no hidden layer will be used.

training cycles \((integer)\) This parameter specifies the number of training cycles used for the neural network training. In back-propagation the output values are compared with the correct answer to compute the value of some predefined error-function. The error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. This process is repeated \(n\) number
of times. \( n \) can be specified using this parameter.

**learning rate** (real) This parameter determines how much we change the weights at each step. It should not be 0.

**momentum** (real) The momentum simply adds a fraction of the previous weight update to the current one. This prevents local maxima and smoothes optimization directions.

**decay** (boolean) This is an expert parameter. It indicates if the learning rate should be decreased during learning.

**shuffle** (boolean) This is an expert parameter. It indicates if the input data should be shuffled before learning. Although it increases memory usage but it is recommended if data is sorted before.

**normalize** (boolean) This is an expert parameter. The Neural Net operator uses an usual sigmoid function as the activation function. Therefore, the value range of the attributes should be scaled to -1 and +1. This can be done through the normalize parameter. Normalization is performed before learning. Although it increases runtime but it is necessary in most cases.

**error epsilon** (real) The optimization is stopped if the training error gets below this epsilon value.

**use local random seed** (boolean) Indicates if a local random seed should be used for randomization.

**local random seed** (integer) This parameter specifies the local random seed. It is only available if the use local random seed parameter is set to true.

---

**Tutorial Processes**

**Introduction to Neural Net**

The 'Ripley' data set is loaded using the Retrieve operator. A breakpoint is inserted here so you can see the data set before the application of the Neural Net operator. You can see that this data set has two regular attributes i.e. att1 and att2. The label attribute has two possible values i.e. 1 or 0. Then the Neural Net operator is applied on it. All parameters are used with default values. When
you run the process, you can see the neural net in the Results Workspace. There are $x+1$ number of nodes in the input, where $x$ is the number of attributes in the input ExampleSet (other than label attribute). The last node is the threshold node. There are $y$ number of nodes in the output, where $y$ is the number of classes in the input ExampleSet (i.e. number of possible values of label attribute). As no value was specified in the hidden layers parameter, the default value is used. Therefore, the number of nodes are created in hidden layer are $= \text{size of hidden layer} = (\text{number of attributes} + \text{number of classes}) / 2 + 1 = (2+2)/2+1= 3$.

The last node (4th node) is a threshold node. The connections between nodes are colored darker if the connection weight is high. You can click on a node in this visualization in order to see the actual weights.

This simple process just provides basic working of this operator. In real scenarios all parameters should be carefully chosen.

**Linear Regression**

This operator calculates a linear regression model from the input ExampleSet.

**Description**

Regression is a technique used for numerical prediction. Regression is a statistical measure that attempts to determine the strength of the relationship between
one dependent variable (i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset.

Linear regression attempts to model the relationship between a scalar variable and one or more explanatory variables by fitting a linear equation to observed data. For example, one might want to relate the weights of individuals to their heights using a linear regression model.

This operator calculates a linear regression model. It uses the Akaike criterion for model selection. The Akaike information criterion is a measure of the relative goodness of a fit of a statistical model. It is grounded in the concept of information entropy, in effect offering a relative measure of the information lost when a given model is used to describe reality. It can be said to describe the tradeoff between bias and variance in model construction, or loosely speaking between accuracy and complexity of the model.

**Differentiation**

**Polynomial Regression** Polynomial regression is a form of linear regression in which the relationship between the independent variable $x$ and the dependent variable $y$ is modeled as an $n$th order polynomial. See page 705 for details.

**Input Ports**

**training set (tra)** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes.
Thus often you may have to use the Nominal to Numerical operator before application of this operator.

**Output Ports**

- **model** *(mod)*: The regression model is delivered from this output port. This model can now be applied on unseen data sets.
- **example set** *(exa)*: The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
- **weights** *(wei)*: This port delivers the attribute weights.

**Parameters**

- **feature selection** *(selection)*: This is an expert parameter. It indicates the feature selection method to be used during regression. Following options are available: none, M5 prime, greedy, T-Test, iterative T-Test
- **alpha** *(real)*: This parameter is available only when the **feature selection** parameter is set to 'T-Test'. It specifies the value of alpha to be used in the T-Test feature selection.
- **max iterations** *(integer)*: This parameter is only available when the **feature selection** parameter is set to 'iterative T-Test'. It specifies the maximum number of iterations of the iterative T-Test for feature selection.
- **forward alpha** *(real)*: This parameter is only available when the **feature selection** parameter is set to 'iterative T-Test'. It specifies the value of forward alpha to be used in the T-Test feature selection.
- **backward alpha** *(real)*: This parameter is only available when the **feature selection** parameter is set to 'iterative T-Test'. It specifies the value of backward alpha to be used in the T-Test feature selection.
- **eliminate colinear features** *(boolean)*: This parameter indicates if the algorithm should try to delete collinear features during the regression or not.
min tolerance (real) This parameter is only available when the eliminate collinear features parameter is set to true. It specifies the minimum tolerance for eliminating collinear features.

use bias (boolean) This parameter indicates if an intercept value should be calculated or not.

ridge (real) This parameter specifies the ridge parameter for using in ridge regression.

Related Documents

Polynomial Regression (705)

Tutorial Processes

Applying the Linear Regression operator on the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it. The first example parameter of the Filter Example Range parameter is set to 1 and the last example parameter is set to 100. Thus the first 100 examples of the 'Polynomial' data set are selected. The Linear Regression operator is applied on it with default values of all parameters. The regression model generated by the Linear Regression operator is applied on the last 100 examples of the 'Polynomial' data set using the Apply Model operator. Labeled data from the Apply Model operator is provided to the Performance (Regression) operator. The absolute error and the prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and the prediction average in the labeled data set. The absolute error is calculated by adding the difference of all predicted values from the actual values.
of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

Polynomial Regression

This operator generates a polynomial regression model from the given ExampleSet. Polynomial regression is considered to be a special case of multiple linear regression.

Description

Polynomial regression is a form of linear regression in which the relationship between the independent variable $x$ and the dependent variable $y$ is modeled as an $n$th order polynomial. In RapidMiner, $y$ is the label attribute and $x$ is the set of...
regular attributes that are used for the prediction of $y$. Polynomial regression fits a nonlinear relationship between the value of $x$ and the corresponding conditional mean of $y$, denoted $E(y \mid x)$, and has been used to describe nonlinear phenomena such as the growth rate of tissues and the progression of disease epidemics. Although polynomial regression fits a nonlinear model to the data, as a statistical estimation problem it is linear, in the sense that the regression function $E(y \mid x)$ is linear in the unknown parameters that are estimated from the data. For this reason, polynomial regression is considered to be a special case of multiple linear regression.

The goal of regression analysis is to model the expected value of a dependent variable $y$ in terms of the value of an independent variable (or vector of independent variables) $x$. In simple linear regression, the following model is used:

$$y = w0 + (w1 * x)$$

In this model, for each unit increase in the value of $x$, the conditional expectation of $y$ increases by $w1$ units.

In many settings, such a linear relationship may not hold. For example, if we are modeling the yield of a chemical synthesis in terms of the temperature at which the synthesis takes place, we may find that the yield improves by increasing amounts for each unit increase in temperature. In this case, we might propose a quadratic model of the form:

$$y = w0 + (w1 * x1^1) + (w2 * x2^2)$$

In this model, when the temperature is increased from $x$ to $x + 1$ units, the expected yield changes by $w1 + w2 + 2(w2 * x)$. The fact that the change in yield depends on $x$ is what makes the relationship nonlinear (this must not be confused with saying that this is nonlinear regression; on the contrary, this is still a case of linear regression). In general, we can model the expected value of $y$ as an $nth$ order polynomial, yielding the general polynomial regression model:

$$y = w0 + (w1 * x1^1) + (w2 * x2^2) + \ldots + (wm * xm^m)$$

Regression is a technique used for numerical prediction. It is a statistical measure
that attempts to determine the strength of the relationship between one dependent variable (i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset.

**Differentiation**

**Linear Regression** Polynomial regression is a form of linear regression in which the relationship between the independent variable $x$ and the dependent variable $y$ is modeled as an $n$th order polynomial. See page 701 for details.

**Input Ports**

**training set** $(tra)$ This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

**Output Ports**

**model** $(mod)$ The regression model is delivered from this output port. This model can now be applied on unseen data sets.

**example set** $(exa)$ The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the
7. Modeling

Results Workspace.

Parameters

**max iterations** *(integer)* This parameter specifies the maximum number of iterations to be used for the model fitting.

**replication factor** *(integer)* This parameter specifies the amount of times each input variable is replicated, i.e. how many different degrees and coefficients can be applied to each variable.

**max degree** *(integer)* This parameter specifies the maximal degree to be used for the final polynomial.

**min coefficient** *(real)* This parameter specifies the minimum number to be used for the coefficients and the offset.

**max coefficient** *(real)* This parameter specifies the maximum number to be used for the coefficients and the offset.

**use local random seed** *(boolean)* This parameter indicates if a local random seed should be used for randomization. Using the same value of the local random seed will produce the same randomization.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

Related Documents

Linear Regression (701)

Tutorial Processes

Applying the Polynomial Regression operator on the Polynomial data set
7.1. Classification and Regression

The 'Polynomial' data set is loaded using the Retrieve operator. The Split Data operator is applied on it to split the ExampleSet into training and testing data sets. The Polynomial Regression operator is applied on the training data set with default values of all parameters. The regression model generated by the Polynomial Regression operator is applied on the testing data set of the 'Polynomial' data set using the Apply Model operator. The labeled data set generated by the Apply Model operator is provided to the Performance (Regression) operator. The absolute error and the prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and the prediction average in the labeled data set. The absolute error is calculated by adding the difference of all predicted values from the actual values of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

Logistic Regression
This operator is a Logistic Regression Learner. It is based on the internal Java implementation of the *myKLR* by Stefan Rueping.

**Description**

This learner uses the Java implementation of the *myKLR* by Stefan Rueping. *myKLR* is a tool for large scale kernel logistic regression based on the algorithm of Keerthi et al (2003) and the code of *mySVM*. For compatibility reasons, the model of *myKLR* differs slightly from that of Keerthi et al (2003). As *myKLR* is based on the code of *mySVM*, the format of example files, parameter files and kernel definition are identical. Please see the documentation of the SVM operator for further information. This learning method can be used for both regression and classification and provides a fast algorithm and good results for many learning tasks. *mySVM* works with linear or quadratic and even asymmetric loss functions.

This operator supports various kernel types including *dot*, *radial*, *polynomial*, *neural*, *anova*, *epachnenikov*, *gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

**Input Ports**

training set (tra) This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

**Output Ports**

model (mod) The Logistic Regression model is delivered from this output port. This model can now be applied on unseen data sets.
weights (wei) This port delivers the attribute weights. This is only possible
7.1. Classification and Regression

when the dot kernel type is used, it is not possible with other kernel types.

**example set** *(exa)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**kernel type** *(selection)* The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot*, *radial*, *polynomial*, *neural*, *anova*, *epachnenikov*, *gaussian combination*, *multiquadric*

- **dot** The dot kernel is defined by $k(x,y)=x*y$ i.e. it is inner product of $x$ and $y$.

- **radial** The radial kernel is defined by $\exp(-g \|x-y\|^2)$ where $g$ is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x,y)=(x*y+1)^d$ where $d$ is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net $\tanh(a x*y+b)$ where $a$ is *alpha* and $b$ is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is $1/N$, where $N$ is the data dimension. Note that not all choices of $a$ and $b$ lead to a valid kernel function.

- **anova** The anova kernel is defined by raised to power $d$ of summation of $\exp(-g (x-y))$ where $g$ is *gamma* and $d$ is *degree*. gamma and degree are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.
7. Modeling

- **epachnenikov** The epachnenikov kernel is this function \((3/4)(1-u^2)\) for \(u\) between -1 and 1 and zero for \(u\) outside that range. It has two adjustable parameters \(kernel\ \sigma_1\) and \(kernel\ degree\).

- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters \(kernel\ \sigma_1, kernel\ \sigma_2\) and \(kernel\ \sigma_3\).

- **multiquadric** The multiquadric kernel is defined by the square root of \(|x-y|^2 + c^2\). It has adjustable parameters \(kernel\ \sigma_1\) and \(kernel\ \sigma\) shift.

**kernel gamma** (real) This is the SVM kernel parameter gamma. This is only available when the \(kernel\ type\) parameter is set to radial or anova.

**kernel sigma1** (real) This is the SVM kernel parameter sigma1. This is only available when the \(kernel\ type\) parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** (real) This is the SVM kernel parameter sigma2. This is only available when the \(kernel\ type\) parameter is set to gaussian combination.

**kernel sigma3** (real) This is the SVM kernel parameter sigma3. This is only available when the \(kernel\ type\) parameter is set to gaussian combination.

**kernel shift** (real) This is the SVM kernel parameter shift. This is only available when the \(kernel\ type\) parameter is set to multiquadric.

**kernel degree** (real) This is the SVM kernel parameter degree. This is only available when the \(kernel\ type\) parameter is set to polynomial, anova or epachnenikov.

**kernel a** (real) This is the SVM kernel parameter a. This is only available when the \(kernel\ type\) parameter is set to neural.

**kernel b** (real) This is the SVM kernel parameter b. This is only available when the \(kernel\ type\) parameter is set to neural.

**kernel cache** (real) This is an expert parameter. It specifies the size of the cache for kernel evaluations in megabytes.

**C** (real) This is the SVM complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.
7.1. Classification and Regression

**convergence epsilon** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations (integer)** This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**scale (boolean)** This is a global parameter. If checked, the example values are scaled and the scaling parameters are stored for a test set.

## Tutorial Processes

### Introduction to the Logistic Regression operator

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a regression model. The Logistic Regression operator is applied in the training subprocess of the Split Validation operator. All parameters are used with default values. The Logistic Regression operator generates a regression model. The Apply Model operator is used in the testing subprocess to apply this model on the testing data set. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The regression model and its performance vector are connected to the output and it can be seen in the Results Workspace.
Support Vector Machine

This operator is an SVM (Support Vector Machine) Learner. It is based on the internal Java implementation of the *mySVM* by Stefan Rueping.

Description

This learner uses the Java implementation of the support vector machine *mySVM* by Stefan Rueping. This learning method can be used for both regression and classification and provides a fast algorithm and good results for many learning tasks. *mySVM* works with linear or quadratic and even asymmetric loss functions.

This operator supports various kernel types including *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

Here is a basic description of the SVM. The standard SVM takes a set of input data and predicts, for each given input, which of the two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classifica-
7.1. Classification and Regression

tion, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by the SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function $K(x,y)$ selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that space is constant.

Input Ports

training set (tra) This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

Output Ports

model (mod) The SVM model is delivered from this output port. This model can now be applied on unseen data sets.

element set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

estimated performance (est) This port delivers a performance vector of the SVM model which gives an estimation of statistical performance of this model.

weights (wei) This port delivers the attribute weights. This is possible only
when the dot kernel type is used, it is not possible with other kernel types.

Parameters

kernel type (selection) The type of the kernel function is selected through this parameter. Following kernel types are supported: dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric

- **dot** The dot kernel is defined by \( k(x,y) = x^*y \) i.e. it is inner product of \( x \) and \( y \).

- **radial** The radial kernel is defined by \( \exp(-g ||x-y||^2) \) where \( g \) is the gamma, it is specified by the kernel gamma parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by \( k(x,y) = (x^*y + 1)^d \) where \( d \) is the degree of polynomial and it is specified by the kernel degree parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net \( \tanh(a x^*y + b) \) where \( a \) is alpha and \( b \) is the intercept constant. These parameters can be adjusted using the kernel a and kernel b parameters. A common value for alpha is \( 1/N \), where \( N \) is the data dimension. Note that not all choices of \( a \) and \( b \) lead to a valid kernel function.

- **anova** The anova kernel is defined by raised to power \( d \) of summation of \( \exp(-g (x-y)) \) where \( g \) is gamma and \( d \) is degree. gamma and degree are adjusted by the kernel gamma and kernel degree parameters respectively.

- **epachnenikov** The epachnenikov kernel is this function \( (3/4)(1-u^2) \) for \( u \) between -1 and 1 and zero for \( u \) outside that range. It has two adjustable parameters kernel sigma1 and kernel degree.

- **gaussian_combination** This is the gaussian combination kernel. It has
adjustable parameters kernel sigma1, kernel sigma2 and kernel sigma3.

- multiquadric The multiquadric kernel is defined by the square root of \( ||x-y||^2 + c^2 \). It has adjustable parameters kernel sigma1 and kernel sigma shift.

**kernel gamma** (real) This is the SVM kernel parameter gamma. This is available only when the kernel type parameter is set to radial or anova.

**kernel sigma1** (real) This is the SVM kernel parameter sigma1. This is available only when the kernel type parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** (real) This is the SVM kernel parameter sigma2. This is available only when the kernel type parameter is set to gaussian combination.

**kernel sigma3** (real) This is the SVM kernel parameter sigma3. This is available only when the kernel type parameter is set to gaussian combination.

**kernel shift** (real) This is the SVM kernel parameter shift. This is available only when the kernel type parameter is set to multiquadric.

**kernel degree** (real) This is the SVM kernel parameter degree. This is available only when the kernel type parameter is set to polynomial, anova or epachnenikov.

**kernel a** (real) This is the SVM kernel parameter a. This is available only when the kernel type parameter is set to neural.

**kernel b** (real) This is the SVM kernel parameter b. This is available only when the kernel type parameter is set to neural.

**kernel cache** (real) This is an expert parameter. It specifies the size of the cache for kernel evaluations in megabytes.

**C** (real) This is the SVM complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**convergence epsilon** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations** (integer) This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**scale** (boolean) This is a global parameter. If checked, the example values are scaled and the scaling parameters are stored for a test set.
7. Modeling

**L pos** *(real)* A factor for the SVM complexity constant for positive examples. This parameter is part of the loss function.

**L neg** *(real)* A factor for the SVM complexity constant for negative examples. This parameter is part of the loss function.

**epsilon** *(real)* This parameter specifies the insensitivity constant. No loss if the prediction lies this close to true value. This parameter is part of the loss function.

**epsilon plus** *(real)* This parameter is part of the loss function. It specifies epsilon for positive deviation only.

**epsilon minus** *(real)* This parameter is part of the loss function. It specifies epsilon for negative deviation only.

**balance cost** *(boolean)* If checked, adapts Cpos and Cneg to the relative size of the classes.

**quadratic loss pos** *(boolean)* Use quadratic loss for positive deviation. This parameter is part of the loss function.

**quadratic loss neg** *(boolean)* Use quadratic loss for negative deviation. This parameter is part of the loss function.

---

**Tutorial Processes**

**Getting started with SVM**

This is a simple Example Process which gets you started with the SVM operator. The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it to convert its nominal attributes to numerical form. This step is necessary because the SVM operator cannot take nominal attributes, it can only classify using numerical attributes. The model generated from the SVM operator is then applied on the 'Golf-Testset' data set. Nominal to Numerical operator was applied on this data set as well. This is necessary because the testing and training data set should be in the same format. The statistical performance of this model is measured using the Performance operator. This is a very basic process. It is recommended that you develop a deeper understanding of SVM for getting better results through this operator. The support
vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps.

Using 'm' numbers to represent an m-category attribute is recommended. Only one of the 'm' numbers is 1, the others are 0. For example, a three-category attribute such as Outlook {overcast, sunny, rain} can be represented as (0,0,1), (0,1,0), and (1,0,0). This can be achieved by setting the coding type parameter to 'dummy coding' in the Nominal to Numerical operator. Generally, if the number of values in an attribute is not too large, this coding might be more stable than using a single number.

To get a more accurate classification model from SVM, scaling is recommended. The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. Because kernel values usually depend on the inner products of feature vectors, e.g. the linear kernel and the polynomial kernel, large attribute values might cause numerical problems. Scaling should be performed on both training and testing data sets. In this process the scale parameter is checked. Uncheck the scale parameter and run the process again. You will see that this time it takes a lot longer than the time taken with scaling.

You should have a good understanding of kernel types and different parameters associated with each kernel type in order to get better results from this operator. The gaussian combination kernel was used in this example process. All parameters were used with default values. The accuracy of this model was just 35.71%. Try changing different parameters to get better results. If you change the parameter $C$ to 1 instead of 0, you will see that accuracy of the model rises to 64.29%. Thus, you can see how making small changes in parameters can have a significant effect on overall results. Thus it is very necessary to have a good understanding of parameters of kernel type in use. It is equally important to have a good understanding of different kernel types, and choosing the most suitable kernel type for your ExampleSet. Try using the polynomial kernel in this Example Process (also set the parameter $C$ to 0); you will see that accuracy is around 71.43% with default values for all parameters. Change the value of the
parameter $C$ to 1 instead of 0. Doing this increased the accuracy of model with *gaussian combination* kernel, but here you will see that accuracy of the model drops.

We used default values for most of the parameters. To get more accurate results these values should be carefully selected. Usually techniques like cross-validation are used to find the best values of these parameters for the ExampleSet under consideration.

**Support Vector Machine (LibSVM)**

This operator is an SVM (Support vector machine) Learner. It is based on the Java libSVM.

**Description**

This operator applies the [http://www.csie.ntu.edu.tw/~cjlin/libsvm/libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/libsvm) learner by Chih-Chung Chang and Chih-Jen Lin. SVM is a powerful method for both
classification and regression. This operator supports the C-SVC and nu-SVC SVM types for classification tasks as well as the epsilon-SVR and nu-SVR SVM types for regression tasks. Additionally one-class SVM type is supported for distribution estimation. The one-class SVM type gives the possibility to learn from just one class of examples and later on test if new examples match the known ones. In contrast to other SVM learners, the libsvm supports internal multiclass learning and probability estimation based on Platt scaling for proper confidence values after applying the learned model on a classification data set.

Here is a basic description of SVM. The standard SVM takes a set of input data and predicts, for each given input, which of two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite- dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space would be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function $K(x,y)$ selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that
space is constant.

For more information regarding libsvm you can visit “http://www.csie.ntu.edu.tw/~cjlin/libsvm”.

Input Ports

**training set (tra)** This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before applying this operator.

Output Ports

**model (mod)** The SVM model is delivered from this output port. This model can now be applied on unseen data sets.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**svm type (selection)** The SVM type is selected through this parameter. This operator supports the *C-SVC* and *nu-SVC* SVM types for classification tasks. The *epsilon-SVR* and *nu-SVR* SVM types are for regression tasks. The *one-class* SVM type is for distribution estimation. The *one-class* SVM type gives the possibility to learn from just one class of examples and later on test if new examples match the known ones.

**kernel type (selection)** The type of the kernel function is selected through this parameter. Following kernel types are supported: *linear, poly, rbf, sigmoid,*
7.1. Classification and Regression

*precomputed*. The *rbf* kernel type is the default value. In general, the *rbf* kernel is a reasonable first choice. Here are a few guidelines regarding different kernel types.

- the *rbf* kernel nonlinearly maps samples into a higher dimensional space
- the *rbf* kernel, unlike the *linear* kernel, can handle the case when the relation between class labels and attributes is nonlinear
- the *linear* kernel is a special case of the *rbf* kernel
- the *sigmoid* kernel behaves like the *rbf* kernel for certain parameters
- the number of hyperparameters influence the complexity of model selection. The *poly* kernel has more hyperparameters than the *rbf* kernel
- the *rbf* kernel has fewer numerical difficulties
- the *sigmoid* kernel is not valid under some parameters
- There are some situations where the *rbf* kernel is not suitable. In particular, when the number of features is very large, one may just use the *linear* kernel.

**degree** *(real)* This parameter is only available when the *kernel type* parameter is set to 'poly'. This parameter is used to specify the degree for a polynomial kernel function.

**gamma** *(real)* This parameter is only available when the *kernel type* parameter is set to 'poly', 'rbf' or 'sigmoid'. This parameter specifies *gamma* for 'polynomial', 'rbf', and 'sigmoid' kernel functions. The value of *gamma* may play an important role in the SVM model. Changing the value of *gamma* may change the accuracy of the resulting SVM model. So, it is a good practice to use cross-validation to find the optimal value of *gamma*.

**coef0** *(real)* This parameter is only available when the *kernel type* parameter is set to 'poly' or 'precomputed'. This parameter specifies coef0 for 'poly' and 'precomputed' kernel functions.

**C** *(real)* This parameter is only available when the *svm type* parameter is set to 'c-SVC', 'epsilon-SVR' or 'nu-SVR'. This parameter specifies the cost parameter C for 'c-SVC', 'epsilon-SVR' and 'nu-SVR'. C is the penalty parameter of the
7. Modeling

error term.

nu (real) This parameter is only available when the svm type parameter is set to 'nu-SVC', 'one-class' and 'nu-SVR'. This parameter specifies the nu parameter for 'nu-SVC', 'one-class' and 'nu-SVR'. Its value should be between 0.0 and 0.5.

cache size (real) This is an expert parameter. It specifies the Cache size in Megabyte.

epsilon (real) This parameter specifies the tolerance of the termination criterion.

p (real) This parameter is only available when the svm type parameter is set to 'epsilon-SVR'. This parameter specifies tolerance of loss function of 'epsilon-SVR'.

class weights (list) This is an expert parameter. It specifies the weights 'w' for all classes. The Edit List button opens a new window with two columns. The first column specifies the class name and the second column specifies the weight for that class. Parameter C is calculated as weight of class multiplied by C. If weight of a class is not specified, that class is assigned weight = 1.

shrinking (boolean) This is an expert parameter. It specifies whether to use the shrinking heuristics.

calculate confidences (boolean) This parameter indicates if proper confidence values should be calculated.

confidence for multiclass (boolean) This is an expert parameter. It indicates if the class with the highest confidence should be selected in the multiclass setting. Uses binary majority vote over all 1-vs-1 classifiers otherwise (selected class must not be the one with highest confidence in that case).

Tutorial Processes

SVM with rbf kernel

This is a simple Example Process which gets you started with the SVM(libSVM) operator. The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it to convert its nominal attributes to numerical form. This step is necessary because the SVM(libSVM) operator can-
7.1. Classification and Regression

not take nominal attributes, it can only classify using numerical attributes. The model generated from the SVM(libSVM) operator is then applied on the 'Golf-Testset' data set using the Apply Model operator. The Nominal to Numerical operator was also applied on this data set. This is necessary because the testing and training data sets should be in the same format. The statistical performance of this model is measured using the Performance operator. This is a very basic process. It is recommended that you develop a deeper understanding of the SVM(libSVM) for getting better results through this operator. The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps.

Using 'm' numbers to represent an m-category attribute is recommended. Only one of the 'm' numbers is 1, and others are 0. For example, a three-category attribute such as Outlook \{overcast, sunny, rain\} can be represented as (0,0,1), (0,1,0), and (1,0,0). This can be achieved by setting the coding type parameter to 'dummy coding' in the Nominal to Numerical operator. Generally, if the number of values in an attribute is not too large, this coding might be more stable than using a single number.

This basic process omitted various essential steps that are necessary for getting acceptable results from this operator. For example to get a more accurate classification model from SVM, scaling is recommended. The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. Because kernel values usually depend on the inner products of feature vectors, e.g. the linear kernel and the polynomial kernel, large attribute values might cause numerical problems. Scaling should be performed on both training and testing data sets.

We have used default values of the parameters $C$, $gamma$ and $epsilon$. To get more accurate results these values should be carefully selected. Usually techniques like cross-validation are used to find best values of these parameters for the ExampleSet under consideration.
Support Vector Machine (Evolutionary)

This operator is SVM implementation using an evolutionary algorithm to solve the dual optimization problem of an SVM.

Description

The Support Vector Machine (Evolutionary) uses an Evolutionary Strategy for optimization. This operator is SVM implementation using an evolutionary algorithm to solve the dual optimization problem of an SVM. It turns out that on many datasets this simple implementation is as fast and accurate as the usual SVM implementations. In addition, it is also capable of learning with Kernels which are not positive semi-definite and can also be used for multi-objective learning which makes the selection of the parameter $C$ unnecessary before learning. For more information please study 'Evolutionary Learning with Kernels: A
Generic Solution for Large Margin Problems' by Ingo Mierswa

This operator supports various kernel types including *dot, radial, polynomial, sigmoid, anova, epachnenikov, gaussian combination* and *multiquadric*. Explanation of these kernel types is given in the parameters section.

Here is a basic description of the SVM. The standard SVM takes a set of input data and predicts, for each given input, which of the two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. Whereas the original problem may be stated in a finite dimensional space, it often happens that the sets to discriminate are not linearly separable in that space. For this reason, it was proposed that the original finite-dimensional space be mapped into a much higher-dimensional space, presumably making the separation easier in that space. To keep the computational load reasonable, the mapping used by the SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function \( K(x,y) \) selected to suit the problem. The hyperplanes in the higher dimensional space are defined as the set of points whose inner product with a vector in that space is constant.
7. Modeling

Input Ports

training set (tra) This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Thus often you may have to use the Nominal to Numerical operator before application of this operator.

Output Ports

model (mod) The SVM model is delivered from this output port. This model can now be applied on unseen data sets.
example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

kernel type (selection) The type of the kernel function is selected through this parameter. Following kernel types are supported: dot, radial, polynomial, sigmoid, anova, epachnenikov, gaussian combination, multiquadric

- dot The dot kernel is defined by $k(x,y) = x^*y$ i.e. it is inner product of $x$ and $y$.
- radial The radial kernel is defined by $exp(-g ||x-y||^2)$ where $g$ is the gamma, it is specified by the kernel gamma parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.
- polynomial The polynomial kernel is defined by $k(x,y) = (x^*y+1)^d$ where $d$ is the degree of polynomial and it is specified by the kernel degree parameter.
The polynomial kernels are well suited for problems where all the training data is normalized.

- **sigmoid** The sigmoid kernel is defined by a two layered neural net \( \tanh(a x^*y+b) \) where \( a \) is alpha and \( b \) is the intercept constant. These parameters can be adjusted using the kernel \( a \) and kernel \( b \) parameters. A common value for alpha is \( 1/N \), where \( N \) is the data dimension. Note that not all choices of \( a \) and \( b \) lead to a valid kernel function.

- **anova** The anova kernel is defined by raised to power \( d \) of summation of \( \exp(-g (x-y)) \) where \( g \) is gamma and \( d \) is degree. gamma and degree are adjusted by the kernel gamma and kernel degree parameters respectively.

- **epachnenikov** The epachnenikov kernel is this function \( (3/4)(1-u^2) \) for \( u \) between -1 and 1 and zero for \( u \) outside that range. It has two adjustable parameters kernel \( sigma1 \) and kernel degree.

- **gaussian combination** This is the gaussian combination kernel. It has adjustable parameters kernel \( sigma1 \), kernel \( sigma2 \) and kernel \( sigma3 \).

- **multiquadric** The multiquadric kernel is defined by the square root of \( ||x-y||^2 + c^2 \). It has adjustable parameters kernel \( sigma1 \) and kernel \( sigma2 \).

**kernel gamma** (*real*) This is the kernel parameter gamma. This is only available when the kernel type parameter is set to radial or anova.

**kernel sigma1** (*real*) This is the kernel parameter sigma1. This is only available when the kernel type parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** (*real*) This is the kernel parameter sigma2. This is only available when the kernel type parameter is set to gaussian combination.

**kernel sigma3** (*real*) This is the kernel parameter sigma3. This is only available when the kernel type parameter is set to gaussian combination.

**kernel shift** (*real*) This is the kernel parameter shift. This is only available when the kernel type parameter is set to multiquadric.

**kernel degree** (*real*) This is the kernel parameter degree. This is only available when the kernel type parameter is set to polynomial, anova or epachnenikov.
7. Modeling

kernel a (real) This is the kernel parameter a. This is only available when the kernel type parameter is set to sigmoid

kernel b (real) This is the kernel parameter b. This is only available when the kernel type parameter is set to sigmoid

C (real) This is the complexity constant which sets the tolerance for misclassification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

epsilon This parameter specifies the width of the regression tube loss function of the regression SVM.

start population type (selection) This parameter specifies the type of start population initialization.

max generations (integer) This parameter specifies the number of generations after which the algorithm should be terminated.

generations without improvement (integer) This parameter specifies the stop criterion for early stopping i.e. it stops after n generations without improvement in the performance. n is specified by this parameter.

population size (integer) This parameter specifies the population size i.e. the number of individuals per generation. If set to -1, all examples are selected.

tournament fraction (real) This parameter specifies the fraction of the current population which should be used as tournament members.

keep best (boolean) This parameter specifies if the best individual should survive. This is also called elitist selection. Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection.

mutation type (selection) This parameter specifies the type of the mutation operator.

selection type (selection) This parameter specifies the selection scheme of this evolutionary algorithms.

crossover prob (real) The probability for an individual to be selected for crossover is specified by this parameter.

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

local random seed (integer) This parameter specifies the local random seed.
This parameter is only available if the use local random seed parameter is set to true.

**hold out set ratio (real)** This operator uses this amount as a hold out set to estimate generalization error after learning.

**show convergence plot (boolean)** This parameter indicates if a dialog with a convergence plot should be drawn.

**show population plot (boolean)** This parameter indicates if the population plot in case of the non-dominated sorting should be shown.

**return optimization performance (boolean)** This parameter indicates if final optimization fitness should be returned as performance.

## Tutorial Processes

### Introduction to the Support Vector Machine (Evolutionary) operator

This is a simple Example Process which gets you started with the SVM (Evolutionary) operator. The Retrieve operator is used to load the 'Golf' data set. The Nominal to Numerical operator is applied on it to convert its nominal attributes to numerical form. This step is necessary because the SVM (Evolutionary) operator cannot take nominal attributes, it can only classify using numerical attributes. The model generated from the SVM (Evolutionary) operator is then applied on the 'Golf-Testset' data set. Nominal to Numerical operator was applied on this data set as well. This is necessary because the testing and training data set should be in the same format. The statistical performance of this model is measured using the Performance operator. This is a very basic process. It is recommended that you develop a deeper understanding of SVM for getting better results through this operator. The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps.
7. Modeling

You should have a good understanding of kernel types and different parameters associated with each kernel type in order to get better results from this operator. The *gaussian combination* kernel was used in this example process. All parameters were used with default values. The accuracy of this model was just 35.71%. Try changing different parameters to get better results. If you change the parameter $C$ to 1 instead of 0, you will see that accuracy of the model rises to 64.29%. Thus, you can see how making small changes in parameters can have a significant effect on overall results. Thus it is very necessary to have a good understanding of parameters of kernel type in use. It is equally important to have a good understanding of different kernel types, and choosing the most suitable kernel type for your ExampleSet. Try using the *polynomial* kernel in this Example Process (also set the parameter $C$ to 0); you will see that accuracy is around 64.29% with default values for all parameters. Change the value of the parameter $C$ to 1 instead of 0. Doing this increased the accuracy of model with *gaussian combination* kernel, but here you will see that accuracy of the model drops.

Default values were used for most of the parameters in the process. To get more accurate results these values should be carefully selected. Usually techniques like cross-validation are used to find the best values of these parameters for the ExampleSet under consideration.
Support Vector Machine (PSO)

This operator is a Support Vector Machine (SVM) Learner which uses Particle Swarm Optimization (PSO) for optimization. PSO is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality.

Description

This operator implements a hybrid approach which combines support vector classifier with particle swarm optimization, in order to improve the strength of each individual technique and compensate for each other's weaknesses. Particle Swarm Optimization (PSO) is an evolutionary computation technique in which each potential solution is seen as a particle with a certain velocity flying through the problem space. Support Vector Machine (SVM) classification operates a linear separation in an augmented space by means of some defined kernels satisfying Mercer's condition. These kernels map the input vectors into a very high di-
7. Modeling

mensional space, possibly of infinite dimension, where linear separation is more likely. Then a linear separating hyper plane is found by maximizing the margin between two classes in this space. Hence the complexity of the separating hyper plane depends on the nature and the properties of the used kernel.

Support Vector Machine (SVM)

Here is a basic description of the SVM. The standard SVM takes a set of input data and predicts, for each given input, which of the two possible classes comprises the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. For more information about SVM please study the description of the SVM operator.

Particle Swarm Optimization (PSO)

Particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found. More specifically, PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by most classic optimization methods. PSO can therefore also be used on optimization problems that are
7.1. Classification and Regression

partially irregular, noisy, change over time, etc.

Input Ports

training set \((tra)\) This input port expects an ExampleSet. This operator cannot handle nominal attributes; it can be applied on data sets with numeric attributes. Moreover, this operator can only be applied on ExampleSets with binominal label.

Output Ports

model \((mod)\) The SVM model is delivered from this output port. This model can now be applied on unseen data sets.
example set \((exa)\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

show convergence plot \((boolean)\) This parameter indicates if a dialog with a convergence plot should be drawn.
kernel type \((selection)\) The type of the kernel function is selected through this parameter. Following kernel types are supported: dot, radial, polynomial, neural, anova, epanchnenikov, gaussian combination, multiquadric

- \textbf{dot} The dot kernel is defined by \(k(x,y)=x^Ty\) i.e. it is inner product of \(x\) and \(y\).
- \textbf{radial} The radial kernel is defined by \(exp(-g \|x-y\|^2)\) where \(g\) is the \textit{gamma}, it is specified by the \textit{kernel gamma} parameter. The adjustable parameter \textit{gamma} plays a major role in the performance of the kernel, and should be
7. Modeling

carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x,y) = (x*y + 1)^d$ where $d$ is the degree of polynomial and it is specified by the `kernel degree` parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net $\tanh(\alpha x*y + b)$ where $\alpha$ is `alpha` and $b$ is the `intercept constant`. These parameters can be adjusted using the `kernel a` and `kernel b` parameters. A common value for `alpha` is $1/N$, where $N$ is the data dimension. Note that not all choices of $\alpha$ and $b$ lead to a valid kernel function.

- **anova** The anova kernel is defined by raised to power $d$ of summation of $\exp(-g (x-y))$ where $g$ is `gamma` and $d$ is `degree`. gamma and degree are adjusted by the `kernel gamma` and `kernel degree` parameters respectively.

- **epachnenikov** The epachnenikov kernel is this function $(3/4)(1-u^2)$ for $u$ between -1 and 1 and zero for $u$ outside that range. It has two adjustable parameters `kernel sigma1` and `kernel degree`.

- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters `kernel sigma1`, `kernel sigma2` and `kernel sigma3`.

- **multiquadric** The multiquadric kernel is defined by the square root of $|x-y|^2 + c^2$. It has adjustable parameters `kernel sigma1` and `kernel sigma shift`.

**kernel gamma (real)** This is the SVM kernel parameter gamma. This is available only when the `kernel type` parameter is set to radial or anova.

**kernel sigma1 (real)** This is the SVM kernel parameter sigma1. This is available only when the `kernel type` parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2 (real)** This is the SVM kernel parameter sigma2. This is available only when the `kernel type` parameter is set to gaussian combination.

**kernel sigma3 (real)** This is the SVM kernel parameter sigma3. This is available only when the `kernel type` parameter is set to gaussian combination.
7.1. Classification and Regression

**kernel shift** *(real)* This is the SVM kernel parameter shift. This is available only when the *kernel type* parameter is set to *multiquadric.*

**kernel degree** *(real)* This is the SVM kernel parameter degree. This is available only when the *kernel type* parameter is set to *polynomial, anova* or *epachnenikov.*

**kernel a** *(real)* This is the SVM kernel parameter a. This is available only when the *kernel type* parameter is set to *neural.*

**kernel b** *(real)* This is the SVM kernel parameter b. This is available only when the *kernel type* parameter is set to *neural.*

**C** *(real)* This is the SVM complexity constant which sets the tolerance for mis-classification, where higher C values allow for 'softer' boundaries and lower values create 'harder' boundaries. A complexity constant that is too large can lead to over-fitting, while values that are too small may result in over-generalization.

**max evaluation** *(integer)* This is an optimizer parameter. It specifies to stop evaluations after the specified number of evaluations.

**generations without improvement** *(integer)* This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**population size** *(integer)* This parameter specifies the population size i.e. the number of individuals per generation.

**inertia weight** *(real)* This parameter specifies the (initial) weight for the old weighting.

**local best weight** *(real)* This parameter specifies the weight for the individual's best position during run.

**global best weight** *(real)* This parameter specifies the weight for the population's best position during run.

**dynamic inertia weight** *(boolean)* This parameter specifies if the inertia weight should be improved during run.

**use local random seed** *(boolean)* This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.
7. Modeling

Tutorial Processes

Introduction to the SVM (PSO) operator

The 'Ripley-Set' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The SVM (PSO) operator is applied in the training subprocess of the Split Validation operator. The SVM (PSO) operator is applied with default values of all parameters. The Apply Model operator is used in the testing subprocess for applying the model generated by the SVM (PSO) operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector are connected to the output and they can be seen in the Results Workspace. The accuracy of this model turns out to be around 85%.

Default values were used for most of the parameters. To get more reliable results these values should be carefully selected. Usually techniques like cross-validation are used to find the best values of these parameters for the ExampleSet under consideration.

Linear Discriminant Analysis
This operator performs linear discriminant analysis (LDA). This method tries to find the linear combination of features which best separate two or more classes of examples. The resulting combination is then used as a linear classifier. Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups, it may have a descriptive or a predictive objective.

Description

This operator performs linear discriminant analysis (LDA). This method tries to find the linear combination of features which best separates two or more classes of examples. The resulting combination is then used as a linear classifier. LDA is closely related to ANOVA (analysis of variance) and regression analysis, which also attempt to express one dependent variable as a linear combination of other features or measurements. In the other two methods however, the dependent variable is a numerical quantity, while for LDA it is a categorical variable (i.e. the class label). LDA is also closely related to principal component analysis (PCA) and factor analysis in that both look for linear combinations of variables which best explain the data. LDA explicitly attempts to model the difference between the classes of data. PCA on the other hand does not take into account any difference in class.

Discriminant analysis is used to determine which variables discriminate between two or more naturally occurring groups. For example, an educational researcher may want to investigate which variables discriminate between high school graduates who decide (1) to go to college, (2) NOT to go to college. For that purpose the researcher could collect data on numerous variables prior to students’ graduation. After graduation, most students will naturally fall into one of the two categories. Discriminant Analysis could then be used to determine which variable(s) are the best predictors of students’ subsequent educational choice. Computationally, discriminant function analysis is very similar to analysis of variance (ANOVA). For example, suppose the same student graduation scenario. We could have measured students' stated intention to continue on to college one year prior to graduation. If the means for the two groups (those who actually went to college and those
7. Modeling

who did not) are different, then we can say that intention to attend college as stated one year prior to graduation allows us to discriminate between those who are and are not college bound (and this information may be used by career counselors to provide the appropriate guidance to the respective students). The basic idea underlying discriminant analysis is to determine whether groups differ with regard to the mean of a variable, and then to use that variable to predict group membership (e.g., of new cases).

Discriminant Analysis may be used for two objectives: either we want to assess the adequacy of classification, given the group memberships of the objects under study; or we wish to assign objects to one of a number of (known) groups of objects. Discriminant Analysis may thus have a descriptive or a predictive objective. In both cases, some group assignments must be known before carrying out the Discriminant Analysis. Such group assignments, or labeling, may be arrived at in any way. Hence Discriminant Analysis can be employed as a useful complement to Cluster Analysis (in order to judge the results of the latter) or Principal Components Analysis.

Differentiation

Quadratic Discriminant Analysis The QDA performs a quadratic discriminant analysis (QDA). QDA is closely related to linear discriminant analysis (LDA), where it is assumed that the measurements are normally distributed. Unlike LDA however, in QDA there is no assumption that the covariance of each of the classes is identical. See page ?? for details.

Regularized Discriminant Analysis The RDA regularized discriminant analysis (RDA) which is a generalization of the LDA and QDA. Both algorithms are special cases of this algorithm. If the alpha parameter is set to 1, RDA operator performs LDA. Similarly if the alpha parameter is set to 0, RDA operator performs QDA. See page ?? for details.
7.1. Classification and Regression

Input Ports

**training set (tra)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

**model (mod)** The Discriminant Analysis is performed and the resultant model is delivered from this output port

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Related Documents

Quadratic Discriminant Analysis (??)

Regularized Discriminant Analysis (??)

Tutorial Processes

Introduction to the LDA operator

The 'Sonar' data set is loaded using the Retrieve operator. A **breakpoint** is inserted here so that you can have a look at this ExampleSet. The Linear Discriminant Analysis operator is applied on this ExampleSet. The Linear Discriminant Analysis operator performs the discriminant analysis and the resultant model can
7. Modeling

be seen in the Results Workspace.

![Diagram of Vote operator](image)

**Vote**

This operator uses a majority vote (for classification) or the average (for regression) on top of the predictions of the inner learners (i.e. learning operators in its subprocess).

**Description**

The Vote operator is a nested operator i.e. it has a subprocess. The subprocess must have at least two learners, called base learners. This operator builds a classification model or regression model depending upon the ExampleSet and learners. This operator uses a majority vote (for classification) or the average (for regression) on top of the predictions of the base learners provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses. All the operator chains in the subprocess must accept an ExampleSet and return a model.

In case of a classification task, all the operators in the subprocess of the Vote operator accept the given ExampleSet and generate a classification model. For prediction of an unknown example, the Vote operator applies all the classification
models from its subprocess and assigns the predicted class with maximum votes to the unknown example. Similarly, In case of a regression task, all the operators in the subprocess of the Vote operator accept the given ExampleSet and generate a regression model. For prediction of an unknown example, the Vote operator applies all the regression models from its subprocess and assigns the average of all predicted values to the unknown example.

**Input Ports**

**training set** *(tra)* This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**model** *(mod)* The simple vote model for classification or regression is delivered from this output port. This model can now be applied on unseen data sets for prediction of the *label* attribute.

**Tutorial Processes**

**Using the Vote operator for classification**

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a model. The Vote operator is applied in the training subprocess of the Split Validation operator. Three learners are applied in the subprocess of the Vote operator. These base learners are: Decision Tree, Neural Net and SVM. The Vote operator uses the vote of each learner for classification of an example, the prediction with maximum votes is
assigned to the unknown example. In other words it uses the predictions of the three base learners to make a combined prediction (using simple voting). The Apply Model operator is used in the testing subprocess of the Split Validation operator for applying the model generated by the Vote operator. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The Vote model and its performance vector is connected to the output and it can be seen in the Results Workspace.

Polynomial by Binomial Classification

This operator builds a polynomial classification model through the given binomial classification learner.

Description

The Polynomial by Binomial Classification operator is a nested operator i.e. it has a subprocess. The subprocess must have a binomial classification learner i.e. an operator that generates a binomial classification model. This operator builds a polynomial classification model using the binomial classification learner.
7.1. Classification and Regression

provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Many classification operators (e.g. the SVM operator) allow for classification only for binomial (binary) label. The Polynomial by Binomial Classification operator uses a binomial classifier and generates binomial classification models for different classes and then aggregates the responses of these binomial classification models for classification of polynomial label.

**Input Ports**

**training set** (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**model** (mod) The polynomial classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the label attribute.

**example set** (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**classification strategies** (selection) This parameter specifies the strategy that should be used for multi-class classifications i.e. polynomial classifications.

**random code multiplicator** (real) This parameter is only available when the
7. Modeling

classification strategies parameter is set to 'exhaustive code' or 'random code'. This parameter specifies a multiplicator regulating the codeword length in random code modus.

**use local random seed** *(boolean)* This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same randomization.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

---

**Tutorial Processes**

**Using the SVM operator for polynomial classification**

The 'Iris' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a polynomial classification model. The Polynomial by Binomial Classification operator is applied in the training subprocess of the Split Validation operator. The SVM operator is applied in the subprocess of the Polynomial by Binomial Classification operator. Although SVM is a binomial classification learner but it will be used by the Polynomial by Binomial Classification operator to train a polynomial classification model. The Apply Model operator is used in the testing subprocess to apply the model. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The polynomial classification model and its performance vector is connected to the output and it can be seen in the Results Workspace.
Classification by Regression

This operator builds a polynomial classification model through the given regression learner.

Description

The Classification by Regression operator is a nested operator i.e. it has a subprocess. The subprocess must have a regression learner i.e. an operator that generates a regression model. This operator builds a classification model using the regression learner provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

Here is an explanation of how a classification model is built from a regression learner. For each class $i$ of the given ExampleSet, a regression model is trained after setting the label to +1 if the label is $i$ and to -1 if it is not. Then the regression models are combined into a classification model. This model can be applied using the Apply Model operator. In order to determine the prediction for an unlabeled example, all regression models are applied and the class belonging to the regression model which predicts the greatest value is chosen.
Input Ports

training set (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

model (mod) The classification model is delivered from this output port. This classification model can now be applied on unseen data sets for prediction of the label attribute.

tutorial set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Tutorial Processes

Using the Linear Regression operator for classification

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Classification by Regression operator is applied in the training subprocess of the Split Validation operator. The Linear Regression operator is applied in the subprocess of the Classification by Regression operator. Although Linear Regression is a regression learner but it will be used by the Classification by Regression operator to train a classification model. The Apply Model operator is used in the testing subprocess to apply the model. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector is connected to
Bayesian Boosting

This operator is a boosting operator based on Bayes' theorem. It implements a meta-algorithm which can be used in conjunction with many other learning algorithms to improve their performance.

Description

The Bayesian Boosting operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

This operator trains an ensemble of classifiers for boolean target attributes. In each iteration the training set is reweighted, so that previously discovered patterns and other kinds of prior knowledge are 'sampled out'. An inner classifier, typically a rule or decision tree induction algorithm, is sequentially applied sev-
7. Modeling

eral times, and the models are combined to a single global model. The maximum number of models to be trained are specified by the \textit{iterations} parameter.

If the \textit{rescale label priors} parameter is set to true, then the ExampleSet is reweighted, so that all classes are equally probable (or frequent). For two-class problems this turns the problem of fitting models to maximize weighted relative accuracy into the more common task of classifier induction. Applying a rule induction algorithm as an inner learner allows to do subgroup discovery. This option is also recommended for data sets with class skew, if a very weak learner like a decision stump is used. If the \textit{rescale label priors} parameter is not set, then the operator performs boosting based on probability estimates.

If the \textit{allow marginal skews} parameter is not set, then the support of each subset defined in terms of common base model predictions does not change from one iteration to the next. Analogously the class priors do not change. This is the procedure originally described in 'Scholz/2005b' in the context of subgroup discovery. Setting the \textit{allow marginal skews} option to true leads to a procedure that changes the marginal weights/probabilities of subsets, if this is beneficial in a boosting context, and stratifies the two classes to be equally likely. As for AdaBoost, the total weight upper-bounds the training error in this case. This bound is reduced more quickly by the Bayesian Boosting operator.

To reproduce the sequential sampling, or knowledge-based sampling, from 'Scholz/2005b' for subgroup discovery, two of the default parameter settings of this operator have to be changed: \textit{rescale label priors} must be set to true, and \textit{allow marginal skews} must be set to false. In addition, a boolean (binomial) label has to be used.

This operator requires an ExampleSet as its input. To sample out prior knowledge of a different form it is possible to provide another model as an optional additional input. The predictions of this model are used to produce an initial weighting of the training set. The output of the operator is a classification model applicable for estimating conditional class probabilities or for plain crisp classification. It contains up to the specified number of inner base models. In the case of an optional initial model, this model will also be stored in the output model, in order to produce the same initial weighting during model application.
7.1. Classification and Regression

Ensemble Theory

Boosting is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

Input Ports

training set (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other
Modeling

operators can also be used as input.

**model (mod)** The input port expects a model. This is an optional port. To sample out prior knowledge of a different form it is possible to provide a model as an optional input. The predictions of this model are used to produce an initial weighting of the training set. The output of the operator is a classification model applicable for estimating conditional class probabilities or for plain crisp classification. It contains up to the specified number of inner base models. In the case of an optional initial model, this model will also be stored in the output model, in order to produce the same initial weighting during model application.

**Output Ports**

**model (mod)** The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the label attribute.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**use subset for training (real)** This parameter specifies the fraction of examples to be used for training, remaining examples are used to estimate the confusion matrix. If set to 1, the test set is turned off.

**iterations (integer)** This parameter specifies the maximum number of iterations of this algorithm.

**rescale label priors (boolean)** This parameter specifies whether the proportion of labels should be equal by construction after first iteration. Please study the description of this operator for more information about this parameter.

**allow marginal skews (boolean)** This parameter specifies if the skewing of the marginal distribution \(P(x)\) should be allowed during learning. Please study the description of this operator for more information about this parameter.
7.1. Classification and Regression

**use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of values.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

---

**Tutorial Processes**

**Using the Bayesian Boosting operator for generating a better Decision Tree**

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Bayesian Boosting operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the Bayesian Boosting operator. The *iterations* parameter of the Bayesian Boosting operator is set to 10, thus there will be at maximum 10 iterations of its subprocess. The Apply Model operator is used in the testing subprocess for applying the model generated by the Bayesian Boosting operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector is connected to the output and it can be seen in the Results Workspace. You can see that the Bayesian Boosting operator produced a new model in each iteration. The accuracy of this model turns out to be around 67.74%. If the same process is repeated without Bayesian Boosting operator i.e. only the Decision Tree operator is used in training subprocess. The accuracy of that model turns out to be around 66%. Thus Bayesian Boosting generated a combination of models that performed better than the original model.
7. Modeling

AdaBoost

This operator is an implementation of the AdaBoost algorithm and it can be used with all learners available in RapidMiner. AdaBoost is a meta-algorithm which can be used in conjunction with many other learning algorithms to improve their performance.

Description

The AdaBoost operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

AdaBoost, short for Adaptive Boosting, is a meta-algorithm, and can be used in conjunction with many other learning algorithms to improve their performance. AdaBoost is adaptive in the sense that subsequent classifiers built are tweaked in favor of those instances misclassified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. In some problems, however, it can be less susceptible to the overfitting problem than most learning algorithms. The classifiers it uses can be weak (i.e., display a substantial error rate), but as long as their performance is not random (resulting in an error rate of 0.5 for binary classification), they will
improve the final model.

AdaBoost generates and calls a new weak classifier in each of a series of rounds $t = 1, \ldots, T$. For each call, a distribution of weights $D(t)$ is updated that indicates the importance of examples in the data set for the classification. On each round, the weights of each incorrectly classified example are increased, and the weights of each correctly classified example are decreased, so the new classifier focuses on the examples which have so far eluded correct classification.

**Ensemble Theory**

Boosting is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be
more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

**Input Ports**

**training set** (*tra*) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**model** (*mod*) The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set** (*exa*) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**iterations** (*integer*) This parameter specifies the maximum number of iterations of the AdaBoost algorithm.

**Tutorial Processes**

Using the AdaBoost operator for generating a better Decision Tree
The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The AdaBoost operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the AdaBoost operator. The iterations parameter of the AdaBoost operator is set to 10, thus there will be at maximum 10 iterations of its subprocess. The Apply Model operator is used in the testing subprocess for applying the model generated by the AdaBoost operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector is connected to the output and it can be seen in the Results Workspace. You can see that the AdaBoost operator produced a new model in each iteration and there are different weights for each model. The accuracy of this model turns out to be around 69%. If the same process is repeated without AdaBoost operator i.e. only the Decision Tree operator is used in training subprocess. The accuracy of that model turns out to be around 66%. Thus AdaBoost generated a combination of models that performed better than the original model.

Bagging

Bootstrap aggregating (bagging) is a machine learning ensemble meta-algorithm to improve classification and regression models in terms of stability and classification accuracy. It also reduces variance and helps to avoid overfitting. Although it is usually applied to decision tree
models, it can be used with any type of model.

Description

The Bagging operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

The concept of bagging (voting for classification, averaging for regression-type problems with continuous dependent variables of interest) applies to the area of predictive data mining, to combine the predicted classifications (prediction) from multiple models, or from the same type of model for different learning data. It is also used to address the inherent instability of results when applying complex models to relatively small data sets. Suppose your data mining task is to build a model for predictive classification, and the dataset from which to train the model (learning data set, which contains observed classifications) is relatively small. You could repeatedly sub-sample (with replacement) from the dataset, and apply, for example, a tree classifier (e.g., CHAID) to the successive samples. In practice, very different trees will often be grown for the different samples, illustrating the instability of models often evident with small data sets. One method of deriving a single prediction (for new observations) is to use all trees found in the different samples, and to apply some simple voting: The final classification is the one most often predicted by the different trees. Note that some weighted combination of predictions (weighted vote, weighted average) is also possible, and commonly used. A sophisticated algorithm for generating weights for weighted prediction or voting is the Boosting procedure which is available in RapidMiner as AdaBoost operator.

Ensemble Theory

Bagging is an ensemble method, therefore an overview of the Ensemble The-
ory has been discussed here. Ensemble methods use multiple models to obtain better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

Input Ports

training set \( (\text{tra}) \) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.
Output Ports

**model**  
The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the *label* attribute.

**example set**  
The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**sample ratio** *(real)*  
This parameter specifies the fraction of examples to be used for training. Its value must be greater than 0 (i.e. zero examples) and should be lower than or equal to 1 (i.e. entire data set).

**iterations** *(integer)*  
This parameter specifies the maximum number of iterations of the Bagging algorithm.

**average confidences** *(boolean)*  
This parameter specifies whether to average available prediction confidences or not.

**use local random seed** *(boolean)*  
This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of values.

**local random seed** *(integer)*  
This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

Tutorial Processes

Using the Bagging operator for generating a better Decision Tree
The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Bagging operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the Bagging operator. The iterations parameter of the Bagging operator is set to 10, thus there will be 10 iterations of its subprocess. The Apply Model operator is used in the testing subprocess for applying the model generated by the Bagging operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector are connected to the output and they can be seen in the Results Workspace. You can see that the Bagging operator produced a new model in each iteration. The accuracy of this model turns out to be around 75.81%. If the same process is repeated without Bagging operator i.e. only Decision Tree operator is used in training subprocess then the accuracy of that model turns out to be around 66%. Thus Bagging improved the performance of the base learner (i.e. Decision Tree).

Stacking

This operator is an implementation of Stacking which is used for combining the models rather than choosing among them, thereby typically getting a performance better than any single one of the trained models.
7. Modeling

Description

Stacked generalization (or stacking) is a way of combining multiple models, that introduces the concept of a meta learner. Unlike bagging and boosting, stacking may be (and normally is) used to combine models of different types. The procedure is as follows:

1. Split the training set into two disjoint sets.
2. Train several base learners on the first part.
3. Test the base learners on the second part.
4. Using the predictions from step 3 as the inputs, and the correct responses as the outputs, train a higher level learner.

Note that steps 1 to 3 are the same as cross-validation, but instead of using a winner-takes-all approach, we combine the base learners, possibly nonlinearly.

The crucial prior belief underlying the scientific method is that one can judge among a set of models by comparing them on data that was not used to create any of them. This prior belief is used in the cross-validation technique, to choose among a set of models based on a single data set. This is done by partitioning the data set into a training data set and a testing data set; training the models on the training data; and then choosing whichever of those trained models performs best on the testing data.

Stacking exploits this prior belief further. It does this by using performance on the testing data to combine the models rather than choose among them, thereby typically getting a better performance than any single one of the trained models. It has been successfully used on both supervised learning tasks (e.g. regression) and unsupervised learning (e.g. density estimation).

The Stacking operator is a nested operator. It has two subprocess: the Base Learners and the Stacking Model Learner subprocess. You need to have a basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.
7.1. Classification and Regression

Ensemble Theory

Stacking is an ensemble method, therefore an overview of the Ensemble Theory has been discussed here. Ensemble methods use multiple models to obtain a better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners in an attempt to produce a strong learner. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to over-fit the training data more than a single model would, but in practice, some ensemble techniques (especially bagging) tend to reduce problems related to over-fitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.
7. Modeling

Input Ports

training set (tra) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

model (mod) The stacking model is delivered from this output port which can be applied on unseen data sets.

Parameters

keep all attributes (boolean) This parameter indicates if all attributes (including the original ones) should be kept in order to learn the stacked model.

Tutorial Processes

Introduction to Stacking

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a model. The Stacking operator is applied in the training subprocess of the Split Validation operator. Three learners are applied in the Base Learner subprocess of the Stacking operator. These learners are: Decision Tree, K-NN and Linear Regression operators. The Naive Bayes operator is applied in the Stacking Model Learner subprocess of the Stacking operator. The Naive Bayes learner is used as a stacking learner which uses the predictions of the preceding three learners to make a combined
7.1. Classification and Regression

prediction. The Apply Model operator is used in the testing subprocess of the Split Validation operator for applying the model generated by the Stacking operator. The resultant labeled ExampleSet is used by the Performance operator for measuring the performance of the model. The stacking model and its performance vector is connected to the output and it can be seen in the Results Workspace.

MetaCost

This metaclassifier makes its base classifier cost-sensitive by using the given cost matrix to compute label predictions according to classification costs.

Description

The MetaCost operator makes its base classifier cost-sensitive by using the cost matrix specified in the cost matrix parameter. The method used by this operator is similar to the MetaCost method described by Pedro Domingos (1999).

The MetaCost operator is a nested operator i.e. it has a subprocess. The subprocess must have a learner i.e. an operator that expects an ExampleSet and generates a model. This operator tries to build a better model using the learner provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the
Subprocess operator for basic understanding of subprocesses.

Most classification algorithms assume that all errors have the same cost, which is seldom the case. For example, in database marketing the cost of mailing to a non-respondent is very small, but the cost of not mailing to someone who would respond is the entire profit lost. In general, misclassification costs may be described by an arbitrary cost matrix $C$, with $C(i,j)$ being the cost of predicting that an example belongs to class $i$ when in fact it belongs to class $j$. Individually making each classification learner cost-sensitive is laborious, and often non-trivial. MetaCost is a principled method for making an arbitrary classifier cost-sensitive by wrapping a cost-minimizing procedure around it. This procedure treats the underlying classifier as a black box, requiring no knowledge of its functioning or change to it. MetaCost is applicable to any number of classes and to arbitrary cost matrices.

**Input Ports**

**training set (tra)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**model (mod)** The meta model is delivered from this output port which can now be applied on unseen data sets for prediction of the label attribute.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

**cost matrix** *(string)* This parameter is used for specifying the cost matrix. The cost matrix is similar in structure to a confusion matrix because it has predicted classes in one dimension and actual classes on the other dimension. Therefore the cost matrix can denote the costs for every possible classification outcome: predicted label vs. actual label. Actually this matrix is a matrix of misclassification costs because you can specify different weights for certain classes misclassified as other classes. Weights can also be assigned to correct classifications but usually they are set to 0. The classes in the matrix are labeled as Class 1, Class 2 etc where classes are numbered according to their order in the internal mapping.

**use subset for training** *(real)* This parameter specifies the fraction of examples to be used for training. Its value must be greater than 0 (i.e. zero examples) and should be lower than or equal to 1 (i.e. entire data set).

**iterations** *(integer)* This parameter specifies the maximum number of iterations of the MetaCost algorithm.

**sampling with replacement** *(boolean)* This parameter indicates if sampling with replacement should be used. In sampling with replacement, at every step all examples have equal probability of being selected. Once an example has been selected for the sample, it remains candidate for selection and it can be selected again in any other coming steps. Thus a sample with replacement can have the same example multiple number of times.

**use local random seed** *(boolean)* This parameter indicates if a local random seed should be used for randomization. Using the same value of local random seed will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of values.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.
7. Modeling

Tutorial Processes

Using the MetaCost operator for generating a better Decision Tree

The 'Sonar' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The MetaCost operator is applied in the training subprocess of the Split Validation operator. The Decision Tree operator is applied in the subprocess of the MetaCost operator. The iterations parameter of the MetaCost operator is set to 10, thus there will be 10 iterations of its subprocess. Have a look at the cost matrix specified in the cost matrix parameter of the MetaCost operator. You can see that the misclassification costs are not equal. The Apply Model operator is used in the testing subprocess for applying the model generated by the MetaCost operator. The resultant labeled ExampleSet is used by the Performance (Classification) operator for measuring the performance of the model. The classification model and its performance vector are connected to the output and they can be seen in the Results Workspace. You can see that the MetaCost operator produced a new model in each iteration. The accuracy of this model turns out to be around 82%. If the same process is repeated without MetaCost operator i.e. only Decision Tree operator is used in training subprocess then the accuracy of that model turns out to be around 66%. Thus MetaCost improved the performance of the base learner (i.e. Decision Tree) by using the cost matrix to compute label predictions according to classification costs.
Weight by Information Gain

This operator calculates the relevance of the attributes based on information gain and assigns weights to them accordingly.

Description

The Weight by Information Gain operator calculates the weight of attributes with respect to the class attribute by using the information gain. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be applied only on ExampleSets with nominal label.

Although information gain is usually a good measure for deciding the relevance of an attribute, it is not perfect. A notable problem occurs when information gain is applied to attributes that can take on a large number of distinct values. For example, suppose some data that describes the customers of a business. When information gain is used to decide which of the attributes are the most relevant, the customer's credit card number may have high information gain. This attribute has a high information gain, because it uniquely identifies each customer, but we may not want to assign high weights to such attributes.

Information gain ratio is sometimes used instead. This method biases against considering attributes with a large number of distinct values. However, attributes with very low information values then appear to receive an unfair advantage. The Weight by Information Gain Ratio operator uses information gain ratio for generating attribute weights.
7. Modeling

Input Ports

**example set** *(exa)* This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

**weights** *(wei)* This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

**example set** *(exa)* The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**normalize weights** *(boolean)* This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights** *(boolean)* This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the **sort direction** parameter.

**sort direction** *(selection)* This parameter is only available when the **sort weights** parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.
7.2. Attribute Weighting

Tutorial Processes

Calculating the weights of the attributes of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Information Gain operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

Weight by Information Gain Ratio

This operator calculates the relevance of the attributes based on the information gain ratio and assigns weights to them accordingly.
7. Modeling

Description

The Weight by Information Gain Ratio operator calculates the weight of attributes with respect to the label attribute by using the information gain ratio. The higher the weight of an attribute, the more relevant it is considered. Please note that this operator can be only applied on ExampleSets with nominal label.

Information gain ratio is used because it solves the drawback of information gain. Although information gain is usually a good measure for deciding the relevance of an attribute, it is not perfect. A notable problem occurs when information gain is applied to attributes that can take on a large number of distinct values. For example, suppose some data that describes the customers of a business. When information gain is used to decide which of the attributes are the most relevant, the customer's credit card number may have high information gain. This attribute has a high information gain, because it uniquely identifies each customer, but we may not want to assign high weights to such attributes. The Weight by Information Gain operator uses information gain for generating attribute weights.

Information gain ratio is sometimes used instead of information gain. Information gain ratio biases against considering attributes with a large number of distinct values. However, attributes with very low information values then appear to receive an unfair advantage.

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.
Output Ports

weights \((\text{wei})\) This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.
exmaple set \((\text{exa})\) ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

normalize weights \((\text{boolean})\) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.
sort weights \((\text{boolean})\) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the sort direction parameter.
sort direction \((\text{selection})\) This parameter is only available when the sort weights parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

Tutorial Processes

Calculating the weights of the attributes of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Information Gain Ratio operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set
to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.

Weight by Correlation

This operator calculates the relevance of the attributes by computing the value of correlation for each attribute of the input ExampleSet with respect to the label attribute. This weighting scheme is based upon correlation and it returns the absolute or squared value of correlation as attribute weight.

Description

The Weight by Correlation operator calculates the weight of attributes with respect to the label attribute by using correlation. The higher the weight of an attribute, the more relevant it is considered. Please note that the Weight by Correlation operator can be applied only on ExampleSets with numerical or binomial label. It cannot be applied on Polynominal attributes because the polynominal classes provide no information about their ordering, therefore the weights are more or less random depending on the internal numerical representation of the classes. Binominal labels work because of the representation as 0 and 1, as
A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.

Suppose we have two attributes X and Y, with means $X'$ and $Y'$ and standard deviations $S(X)$ and $S(Y)$ respectively. The correlation is computed as summation from 1 to n of the product $(X(i)-X')(Y(i)-Y')$ and then dividing this summation by the product $(n-1).S(X).S(Y)$ where $n$ is the total number of examples and $i$ is the increment variable of summation. There can be other formulas and definitions but let us stick to this one for simplicity.

As discussed earlier a positive value for the correlation implies a positive association. Suppose that an X value was above average, and that the associated Y value was also above average. Then the product $(X(i)-X')(Y(i)-Y')$ would be the product of two positive numbers which would be positive. If the X value and the Y value were both below average, then the product above would be of two negative numbers, which would also be positive. Therefore, a positive correlation is evidence of a general tendency that large values of X are associated with large values of Y and small values of X are associated with small values of Y.

As discussed earlier a negative value for the correlation implies a negative or inverse association. Suppose that an X value was above average, and that the associated Y value was instead below average. Then the product $(X(i)-X')(Y(i)-Y')$ would be the product of a positive and a negative number which would make the product negative. If the X value was below average and the Y value was above average, then the product above would also be negative. Therefore, a negative correlation is evidence of a general tendency that large values of X are associated with small values of Y and small values of X are associated with large values of Y.
7. Modeling

Input Ports

data port (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

weights (wei) This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

normalize weights (boolean) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

sort weights (boolean) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the sort direction parameter.

sort direction (selection) This parameter is only available when the sort weights parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

squared correlation (boolean) This parameter indicates if the squared correlation should be calculated instead of simple correlation. If set to true, the attribute weights are calculated as squares of correlations instead of simple correlations.
7.2. Attribute Weighting

Tutorial Processes

Calculating the attribute weights of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Weight by Correlation operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace. Now set the squared correlation parameter to true and run the process again. You will see that these weights are the squares of the previous weights.

Weight by Chi Squared Statistic

This operator calculates the relevance of the attributes by computing for each attribute of the input ExampleSet the value of the chi-squared statistic with respect to the class attribute.
7. Modeling

Description

The Weight by Chi Squared Statistic operator calculates the weight of attributes with respect to the class attribute by using the chi-squared statistic. The higher the weight of an attribute, the more relevant it is considered. Please note that the chi-squared statistic can only be calculated for nominal labels. Thus this operator can be applied only on ExampleSets with nominal label.

The chi-square statistic is a nonparametric statistical technique used to determine if a distribution of observed frequencies differs from the theoretical expected frequencies. Chi-square statistics use nominal data, thus instead of using means and variances, this test uses frequencies. The value of the chi-square statistic is given by

\[ X^2 = \sum \left( \frac{(O-E)^2}{E} \right) \]

where \( X^2 \) is the chi-square statistic, \( O \) is the observed frequency and \( E \) is the expected frequency. Generally the chi-squared statistic summarizes the discrepancies between the expected number of times each outcome occurs (assuming that the model is true) and the observed number of times each outcome occurs, by summing the squares of the discrepancies, normalized by the expected numbers, over all the categories.

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

weights (wei) This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.
example set (exa) ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

normalize weights (boolean) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.
sort weights (boolean) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the sort direction parameter.
sort direction (selection) This parameter is available only when the sort weights parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.
number of bins (integer) This parameter specifies the number of bins used for discretization of numerical attributes before the chi-squared test can be performed.

Tutorial Processes

Calculating the weights of the attributes of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Chi Squared Statistic operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can
verify this by viewing the results of this process in the Results Workspace.

**Weight by Relief**

This operator calculates the relevance of the attributes by Relief. The key idea of Relief is to estimate the quality of features according to how well their values distinguish between the instances of the same and different classes that are near each other.

**Description**

Relief is considered one of the most successful algorithms for assessing the quality of features due to its simplicity and effectiveness. The key idea of Relief is to estimate the quality of features according to how well their values distinguish between the instances of the same and different classes that are near each other. Relief measures the relevance of features by sampling examples and comparing the value of the current feature for the nearest example of the same and of a different class. This version also works for multiple classes and regression data sets. The resulting weights are normalized into the interval between 0 and 1 if the `normalize weights` parameter is set to true.
7.2. Attribute Weighting

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

weights (wei) This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.
example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

normalize weights (boolean) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.
sort weights (boolean) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the sort direction parameter.
sort direction (selection) This parameter is only available when the sort weights parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.
number of neighbors (integer) This parameter specifies the number of nearest neighbors for relevance calculation.
sample ratio (real) This parameter specifies the ratio of examples to be used for determining the weights.
use local random seed (boolean) This parameter indicates if a local random seed should be used for randomizing examples of a subset. Using the same value
7. Modeling

of the *local random seed* will produce the same sample. Changing the value of this parameter changes the way examples are randomized, thus the sample will have a different set of examples.

**local random seed (integer)** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

**Tutorial Processes**

**Calculating the attribute weights of the Polynomial data set**

The 'Polynomial' data set is loaded using the Retrieve operator. The Weight by Relief operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The *normalize weights* parameter is set to true, thus all the weights will be normalized in range 0 to 1. The *sort weights* parameter is set to true and the *sort direction* parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.
7.2. Attribute Weighting

**Weight by SVM**

This operator calculates the relevance of the attributes by computing for each attribute of the input ExampleSet the weight with respect to the class attribute. The coefficients of a hyperplane calculated by an SVM (Support Vector Machine) are set as attribute weights.

**Description**

The Weight by SVM operator uses the coefficients of the normal vector of a linear SVM as attribute weights. In contrast to most of the SVM based operators available in RapidMiner, this operator works for multiple classes too. Please note that the attribute values still have to be numerical. This operator can be applied only on ExampleSets with numerical label. Please use appropriate preprocessing operators (type conversion operators) in order to ensure this. For more information about SVM please study the documentation of the SVM operator.

**Input Ports**

**example set (exa)** This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

**Output Ports**

**weights (wei)** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant. **example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

normalize weights (boolean) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in the range from 0 to 1.

sort weights (boolean) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of sorting is specified using the sort direction parameter.

sort direction (selection) This parameter is only available when the sort weights parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

C (real) This parameter specifies the SVM complexity weighting factor.

Tutorial Processes

Calculating the weights of the attributes of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Weight by SVM operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in the range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. You can verify this by viewing the results of this process in the Results Workspace.
Weight by PCA

This operator creates attribute weights of the ExampleSet by using a component created by the PCA. This operator behaves exactly the same way as if a PCA model is given to the Weight by Component Model operator.

Description

The Weight by PCA operator generates attribute weights of the given ExampleSet using a component created by the PCA. The component is specified by the component number parameter. If the normalize weights parameter is not set to true, exact values of the selected component are used as attribute weights. The normalize weights parameter is usually set to true to spread the weights between 0 and 1. The attribute weights reflect the relevance of the attributes with respect to the class attribute. The higher the weight of an attribute, the more relevant it is considered.

Principal Component Analysis (PCA) is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated attributes into a set of values of uncorrelated attributes called principal components. The number of principal components is less than or equal to the number
of original attributes. This transformation is defined in such a way that the first principal component's variance is as high as possible (accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it should be orthogonal to (uncorrelated with) the preceding components.

**Input Ports**

**example set (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

**Output Ports**

**weights (wei)** This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant. **example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

**normalize weights (boolean)** This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

**sort weights (boolean)** This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the **sort direction** parameter.

**sort direction (selection)** This parameter is only available when the **sort weights** parameter is set to true. This parameter specifies the sorting order of the at-
tributes according to their weights.

**component number (integer)** This parameter specifies the number of the component that should be used as attribute weights.

## Tutorial Processes

### Calculating the attribute weights of the Sonar data set by PCA

The 'Sonar' data set is loaded using the Retrieve operator. The PCA operator is applied on it. The *dimensionality reduction* parameter is set to 'none'. A *breakpoint* is inserted here so that you can have a look at the components created by the PCA operator. Have a look at the *EigenVectors* generated by the PCA operator especially 'PC1' because it will be used as weights by using the Weight by Component Model operator. The Weight by Component Model operator is applied next. The *ExampleSet* and *Model* ports of the PCA operator are connected to the corresponding ports of the Weight by Component Model operator. The `normalize weights` and `sort weights` parameters are set to false, thus all the weights will be exactly the same as the selected component. The *component number* parameter is set to 1, thus 'PC1' will be used as attribute weights. The weights can be seen in the Results Workspace. You can see that these weights are exactly the same as the values of 'PC1'.

In the second operator chain the Weight by PCA operator is applied on the 'Sonar' data set to perform exactly the same task. The parameters of the Weight by PCA operator are set exactly the same as the parameters of the Weight by Component Model operator. As it can be seen in the Results Workspace, exactly same weights are generated here.
Weight by Component Model

This operator creates attribute weights of the ExampleSet by using a component created by operators like the PCA, GHA or ICA. If the model given to this operator is PCA then this operator behaves exactly as the Weight by PCA operator.

Description

The Weight by Component Model operator always comes after operators like the PCA, GHA or ICA. The ExampleSet and Preprocessing model generated by these operators is connected to the ExampleSet and Model ports of the Weight by Component Model operator. The Weight by Component Model operator then generates attribute weights of the original ExampleSet using a component created by the previous operator (i.e. PCA, GHA, ICA etc). The component is specified by the component number parameter. If the normalize weights parameter is not set to true exact values of the selected component are used as attribute weights. The normalize weights parameter is usually set to true to spread the weights between 0 and 1.
7.2. Attribute Weighting

The attribute weights reflect the relevance of the attributes with respect to the class attribute. The higher the weight of an attribute, the more relevant it is considered.

Input Ports

example set \((\text{exa})\) This input port expects an ExampleSet. It is the output of the PCA operator in the attached Example Process.
model \((\text{mod})\) This input port expects a model. Usually the Preprocessing model generated by the operators like PCA, GHA or ICA is provided here.

Output Ports

weights \((\text{wei})\) This port delivers the weights of the attributes with respect to the label attribute. The attributes with higher weight are considered more relevant.
example set \((\text{exa})\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
model \((\text{mod})\) The model that was given as input is passed without changing to the output through this port.

Parameters

normalize weights \((\text{boolean})\) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.
sort weights \((\text{boolean})\) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the sort direction parameter.
7. Modeling

**sort direction** *(selection)* This parameter is only available when the *sort weights* parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

**component number** *(integer)* This parameter specifies the number of the component that should be used as attribute weights.

---

**Tutorial Processes**

**Calculating the attribute weights of the Sonar data set by PCA**

The 'Sonar' data set is loaded using the Retrieve operator. The PCA operator is applied on it. The *dimensionality reduction* parameter is set to 'none'. A *breakpoint* is inserted here so that you can have a look at the components created by the PCA operator. Have a look at the *EigenVectors* generated by the PCA operator especially 'PC1' because it will be used as weights by using the Weight by Component Model operator. The Weight by Component Model operator is applied next. The *ExampleSet* and *Model* ports of the PCA operator are connected to the corresponding ports of the Weight by Component Model operator. The *normalize weights* and *sort weights* parameters are set to false, thus all the weights will be exactly the same as the selected component. The *component number* parameter is set to 1, thus 'PC1' will be used as attribute weights. The weights can be seen in the Results Workspace. You can see that these weights are exactly the same as the values of 'PC1'.

In the second operator chain the Weight by PCA operator is applied on the 'Sonar' data set. The parameters of the Weight by PCA operator are set exactly the same as the parameters of the Weight by Component Model operator. As it can be seen in the Results Workspace, exactly same weights are generated here.
7.2. Attribute Weighting

Data to Weights

This operator simply generates an attribute weights vector with weight 1.0 for each input attribute.

Description

The Data to Weights operator creates a new attribute weights IOObject from the given ExampleSet. The result is a vector of attribute weights containing the weight 1.0 for each attribute of the input ExampleSet.

Input Ports

example set (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.
Output Ports

weights (wei) This port delivers the weights of the attributes with respect to the label attribute.

eexample set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

normalize weights (boolean) This parameter indicates if the calculated weights should be normalized or not. If set to true, all weights are normalized in range from 0 to 1.

sort weights (boolean) This parameter indicates if the attributes should be sorted according to their weights in the results. If this parameter is set to true, the order of the sorting is specified using the sort direction parameter.

sort direction (selection) This parameter is only available when the sort weights parameter is set to true. This parameter specifies the sorting order of the attributes according to their weights.

Tutorial Processes

Generating a weight vector with weight 1.0 for all the attributes

The 'Golf' data set is loaded using the Retrieve operator. The Data to Weights operator is applied on it to generate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, the sort weights parameter is set to true and the sort direction parameter is set to
'ascending'. Run the process and see the results of this process in the Results Workspace. You can see that all attributes have been assigned to weight 1.0.

Weights to Data

This operator takes attribute weights as input and delivers an ExampleSet with attribute names and corresponding weights as output.

Description

The Weights to Data operator takes attribute weights as input and delivers an ExampleSet with attribute names and corresponding weights as output. The resultant ExampleSet has two attributes 'Attribute' and 'Weight'. The 'Attribute' attribute stores the names of attributes and the 'Weight' attribute stores the weight of the corresponding attribute. This ExampleSet has \( n \) number of examples; where \( n \) is the number of attributes in the input weights vector. There are numerous operators that provide attribute weights at 'Modeling/Attribute Weighting' in the Operators Window.
Input Ports

attribute weights (att) This input port expects weights of attributes. It is the output of the Weight by Chi Squared Statistic operator in the attached Example Process.

Output Ports

example set (exa) The ExampleSet that contains attribute names and corresponding weights is delivered through this port.

Tutorial Processes

Calculating the weights of the attributes of the Golf data set and storing them in an ExampleSet

The 'Golf' data set is loaded using the Retrieve operator. The Weight by Chi Squared Statistic operator is applied on it to calculate the weights of the attributes. All parameters are used with default values. The normalize weights parameter is set to true, thus all the weights will be normalized in range 0 to 1. The sort weights parameter is set to true and the sort direction parameter is set to 'ascending', thus the results will be in ascending order of the weights. A breakpoint is inserted here to show the weights produced by the Weight by Chi Squared Statistic operator. These weights are provided as input to the Weights to Data operator which stores these weights in form of an ExampleSet. The ExampleSet can be seen in the Results Workspace.
Optimize Weights (Evolutionary)

This operator calculates the relevance of the attributes of the given ExampleSet by using an evolutionary approach. The weights of the attributes are calculated using a Genetic Algorithm.

Description

The Optimize Weights (Evolutionary) operator is a nested operator i.e. it has a subprocess. The subprocess of the Optimize Weights (Evolutionary) operator must always return a performance vector. For more information regarding subprocesses please study the Subprocess operator. The Optimize Weights (Evolutionary) operator calculates the weights of the attributes of the given ExampleSet by using a Genetic Algorithm. The higher the weight of an attribute, the more relevant it is considered.

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems.
7. Modeling

using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

In genetic algorithm 'mutation' means switching features on and off and 'crossover' means interchanging used features. Selection is done by the specified selection scheme which is selected by the selection scheme parameter. A genetic algorithm works as follows:

Generate an initial population consisting of $p$ individuals. The number $p$ can be adjusted by the population size parameter.

For all individuals in the population

1. Perform mutation, i.e. set used attributes to unused with probability $p_m$ and vice versa. The probability $p_m$ can be adjusted by the corresponding parameters.

2. Choose two individuals from the population and perform crossover with probability $p_c$. The probability $p_c$ can be adjusted by the $p$ crossover parameter. The type of crossover can be selected by the crossover type parameter.

3. Perform selection, map all individuals according to their fitness and draw $p$ individuals at random according to their probability where $p$ is the population size which can be adjusted by the population size parameter.

4. As long as the fitness improves, go to step number 2.

If the ExampleSet contains value series attributes with block numbers, the whole block will be switched on and off. Exact, minimum or maximum number of attributes in combinations to be tested can be specified by the appropriate parameters. Many other options are also available for this operator. Please study the parameters section for more information.
7.2. Attribute Weighting

Input Ports

**example set in** *(exa)* This input port expects an ExampleSet. This ExampleSet is available at the first port of the nested chain (inside the subprocess) for processing in the subprocess.

**attribute weights in** *(att)* This port expects attribute weights. It is not compulsory to use this port.

**through** *(thr)* This operator can have multiple *through* ports. When one input is connected with the *through* port, another *through* port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The Object supplied at the first *through* port of this operator is available at the first *through* port of the nested chain (inside the subprocess). Do not forget to connect all inputs in correct order. Make sure that you have connected right number of ports at subprocess level.

Output Ports

**example set out** *(exa)* The genetic algorithm is applied on the input ExampleSet. The resultant ExampleSet with reduced attributes is delivered through this port.

**weights** *(wei)* The attribute weights are delivered through this port.

**performance** *(per)* This port delivers the Performance Vector for the selected attributes. A Performance Vector is a list of performance criteria values.

Parameters

**population size** *(integer)* This parameter specifies the population size i.e. the number of individuals per generation.

**maximum number of generations** *(integer)* This parameter specifies the number of generations after which the algorithm should be terminated.

**use early stopping** *(boolean)* This parameter enables early stopping. If not set
to true, always the maximum number of generations are performed.

**generations without improvement** *(integer)* This parameter is available only when the *use early stopping* parameter is set to true. This parameter specifies the stop criterion for early stopping i.e. it stops after *n* generations without improvement in the performance. *n* is specified by this parameter.

**normalize weights** *(boolean)* This parameter indicates if the final weights should be normalized. If set to true, the final weights are normalized such that the maximum weight is 1 and the minimum weight is 0.

**use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomization. Using the same value of *local random seed* will produce the same randomization.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.

**show stop dialog** *(boolean)* This parameter determines if a dialog with a *stop* button should be displayed which stops the search for the best feature space. If the search for best feature space is stopped, the best individual found till then will be returned.

**user result individual selection** *(boolean)* If this parameter is set to true, it allows the user to select the final result individual from the last population.

**show population plotter** *(boolean)* This parameter determines if the current population should be displayed in performance space.

**population criteria data file** *(filename)* This parameter specifies the path to the file in which the criteria data of the final population should be saved.

**maximal fitness** *(real)* This parameter specifies the maximal fitness. The optimization will stop if the fitness reaches this value.

**selection scheme** *(selection)* This parameter specifies the selection scheme of this evolutionary algorithms.

**tournament size** *(real)* This parameter is available only when the *selection scheme* parameter is set to 'tournament'. It specifies the fraction of the current population which should be used as tournament members.

**start temperature** *(real)* This parameter is available only when the *selection scheme* parameter is set to 'Boltzmann'. It specifies the scaling temperature.

**dynamic selection pressure** *(boolean)* This parameter is available only when
7.2. Attribute Weighting

the *selection scheme* parameter is set to 'Boltzmann' or 'tournament'. If set to true the selection pressure is increased to maximum during the complete optimization run.

**keep best individual (boolean)** If set to true, the best individual of each generations is guaranteed to be selected for the next generation.

**save intermediate weights (boolean)** This parameter determines if the intermediate best results should be saved.

**intermediate weights generations (integer)** This parameter is available only when the *save intermediate weights* parameter is set to true. The intermediate best results would be saved every *k* generations where *k* is specified by this parameter.

**intermediate weights file (filename)** This parameter specifies the file into which the intermediate weights should be saved.

**mutation variance (real)** This parameter specifies the (initial) variance for each mutation.

**1 5 rule (boolean)** This parameter determines if the 1/5 rule for variance adaptation should be used.

**bounded mutation (boolean)** This parameter determines if the weights should be bounded between 0 and 1. If set to true, the weights are bounded between 0 and 1.

**p crossover (real)** The probability for an individual to be selected for crossover is specified by this parameter.

**crossover type (selection)** The type of the crossover can be selected by this parameter.

**use default mutation rate (boolean)** This parameter determines if the default mutation rate should be used for nominal attributes.

**nominal mutation rate (real)** This parameter specifies the probability to switch nominal attributes between 0 and 1.

**initialize with input weights (boolean)** This parameter indicates if this operator should look for attribute weights in the given input and use the input weights of all known attributes as starting point for the optimization.
7. Modeling

Tutorial Processes

Calculating the weights of the attributes of the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has 5 regular attributes other than the label attribute. The Optimize Weights (Evolutionary) operator is applied on the ExampleSet which is a nested operator i.e. it has a subprocess. It is necessary for the subprocess to deliver a performance vector. This performance vector is used by the underlying Genetic Algorithm. Have a look at the subprocess of this operator. The Split Validation operator has been used there which itself is a nested operator. Have a look at the subprocesses of the Split Validation operator. The SVM operator is used in the 'Training' subprocess to train a model. The trained model is applied using the Apply Model operator in the 'Testing' subprocess. The performance is measured through the Performance operator and the resultant performance vector is used by the underlying algorithm. Run the process and switch to the Results Workspace. You can see that the ExampleSet that had 5 attributes has now been reduced to 2 attributes. Also take a look at the weights of the attributes in the Results Workspace. You can see that two attributes have weight 1 and the remaining attributes have weight 0.
K-Means

This operator performs clustering using the \emph{k-means} algorithm. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabelled data. \emph{k-means} clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters.

Description

This operator performs clustering using the \emph{k-means} algorithm. \emph{k-means} clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data. Clustering can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Here is a simple explanation of how the \emph{k-means} algorithm works. First of all we need to introduce the notion of the center of a cluster, generally called its centroid. Assuming that we are using Euclidean distance or something similar as a measure we can define the centroid of a cluster to be the point for which each attribute value is the average of the values of the corresponding attribute for all the points in the cluster. The centroid of a cluster will sometimes be one of the points in the cluster, but frequently it will be an imaginary point, not part of the cluster itself, which we can take as marking its center.

For this method of clustering we start by deciding how many clusters we would like to form from our data. We call this value \emph{k}. The value of \emph{k} is generally a small integer, such as 2, 3, 4 or 5, but may be larger.
7. Modeling

We next select \( k \) points. These are treated as the centroids of \( k \) clusters, or to be more precise as the centroids of \( k \) potential clusters, which at present have no members. We can select these points in any way we wish, but the method may work better if we pick \( k \) initial points that are fairly far apart. We now assign each of the points one by one to the cluster which has the nearest centroid. When all the objects have been assigned we will have \( k \) clusters based on the original \( k \) centroids but the centroids will no longer be the true centroids of the clusters. Next we recalculate the centroids of the clusters, and then repeat the previous steps, assigning each object to the cluster with the nearest centroid etc. The entire algorithm can be summarized as

1. Choose a value of \( k \).
2. Select \( k \) objects in an arbitrary fashion. Use these as the initial set of \( k \) centroids.
3. Assign each of the objects to the cluster for which it is nearest to the centroid.
4. Recalculate the centroids of the \( k \) clusters.
5. Repeat steps 3 and 4 until the centroids no longer move.

Differentiation

**k-Medoids** In case of the k-medoids algorithm the centroid of a cluster will always be one of the points in the cluster. This is the major difference between the k-means and k-medoids algorithm. In k-means algorithm the centroid of a cluster will frequently be an imaginary point, not part of the cluster itself, which we can take as marking its center. See page 810 for details.

**k-Means (Kernel)** Kernel k-means uses kernels to estimate distance between objects and clusters. Because of the nature of kernels it is necessary to sum over all elements of a cluster to calculate one distance. So this algorithm is quadratic in number of examples and does not return a Centroid Cluster Model (on the contrary the K-Means operator returns a Centroid Cluster Model). See page 805.
for details.

### Input Ports

**example set input** *(exa)* The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

### Output Ports

**cluster model** *(clu)* This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

**clustered set** *(clu)* The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

### Parameters

**add cluster attribute** *(boolean)* If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label** *(boolean)* If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role.

**remove unlabeled** *(boolean)* If set to true, unlabeled examples are deleted.

**k** *(integer)* This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have small number of clusters with examples scattered (not too scattered) around...
7. Modeling

them in a balanced way.

**max runs (integer)** This parameter specifies the maximal number of runs of k-Means with random initialization that are performed.

**max optimization steps (integer)** This parameter specifies the maximal number of iterations performed for one run of k-Means

**use local random seed (boolean)** Indicates if a *local random seed* should be used for randomization. Randomization may be used for selecting *k* different points at the start of the algorithm as potential centroids.

**local random seed (integer)** This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.

Related Documents

- k-Medoids (810)
- k-Means (Kernel) (805)

Tutorial Processes

Simple clustering of Ripley-Set data set

In many cases, no target attribute (i.e. *label*) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the *label* is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A *breakpoint* is inserted at this step so that you can have a look at the ExampleSet before application
of the K-Means operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Means operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format. You can see information regarding centroids under the Centroid Table and Centroid Plot View tabs.

K-Means (Kernel)

This operator performs clustering using the kernel k-means algorithm. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. Kernel k-means uses kernels to estimate the distance between objects and clusters. K-means is an exclusive clustering algorithm.
7. Modeling

Description

This operator performs clustering using the kernel k-means algorithm. The k-means is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them. Kernel k-means uses kernels to estimate the distance between objects and clusters. Because of the nature of kernels it is necessary to sum over all elements of a cluster to calculate one distance. So this algorithm is quadratic in number of examples and does not return a Centroid Cluster Model contrary to the K-Means operator. This operator creates a cluster attribute in the resultant ExampleSet if the add cluster attribute parameter is set to true.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data. Clustering can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Differentiation

k-Means Kernel k-means uses kernels to estimate the distance between objects and clusters. Because of the nature of kernels it is necessary to sum over all elements of a cluster to calculate one distance. So this algorithm is quadratic in number of examples and does not return a Centroid Cluster Model which does the K-Means operator. See page 801 for details.

Input Ports

example set (exa) The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other
operators can also be used as input.

### Output Ports

**cluster model** *(clu)* This port delivers the cluster model which has information regarding the clustering performed. It tells which examples are part of which cluster.

**clustered set** *(clu)* The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

### Parameters

**add cluster attribute** *(boolean)* If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.

**add as label** *(boolean)* If true, the cluster id is stored in an attribute with the *label* role instead of *cluster* role.

**remove unlabeled** *(boolean)* If set to true, unlabeled examples are deleted.

**use weights** *(boolean)* This parameter indicates if the weight attribute should be used.

**k** *(integer)* This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have small number of clusters with examples scattered (not too scattered) around them in a balanced way.

**max optimization steps** *(integer)* This parameter specifies the maximal number of iterations performed for one run of k-Means.

**use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomization.

**local random seed** *(integer)* This parameter specifies the *local random seed*. 

807
7. Modeling

This parameter is only available if the use local random seed parameter is set to true.

**kernel type (selection)** The type of the kernel function is selected through this parameter. Following kernel types are supported: *dot, radial, polynomial, neural, anova, epachnenikov, gaussian combination, multiquadric*

- **dot** The dot kernel is defined by $k(x,y)=x^*y$ i.e. it is inner product of $x$ and $y$.

- **radial** The radial kernel is defined by $\exp(-g \|x-y\|^2)$ where $g$ is the *gamma*, it is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x,y)=(x^*y+1)^d$ where $d$ is the degree of polynomial and it is specified by the *kernel degree* parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net $\tanh(a \ x^*y+b)$ where $a$ is *alpha* and $b$ is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is $1/N$, where $N$ is the data dimension. Note that not all choices of $a$ and $b$ lead to a valid kernel function.

- **anova** The anova kernel is defined by raised to power $d$ of summation of $\exp(-g \ (x-y))$ where $g$ is *gamma* and $d$ is *degree*. *gamma* and *degree* are adjusted by the *kernel gamma* and *kernel degree* parameters respectively.

- **epachnenikov** The epachnenikov kernel is this function $(3/4)(1-u^2)$ for $u$ between -1 and 1 and zero for $u$ outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.

- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1*, *kernel sigma2* and *kernel sigma3*.

- **multiquadric** The multiquadric kernel is defined by the square root of $\|x-y\|^2$.
7.3. Clustering and Segmentation

$y||^2 + c^2$. It has adjustable parameters $kernel\ sigma1$ and $kernel\ shift$.

**kernel gamma** *(real)* This is the kernel parameter gamma. This is only available when the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** *(real)* This is the kernel parameter sigma1. This is only available when the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** *(real)* This is the kernel parameter sigma2. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel sigma3** *(real)* This is the kernel parameter sigma3. This is only available when the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** *(real)* This is the kernel parameter shift. This is only available when the *kernel type* parameter is set to *multiquadric*.

**kernel degree** *(real)* This is the kernel parameter degree. This is only available when the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** *(real)* This is the kernel parameter a. This is only available when the *kernel type* parameter is set to *neural*.

**kernel b** *(real)* This is the kernel parameter b. This is only available when the *kernel type* parameter is set to *neural*.

**Related Documents**

*k*-Means *(801)*

**Tutorial Processes**

Clustering of the Ripley-Set data set using the Kernel K-Means operator

In many cases, no target attribute (i.e. *label*) can be defined and the data should
be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the Kernel K-Means operator. Besides the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The Kernel K-Means operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the Kernel K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter $k$ was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

K-Medoids

This operator performs clustering using the $k$-medoids algorithm. Clustering is concerned with grouping objects together that are similar to
7.3. Clustering and Segmentation

each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabelled data. k-medoids clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters.

Description

This operator performs clustering using the k-medoids algorithm. K-medoids clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Here is a simple explanation of how the k-medoids algorithm works. First of all we need to introduce the notion of the center of a cluster, generally called its centroid. Assuming that we are using Euclidean distance or something similar as a measure we can define the centroid of a cluster to be the point for which each attribute value is the average of the values of the corresponding attribute for all the points in the cluster. The centroid of a cluster will always be one of the points in the cluster. This is the major difference between the k-means and k-medoids algorithm. In the k-means algorithm the centroid of a cluster will frequently be an imaginary point, not part of the cluster itself, which we can take to mark its center. For more information about the k-means algorithm please study the k-means operator.
7. Modeling

Differentiation

**k-Means** In case of the k-medoids algorithm the centroid of a cluster will always be one of the points in the cluster. This is the major difference between the k-means and k-medoids algorithm. In the k-means algorithm the centroid of a cluster will frequently be an imaginary point, not part of the cluster itself, which we can take to mark its center. See page 801 for details.

Input Ports

**example set input** *(exa)* The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

**cluster model** *(clu)* This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

**clustered set** *(clu)* The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with *id* role is added to the input ExampleSet to distinguish examples. An attribute with *cluster* role may also be added depending on the state of the *add cluster attribute* parameter.

Parameters

**add cluster attribute** *(boolean)* If enabled, a new attribute with *cluster* role is generated directly in this operator, otherwise this operator does not add the *cluster* attribute. In the latter case you have to use the Apply Model operator to generate the *cluster* attribute.
add as label (boolean) If true, the cluster id is stored in an attribute with the label role instead of cluster role

remove unlabeled (boolean) If set to true, unlabeled examples are deleted.

k (integer) This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have a small number of clusters with examples scattered (not too scattered) around them in a balanced way.

max runs (integer) This parameter specifies the maximal number of runs of k-medoids with random initialization that are performed.

max optimization steps (integer) This parameter specifies the maximal number of iterations performed for one run of k-medoids.

use local random seed (boolean) Indicates if a local random seed should be used for randomization. Randomization may be used for selecting k different points at the start of the algorithm as potential centroids.

local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

measure types (selection) This parameter is used for selecting the type of measure to be used for measuring the distance between points. The following options are available: mixed measures, nominal measures, numerical measures and Bregman divergences.

mixed measure (selection) This parameter is available when the measure type parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'.

nominal measure (selection) This parameter is available when the measure type parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

numerical measure (selection) This parameter is available when the measure type parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

divergence (selection) This parameter is available when the measure type parameter is set to 'bregman divergences'.
7. Modeling

kernel type (selection) This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by \( k(x,y) = x^T y \) i.e. it is inner product of \( x \) and \( y \).

- **radial** The radial kernel is defined by \( \exp(-g ||x-y||^2) \) where \( g \) is the gamma that is specified by the kernel gamma parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by \( k(x,y) = (x^T y + 1)^d \) where \( d \) is the degree of the polynomial and it is specified by the kernel degree parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net \( \tanh(a \cdot x^T y + b) \) where \( a \) is alpha and \( b \) is the intercept constant. These parameters can be adjusted using the kernel a and kernel b parameters. A common value for alpha is \( 1/N \), where \( N \) is the data dimension. Note that not all choices of \( a \) and \( b \) lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that the sigmoid kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has adjustable parameters gamma and degree.

- **epachnenikov** The Epanechnikov kernel is this function \( (3/4)(1-u^2) \) for \( u \) between -1 and 1 and zero for \( u \) outside that range. It has two adjustable parameters kernel sigma1 and kernel degree.

- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters kernel sigma1, kernel sigma2 and kernel sigma3.

- **multiquadric** The multiquadric kernel is defined by the square root of \( ||x-y||^2 + c^2 \). It has adjustable parameters kernel sigma1 and kernel sigma
7.3. Clustering and Segmentation

kernel gamma (real) This is the SVM kernel parameter gamma. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to radial or anova.

kernel sigma1 (real) This is the SVM kernel parameter sigma1. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to epachnenikov, gaus-sian combination or multiquadric.

kernel sigma2 (real) This is the SVM kernel parameter sigma2. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to gaussian combination.

kernel sigma3 (real) This is the SVM kernel parameter sigma3. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to gaussian combination.

kernel shift (real) This is the SVM kernel parameter shift. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to multiquadric.

kernel degree (real) This is the SVM kernel parameter degree. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to polynomial, anova or epachnenikov.

kernel a (real) This is the SVM kernel parameter a. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to neural.

kernel b (real) This is the SVM kernel parameter b. This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to neural.
7. Modeling

Related Documents

k-Means (801)

Tutorial Processes

Clustering of Ripley-Set data set by the K-Medoids operator

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the K-Medoids operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Medoids operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the K-Medoids operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Also note the Plot View of this data. You can clearly see how the algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format. You can see information regarding centroids under the Centroid
7.3. Clustering and Segmentation

Table and Centroid Plot View tabs.

DBSCAN

This operator performs clustering with DBSCAN. DBSCAN (for density-based spatial clustering of applications with noise) is a density-based clustering algorithm because it finds a number of clusters starting from the estimated density distribution of corresponding nodes.

Description

DBSCAN's definition of a cluster is based on the notion of density reachability. Basically, a point $q$ is directly density-reachable from a point $p$ if it is not farther away than a given distance epsilon (i.e. it is part of its epsilon-neighborhood) and if $p$ is surrounded by sufficiently many points such that one may consider $p$ and $q$ to be part of a cluster. $q$ is called density-reachable (note the distinction from “directly density-reachable“) from $p$ if there is a sequence $p(1), \ldots, p(n)$ of points with $p(1) = p$ and $p(n) = q$ where each $p(i+1)$ is directly density-reachable from $p(i)$.

Note that the relation of density-reachable is not symmetric. $q$ might lie on the edge of a cluster, having insufficiently many neighbors to count as dense itself. This would halt the process of finding a path that stops with the first non-dense point. By contrast, starting the process with $q$ would lead to $p$ (though the process would halt there, $p$ being the first non-dense point). Due to this asymmetry, the notion of density-connected is introduced: two points $p$ and $q$ are
density-connected if there is a point \( o \) such that both \( p \) and \( q \) are density-reachable from \( o \). Density-connectedness is symmetric.

A cluster, which is a subset of the points of the data set, satisfies two properties:

1. All points within the cluster are mutually density-connected.
2. If a point is density-connected to any point of the cluster, it is part of the cluster as well.

DBSCAN requires two parameters: \( \text{epsilon} \) and the minimum number of points required to form a cluster (\( \text{minPts} \)). \( \text{epsilon} \) and \( \text{minPts} \) can be specified through the \( \text{epsilon} \) and \( \text{min points} \) parameters respectively. DBSCAN starts with an arbitrary starting point that has not been visited. This point's \( \text{epsilon} \)-neighborhood is retrieved, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. Note that this point might later be found in a sufficiently sized \( \text{epsilon} \)-environment of a different point and hence be made part of a cluster.

If a point is found to be a dense part of a cluster, its \( \text{epsilon} \)-neighborhood is also part of that cluster. Hence, all points that are found within the \( \text{epsilon} \)-neighborhood are added, as is their own \( \text{epsilon} \)-neighborhood when they are also dense. This process continues until the density-connected cluster is completely found. Then, a new unvisited point is retrieved and processed, leading to the discovery of a further cluster or noise.

If no id attribute is present, this operator will create one. The 'Cluster 0' assigned by DBSCAN operator corresponds to points that are labeled as noise. These are the points that have less than \( \text{min points} \) points in their \( \text{epsilon} \)-neighborhood.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.
7.3. Clustering and Segmentation

Input Ports

d_{example set} (exa) This input port expects an ExampleSet. It is output of the Retrieve operator in the attached Example Process.

Output Ports

d_{cluster model} (clu) This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster.
d_{clustered set} (clu) The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with id role is added to the input ExampleSet to distinguish examples. An attribute with cluster role may also be added depending on the state of the add cluster attribute parameter.

Parameters

d_{epsilon} (real) This parameter specifies the epsilon parameter of the DBSCAN algorithm. Epsilon specifies the size of the neighborhood.
d_{min points} (integer) This parameter specifies the minimal number of points forming a cluster.
d_{add cluster attribute} (boolean) If this parameter is set to true, a new attribute with cluster role is generated in the resultant ExampleSet, otherwise this operator does not add the cluster attribute. In the latter case you have to use the Apply Model operator to generate the cluster attribute.
d_{add as label} (boolean) If this parameter is set to true, the cluster id is stored in an attribute with the label role instead of cluster role
d_{remove unlabeled} (boolean) If this parameter is set to true, unlabeled examples are deleted from the ExampleSet.
d_{measure types} (selection) This parameter is used for selecting the type of measure to be used for measuring the distance between points. The following options
7. Modeling

are available: mixed measures, nominal measures, numerical measures and Bregman divergences.

**mixed measure** *(selection)* This parameter is available when the measure type parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'

**nominal measure** *(selection)* This parameter is available when the measure type parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure** *(selection)* This parameter is available when the measure type parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence** *(selection)* This parameter is available when the measure type parameter is set to 'bregman divergences'.

**kernel type** *(selection)* This parameter is only available when the numerical measure parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by \( k(x,y) = x^*y \) i.e. it is inner product of \( x \) and \( y \).

- **radial** The radial kernel is defined by \( \exp(-g \|x-y\|^2) \) where \( g \) is the gamma that is specified by the kernel gamma parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by \( k(x,y) = (x^*y + 1)^d \) where \( d \) is the degree of the polynomial and it is specified by the kernel degree parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net \( \tanh(a x^*y + b) \) where \( a \) is alpha and \( b \) is the intercept constant. These parameters can be adjusted using the kernel a and kernel b parameters. A common value for alpha is \( 1/N \), where \( N \) is the data dimension. Note that not all
7.3. Clustering and Segmentation

choices of $a$ and $b$ lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has adjustable parameters $\gamma$ and $\text{degree}$.

- **epachnenikov** The Epanechnikov kernel is this function $(3/4)(1-u^2)$ for $u$ between -1 and 1 and zero for $u$ outside that range. It has two adjustable parameters kernel $\sigma_1$ and kernel $\text{degree}$.

- **gaussian combination** This is the gaussian combination kernel. It has adjustable parameters kernel $\sigma_1$, kernel $\sigma_2$ and kernel $\sigma_3$.

- **multiquadric** The multiquadric kernel is defined by the square root of $||x-y||^2 + c^2$. It has adjustable parameters kernel $\sigma_1$ and kernel $\text{shift}$.

**kernel gamma** *(real)* This is the SVM kernel parameter gamma. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to radial or anova.

**kernel sigma1** *(real)* This is the SVM kernel parameter $\sigma_1$. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** *(real)* This is the SVM kernel parameter $\sigma_2$. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to gaussian combination.

**kernel sigma3** *(real)* This is the SVM kernel parameter $\sigma_3$. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to gaussian combination.

**kernel shift** *(real)* This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to multiquadric.
7. Modeling

**kernel degree (real)** This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a (real)** This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b (real)** This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

## Tutorial Processes

**Clustering of Ripley-Set data set by the DBSCAN operator**

In many cases, no target attribute (i.e. *label*) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the *label* is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A *breakpoint* is inserted at this step so that you can have a look at the ExampleSet before application of the DBSCAN operator. Other than the *label* attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The DBSCAN operator is applied on this data set with default values for all parameters except the *epsilon* parameter which is set to 0.1. Run the process and you will see that two new attributes are created by the DBSCAN operator. The *id* attribute is created to distinguish examples clearly. The *cluster* attribute is created to show which cluster the examples belong to. Each example is assigned to a particular cluster. The examples in 'cluster_0' are
considered as noise. Also note the Plot View of this data set. Switch to Plot View and set the the Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'cluster'. You can clearly see how the algorithm has created three separate groups (noise i.e. cluster_0 is also visible separately). A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

![Image](image-url)

**Expectation Maximization Clustering**

This operator performs clustering using the Expectation Maximization algorithm. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. But the Expectation Maximization algorithm extends this basic approach to clustering in some important ways.

**Description**

The general purpose of clustering is to detect clusters in examples and to assign those examples to the clusters. A typical application for this type of analysis is a marketing research study in which a number of consumer behavior related variables are measured for a large sample of respondents. The purpose of the study is to detect & market segments, i.e., groups of respondents that are
7. Modeling

somehow more similar to each other (to all other members of the same cluster) when compared to respondents that belong to other clusters. In addition to identifying such clusters, it is usually equally of interest to determine how the clusters are different, i.e., determine the specific variables or dimensions that vary and how they vary in regard to members in different clusters.

The EM (expectation maximization) technique is similar to the K-Means technique. The basic operation of K-Means clustering algorithms is relatively simple: Given a fixed number of \( k \) clusters, assign observations to those clusters so that the means across clusters (for all variables) are as different from each other as possible. The EM algorithm extends this basic approach to clustering in two important ways:

- Instead of assigning examples to clusters to maximize the differences in means for continuous variables, the EM clustering algorithm computes probabilities of cluster memberships based on one or more probability distributions. The goal of the clustering algorithm then is to maximize the overall probability or likelihood of the data, given the (final) clusters.

- Unlike the classic implementation of k-means clustering, the general EM algorithm can be applied to both continuous and categorical variables (note that the classic k-means algorithm can also be modified to accommodate categorical variables).

**Expectation Maximization algorithm**

The basic approach and logic of this clustering method is as follows. Suppose you measure a single continuous variable in a large sample of observations. Further, suppose that the sample consists of two clusters of observations with different means (and perhaps different standard deviations); within each sample, the distribution of values for the continuous variable follows the normal distribution. The goal of EM clustering is to estimate the means and standard deviations for each cluster so as to maximize the likelihood of the observed data (distribution). Put another way, the EM algorithm attempts to approximate the observed distributions of values based on mixtures of different distributions in different clusters.
The results of EM clustering are different from those computed by k-means clustering. The latter will assign observations to clusters to maximize the distances between clusters. The EM algorithm does not compute actual assignments of observations to clusters, but classification probabilities. In other words, each observation belongs to each cluster with a certain probability. Of course, as a final result you can usually review an actual assignment of observations to clusters, based on the (largest) classification probability.

**Differentiation**

**k-Means** The K-Means operator performs clustering using the k-means algorithm. k-means clustering is an exclusive clustering algorithm i.e. each object is assigned to precisely one of a set of clusters. Objects in one cluster are similar to each other. The similarity between objects is based on a measure of the distance between them. The K-Means operator assigns observations to clusters to maximize the distances between clusters. The Expectation Maximization Clustering operator, on the other hand, computes classification probabilities. See page 801 for details.

**Input Ports**

**example set (exa)** The input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

**cluster model (clu)** This port delivers the cluster model which has information regarding the clustering performed. It has information about cluster probabilities and cluster means.
7. Modeling

clustered set (clu) The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with id role is added to the input ExampleSet to distinguish examples. An attribute with cluster role may also be added depending on the state of the add cluster attribute parameter. If the show probabilities parameter is set to true, one probability column is added for each cluster.

Parameters

k (integer) This parameter specifies the number of clusters to form. There is no hard and fast rule of number of clusters to form. But, generally it is preferred to have small number of clusters with examples scattered (not too scattered) around them in a balanced way.

add cluster attribute (boolean) If enabled, a new attribute with cluster role is generated directly in this operator, otherwise this operator does not add the cluster attribute. In the latter case you have to use the Apply Model operator to generate the cluster attribute.

add as label (boolean) If true, the cluster id is stored in an attribute with the label role instead of cluster role

remove unlabeled (boolean) If set to true, unlabeled examples are deleted.

max runs (integer) This parameter specifies the maximal number of runs of this operator to be performed with random initialization.

max optimization steps (integer) This parameter specifies the maximal number of iterations performed for one run of this operator.

quality (real) This parameter specifies the quality that must be fulfilled before the algorithm stops (i.e. the rising of the log-likelihood that must be undercut).

use local random seed (boolean) This parameter indicates if a local random seed should be used for randomization.

local random seed (integer) This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

show probabilities (boolean) This parameter indicates if the probabilities for every cluster should be inserted with every example in the ExampleSet.
initial distribution \textit{(selection)} This parameter indicates the initial distribution of the centroids.

correlated attributes \textit{(boolean)} This parameter should be set to true if the ExampleSet contains correlated attributes.

**Related Documents**

k-Means (801)

**Tutorial Processes**

**Clustering of the Ripley-Set data set using the Expectation Maximization Clustering operator**

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the Expectation Maximization Clustering operator. Besides the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The Expectation Maximization Clustering operator is applied on this data set with default values for all parameters. Run the process and you will see that a few new attributes are created by the Expectation Maximization Clustering operator. The \textit{id} attribute is created to distinguish examples clearly. The \textit{cluster} attribute is created to show which cluster the examples belong to. As parameter \(k\) was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Note that the Expectation Maximization Clustering operator has added probability attributes for each cluster that show the probability of an example to be part of that cluster. This operator assigns an example to the cluster with maximum probability. Also note the \textit{Plot View} of this data. You can clearly see how the
algorithm has created two separate groups in the Plot View. A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. It also has information about cluster probabilities and cluster means. Under Folder View you can see members of each cluster in folder format.

Support Vector Clustering

This operator performs clustering with support vectors. Clustering is concerned with grouping objects together that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data.

Description

This operator is an implementation of Support Vector Clustering based on Ben-Hur et al (2001). In this Support Vector Clustering (SVC) algorithm data points are mapped from data space to a high dimensional feature space using a Gaussian kernel. In feature space the smallest sphere that encloses the image of the data is searched. This sphere is mapped back to data space, where it forms a set of contours which enclose the data points. These contours are interpreted as cluster boundaries. Points enclosed by each separate contour are associated with the same cluster. As the width parameter of the Gaussian kernel is decreased, the number of disconnected contours in data space increases, leading to an increasing number of clusters. Since the contours can be interpreted as delineating the
7.3. Clustering and Segmentation

support of the underlying probability distribution, this algorithm can be viewed as one identifying valleys in this probability distribution.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Input Ports

data set (exa) This input port expects an ExampleSet. It is output of the Generate Data operator in the attached Example Process.

Output Ports

data model (clu) This port delivers the cluster model. It has information regarding the clustering performed. It tells which examples are part of which cluster.
data set (clu) The ExampleSet that was given as input is passed with minor changes to the output through this port. An attribute with id role is added to the input ExampleSet to distinguish examples. An attribute with cluster role may also be added depending on the state of the add cluster attribute parameter.

Parameters

add cluster attribute (boolean) If this parameter is set to true, a new attribute with cluster role is generated in the resultant ExampleSet, otherwise this operator does not add the cluster attribute. In the latter case you have to use the Apply Model operator to generate the cluster attribute.
add as label (boolean) If this parameter is set to true, the cluster id is stored in
an attribute with the label role instead of cluster role

**remove unlabeled (boolean)** If this parameter is set to true, unlabeled examples are deleted from the ExampleSet.

**min pts (integer)** This parameter specifies the minimal number of points in each cluster.

**kernel type (selection)** The type of the kernel function is selected through this parameter. Following kernel types are supported: dot, radial, polynomial, neural

- **dot** The dot kernel is defined by \( k(x,y) = x \cdot y \) i.e. it is inner product of \( x \) and \( y \).

- **radial** The radial kernel is defined by \( \exp(-g \|x-y\|^2) \) where \( g \) is the gamma, it is specified by the kernel gamma parameter. The adjustable parameter gamma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by \( k(x,y) = (x \cdot y + 1)^d \) where \( d \) is the degree of polynomial and it is specified by the kernel degree parameter. The polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net \( \tanh(a \cdot x \cdot y + b) \) where \( a \) is alpha and \( b \) is the intercept constant. These parameters can be adjusted using the kernel a and kernel b parameters. A common value for alpha is \( 1/N \), where \( N \) is the data dimension. Note that not all choices of \( a \) and \( b \) lead to a valid kernel function.

**kernel gamma (real)** This is the SVM kernel parameter gamma. This is available only when the kernel type parameter is set to radial.

**kernel degree (real)** This is the SVM kernel parameter degree. This is available only when the kernel type parameter is set to polynomial.

**kernel a (real)** This is the SVM kernel parameter a. This is available only when the kernel type parameter is set to neural.

**kernel b (real)** This is the SVM kernel parameter b. This is available only when the kernel type parameter is set to neural.

**kernel cache (real)** This is an expert parameter. It specifies the size of the cache...
for kernel evaluations in megabytes.

**convergence epsilon (real)** This is an optimizer parameter. It specifies the precision on the KKT conditions.

**max iterations (integer)** This is an optimizer parameter. It specifies to stop iterations after a specified number of iterations.

**p (real)** This parameter specifies the fraction of allowed outliers.

**r (real)** If this parameter is set to -1 then the the calculated radius is used as radius. Otherwise the value specified in this parameter is used as radius.

**number sample points (real)** This parameter specifies the number of virtual sample points to check for neighborhood.

## Tutorial Processes

### Clustering of Ripley-Set data set by the Support Vector Clustering operator

In many cases, no target attribute (i.e. *label*) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the Generate Data operator is used for generating an ExampleSet. Note that the *label* is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A **breakpoint** is inserted at this step so that you can have a look at the ExampleSet before application of the clustering operator. Other than the *label* attribute the ExampleSet has two real attributes; 'att1' and 'att2'. The Support Vector Clustering operator is applied on this data set. Run the process and you will see that two new attributes are created by the Support Vector Clustering operator. The *id* attribute is created to distinguish examples clearly. The *cluster* attribute is created to show which cluster the examples belong to. Each example is assigned to a particular cluster.
7. Modeling

The examples that are not in any cluster are considered as noise. Also note the Plot View of this data set. Switch to Plot View and set the the Plotter to 'Scatter', x-Axis to 'att1', y-Axis to 'att2' and Color Column to 'cluster'. You can clearly see how the algorithm has created three separate cluster (noise is also visible separately). A cluster model is also delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format.

Agglomerative Clustering

This operator performs Agglomerative clustering which is a bottom-up strategy of Hierarchical clustering. Three different strategies are supported by this operator: single-link, complete-link and average-link. The result of this operator is an hierarchical cluster model, providing distance information to plot as a dendrogram.

Description

Agglomerative clustering is a strategy of hierarchical clustering. Hierarchical clustering (also known as Connectivity based clustering) is a method of cluster analysis which seeks to build a hierarchy of clusters. Hierarchical clustering, is based on the core idea of objects being more related to nearby objects than to objects farther away. As such, these algorithms connect 'objects' (or examples, in case of an ExampleSet) to form clusters based on their distance. A cluster can be described largely by the maximum distance needed to connect parts of the cluster.
At different distances, different clusters will form, which can be represented using a dendrogram, which explains where the common name 'hierarchical clustering' comes from: these algorithms do not provide a single partitioning of the data set, but instead provide an extensive hierarchy of clusters that merge with each other at certain distances. In a dendrogram, the y-axis marks the distance at which the clusters merge, while the objects are placed along the x-axis so the clusters don't mix.

Strategies for hierarchical clustering generally fall into two types:

- **Agglomerative:** This is a bottom-up approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.
- **Divisive:** This is a top-down approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

Hierarchical clustering is a whole family of methods that differ by the way distances are computed. Apart from the usual choice of distance functions, the user also needs to decide on the linkage criterion to use, since a cluster consists of multiple objects, there are multiple candidates to compute the distance to. Popular choices are known as single-linkage clustering (the minimum of object distances), complete-linkage clustering (the maximum of object distances) or average-linkage clustering (also known as UPGMA, 'Unweighted Pair Group Method with Arithmetic Mean').

The algorithm forms clusters in a bottom-up manner, as follows:

1. Initially, put each example in its own cluster.
2. Among all current clusters, pick the two clusters with the smallest distance.
3. Replace these two clusters with a new cluster, formed by merging the two original ones.
4. Repeat the above two steps until there is only one remaining cluster in the pool.

Clustering is concerned with grouping together objects that are similar to each
other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

**Input Ports**

**example set (exa)** This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

**Output Ports**

**cluster model (clu)** This port delivers the hierarchical cluster model. It has information regarding the clustering performed. It explains how clusters were merged to make a hierarchy of clusters.

**example set (exa)** The ExampleSet that was given as input is passed without any modification to the output through this port.

**Parameters**

**mode (selection)** This parameter specifies the cluster mode or the linkage criterion.

- **SingleLink** In single-link hierarchical clustering, we merge in each step the two clusters whose two closest members have the smallest distance (or: the two clusters with the smallest minimum pairwise distance).

- **CompleteLink** In complete-link hierarchical clustering, we merge in each step the two clusters whose merger has the smallest diameter (or: the two clusters with the smallest maximum pairwise distance).
7.3. Clustering and Segmentation

- **AverageLink** Average-link clustering is a compromise between the sensitivity of complete-link clustering to outliers and the tendency of single-link clustering to form long chains that do not correspond to the intuitive notion of clusters as compact, spherical objects.

**measure types (selection)** This parameter is used for selecting the type of measure to be used for measuring the distance between points. The following options are available: *mixed measures, nominal measures, numerical measures* and *Bregman divergences*.

**mixed measure (selection)** This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'.

**nominal measure (selection)** This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure' option should be selected.

**numerical measure (selection)** This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence (selection)** This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type (selection)** This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by $k(x,y)=x^Ty$ i.e. it is inner product of $x$ and $y$.

- **radial** The radial kernel is defined by $\exp(-g ||x-y||^2)$ where $g$ is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x,y)=(x^Ty+1)^d$ where $d$ is the degree of the polynomial and it is specified by the *kernel degree*.
7. Modeling

parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net $tanh(a \times x * y + b)$ where $a$ is $alpha$ and $b$ is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for $alpha$ is $1/N$, where $N$ is the data dimension. Note that not all choices of $a$ and $b$ lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has adjustable parameters $gamma$ and $degree$.

- **epachnenikov** The Epanechnikov kernel is this function $(3/4)(1-u^2)$ for $u$ between -1 and 1 and zero for $u$ outside that range. It has two adjustable parameters *kernel sigma1* and *kernel degree*.

- **gaussian_combination** This is the gaussian combination kernel. It has adjustable parameters *kernel sigma1*, *kernel sigma2* and *kernel sigma3*.

- **multiquadric** The multiquadric kernel is defined by the square root of $||x-y||^2 + c^2$. It has adjustable parameters *kernel sigma1* and *kernel sigma*.

**kernel gamma** *(real)* This is the SVM kernel parameter *gamma*. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *radial* or *anova*.

**kernel sigma1** *(real)* This is the SVM kernel parameter *sigma1*. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *epachnenikov*, *gaussian combination* or *multiquadric*.

**kernel sigma2** *(real)* This is the SVM kernel parameter *sigma2*. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*. 
7.3. Clustering and Segmentation

**kernel sigma3** *(real)* This is the SVM kernel parameter sigma3. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *gaussian combination*.

**kernel shift** *(real)* This is the SVM kernel parameter shift. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** *(real)* This is the SVM kernel parameter degree. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial*, *anova* or *epachnenikov*.

**kernel a** *(real)* This is the SVM kernel parameter a. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** *(real)* This is the SVM kernel parameter b. This parameter is only available when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**Tutorial Processes**

**Agglomerative Clustering of Ripley-Set data set**

The 'Ripley-Set' data set is loaded using the Retrieve operator. A *breakpoint* is inserted at this step so that you can have a look at the ExampleSet. The Agglomerative Clustering operator is applied on this ExampleSet. Run the process and switch to the Results Workspace. Note the *Graph View* of the results. You can see that the algorithm has not created separate groups or clusters as other clustering algorithms (like k-means), instead the result is a hierarchy of clusters. Under the *Folder View* you can see members of each cluster in folder format. You can see that it is an hierarchy of folders. The *Dendogram View* shows the dendrogram for this clustering which shows how single-element clusters were joined step by step to make a hierarchy of clusters.
7. Modeling

Top Down Clustering

This operator performs top down clustering by applying the inner flat clustering scheme recursively. Top down clustering is a strategy of hierarchical clustering. The result of this operator is an hierarchical cluster model.

Description

This operator is a nested operator i.e. it has a subprocess. The subprocess must have a flat clustering operator e.g. the K-Means operator. This operator builds Hierarchical clustering model using the clustering operator provided in its subprocess. You need to have basic understanding of subprocesses in order to apply this operator. Please study the documentation of the Subprocess operator for basic understanding of subprocesses.

The basic idea of Top down clustering is that all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy. Top down clustering is a strategy of hierarchical clustering. Hierarchical clustering (also known as Connectivity based clustering) is a method of cluster analysis which seeks to build a hierarchy of clusters. Hierarchical clustering, is based on the core idea of objects being more related to nearby objects than to objects farther
7.3. Clustering and Segmentation

away. As such, these algorithms connect 'objects' (or examples, in case of an ExampleSet) to form clusters based on their distance. A cluster can be described largely by the maximum distance needed to connect parts of the cluster. At different distances, different clusters will form. These algorithms do not provide a single partitioning of the data set, but instead provide an extensive hierarchy of clusters that merge with each other at certain distances.

Strategies for hierarchical clustering generally fall into two types:

- **Agglomerative**: This is a bottom-up approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. This type of clustering is implemented in RapidMiner as the Agglomerative Clustering operator.

- **Divisive**: This is a top-down approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

**Input Ports**

device set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process.

**Output Ports**

cluster model (clu) This port delivers the hierarchical cluster model. It has information regarding the clustering performed.

clustered set (clu) The ExampleSet that was given as input is passed with minor
7. Modeling

changes to the output through this port. An attribute with id role is added to the input ExampleSet to distinguish examples. An attribute with cluster role may also be added depending on the state of the add cluster label parameter.

Parameters

create cluster label (boolean) This parameter specifies if a cluster label should be created. If this parameter is set to true, a new attribute with cluster role is generated in the resultant ExampleSet, otherwise this operator does not add the cluster attribute.

max depth (integer) This parameter specifies the maximal depth of the cluster tree.

max leaf size (integer) This parameter specifies the maximal number of items in each cluster leaf.

Tutorial Processes

Top down clustering of Ripley-Set data set

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the Top Down Clustering operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The Top Down Clustering operator is applied on this data set. Run the process and you will see that two new attributes are created by the Top Down Clustering operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. Each example is assigned to a particular cluster. Note the Graph View of the results. You can see that the algorithm has not created separate groups.
or clusters as other clustering algorithms (like k-means), instead the result is a hierarchy of clusters. Under the Folder View you can see members of each cluster in folder format. You can see that it is an hierarchy of folders.

Extract Cluster Prototypes

This operator generates an ExampleSet consisting of the Cluster Prototypes from the Cluster Model. This operator is usually applied after clustering operators to store the Cluster Prototypes in form of an ExampleSet.

Description

Most clustering algorithms like K-Means or K-Medoids cluster the data around some prototypical data vectors. For example the K-Means algorithm uses the centroid of all examples of a cluster. The Extract Cluster Prototypes operator extracts these prototypes and stores them in an ExampleSet for further use. This operator expects a cluster model as input. The information about the cluster prototypes can be seen in the cluster models generated by most clustering operators but the Extract Cluster Prototypes operator stores this information in form of an ExampleSet thus it can be used easily.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. Clustering is a technique for extracting information from unlabeled data. Clustering can be very useful in many different scenarios e.g. in a marketing application we may
be interested in finding clusters of customers with similar buying behavior.

**Differentiation**

**k-Medoids** The K-Medoids operator performs the clustering and generates a cluster model and a clustered ExampleSet. The cluster model generated by the K-Medoids operator can be used by the Extract Cluster Prototypes operator to store the Centroid Table in form of an ExampleSet. See page 810 for details.

**k-Means** The K-Means operator performs the clustering and generates a cluster model and a clustered ExampleSet. The cluster model generated by the K-Means operator can be used by the Extract Cluster Prototypes operator to store the Centroid Table in form of an ExampleSet. See page 801 for details.

**Input Ports**

**model** *(mod)* This port expects a cluster model. It has information regarding the clustering performed by a clustering operator. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

**Output Ports**

**example set** *(exa)* The ExampleSet consisting of the Cluster Prototypes is generated from the input Cluster Model and the ExampleSet is delivered through this port.

**model** *(mod)* The cluster model that was given as input is passed without any changes to the output through this port.
7.3. Clustering and Segmentation

Related Documents

k-Medoids (810)
k-Means (801)

Tutorial Processes

Extracting Centroid Table after application of the K-Means operator on Ripley-Set

In many cases, no target attribute (i.e. label) can be defined and the data should be automatically grouped. This procedure is called Clustering. RapidMiner supports a wide range of clustering schemes which can be used in just the same way like any other learning scheme. This includes the combination with all preprocessing operators.

In this Example Process, the 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the K-Means operator. Other than the label attribute the 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Means operator is applied on this data set with default values for all parameters. Run the process and you will see that two new attributes are created by the K-Means operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter k was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. A cluster model is delivered through the cluster model output port. It has information regarding the clustering performed. Under Folder View you can see members of each cluster in folder format. You can see information regarding centroids under the Centroid Table and Centroid Plot View tabs.
**7. Modeling**

*breakpoint* is inserted at this step so that you can have a look at the cluster model (especially the *Centroid Table*) before application of the Extract Cluster Prototypes operator. The Extract Cluster Prototypes operator is applied on the cluster model generated by the K-Means operator which stores the *Centroid Table* in form of an ExampleSet which can be seen in the Results Workspace.

---

**FP-Growth**

This operator efficiently calculates all frequent itemsets from the given ExampleSet using the FP-tree data structure. It is compulsory that all attributes of the input ExampleSet should be binominal.

**Description**

In simple words, frequent itemsets are groups of items that often appear together in the data. It is important to know the basics of market-basket analysis for understanding frequent itemsets.

The market-basket model of data is used to describe a common form of a many-to-many relationship between two kinds of objects. On the one hand, we have items, and on the other we have baskets, also called 'transactions'. The set of items is...
usually represented as set of attributes. Mostly these attributes are binominal. The transactions are usually each represented as examples of the ExampleSet. When an attribute value is 'true' in an example; it implies that the corresponding item is present in that transaction. Each transaction consists of a set of items (an itemset). Usually it is assumed that the number of items in a transaction is small, much smaller than the total number of items i.e. in most of the examples most of the attribute values are 'false'. The number of transactions is usually assumed to be very large i.e. the number of examples in the ExampleSet is assumed to be large. The frequent-itemsets problem is that of finding sets of items that appear together in at least a threshold ratio of transactions. This threshold is defined by the 'minimum support' criteria. The support of an itemset is the number of times that itemset appears in the ExampleSet divided by the total number of examples. The 'Transactions' data set at SSamples/data/Transactions in the repository of RapidMiner is an example of how transactions data usually look like.

The discovery of frequent itemsets is often viewed as the discovery of 'association rules', although the latter is a more complex characterization of data, whose discovery depends fundamentally on the discovery of frequent itemsets. Association rules are derived from the frequent itemsets. The FP-Growth operator finds the frequent itemsets and operators like the Create Association Rules operator uses these frequent itemsets for calculating the association rules.

This operator calculates all frequent itemsets from an ExampleSet by building a FP-tree data structure on the transaction data base. This is a very compressed copy of the data which in many cases fits into main memory even for large data bases. All frequent itemsets are derived from this FP-tree. Many other frequent itemset mining algorithms also exist e.g. the Apriori algorithm. A major advantage of FP-Growth compared to Apriori is that it uses only 2 data scans and is therefore often applicable even on large data sets.

Please note that the given ExampleSet should contain only binominal attributes, i.e. nominal attributes with only two different values. If your ExampleSet does not satisfy this condition, you may use appropriate preprocessing operators to transform it into the required form. The discretization operators can be used for changing the value of numerical attributes to nominal attributes. Then the
Nominal to Binomial operator can be used for transforming nominal attributes into binomial attributes.

Please note that the frequent itemsets are mined for the positive entries in your ExampleSet, i.e. for those nominal values which are defined as positive in your ExampleSet. If your data does not specify the positive entries correctly, you may set them using the positive value parameter. This only works if all your attributes contain this value.

This operator has two basic working modes:

- finding at least the specified number of itemsets with highest support without taking the 'min support' into account. This mode is available when the find min number of itemsets parameter is set to true. Then this operator finds the number of itemsets specified in the min number of itemsets parameter. The min support parameter is ignored in this case.

- finding all itemsets with a support larger than the specified minimum support. The minimum support is specified through the min support parameter. This mode is available when the find min number of itemsets parameter is set to false.

**Input Ports**

example set (exa) This input port expects an ExampleSet. Please make sure that all attributes of the ExampleSet are binomial.

**Output Ports**

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
frequent sets (fre) The frequent itemsets are delivered through this port. Operators like the Create Association Rules operator can use these frequent itemsets to generate association rules.

Parameters

find min number of itemsets (boolean) If this parameter is set to true, this operator finds at least the specified number of itemsets with highest support without taking the min support parameter into account. This operator finds (at least) the number of itemsets specified in the min number of itemsets parameter. The min support parameter is ignored to some extent in this case. The minimal support is decreased automatically until the specified minimum number of frequent itemsets is found. The defined minimal support is lowered by 20 percent each time.

min number of itemsets (integer) This parameter is only available when the find min number of itemsets parameter is set true. This parameter specifies the minimum number of itemsets which should be mined.

max number of retries (integer) This parameter is only available when the find min number of itemsets parameter is set true. This parameter determines how many times the operator should lower the minimal support to find the minimal number of item sets. Each time the minimal support is lowered by 20 percent.

positive value (string) This parameter determines which value of the binominal attributes should be treated as positive. The attributes with that value are considered as part of a transaction. If left blank, the ExampleSet determines which value is used.

min support (real) The minimum support criteria is specified by this parameter. Please study the description of this operator for more information about minimum support.

max items (integer) This parameter specifies the upper bound for the length of the itemsets i.e. the maximum number of items in an itemset. If set to -1, this parameter imposes no upper bound.

must contain This parameter specifies the items that should be part of frequent itemsets. It is specified through a regular expression. If there is no specific item
that you want to have in the frequent itemset, you can leave this blank.

**Tutorial Processes**

**Introduction to the FP-Growth operator**

The 'Iris' data set is loaded using the Retrieve operator. A *breakpoint* is inserted here so that you can view the ExampleSet. As you can see, the ExampleSet has real attributes. Thus the FP-Growth operator cannot be applied on it directly because the FP-Growth operator requires all attributes to be binominal. We have to do some preprocessing to mold the ExampleSet into the desired form. The Discretize by Frequency operator is applied to change the real attributes to nominal attributes. Then the Nominal to Binominal operator is applied to change these nominal attributes to binominal attributes. Finally, the FP-Growth operator is applied to generate frequent itemsets. The frequent itemsets can be viewed in the Results Workspace. Run this process with different values for different parameters to get a better understanding of this operator.
Create Association Rules

This operator generates a set of association rules from the given set of frequent itemsets.

Description

Association rules are if/then statements that help uncover relationships between seemingly unrelated data. An example of an association rule would be ‘If a customer buys eggs, he is 80% likely to also purchase milk.’ An association rule has two parts, an antecedent (if) and a consequent (then). An antecedent is an item (or itemset) found in the data. A consequent is an item (or itemset) that is found in combination with the antecedent.

Association rules are created by analyzing data for frequent if/then patterns and using the criteria support and confidence to identify the most important relationships. Support is an indication of how frequently the items appear in the database. Confidence indicates the number of times the if/then statements have been found to be true. The frequent if/then patterns are mined using the operators like the FP-Growth operator. The Create Association Rules operator takes these frequent itemsets and generates association rules.

Such information can be used as the basis for decisions about marketing activities such as, e.g., promotional pricing or product placements. In addition to the above example from market basket analysis association rules are employed today in many application areas including Web usage mining, intrusion detection and bioinformatics.

Input Ports

item sets (ite) This input port expects frequent itemsets. Operators like the FP-Growth operator can be used for providing these frequent itemsets.
7. Modeling

Output Ports

item sets (ite) The itemsets that were given as input is passed without changing to the output through this port. This is usually used to reuse the same itemsets in further operators or to view the itemsets in the Results Workspace.

rules (rul) The association rules are delivered through this output port.

Parameters

criterion (selection) This parameter specifies the criterion which is used for the selection of rules.

- confidence The confidence of a rule is defined $conf(X \implies Y) = \frac{supp(X \cup Y)}{supp(X)}$. Be careful when reading the expression: here $supp(X \cup Y)$ means support for occurrences of transactions where $X$ and $Y$ both appear, not support for occurrences of transactions where either $X$ or $Y$ appears. Confidence ranges from 0 to 1. Confidence is an estimate of $Pr(Y \mid X)$, the probability of observing $Y$ given $X$. The support $supp(X)$ of an itemset $X$ is defined as the proportion of transactions in the data set which contain the itemset.

- lift The lift of a rule is defined as $lift(X \implies Y) = \frac{supp(X \cup Y)}{supp(Y) \times supp(X)}$ or the ratio of the observed support to that expected if $X$ and $Y$ were independent. Lift can also be defined as $lift(X \implies Y) = \frac{conf(X \implies Y)}{supp(Y)}$. Lift measures how far from independence are $X$ and $Y$. It ranges within 0 to positive infinity. Values close to 1 imply that $X$ and $Y$ are independent and the rule is not interesting.

- conviction conviction is sensitive to rule direction i.e. $conv(X \implies Y)$ is not same as $conv(Y \implies X)$. Conviction is somewhat inspired in the logical definition of implication and attempts to measure the degree of implication of a rule. Conviction is defined as $conv(X \implies Y) = \frac{(1 - supp(Y))}{(1 - conf(X \implies Y))}$
7.4. Association and Item Set Mining

- **gain** When this option is selected, the gain is calculated using the \( gain \ theta \) parameter.

- **laplace** When this option is selected, the Laplace is calculated using the \( laplace \ k \) parameter.

- **ps** When this option is selected, the ps criteria is used for rule selection.

**min confidence** (real) This parameter specifies the minimum confidence of the rules.

**min criterion value** (real) This parameter specifies the minimum value of the rules for the selected criterion.

**gain theta** (real) This parameter specifies the parameter \( Theta \) which is used in the Gain calculation.

**laplace k** (real) This parameter specifies the parameter \( k \) which is used in the Laplace function calculation.

**Tutorial Processes**

**Introduction to the Create Association Rules operator**

The 'Iris' data set is loaded using the Retrieve operator. A **breakpoint** is inserted here so that you can view the ExampleSet. As you can see, the ExampleSet has real attributes. Thus the FP-Growth operator cannot be applied on it directly because the FP-Growth operator requires all attributes to be binominal. We have to do some preprocessing to mold the ExampleSet into desired form. The Discretize by Frequency operator is applied to change the real attributes to nominal attributes. Then the Nominal to Binominal operator is applied to change these nominal attributes to binominal attributes. Finally, the FP-Growth operator is applied to generate frequent itemsets. The frequent itemsets generated from the FP-Growth operator are provided to the Create Association Rules operator. The resultant association rules can be viewed in the Results Workspace. Run this process with different values for different parameters to get a better understanding.
Generalized Sequential Patterns

This operator searches sequential patterns in a set of transactions using the GSP (Generalized Sequential Pattern) algorithm. GSP is a popular algorithm used for sequence mining.

Description

This operator searches sequential patterns in a set of transactions. The Example-Set must contain one attribute for the time and one attribute for the customer. Moreover, each transaction must be encoded as a single example. The time and customer attributes are specified through the time attribute and customer id parameters respectively. This pair of attributes is used for generating one sequence per customer containing every transaction ordered by the time of each transaction. The algorithm then searches sequential patterns in the form of: If a customer bought item 'a' and item 'c' in one transaction, he bought item 'b' in the next. This pattern is represented in this form: \( <a, c> \text{ then } <b> \). The minimal support describes how many customer must support such a pattern for regarding it as frequent. Infrequent patterns will be dropped. A customer
supports such a pattern, if there are some parts of his sequence that includes that pattern. The above pattern would be supported by a customer, for example, with transactions: \(<s, g> \text{ then } <a, s, c> \text{ then } <b> \text{ then } <f, h>\). The minimum support criteria is specified through the \textit{min support} parameter.

The \textit{min gap}, \textit{max gap} and \textit{window size} parameters determine how transactions are handled. For example, if the above customer forgot to buy item 'c', and had to return 5 minutes later to buy it, then his transactions would look like: \(<s, g> \text{ then } <a, s> \text{ then } <c> \text{ then } <b> \text{ then } <f, h>\). This would not support the pattern \(<a, c> \text{ then } <b>\). To avoid this problem, the \textit{window size} determines, how long a subsequent transaction is treated as the same transaction. If the window size is larger than 5 minutes then \(<c>\) would be treated as being part of the second transaction and hence this customer would support the above pattern. The \textit{max gap} parameter causes a customer's sequence not to support a pattern, if the transactions containing this pattern are too widely separated in time. The \textit{min gap} parameter does the same if they are too near.

This technique overcomes some crucial drawbacks of existing mining methods, for example:

- absence of time constraints: This drawback is overcome by the \textit{min gap} and \textit{max gap} parameters.

- rigid definition of a transaction: This drawback is overcome by the sliding time window.

Please note that all attributes (except customer and time attributes) of the given \textit{ExampleSet} should be binominal, i.e. nominal attributes with only two possible values. If your \textit{ExampleSet} does not satisfy this condition, you may use appropriate preprocessing operators to transform it into the required form. The discretization operators can be used for changing the value of numerical attributes to nominal attributes. Then the Nominal to Binominal operator can be used for transforming nominal attributes into binominal attributes.

Please note that the sequential patterns are mined for the positive entries in your \textit{ExampleSet}, i.e. for those nominal values which are defined as positive in your \textit{ExampleSet}. If your data does not specify the positive entries correctly, you may
set them using the positive value parameter. This only works if all your attributes contain this value.

**Input Ports**

**example set (exa)** This input port expects an ExampleSet. Please make sure that all attributes (except customer and time attributes) of the ExampleSet are binominal.

**Output Ports**

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**patterns (pat)** The GSP algorithm is applied on the given ExampleSet and the resultant set of sequential patterns is delivered through this port.

**Parameters**

**customer id (string)** This parameter specifies the name of the attribute that will be used for identifying the customers.

**time attribute (string)** This parameter specifies the name of the numerical attribute that specifies the time of a transaction.

**min support (real)** The minimum support criteria is specified by this parameter. Please study the description of this operator for more information about minimum support.

**window size (real)** This parameter specifies the window size. Please study the description of this operator for more information about window size.

**max gap (real)** This parameter specifies the maximal gap. The max gap param-
7.4. Association and Item Set Mining

Parameter causes a customers sequence not to support a pattern, if the transactions containing this pattern are too widely separated in time.

**min gap** *(real)* This parameter specifies the minimal gap. The min gap parameter causes a customers sequence not to support a pattern, if the transactions containing this pattern are too near in time.

**positive value** *(string)* This parameter determines which value of the binominal attributes should be treated as positive. The attributes with this value in an example are considered to be part of that transaction.

**Tutorial Processes**

**Introduction to the GSP operator**

The ExampleSet expected by the GSP operator should meet the following criteria:

- It should have an attribute that can be used for identifying the customers.
- It should have a numerical attribute that represents the time of the transaction.
- All other attributes are used for representing items of transactions. These attributes should be binominal.

This Example Process starts with the Subprocess operator. A sequence of operators is applied in the subprocess to generate an ExampleSet that satisfies all the above mentioned conditions. A *breakpoint* is inserted after the Subprocess operator so that you can have a look at the ExampleSet. The Customer attribute represents the customers, this ExampleSet has five. The Time attribute represents the time of transaction. For simplicity all transactions have been given the same time. This will not be case in real scenarios. There are 20 binominal attributes in the ExampleSet that represent items that the customer may buy in a transaction. In this ExampleSet, value 'true' for an item in an example
means that this item was bought in this transaction (represented by the current example). The GSP operator is applied on this ExampleSet. The customer id and time attribute parameters are set to 'Customer' and 'Time' respectively. The positive value parameter is set to 'true'. The min support parameter is set 0.9. The resultant set of sequential patterns can be seen in the Results Workspace.

Correlation Matrix

This operator determines correlation between all attributes and it can produce a weights vector based on these correlations. Correlation is a statistical technique that can show whether and how strongly pairs of attributes are related.

Description

A correlation is a number between -1 and +1 that measures the degree of association between two attributes (call them X and Y). A positive value for the correlation implies a positive association. In this case large values of X tend to be associated with large values of Y and small values of X tend to be associated with small values of Y. A negative value for the correlation implies a negative or inverse association. In this case large values of X tend to be associated with small values of Y and vice versa.
Suppose we have two attributes X and Y, with means $X'$ and $Y'$ respectively and standard deviations $S(X)$ and $S(Y)$ respectively. The correlation is computed as summation from 1 to n of the product $(X(i)-X')(Y(i)-Y')$ and then dividing this summation by the product $(n-1).S(X).S(Y)$ where n is total number of examples and $i$ is the increment variable of summation. There can be other formulas and definitions but let us stick to this one for simplicity.

As discussed earlier a positive value for the correlation implies a positive association. Suppose that an X value was above average, and that the associated Y value was also above average. Then the product $(X(i)-X')(Y(i)-Y')$ would be the product of two positive numbers which would be positive. If the X value and the Y value were both below average, then the product above would be of two negative numbers, which would also be positive. Therefore, a positive correlation is evidence of a general tendency that large values of X are associated with large values of Y and small values of X are associated with small values of Y.

As discussed earlier a negative value for the correlation implies a negative or inverse association. Suppose that an X value was above average, and that the associated Y value was instead below average. Then the product $(X(i)-X')(Y(i)-Y')$ would be the product of a positive and a negative number which would make the product negative. If the X value was below average and the Y value was above average, then the product above would also be negative. Therefore, a negative correlation is evidence of a general tendency that large values of X are associated with small values of Y and small values of X are associated with large values of Y.

This operator can be used for creating a correlation matrix that shows correlations of all the attributes of the input ExampleSet. Please note that this operator performs a data scan for each attribute combination and might therefore take some time for non-memory ExampleSets. The attribute weights vector; based on the correlations can also be returned by this operator. Using this weights vector, highly correlated attributes can be removed from the ExampleSet with the help of the Select by Weights operator. Highly correlated attributes can be more easily removed by simply using the Remove Correlated Attributes operator. Correlated attributes are usually removed because they are similar in behavior.
and will have similar impact in prediction calculations, so keeping attributes with similar impacts is redundant. Removing correlated attributes saves space and time of calculation of complex algorithms.

Input Ports

element set (exa) This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

element set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

matrix (mat) The correlations of all attributes of the input ExampleSet are calculated and the resultant correlation matrix is returned from this port.

weights (wei) The attribute weights vector based on the correlations of the attributes is delivered through this output port.

Parameters

normalize weights (boolean) This parameter indicates if the weights of the resultant attribute weights vector should be normalized. If set to true, all weights are normalized such that the minimum weight is 0 and the maximum weight is 1.

squared correlation (boolean) This parameter indicates if the squared correlation should be calculated. If set to true, the correlation matrix shows squares of correlations instead of simple correlations.
7.6. Similarity Computation

Tutorial Processes

Correlation matrix of the Golf data set

The 'Golf' data set is loaded using the Retrieve operator. A breakpoint is inserted here so that you can view the ExampleSet. As you can see, the ExampleSet has 4 regular attributes i.e. 'Outlook', 'Temperature', 'Humidity' and 'Wind'. The Correlation Matrix operator is applied on it. The weights vector generated by this operator is provided to the Select by Weights operator along with the 'Golf' data set. The parameters of the Select by Weights operator are adjusted such that the attributes with weights greater than 0.5 are selected and all other attributes are removed. This is why the resultant ExampleSet does not have the 'Temperature' attribute (weight=0). The correlation matrix, weights vector and the resultant ExampleSet can be viewed in the Results Workspace.
Data to Similarity

This operator measures the similarity of each example of the given ExampleSet with every other example of the same ExampleSet.

Description

The Data to Similarity operator calculates the similarity among examples of an ExampleSet. Same comparisons are not repeated again e.g. if example \( x \) is compared with example \( y \) to compute similarity then example \( y \) will not be compared again with example \( x \) to compute similarity because the result will be the same. Thus if there are \( n \) examples in the ExampleSet, this operator does not return \( n^2 \) similarity comparisons. Instead it returns \((n)(n-1)/2\) similarity comparisons. This operator provides many different measures for similarity computation. The measure to use for calculating the similarity can be specified through the parameters. Four types of measures are provided: mixed measures, nominal measures, numerical measures and Bregman divergences.

The behavior of this operator can be considered close to a certain scenario of the Cross Distances operator, if the same ExampleSet is provided at both inputs of the Cross Distances operator and the compute similarities parameter is also set to true. In this case the Cross Distances operator behaves similar to the Data to Similarity operator. There are a few differences though e.g. in this scenario examples are also compared with themselves and secondly the signs (i.e.+ive or -ive) of the results are also different.

Differentiation

Data to Similarity Data The Data to Similarity Data operator calculates the similarity among all examples of an ExampleSet. Even examples are compared to themselves. Thus if there are \( n \) examples in the ExampleSet, this operator returns \( n^2 \) similarity comparisons. The Data to Similarity Data operator returns an
ExampleSet which is merely a view, so there should be no memory problems. See page ?? for details.

**Input Ports**

`example set (exa)` This input port expects an ExampleSet. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

**Output Ports**

`similarity (sim)` A similarity measure object that contains the calculated similarity between each example of the given ExampleSet with every other example of the same ExampleSet is delivered through this port.

`example set (exa)` The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**Parameters**

`measure types (selection)` This parameter is used for selecting the type of measure to be used for calculating similarity. Following options are available: mixed measures, nominal measures, numerical measures and Bregman divergences.

`mixed measure (selection)` This parameter is available if the `measure type` parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'.

`nominal measure (selection)` This parameter is available if the `measure type` parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. In this case the 'numerical measure'
option should be selected.  
**numerical measure (selection)** This parameter is available if the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. In this case the 'nominal measure' option should be selected.  
**divergence (selection)** This parameter is available if the *measure type* parameter is set to 'bregman divergences'.  
**kernel type (selection)** This parameter is only available if the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by $k(x, y) = x \cdot y$ i.e.it is the inner product of $x$ and $y$.

- **radial** The radial kernel is defined by $\exp(-g ||x-y||^2)$ where $g$ is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x, y) = (x \cdot y + 1)^d$ where $d$ is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.

- **neural** The neural kernel is defined by a two layered neural net $\tanh(a \cdot x \cdot y + b)$ where $a$ is *alpha* and $b$ is the *intercept constant*. These parameters can be adjusted using the *kernel a* and *kernel b* parameters. A common value for *alpha* is $1/N$, where $N$ is the data dimension. Note that not all choices of $a$ and $b$ lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that the *sigmoid* kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has the adjustable parameters *gamma* and *degree*.

- **epachnenikov** The Epanechnikov kernel is this function $(3/4)(1-u^2)$ for $u$
between -1 and 1 and zero for $u$ outside that range. It has the two adjustable parameters \textit{kernel sigma1} and \textit{kernel degree}.

- \textbf{gaussian\_combination} This is the gaussian combination kernel. It has the adjustable parameters \textit{kernel sigma1}, \textit{kernel sigma2} and \textit{kernel sigma3}.

- \textbf{multiquadric} The multiquadric kernel is defined by the square root of $||x-y||^2 + c^2$. It has the adjustable parameters \textit{kernel sigma1} and \textit{kernel sigma shift}.

\textbf{kernel gamma} (real) This is the SVM kernel parameter gamma. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{radial} or \textit{anova}.

\textbf{kernel sigma1} (real) This is the SVM kernel parameter sigma1. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{epachnenikov}, \textit{gaussian combination} or \textit{multiquadric}.

\textbf{kernel sigma2} (real) This is the SVM kernel parameter sigma2. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{gaussian combination}.

\textbf{kernel sigma3} (real) This is the SVM kernel parameter sigma3. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{gaussian combination}.

\textbf{kernel shift} (real) This is the SVM kernel parameter shift. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{multiquadric}.

\textbf{kernel degree} (real) This is the SVM kernel parameter degree. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{polynomial}, \textit{anova} or \textit{epachnenikov}.

\textbf{kernel a} (real) This is the SVM kernel parameter a. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{neural}. 
kernel b \textit{(real)} This is the SVM kernel parameter b. This parameter is only available when the \textit{numerical measure} parameter is set to 'Kernel Euclidean Distance' and the \textit{kernel type} parameter is set to \textit{neural}.

Related Documents

Data to Similarity Data (??)

Tutorial Processes

Introduction to the Data to Similarity operator

The 'Golf' data set is loaded using the Retrieve operator. A \textit{breakpoint} is inserted here so that you can have a look the ExampleSet. You can see that the ExampleSet has 14 examples. The Data to Similarity operator is applied on it to compute the similarity of examples. As there are 14 examples in the given ExampleSet, there will be 91 (i.e. \((14)(14-1)/2\)) similarity comparisons in the resultant similarity measure object which can be seen in the Results Workspace.
7.6. Similarity Computation

Cross Distances

This operator calculates the distance between each example of a 'request set' ExampleSet to each example of a 'reference set' ExampleSet. This operator is also capable of calculating similarity instead of distance.

Description

The Cross Distances operator takes two ExampleSets as input i.e. the 'reference set' and 'request set' ExampleSets. It creates an ExampleSet that contains the distance between each example of the 'request set' ExampleSet to each example of the 'reference set' ExampleSet. Please note that both input ExampleSets should have the same attributes and in the same order. This operator will not work properly if the order of the attributes is different. This operator is also capable of calculating similarity instead of distance. If the compute similarities parameter is set to true, similarities are calculated instead of distances. Please note that both input ExampleSets should have id attributes. If id attributes are not present, this operator automatically creates id attributes for such ExampleSets. The measure to use for calculating the distances can be specified through the parameters. Four type of measures are provided: mixed measures, nominal measures, numerical measures and Bregman divergences.

If data is imported from two different sources that are supposed to represent the same data but which have columns in different orders, the Cross Distances operator will not behave as expected. It is possible to work round this by using the Generate Attributes operator to recreate attributes in both ExampleSets in the same order.
Input Ports

request set *(req)* This input port expects an ExampleSet. This ExampleSet will be used as the 'request set'. Please note that both input ExampleSets ('request set' and 'reference set') should have the same attributes and in the same order. This operator will not work properly if the order of the attributes is different. Also note that both input ExampleSets should have *id* attributes. If *id* attributes are not present, this operator automatically creates *id* attributes for such ExampleSets.

reference set *(ref)* This input port expects an ExampleSet. This ExampleSet will be used as the 'reference set'. Please note that both input ExampleSets ('request set' and 'reference set') should have same attributes and in the same order. This operator will not work properly if the order of the attributes is different. Also note that both input ExampleSets should have *id* attributes. If *id* attributes are not present, this operator automatically creates *id* attributes for such ExampleSets.

Output Ports

result set *(res)* An ExampleSet that contains the distance (or similarity, if the *compute similarities* parameter is set to true) between each example of the 'request set' ExampleSet to each example of the 'reference set' ExampleSet is delivered through this port.

request set *(req)* The 'request set' ExampleSet that was provided at the request set input port is delivered through this port. If the input ExampleSet had an *id* attribute then the ExampleSet is delivered without any modification. Otherwise an *id* attribute is automatically added to the input ExampleSet.

reference set *(ref)* The 'reference set' ExampleSet that was provided at the reference set input port is delivered through this port. If the input ExampleSet had an *id* attribute then the ExampleSet is delivered without any modification. Otherwise an *id* attribute is automatically added to the input ExampleSet.
7.6. Similarity Computation

Parameters

**measure types** *(selection)* This parameter is used for selecting the type of measure to be used for calculating distances (or similarity). The following options are available: *mixed measures, nominal measures, numerical measures* and *Bregman divergences*.

**mixed measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'mixed measures'. The only available option is the 'Mixed Euclidean Distance'.

**nominal measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'nominal measures'. This option cannot be applied if the input ExampleSet has numerical attributes. If the input ExampleSet has numerical attributes the 'numerical measure' option should be selected.

**numerical measure** *(selection)* This parameter is available when the *measure type* parameter is set to 'numerical measures'. This option cannot be applied if the input ExampleSet has nominal attributes. If the input ExampleSet has nominal attributes the 'nominal measure' option should be selected.

**divergence** *(selection)* This parameter is available when the *measure type* parameter is set to 'bregman divergences'.

**kernel type** *(selection)* This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance'. The type of the kernel function is selected through this parameter. Following kernel types are supported:

- **dot** The dot kernel is defined by $k(x,y) = x^*y$ i.e. it is inner product of $x$ and $y$.

- **radial** The radial kernel is defined by $\exp(-g \|x-y\|^2)$ where $g$ is the *gamma* that is specified by the *kernel gamma* parameter. The adjustable parameter *gamma* plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand.

- **polynomial** The polynomial kernel is defined by $k(x,y) = (x^*y+1)^d$ where $d$ is the degree of the polynomial and it is specified by the *kernel degree* parameter. The Polynomial kernels are well suited for problems where all the training data is normalized.
7. Modeling

- **neural** The neural kernel is defined by a two layered neural net $\tanh(a x^*y+b)$ where $a$ is alpha and $b$ is the intercept constant. These parameters can be adjusted using the kernel a and kernel b parameters. A common value for alpha is $1/N$, where $N$ is the data dimension. Note that not all choices of $a$ and $b$ lead to a valid kernel function.

- **sigmoid** This is the sigmoid kernel. Please note that the sigmoid kernel is not valid under some parameters.

- **anova** This is the anova kernel. It has adjustable parameters gamma and degree.

- **epachnenikov** The Epanechnikov kernel is this function $(3/4)(1-u^2)$ for $u$ between -1 and 1 and zero for $u$ outside that range. It has two adjustable parameters kernel sigma1 and kernel degree.

- **gaussian combination** This is the gaussian combination kernel. It has adjustable parameters kernel sigma1, kernel sigma2 and kernel sigma3.

- **multiquadric** The multiquadric kernel is defined by the square root of $||x-y||^2 + c^2$. It has adjustable parameters kernel sigma1 and kernel sigma shift.

**kernel gamma** (real) This is the SVM kernel parameter gamma. This parameter is available when only the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to radial or anova.

**kernel sigma1** (real) This is the SVM kernel parameter sigma1. This parameter is available only when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to epachnenikov, gaussian combination or multiquadric.

**kernel sigma2** (real) This is the SVM kernel parameter sigma2. This parameter is available only when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to gaussian combination.

**kernel sigma3** (real) This is the SVM kernel parameter sigma3. This parameter is available only when the numerical measure parameter is set to 'Kernel Euclidean Distance' and the kernel type parameter is set to gaussian combination.
7.6. Similarity Computation

**kernel shift** *(real)* This is the SVM kernel parameter shift. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *multiquadric*.

**kernel degree** *(real)* This is the SVM kernel parameter degree. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *polynomial, anova* or *epachnenikov*.

**kernel a** *(real)* This is the SVM kernel parameter a. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**kernel b** *(real)* This is the SVM kernel parameter b. This parameter is available only when the *numerical measure* parameter is set to 'Kernel Euclidean Distance' and the *kernel type* parameter is set to *neural*.

**only top k** *(boolean)* This parameter indicates if only the k nearest to each request example should be calculated.

**k** *(integer)* This parameter is only available when the *only top k* parameter is set to true. It determines how many of the nearest examples should be shown in the result.

**search for** *(selection)* This parameter is only available when the *only top k* parameter is set to true. It determines if the nearest or the farthest distances should be selected.

computesimilarities *(boolean)* If this parameter is set true, similarities are computed instead of distances. All measures will still be usable, but measures that are not originally distance or respective similarity measure are transformed to match optimization direction.

### Tutorial Processes

**Introduction to the Cross Distances operator**

This Example Process starts with a Subprocess operator. This subprocess gen-
7. Modeling

erates the 'request set' ExampleSet and the 'reference set' ExampleSet. A breakpoint is inserted here so that you can have a look at the ExampleSets before application of the Cross Distances operator. You can see that the 'request set' has only 1 example with id 'id_1'. The 'reference set' has just two examples with ids 'id_1' and 'id_2'. Both ExampleSets have three attributes in the same order. It is very important that both ExampleSets should have the same attributes and in the same order otherwise the Cross Distances operator will not behave as expected. The Cross Distances operator is applied on these ExampleSets. The resultant ExampleSet that contains the distance between each example of the 'request set' ExampleSet to each example of the 'reference set' ExampleSet is calculated by the Cross Distance operator. The resultant ExampleSet can be viewed in the Results Workspace.

Apply Model

This operator applies an already learnt or trained model on an ExampleSet.
7.7. Model Application

Description

A model is first trained on an ExampleSet; information related to the ExampleSet is learnt by the model. Then that model can be applied on another ExampleSet usually for prediction. All needed parameters are stored within the model object. It is compulsory that both ExampleSets should have exactly the same number, order, type and role of attributes. If these properties of meta data of ExampleSets are not consistent, it may lead to serious errors. If you want to apply several models in a row; for example you want to apply a few preprocessing models before applying a prediction model; then you may group models. This is possible using the Group Models operator.

Input Ports

model (mod) This port expects a model. It should be made sure that number, order, type and role of attributes of the ExampleSet on which this model was trained are consistent with the ExampleSet on the unlabeled data input port.

unlabelled data ( unl) This port expects an ExampleSet. It should be made sure that number, order, type and role of attributes of this ExampleSet are consistent with ExampleSet on which the model delivered to the model input port was trained.

Output Ports

labelled data (lab) Model that was given in input is applied on the given ExampleSet and the updated ExampleSet is delivered from this port. Some information is added to the input ExampleSet before it is delivered through this output port. For example, when a prediction model is applied on an ExampleSet through Apply Model operator, an attribute with prediction role is added to the ExampleSet. This attribute stores predicted values of label attribute using the given model.

model (mod) Model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same model in further
7. Modeling

operators or to view the model in the Results Workspace.

Parameters

application parameters *(menu)* This parameter models parameters for application (usually not needed). This is an expert parameter.

create view *(boolean)* If the model applied at the input port supports Views, it is possible to create a View instead of changing the underlying data. Simply select this parameter to enable this option. The transformation that would be normally performed directly on the data will then be computed every time a value is requested and the result is returned without changing the data. Some models do not support Views.

Tutorial Processes

Applying a model

In this Example Process, Golf data set is loaded by using Retrieve operator. A classification model is trained on this ExampleSet using k-NN operator. This model is then supplied at model input port of Apply Model operator. Golf-Testset data set is loaded using Retrieve operator and provided at the unlabelled data input port of the Apply Model operator. The Apply Model operator applies the model trained by k-NN operator on the Golf-Testset to predict the value of attribute with label role i.e. 'Play' attribute. The original model is also connected to the results port. Breakpoints are added after both Retrieve operators so that the ExampleSets can be viewed before application of the model.

When you run the process, first of all you will see Golf data set. Press the Run button to continue. Now, you will see Golf-Testset data set. Press the Run button again to see the final output of the process. As you can see in the Results Workspace, an attribute with prediction role is added to the original Golf-Testset
data set. This attribute stores values of *label* (Play) predicted by the model (k-NN classification model). This is why now it is called 'labeled data' instead of 'unlabelled data'.

### Group Models

This operator groups the given models into a single combined model. When this combined model is applied, it is equivalent to applying the original models in their respective order.

#### Description

The Group Models operator groups all input models together into a single combined model. This combined model can be applied on ExampleSets (using the Apply Model operator) like other models. When this combined model is applied, it is equivalent to applying the original models in their respective order. This combined model can also be written into a file using the Write Model operator. This operator is useful in cases where preprocessing and prediction models should be applied together on new and unseen data. A grouped model can be ungrouped with the Ungroup Models operator. Please study the attached Example Process
7. Modeling

for more information about the Group Models operator.

Input Ports

**model in (mod)** This input port expects a model. This operator can have multiple inputs but it is mandatory to provide at least two models to this operator as input. When one input is connected, another model in port becomes available which is ready to accept another model (if any). The order of models remains the same i.e. the model supplied at the first model in port of this operator will be the first model to be applied when the resultant combined model is applied.

Output Ports

**model out (mod)** The given models are grouped into a single combined model and the resultant grouped model is returned from this port.

Tutorial Processes

Grouping models and applying the resultant grouped model

The 'Iris' data set is loaded using the Retrieve operator. A **breakpoint** is inserted here so that you can have a look at the ExampleSet. You can see that the ExampleSet has four regular attributes. The Split Data operator is applied on it to split the ExampleSet into training and testing data sets. The training data set (composed of 70% of examples) is passed to the SVD operator. The **dimensionality reduction** and **dimensions** parameters of the SVD operator are set to 'fixed number' and 2 respectively. Thus the given data set will be reduced to a data set with two dimensions (artificial attributes that represent the original attributes). The SVD model (model that reduces the dimensionality of the given Example-
Set) is provided as the first model to the Group Models operator. The Naive Bayes operator is applied on the resultant ExampleSet (i.e. the training data set with reduced dimensions). The classification model generated by the Naive Bayes operator is provided as the second model to the Group Models operator. Thus the Group Models operator combines two models:

1. SVD dimensionality reduction model
2. Naive Bayes classification model.

This combined model is applied on the testing data set (composed of 30% of the 'Iris' data set) using the Apply Model operator. When the combined model is applied, the SVD model is applied first on the testing data set. Then the Naive Bayes classification model is applied on the resultant ExampleSet (i.e. the testing data set with reduced dimensions). The combined model and the labeled ExampleSet can be seen in the Results Workspace after the execution of the process.
Find Threshold

This operator finds the best threshold for crisp classification of soft classified data based on user defined costs. The optimization step is based on ROC analysis.

Description

This operator finds the threshold for given prediction confidences of soft classified predictions in order to turn it into a crisp classification. The optimization step is based on ROC analysis. ROC is discussed at the end of this description.

The Find Threshold operator finds the threshold of a labeled ExampleSet to map a soft prediction to crisp values. The threshold is delivered through the threshold port. Mostly the Apply Threshold operator is used for applying a threshold after it has been delivered by the Find Threshold operator. If the confidence for the second class is greater than the given threshold the prediction is set to this class otherwise it is set to the other class. This can be easily understood by studying the attached Example Process.

Among various classification methods, there are two main groups of methods: soft and hard classification. In particular, a soft classification rule generally estimates the class conditional probabilities explicitly and then makes the class prediction based on the largest estimated probability. In contrast, hard classification bypasses the requirement of class probability estimation and directly estimates the classification boundary.

Receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot of the true positive rate vs. false positive rate for a binary classifier system as its discrimination threshold is varied. The ROC can also be represented equivalently by plotting the fraction of true positives out of the positives (TP/P = true positive
rate) vs. the fraction of false positives out of the negatives (FP/N = false positive rate). TP/P determines a classifier or a diagnostic test performance on classifying positive instances correctly among all positive samples available during the test. FP/N, on the other hand, defines how many incorrect positive results occur among all negative samples available during the test.

A ROC space is defined by FP/N and TP/P as x and y axes respectively, which depicts relative trade-offs between true positive (benefits) and false positive (costs). Each prediction result or one instance of a confusion matrix represents one point in the ROC space. The best possible prediction method would yield a point in the upper left corner or coordinate (0,1) of the ROC space, representing 100% TP/P and 0% FP/N. The (0,1) point is also called a perfect classification. A completely random guess would give a point along a diagonal line from the left bottom to the top right corners.

The diagonal divides the ROC space. Points above the diagonal represent good classification results, points below the line represent poor results. Note that the Find Threshold operator finds a threshold where points of bad classification are inverted to convert them to good classification.

Input Ports

data set (exa) This input port expects a labeled ExampleSet. The ExampleSet should have label and prediction attributes as well as attributes for the confidence of predictions.

Output Ports

data set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
7. Modeling

**threshold** (*thr*) The threshold is delivered through this output port. Frequently, the Apply Threshold operator is used for applying this threshold on the soft classified data.

**Parameters**

**define labels** (*boolean*) This is an expert parameter. If set to true, the first and second label can be defined explicitly using the *first label* and *second label* parameters.

**first label** (*string*) This parameter is only available when the *define labels* parameter is set to true. It explicitly defines the first label.

**second label** (*string*) This parameter is only available when the *define labels* parameter is set to true. It explicitly defines the second label.

**misclassification costs first** (*real*) This parameter specifies the costs assigned when an example of the first class is misclassified as one of the second.

**misclassification costs second** (*real*) This parameter specifies the costs assigned when an example of the second class is misclassified as one of the first.

**show roc plot** (*boolean*) This parameter indicates whether to display a plot of the ROC curve.

**use example weights** (*boolean*) This parameter indicates if example weights should be used.

**roc bias** (*selection*) This is an expert parameter. It determines how the ROC (and AUC) are evaluated.

**Tutorial Processes**

**Introduction to the Find Threshold operator**

This Example Process starts with a Subprocess operator. This subprocess provides the labeled ExampleSet. Double-click on the Subprocess operator to see
what is happening inside although it is not directly relevant to the understanding of the Find Threshold operator. In the subprocess, the Generate Data operator is used for generation of testing and training data sets with binominal label. The SVM classification model is learned and applied on training and testing data sets respectively. The resultant labeled ExampleSet is output of this subprocess. A breakpoint is inserted after this subprocess so that you can have a look at the labeled ExampleSet before application of the Find Threshold operator. You can see that the ExampleSet has 500 examples. If you sort the results according to the confidence of positive prediction, and scroll through the data set, you will see that all examples with 'confidence(positive)' greater than 0.500 are classified as positive and all examples with 'confidence(positive)' less than 0.500 are classified as negative.

Now have a look at what is happening outside the subprocess. The Find Threshold operator is used for finding a threshold. All its parameters are used with default values. The Find Threshold operator delivers a threshold through the threshold port. This threshold is applied on the labeled ExampleSet using the Apply Threshold operator. We know that when the Apply Threshold operator is applied on an ExampleSet, if the confidence for the second class is greater than the given threshold the prediction is set to this class otherwise it is set to the other class. Have a look at the resultant ExampleSet. Sort the ExampleSet according to 'confidence(positive)' and scroll through the ExampleSet. You will see that all examples where 'confidence(positive)' is greater than 0.306 are classified as positive and all examples where 'confidence(positive)' is less than or equal to 0.306 are classified as negative. In the original ExampleSet the boundary value was 0.500 but the Find Threshold operator found a better threshold for a crisp classification of soft classified data.
Create Threshold

This operator creates a user defined threshold for crisp classification based on the prediction confidences (soft predictions). This threshold can be applied by using the Apply Threshold operator.

Description

The \textit{threshold} parameter specifies the required threshold. The \textit{first class} and \textit{second class} parameters are used for specifying the classes of the ExampleSet that should be considered as first and second class respectively. The threshold created by this operator can be applied on the labeled ExampleSet using the Apply Threshold operator. Should it occur that the confidence for the second class is greater than the given threshold then the prediction is set to this second class otherwise it is set to the first class. This can be easily understood by studying the attached Example Process.

The Apply Threshold operator applies the given threshold to a labeled ExampleSet and maps a soft prediction to crisp values. The threshold is provided through the \textit{threshold} port. Mostly the Create Threshold operator is used for creating thresholds before they are applied using the Apply Threshold operator.

Among various classification methods, there are two main groups of methods: soft
and hard classification. In particular, a soft classification rule generally estimates the class conditional probabilities explicitly and then makes the class prediction based on the largest estimated probability. In contrast, hard classification bypasses the requirement of class probability estimation and directly estimates the classification boundary.

Output Ports

output (out) This port delivers the threshold. This threshold can be applied on a labeled ExampleSet by using the Apply Threshold operator.

Parameters

threshold (real) This parameter specifies the threshold of the prediction confidence. It should be in range 0.0 to 1.0. If the prediction confidence for the second class is greater than this threshold the prediction is set to second class (i.e. the class specified through the second class parameter) otherwise it is set to the first class (i.e. the class specified through the first class parameter).

first class (string) This parameter specifies the class which should be considered as the first class.

second class (string) This parameter specifies the class which should be considered as the second class.

Tutorial Processes

Creating and Applying thresholds

This Example Process starts with a Subprocess operator. This subprocess provides the labeled ExampleSet. Double-click on the Subprocess operator to see
what is happening inside the subprocess although it is not directly relevant to
the use of the Create Threshold operator. In the subprocess, the K-NN classification model is learned and applied on different samples of the 'Weighting' data set. The resultant labeled ExampleSet is output of this subprocess. A breakpoint is inserted after this subprocess so that you can have a look at the labeled ExampleSet before the application of the Create Threshold and Apply Threshold operators. You can see that the ExampleSet has 20 examples. 11 of them are predicted as 'positive' and the remaining 9 examples are predicted as 'negative'. If you sort the results according to the confidence of positive prediction, you will easily see that among 11 examples predicted as 'positive', 3 examples have confidence 0.600, 4 examples have confidence 0.700, 3 examples have confidence 0.800 and 1 example has confidence 0.900.

Now let us have a look at what is happening outside the subprocess. The Create Threshold operator is used for creating a threshold. The threshold parameter is set to 0.700 and the first class and second class parameters are set to 'negative' and 'positive' respectively. A breakpoint is inserted here so that you can see the threshold in the Results Workspace. This statement in the Results Workspace explains everything:

if confidence(positive) > 0.7 then positive; else negative

This statement means that if confidence(positive) is greater than 0.7 then the class should be predicted as positive otherwise it should be predicted as negative. In a general form this statement would look something like this:

if confidence(second) > T then second; else first.

where T, second and first are the values of the threshold, second class and first class parameters respectively.

This threshold is applied on the labeled ExampleSet using the Apply Threshold operator. We know that when the Apply Threshold operator is applied on an ExampleSet there are two possibilities: if the confidence for the second class is greater than the given threshold the prediction is set to second otherwise to the first class. In this process, if the confidence for the second class i.e. 'positive' (class specified in the second class parameter of the Create Threshold operator)
is greater than the given threshold i.e. 0.700 (threshold specified in the *threshold* parameter of the Create Threshold operator) the prediction is set to 'positive' otherwise it is set to 'negative'. In the labeled ExampleSet only 4 examples had confidence (positive) greater than 0.700. When the Apply Threshold operator is applied only these 4 examples are assigned 'positive' predictions and all other examples are assigned 'negative' predictions.

**Apply Threshold**

This operator applies a threshold on soft classified data.

**Description**

The Apply Threshold operator applies the given threshold to a labeled ExampleSet and maps a soft prediction to crisp values. The threshold is provided through the *threshold* port. Mostly the Create Threshold operator is used for creating thresholds before it is applied using the Apply Threshold operator. If the confidence for the second class is greater than the given threshold the predic-
7. Modeling

tion is set to this class otherwise it is set to the other class. This can be easily understood by studying the attached Example Process.

Among various classification methods, there are two main groups of methods: soft and hard classification. In particular, a soft classification rule generally estimates the class conditional probabilities explicitly and then makes the class prediction based on the largest estimated probability. In contrast, hard classification bypasses the requirement of class probability estimation and directly estimates the classification boundary.

Input Ports

example set (exa) This input port expects a labeled ExampleSet. The ExampleSet should have label and prediction attributes as well as attributes for confidence of predictions.

threshold (thr) The threshold is provided through this input port. Frequently, the Create Threshold operator is used for providing threshold at this port.

Output Ports

example set (exa) The predictions of the input ExampleSet are changed according to the threshold given at the threshold port and the modified ExampleSet is delivered through this port.

Tutorial Processes

Creating and Applying thresholds

This Example Process starts with a Subprocess operator. This subprocess provides the labeled ExampleSet. Double-click on the Subprocess operator to see
what is happening inside the subprocess although it is not directly relevant to the use of the Apply Threshold operator. In the subprocess, the K-NN classification model is learned and applied on different samples of the 'Weighting' data set. The resultant labeled ExampleSet is output of this subprocess. A breakpoint is inserted after this subprocess so that you can have a look at the labeled ExampleSet before the application of the Apply Threshold operator. You can see that the ExampleSet has 20 examples. 11 of them are predicted as 'positive' and the remaining 9 examples are predicted as 'negative'. If you sort the results according to the confidence of positive prediction, you will easily see that among 11 examples predicted as 'positive', 3 examples have confidence 0.600, 4 examples have confidence 0.700, 3 examples have confidence 0.800 and 1 example has confidence 0.900.

Now let us have a look at what is happening outside the subprocess. The Create Threshold operator is used for creating a threshold. The threshold parameter is set to 0.700 and the first class and second class parameters are set to 'negative' and 'positive' respectively. This threshold is applied on the labeled ExampleSet using the Apply Threshold operator. We know that when the Apply Threshold operator is applied on an ExampleSet, if the confidence for the second class is greater than the given threshold then the prediction is set to this class otherwise it is set to the other class. In this process, if the confidence for the second class i.e. 'positive' (class specified in the second class parameter of the Create Threshold operator) is greater than the given threshold i.e. 0.700 (threshold specified in the threshold parameter of the Create Threshold operator) the prediction is set to 'positive' otherwise it is set to 'negative'. In the labeled ExampleSet only 4 examples had confidence (positive) greater than 0.700. When the Apply Threshold operator is applied only these 4 examples are assigned 'positive' prediction and all other examples are assigned 'negative' predictions.
Drop Uncertain Predictions

This operator sets all predictions to 'unknown' (missing value) if the corresponding confidence is less than the specified minimum confidence. This operator is used for dropping predictions with low confidence values.

Description

The Drop Uncertain Predictions operator expects a labeled ExampleSet i.e. an ExampleSet with label and prediction attributes along with prediction confidences. The minimum confidence threshold is specified through the \textit{min confidence} parameter. All those predictions of the given ExampleSet are dropped where the corresponding prediction confidence is below the specified threshold. Suppose an ExampleSet with two possible classes 'positive' and 'negative'. If the \textit{min confidence} parameter is set to 0.700, all the examples that were predicted as
'positive' but their corresponding 'confidence (positive)' value is less than 0.700 are classified as missing values. Similarly the label value is set to missing value for all those examples that were predicted as 'negative' but their corresponding confidence '(negative)' value is less than 0.700. This operator also allows you to define different minimum confidence thresholds for different classes through the \textit{min confidences} parameter.

\textbf{Input Ports}

\textbf{example set input (exa)} This input port expects a labeled ExampleSet. It is the output of the Apply Model operator in the attached Example Process. The output of other operators can also be used as input if it is a labeled ExampleSet.

\textbf{Output Ports}

\textbf{example set output (exa)} The uncertain predictions are dropped and the resultant ExampleSet is delivered through this port.

\textbf{original (ori)} The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

\textbf{Parameters}

\textbf{class handling (selection)} This parameter specifies the mode of class handling which defines if all classes are handled equally or if individual class thresholds are set.

- \textbf{balanced} In this case all classes are handled equally i.e. the same confidence threshold is applied on all possible values of the label. The minimum confidence threshold is specified through the \textit{min confidence} parameter.
7. Modeling

- **unbalanced** In this case classes are not handled equally i.e. different confidence thresholds can be specified for different classes through the `min confidences` parameter.

**min confidence** *(real)* This parameter is only available when the `class handling` parameter is set to 'balanced'. This parameter sets the minimum confidence threshold for all the classes. Predictions below this confidence will be dropped.

**min confidences** *(list)* This parameter is only available when the `class handling` parameter is set to 'unbalanced'. This parameter specifies individual thresholds for classes. Predictions below these confidences will be dropped.

---

**Tutorial Processes**

**Dropping uncertain predictions of the Naive Bayes operator**

The 'Golf' data set is loaded using the Retrieve operator. The Naive Bayes operator is applied on it to generate a classification model. The resultant classification model is applied on the 'Golf-Testset' data set by using the Apply Model operator. A breakpoint is inserted here so that you can see the labeled ExampleSet generated by the Apply Model operator. You can see that 10 examples have been classified as 'yes' but only 6 of them have 'confidence (yes)' above 0.700. Only 2 examples have been classified as 'no' but only 1 of them has 'confidence (no)' above 0.700. This labeled ExampleSet is provided to the Drop Uncertain Predictions operator. The `min confidence` parameter is set to 0.7. Thus all the examples where the prediction confidence is below 0.7 are set to missing values. This can be seen in the Results Workspace. 7 examples had a prediction confidence below 0.7 and all of them have been dropped.
7.7. Model Application
8 Evaluation

Split Validation

This operator performs a simple validation i.e. randomly splits up the ExampleSet into a training set and test set and evaluates the model. This operator performs a split validation in order to estimate the performance of a learning operator (usually on unseen data sets). It is mainly used to estimate how accurately a model (learnt by a particular learning operator) will perform in practice.

Description

The Split Validation operator is a nested operator. It has two subprocesses: a training subprocess and a testing subprocess. The training subprocess is used for learning or building a model. The trained model is then applied in the testing subprocess. The performance of the model is also measured during the testing phase.

The input ExampleSet is partitioned into two subsets. One subset is used as the training set and the other one is used as the test set. The size of two subsets can be adjusted through different parameters. The model is learned on the training set and is then applied on the test set. This is done in a single iteration, as
8. Evaluation

compared to the X-Validation operator that iterates a number of times using different subsets for testing and training purposes.

Usually the learning process optimizes the model parameters to make the model fit the training data as well as possible. If we then take an independent sample of testing data, it will generally turn out that the model does not fit the testing data as well as it fits the training data. This is called 'over-fitting', and is particularly likely to happen when the size of the training data set is small, or when the number of parameters in the model is large. Split Validation is a way to predict the fit of a model to a hypothetical testing set when an explicit testing set is not available. The Split Validation operator also allows training on one data set and testing on another explicit testing data set.

Input Ports

training example set (tra) This input port expects an ExampleSet for training a model (training data set). The same ExampleSet will be used during the testing subprocess for testing the model if no other data set is provided.

Output Ports

model (mod) The training subprocess must return a model, which is trained on the input ExampleSet. Please note that the model built on the complete input ExampleSet is delivered from this port.

training example set (tra) The ExampleSet that was given as input at the training input port is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

averagable (ave) The testing subprocess must return a Performance Vector. This is usually generated by applying the model and measuring its performance. Two such ports are provided but more can also be used if required. Please note that the performance calculated by this estimation scheme is only an estimate.
8.1. Validation

(instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

Parameters

split (selection) This parameter specifies how the ExampleSet should be split

- relative If a relative split is required, the relative size of the training set should be provided in the split ratio parameter. Afterwards the relative size of the test set is automatically calculated by subtracting the value of the split ratio from 1.

- absolute If an absolute split is required, you have to specify the exact number of examples to use in the training or test set in the training set size parameter or in the test set size parameter. If either of these parameters is set to -1, its value is calculated automatically using the other one.

split ratio (real) This parameter is only available when the split parameter is set to 'relative'. It specifies the relative size of the training set. It should be between 1 and 0, where 1 means that the entire ExampleSet will be used as training set.

training set size (integer) This parameter is only available when the split parameter is set to 'absolute'. It specifies the exact number of examples to be used as training set. If it is set to -1, the test size set number of examples will be used for the test set and the remaining examples will be used as training set.

test set size (integer) This parameter is only available when the split parameter is set to 'absolute'. It specifies the exact number of examples to be used as test set. If it is set to -1, the training size set number of examples will be used for training set and the remaining examples will be used as test set.

sampling type (selection) The Split Validation operator can use several types of sampling for building the subsets. Following options are available:

- Linear sampling Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.
8. Evaluation

- **Shuffled sampling** Shuffled sampling builds random subsets of ExampleSet. Examples are chosen randomly for making subsets.

- **Stratified sampling** Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class labels.

**use local random seed** *(boolean)* Indicates if a local random seed should be used for randomizing examples of a subset. Using the same value of local random seed will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed** *(integer)* This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

## Tutorial Processes

### Validating Models using Split Validation

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to uniquely identify examples. This is done so that you can understand this process easily; otherwise IDs are not required here. A breakpoint is added after this operator so that you can preview the data before the Split Validation operator starts. Double click the Split Validation operator and you will see training and testing subprocesses. The Decision Tree operator is used in the training subprocess. The trained model (i.e. Decision Tree) is passed to the testing subprocess through the model ports. The testing subprocess receives testing data from the testing port.
8.1. Validation

Now, have a look at the parameters of the Split Validation operator. The `split` parameter is set to 'absolute'. The `training set size` parameter is set to 10 and the `test set size` parameter is set to -1. As there are 14 total examples in the 'Golf' data set, the test set automatically gets 4 remaining examples. The `sampling type` parameter is set to `Linear Sampling`. Remaining parameters have default values. Thus two subsets of the 'Golf' data set will be created. You will observe later that these two subsets are created:

- training set: examples with IDs 1 to 10 (10 examples)
- test set: examples with IDs 11 to 14 (4 examples)

You can see that all examples in a subset are consecutive (i.e. with consecutive IDs). This is because `Linear Sampling` is used.

Breakpoints are inserted to make you understand the process. Here is what happens when you run the process:

- First the 'Golf' data set is displayed with all rows uniquely identified using the ID attribute. There are 14 rows with ids 1 to 14. Press the green-colored Run button to continue.

- Now a Decision tree is shown. This was trained from the training set of the 'Golf' data set. Hit the Run button to continue.

- The Decision tree was applied on the testing data. Here you can see the results after application of the Decision Tree model. Have a look at IDs of the testing data here. They are 11 to 14. Compare the `label` and `prediction` columns and you will see that only 2 predictions out of 4 are correct (only ID 1 and 3 are correct predictions). Hit the Run button again.

- Now the Performance Vector of the Decision tree is shown. As only 2 out of 4 predictions were correct, the accuracy is 50%. Press the Run button again.

- Now you can see a different Decision tree. It was trained on the complete 'Golf' data set that is why it is different from the previous decision tree.
8. Evaluation

You can run the same process with different values of *sampling type* parameter. If *linear sampling* is used, as in our example process, you will see that IDs of examples in subsets will be consecutive values. If *shuffled sampling* is used you will see that IDs of examples in subsets will be random values. If *stratified sampling* is used you will see that IDs of examples in subsets will be random values but the class distribution in the subsets will be nearly the same as in the whole 'Golf' data set.

To get an understanding of how objects are passed using *through* ports please study the Example Process of X-Validation operator.

---

### X-Validation

This operator performs a cross-validation in order to estimate the statistical performance of a learning operator (usually on unseen data sets). It is mainly used to estimate how accurately a model (learnt by a particular learning operator) will perform in practice.

### Description

The X-Validation operator is a nested operator. It has two subprocesses: a training subprocess and a testing subprocess. The training subprocess is used for training a model. The trained model is then applied in the testing subprocess. The performance of the model is also measured during the testing phase.
8.1. Validation

The input ExampleSet is partitioned into $k$ subsets of equal size. Of the $k$ subsets, a single subset is retained as the testing data set (i.e. input of the testing subprocess), and the remaining $k - 1$ subsets are used as training data set (i.e. input of the training subprocess). The cross-validation process is then repeated $k$ times, with each of the $k$ subsets used exactly once as the testing data. The $k$ results from the $k$ iterations then can be averaged (or otherwise combined) to produce a single estimation. The value $k$ can be adjusted using the number of validations parameter.

Usually the learning process optimizes the model parameters to make the model fit the training data as well as possible. If we then take an independent sample of testing data, it will generally turn out that the model does not fit the testing data as well as it fits the training data. This is called 'over-fitting', and is particularly likely to happen when the size of the training data set is small, or when the number of parameters in the model is large. Cross-validation is a way to predict the fit of a model to a hypothetical testing set when an explicit testing set is not available.

Differentiation

X-Prediction The X-Validation and X-Prediction operators work in the same way. The major difference is the objects returned by these operators. The X-Validation operator returns a performance vector whereas the X-Prediction operator returns labeled ExampleSet. See page 66 for details.

Input Ports

training example set (tra) This input port expects an ExampleSet for training a model (training data set). The same ExampleSet will be used during the testing subprocess for testing the model.
8. Evaluation

Output Ports

**model** (*mod*) The training subprocess must return a model, which is trained on the input ExampleSet. Please note that model built on the complete input ExampleSet is delivered from this port.

**training example set** (*tra*) The ExampleSet that was given as input at the training input port is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**averagable** (*ave*) The testing subprocess must return a Performance Vector. This is usually generated by applying the model and measuring its performance. Two such ports are provided but more can also be used if required. Please note that the statistical performance calculated by this estimation scheme is only an estimate (instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

Parameters

**average performances only** (*boolean*) This is an expert parameter which indicates if only performance vectors should be averaged or all types of averagable result vectors.

**leave one out** (*boolean*) As the name suggests, the leave one out cross-validation involves using a single example from the original ExampleSet as the testing data (in testing subprocess), and the remaining examples as the training data (in training subprocess). This is repeated such that each example in the ExampleSet is used once as the testing data. Thus, it is repeated 'n' number of times, where 'n' is the total number of examples in the ExampleSet. This is the same as applying the X-Validation operator with the number of validations parameter set equal to the number of examples in the original ExampleSet. This is usually very expensive for large ExampleSets from a computational point of view because the training process is repeated a large number of times (number of examples time). If set to true, the number of validations parameter is ignored.
8.1. Validation

**number of validations** *(integer)* This parameter specifies the number of subsets the ExampleSet should be divided into (each subset has equal number of examples). Also the same number of iterations will take place. Each iteration involves training a model and testing that model. If this is set equal to total number of examples in the ExampleSet, it is equivalent to the X-Validation operator with the *leave one out* parameter set to true.

**sampling type** *(selection)* The X-Validation operator can use several types of sampling for building the subsets. Following options are available:

- **linear_sampling** The Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.

- **shuffled_sampling** The Shuffled sampling builds random subsets of ExampleSet. Examples are chosen randomly for making subsets.

- **stratified_sampling** The Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets such that each subset contains roughly the same proportions of the two values of class labels.

**use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of the *local random seed* will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is available only if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is available only if the *use local random seed* parameter is set to true.
8. Evaluation

Related Documents

X-Prediction (66)

Tutorial Processes

Validating Models using X-Validation

The Golf data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to uniquely identify examples. This is done so that you can understand this process easily; otherwise IDs are not required here. The breakpoint is added after this operator so that you can preview the data before the X-Validation operator starts. Double click the X-Validation operator and you will see the training and testing subprocesses. The Decision Tree operator is used in the training subprocess. The trained model (i.e. Decision Tree) is passed to the testing subprocess through the model ports. The testing subprocess receives testing data from the testing port.

Now, have a look at the parameters of the X-Validation operator. The no of validations parameter is set to 3 and the sampling type parameter is set to linear sampling. Remaining parameters have default values. The no of validations is set to 3, which implies that 3 subsets of the Golf data set will be created. You will observe later that these three subsets are created:

- sub1: examples with IDs 1 to 5 (5 examples)
- sub2: examples with IDs 6 to 9 (4 examples)
- sub3: examples with IDs 10 to 14 (5 examples)

You can see that all examples in a subset are consecutive (i.e. with consecutive IDs). This is because linear sampling is used. Also note that all subsets have almost an equal number of elements. An exactly equal number of elements was
8.1. Validation

not possible because 14 examples could not be divided equally in 3 subsets.

As the no of validations parameter is set to 3, there will be three iterations.

- Iteration 1: A model (decision tree) will be trained on sub2 and sub3 during the training subprocess. Trained model will be applied on sub1 during testing subprocess.

- Iteration 2: A model (decision tree) will be trained on sub1 and sub3 during the training subprocess. Trained model will be applied on sub2 during testing subprocess.

- Iteration 3: A model (decision tree) will be trained on sub1 and sub2 during the training subprocess. Trained model will be applied on sub3 during testing subprocess.

Breakpoints are inserted to make you understand the process. Here is what happens when you run the process:

- First the Golf data set is displayed with all rows uniquely identified using the ID attribute. There are 14 rows with ids 1 to 14. Press the green-colored Run button to continue.

- Now a Decision tree is shown. This was trained from a subset (combination of sub2 and sub3) of the Golf data set. Press the Run button to continue.

- The Decision tree was applied on the testing data. Testing data for this iteration was sub1. Here you can see the results after application of the Decision Tree model. Have a look at the IDs of the testing data here. They are 1 to 5. This means that the tree was trained on the remaining examples i.e. examples with IDs 6 to 14 thus sub2 + sub3. Compare the label and prediction columns and you will see that only 1 prediction out of 5 is correct (only ID 3 has a correct prediction). Press the Run button again.

- Now the Performance Vector of the Decision tree is shown. As only 1 out of 5 predictions was correct, the accuracy is 20%. Press the Run button again.
8. Evaluation

- Now you can see a different Decision tree. It was trained on another subset of Golf data set that is why it is different from the previous decision tree. Keep pressing the Run button and you will see testing data and the Performance Vector for this tree. This process will repeat 3 times because there will be three iterations because the *number of validations* parameter was set to 3.

- At the end of 3 iterations, you will see the Average Performance Vector in the Results Workspace it averages all the performance vectors. Accuracy was 20%, 50% and 60% respectively in the three iterations. Thus the Average Performance Vector has accuracy = 43.33% (i.e. accuracy = (20+50+60)/3).

You can run the same process with different values of the *sampling type* parameter. If *linear sampling* is used, as in our example process, you will see that the IDs of the examples in the subsets will be consecutive values. If *shuffled sampling* is used you will see that the IDs of the examples in the subsets will be randomized. If *stratified sampling* is used you will also see randomized IDs but the class distribution in the subsets will be nearly the same as in the whole Golf data set.

Passing results from training to testing process using through ports

This process is similar to the first process, but it will apply a weighting operator for selecting a subset of the attributes before applying the Decision Tree operator. The focus of this Example Process is to highlight the usage of *through*
8.1. Validation

ports.

Please note the use of *through* ports for transferring objects between the training and testing subprocesses. The results generated during training, that have to be applied in the same way on the test set before the model can be applied, may be passed using the *through* ports.

Have a look at the subprocesses. In the training subprocess, the Weight by Correlation operator is applied on the training data set. Then the Select by Weights operator is applied on it. Then the Decision Tree operator is applied on it. Note the use of the *through* ports. The *through* ports are used to transfer the weights from the training subprocess to the testing subprocess. The Select by Weights operator is applied on the testing data set using the same parameter values as used in the Select by Weights operator of the training subprocess. The Select by Weights operator uses the weights transferred through the *through* ports.

---

**Bootstrapping Validation**

This operator performs validation after bootstrapping a sampling of training data set in order to estimate the statistical performance of a learning operator (usually on unseen data sets). It is mainly used to estimate how accurately a model (learnt by a particular learning operator) will perform in practice.
Description

The Bootstrapping Validation operator is a nested operator. It has two subprocesses: a training subprocess and a testing subprocess. The training subprocess is used for training a model. The trained model is then applied in the testing subprocess. The performance of the model is also measured during the testing phase. The training subprocess must provide a model and the testing subprocess must provide a performance vector.

The input ExampleSet is partitioned into two subsets. One subset is used as the training set and the other one is used as the test set. The size of two subsets can be adjusted through the sample ratio parameter. The sample ratio parameter specifies the ratio of examples to be used in the training set. The ratio of examples in the testing set is automatically calculated as $1-n$ where $n$ is the ratio of examples in the training set. The important thing to note here is that this operator performs bootstrapping sampling (explained in the next paragraph) on the training set before training a model. The model is learned on the training set and is then applied on the test set. This process is repeated $m$ number of times where $m$ is the value of the number of validations parameter.

Bootstrapping sampling is sampling with replacement. In sampling with replacement, at every step all examples have equal probability of being selected. Once an example has been selected for the sample, it remains candidate for selection and it can be selected again in any other coming steps. Thus a sample with replacement can have the same example multiple number of times. More importantly, a sample with replacement can be used to generate a sample that is greater in size than the original ExampleSet.

Usually the learning process optimizes the model parameters to make the model fit the training data as well as possible. If we then take an independent sample of testing data, it will generally turn out that the model does not fit the testing data as well as it fits the training data. This is called 'over-fitting', and is particularly likely to happen when the size of the training data set is small, or when the number of parameters in the model is large. Bootstrapping Validation is a way to predict the fit of a model to a hypothetical testing set when an explicit testing
8.1. Validation

set is not available.

**Differentiation**

**Split Validation** Its validation subprocess executes just once. It provides linear, shuffled and stratified sampling. See page 891 for details.

**X-Validation** The input ExampleSet is partitioned into $k$ subsets of equal size. Of the $k$ subsets, a single subset is retained as the testing data set (i.e. input of the testing subprocess), and the remaining $k - 1$ subsets are used as the training data set (i.e. input of the training subprocess). The cross-validation process is then repeated $k$ times, with each of the $k$ subsets used exactly once as the testing data. The $k$ results from the $k$ iterations can then be averaged (or otherwise combined) to produce a single estimation. See page 896 for details.

**Input Ports**

**training (tra)** This input port expects an ExampleSet for training a model (training data set). The same ExampleSet will be used during the testing subprocess for testing the model.

**Output Ports**

**model (mod)** The training subprocess must return a model, which is trained on the input ExampleSet. Please note that model built on the complete input ExampleSet is delivered from this port.

**training (tra)** The ExampleSet that was given as input at the training input port is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**averagable (ave)** The testing subprocess must return a Performance Vector.
8. Evaluation

This is usually generated by applying the model and measuring its performance. Two such ports are provided but more can also be used if required. Please note that the statistical performance calculated by this estimation scheme is only an estimate (instead of an exact calculation) of the performance which would be achieved with the model built on the complete delivered data set.

Parameters

**number of validations** *(integer)* This parameter specifies the number of times the validation should be repeated i.e. the number of times the inner subprocess should be executed.

**sample ratio** *(real)* This parameter specifies the relative size of the training set. In other validation schemes this parameter should be between 1 and 0, where 1 means that the entire ExampleSet will be used as training set. In this operator its value can be greater than 1 because bootstrapping sampling can generate an ExampleSet with a number of examples greater than the original ExampleSet. All examples that are not selected for the training set are automatically selected for the test set.

**use weights** *(boolean)* If this parameter is checked, example weights will be used for bootstrapping if such weights are available.

**average performances only** *(boolean)* This parameter indicates if only performance vectors should be averaged or all types of averagable result vectors.

**use local random seed** *(boolean)* This parameter indicates if a *local random seed* should be used for randomizing examples of a subset. Using the same value of the *local random seed* will produce the same samples. Changing the value of this parameter changes the way examples are randomized, thus samples will have a different set of examples.

**local random seed** *(integer)* This parameter specifies the *local random seed*. This parameter is only available if the *use local random seed* parameter is set to true.
Related Documents

Split Validation (891)
X-Validation (896)

Tutorial Processes

Validating Models using Bootstrapping Validation

The 'Golf' data set is loaded using the Retrieve operator. The Generate ID operator is applied on it to uniquely identify examples. This is done so that you can understand this process easily; otherwise IDs are not required here. A breakpoint is added after this operator so that you can preview the data before the application of the Bootstrapping Validation operator. You can see that the ExampleSet has 14 examples with ids from 1 to 14. Double click the Bootstrapping Validation operator and you will see the training and testing subprocesses. The Decision Tree operator is used in the training subprocess. The trained model (i.e. Decision Tree) is passed to the testing subprocess through the model ports. The testing subprocess receives testing data from the testing port.

Now, have a look at the parameters of the Bootstrapping Validation operator. The no of validations parameter is set to 2 thus the inner subprocess will execute just twice. The sample ratio parameter is set to 0.5. The number of examples in the ExampleSet is 14 and sample ratio is 0.5, thus the training set will be composed of 7 (i.e. 14 x 0.5) examples. But it is not necessary that these examples will be unique because bootstrapping sampling can select an example multiple number of time. All the examples that are not selected for the training set automatically become part of the testing set. You can verify this by running the process. You will see that the training set has 7 examples but they are not all unique and all the examples that were not part of the training set are part of the testing set.
Performance

This operator is used for performance evaluation. This operator delivers a list of performance criteria values. These performance criteria are automatically determined in order to fit the learning task type.

Description

In contrast to the other performance evaluation operators like Performance (Classification) operator, Performance (Binominal Classification) operator or Performance (Regression) operator, this operator can be used for all types of learning tasks. It automatically determines the learning task type and calculates the most common criteria for that type. For more sophisticated performance calculations, you should use the operators mentioned above. If none of them meets your requirements, you can use Performance (User-Based) operator which allows you to write your own performance measure.

The following criteria are added for binominal classification tasks:

- Accuracy
- Precision
8.2. Performance Measurement

- Recall
- AUC (optimistic)
- AUC (neutral)
- AUC (pessimistic)

The following criteria are added for polynominal classification tasks:

- Accuracy
- Kappa statistic

The following criteria are added for regression tasks:

- Root Mean Squared Error
- Mean Squared Error

Input Ports

labelled data (lab) This input port expects a labelled ExampleSet. Apply Model is a good example of such operators that provide labeled data. Make sure that the ExampleSet has label attribute and prediction attribute. See the Set Role operator for more details.

performance (per) This is an optional parameter. It requires a Performance Vector.

Output Ports

performance (per) This port delivers a Performance Vector (we call it output-performance-vector for now). Performance Vector is a list of performance criteria values. The output-performance-vector contains performance criteria calculated by this Performance operator (we call it calculated-performance-vector here). If
8. Evaluation

A Performance Vector was also fed at performance input port (we call it input-performance-vector here), criteria of input-performance-vector are also added in the output-performance-vector. If input-performance-vector and calculated-performance-vector both have same criteria but with different values, the values of calculated-performance-vector are delivered through the output port. This concept can be easily understood by studying the attached Example Process.

**Example Set** *(exa)* ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

### Parameters

**use example weights** *(boolean)* This parameter allows example weights to be used for performance calculations if possible. This parameter has no effect if no attribute has weight role. In order to consider weights of examples the ExampleSet should have an attribute with weight role. Several operators are available that assign weights e.g. Generate Weights operator. Study Set Roles operator for more information regarding weight roles.

### Tutorial Processes

**Assessing the performance of a prediction**

This process is composed of two Subprocess operators and one Performance operator. Double click on the first Subprocess operator and you will see the operators within this subprocess. The first subprocess 'Subprocess (labeled data provider)' loads Golf data set using Retrieve operator and then learns a classification model using k-NN operator. Then the learnt model is applied on Golf-Testset data set using Apply Model operator. Then Generate Weight operator is used to add an attribute with weight role. Thus, this subprocess provides a labeled Example-
Set with *weight* attribute. Breakpoint is inserted after this subprocess to show this ExampleSet. This ExampleSet is provided at *labeled data* input port of the Performance operator in main process.

The second Subprocess operator 'Subprocess (performance vector provider)' loads Golf data set using Retrieve operator and then learns a classification model using k-NN operator. Then the learnt model is applied on Golf data set using Apply Model operator. Then Performance (Classification) operator is applied on the labeled data to produce a Performance Vector. Breakpoint is inserted after this subprocess to show this Performance Vector. Note that this model was trained and tested on the same data set (Golf data set), so its accuracy is 100%. Thus this subprocess provides a Performance Vector with 100% accuracy and 0.00% classification error. This Performance Vector is connected to *performance* input port of the Performance operator in main process.

When you run the process, first you will see an ExampleSet which is output of the first Subprocess operator. Press the Run button again and now you will see a Performance Vector. This is output of the second Subprocess operator. Press the Run button again and you will see various criteria in *criterion selector* window in Results Workspace. These include classification error, accuracy, precision, recall, AUC (optimistic), AUC and AUC (pessimistic). Now select accuracy from *criterion selector* window, its value is 71.43%. On the contrary the accuracy of input Performance Vector provided by second subprocess was 100%. Accuracy of final Performance Vector is 71.43% instead of 100% because if input Performance Vector and calculated Performance Vector both have same criteria but with different values, the values of calculated Performance Vector are delivered through the output port. Now, note that *classification error* criterion is added to criteria list because of the Performance Vector provided at *performance* input port. Disable the second Subprocess operator and run the same process again, you will see that *classification error* criterion does not appear now. This is because if a Performance Vector is fed at *performance* input port, its criteria are also added to the output Performance Vector.

Accuracy is calculated by taking the percentage of correct predictions over the total number of examples. Correct prediction means examples where value of
8. Evaluation

*prediction* attribute is equal to the value of *label* attribute. If you look at the ExampleSet in Results Workspace, you can see that there are 14 examples in this data set. 10 out 14 examples are correct predictions i.e. their *label* and *prediction* attributes have the same values. This is why accuracy was 71.43% (10 x 100 /14 = 71.43%). Now run the same process again but this time set *use example weights* parameter to true. Check the results again. They have changed now because the weight of each example was taken into account this time. Accuracy is 68.89% this time. If you take percentage of weight of correct predictions and total weight you get the same answer (0.6889 x 100/1 = 68.89%). In this Example Process, using weights reduced the accuracy but this is not always the case.

Note: This Example Process is just for highlighting different perspectives of Performance operator. It may not be very useful in real scenarios.

![Diagram](image)

**Extract Performance**

This operator can be used for deriving a performance measure (in form of a performance vector) from the given ExampleSet.
8.2. Performance Measurement

Description

This operator can be used for generating a performance vector from the properties of the given ExampleSet. This includes properties like the number of examples or number of attributes of the input ExampleSet. Specific data value of the input ExampleSet can also be used as the value of the performance vector. Various statistical properties of the input ExampleSet e.g. average, min or max value of an attribute can also be used as the value of the performance vector. All these options can be understood by studying the parameters and the attached Example Process.

Input Ports

example set (exa) This input port expects an ExampleSet. The performance vector value will be extracted from this ExampleSet.

Output Ports

performance (per) This port delivers a performance vector. A performance vector is a list of performance criteria values.
example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators.

Parameters

performance type (selection) This parameter indicates the way the input ExampleSet should be used to define the performance vector.

- number_of_examples If this option is selected, the performance vector
8. Evaluation

value is set to the total number of examples in the input ExampleSet.

- **number_of_attributes** If this option is selected, the performance vector value is set to the total number of attributes in the input ExampleSet.

- **data_value** If this option is selected, the performance vector value is set to the value of the specified attribute at the specified index. The attribute is specified using the `attribute name` parameter and the index is specified using the `example index` parameter.

- **statistics** If this option is selected, the performance vector value is set to the value obtained by applying the selected statistical operation on the specified attribute. The attribute is specified using the `attribute name` parameter and the statistical operation is selected using the `statistics` parameter.

**statistics** (*selection*) This parameter is only available when the `performance type` parameter is set to 'statistics'. This parameter allows you to select the statistical operation to be applied on the attribute specified by the `attribute name` parameter.

**attribute name** (*string*) This parameter is only available when the `performance type` parameter is set to 'statistics' or 'data value'. This parameter allows you to select the required attribute.

**attribute value** (*string*) This parameter is only available when the `performance type` parameter is set to 'statistics' and the `statistics` parameter is set to 'count'. This parameter is used for specifying a particular value of the specified attribute. The performance vector value will be set to the number of occurrences of this value in the specified attribute. The attribute is specified by the `attribute name` parameter.

**example index** (*integer*) This parameter is only available when the `performance type` parameter is set to 'data value'. This parameter allows you to select the index of the required example of the attribute specified by the `attribute name` parameter.

**optimization direction** (*selection*) This parameter indicates if the performance value should be minimized or maximized.
8.2. Performance Measurement

Tutorial Processes

Introduction to the Extract Performance operator

This is a very basic process that demonstrates the use of the Extract Performance operator. The 'Golf' data set is loaded using the Retrieve operator. The Extract Performance operator is applied on it. The performance type parameter is set to 'statistics', the statistics parameter is set to 'average' and the attribute name parameter is set to 'Temperature'. Thus the value of the resultant performance vector will be the average of values of the Temperature attribute. The average of the Temperature attribute in all 14 examples of the 'Golf' data set is 73.571. The resultant performance vector and the 'Golf' data set can be seen in the Results Workspace. You can see that the value of the performance vector is 73.571.

Combine Performances

This operator takes a performance vector as input and returns a performance vector containing the weighted fitness value of the specified criteria.
8. Evaluation

Description

This Combine Performances operator takes a performance vector as input and returns a performance vector containing the weighted fitness value of the specified criteria. The user can specify the weights of different criteria. This operator takes the weighted average of the values of the specified criteria. It should be noted that some criteria values are considered positive by this operator e.g. accuracy. On the other hand some criteria values (usually error related) are considered negative by this operator e.g. relative error. Please study the attached Example Process for better understanding of this operator.

Input Ports

performance (per) This port expects a performance vector. A performance vector is a list of performance criteria values.

Output Ports

performance (per) The performance vector containing the weighted fitness value of the specified criteria is returned through this port.

Parameters

default weight (real) This parameter specifies the default weight for all criteria that are not assigned a weight through the criteria weights parameter.

criteria weights (list) Different performance criteria can be assigned different weights through this parameter. The criteria that are not assigned a weight through this parameter will have the default weight (i.e. specified by the default weight parameter).
Tutorial Processes

Introduction to the Combine Performances operator

This Example Process starts with the Subprocess operator. The subprocess is used for generating a sample performance vector. Therefore it is not necessary to understand the operators in the subprocess. A breakpoint is inserted after the Subprocess operator so that you can have a look at the performance vector. The performance vector has the following criteria values:

- Accuracy: 0.250
- Absolute error: 0.750
- Root mean squared error: 0.866

It is important to note that the accuracy is considered positive and the remaining two criteria are considered negative in the calculations by the Combine Performances operator.

The Combine Performances operator is applied on this performance vector. Have a look at the criteria weights parameter of the Combine Performances operator. The following weights are assigned to criteria:

- Accuracy: 2.0
- Absolute error: 1.0
- Root mean squared error: 0.0

The weighted fitness value is calculated by multiplying the weight with the corresponding value and finally averaging the results. In this case the following calculation is performed:

\[
\frac{(2(0.250) + 1(-0.750) + 0(0.866))}{3} = \frac{(0.500 - 0.750 + 0.000)}{3}
\]
Performance (Classification)

This operator is used for statistical performance evaluation of classification tasks. This operator delivers a list of performance criteria values of the classification task.

Description

This operator should be used for performance evaluation of only classification tasks. Many other performance evaluation operators are also available in RapidMiner e.g. Performance operator, Performance (Binominal Classification) operator, Performance (Regression) operator etc. The Performance (Classification) operator is used with classification tasks only. On the other hand, the Performance operator automatically determines the learning task type and calculates the most common criteria for that type. You can use the Performance (User-Based) operator if you want to write your own performance measure.

Classification is a technique used to predict group membership for data instances. For example, you may wish to use classification to predict whether the train on a particular day will be 'on time', 'late' or 'very late'. Predicting whether a number of people on a particular event would be 'below-average', 'average' or
8.2. Performance Measurement

'above-average' is another example. For evaluating the statistical performance of a classification model the data set should be labeled i.e. it should have an attribute with label role and an attribute with prediction role. The label attribute stores the actual observed values whereas the prediction attribute stores the values of label predicted by the classification model under discussion.

Input Ports

labeled data (lab) This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has a label attribute and a prediction attribute. See the Set Role operator for more details regarding label and prediction roles of attributes.

performance (per) This is an optional parameter. It requires a Performance Vector.

Output Ports

performance (per) This port delivers a Performance Vector (we call it output performance-vector for now). The Performance Vector is a list of performance criteria values. The Performance vector is calculated on the basis of the label attribute and the prediction attribute of the input ExampleSet. The output performance-vector contains performance criteria calculated by this Performance operator (we call it calculated-performance-vector here). If a Performance Vector was also fed at the performance input port (we call it input-performance-vector here), criteria of the input-performance-vector are also added in the output-performance-vector. If the input-performance-vector and the calculated-performance-vector both have the same criteria but with different values, the values of calculated-performance-vector are delivered through the output port. This concept can be easily understood by studying the attached Example Process.

example set (exa) ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same Exam-
8. Evaluation

pleSet in further operators or to view the ExampleSet in the Results Workspace.

Parameters

**main criterion** The main criterion is used for comparisons and needs to be specified only for processes where performance vectors are compared, e.g. attribute selection or other meta optimization process setups. If no *main criterion* is selected, the first criterion in the resulting performance vector will be assumed to be the *main criterion*.

**accuracy** *(boolean)* Relative number of correctly classified examples or in other words percentage of correct predictions.

**classification error** *(boolean)* Relative number of misclassified examples or in other words percentage of incorrect predictions.

**kappa** *(boolean)* The kappa statistics for the classification. It is generally thought to be a more robust measure than simple percentage correct prediction calculation since it takes into account the correct prediction occurring by chance.

**weighted mean recall** *(boolean)* The weighted mean of all per class recall measurements. It is calculated through class recalls for individual classes. Class recalls are mentioned in the last row of the matrix displayed in the Results Workspace.

**weighted mean precision** *(boolean)* The weighted mean of all per class precision measurements. It is calculated through class precisions for individual classes. Class precisions are mentioned in the last column of the matrix displayed in the Results Workspace.

**spearman rho** *(boolean)* The rank correlation between the actual and predicted *labels*, using Spearman's rho. Spearman's rho is a measure of the linear relationship between two variables. The two variables in this case are *label* attribute and *prediction* attribute.

**kendall tau** *(boolean)* The rank correlation between the actual and predicted *labels*, using Kendall's tau. Kendall's tau is a measure of correlation, and so measures the strength of the relationship between two variables. The two variables in this case are the *label* attribute and the *prediction* attribute.

**absolute error** *(boolean)* Average absolute deviation of the prediction from the
actual value. The values of the label attribute are the actual values.

**relative error (boolean)** Average relative error is the average of the absolute deviation of the prediction from the actual value divided by the actual value. The values of the label attribute are the actual values.

**relative error lenient (boolean)** Average lenient relative error is the average of the absolute deviation of the prediction from the actual value divided by the maximum of the actual value and the prediction. The values of the label attribute are the actual values.

**relative error strict (boolean)** Average strict relative error is the average of the absolute deviation of the prediction from the actual value divided by the minimum of the actual value and the prediction. The values of the label attribute are the actual values.

**normalized absolute error (boolean)** The absolute error divided by the error made if the average would have been predicted.

**root mean squared error (boolean)** The averaged root-mean-squared error.

**root relative squared error (boolean)** The averaged root-relative-squared error.

**squared error (boolean)** The averaged squared error.

**correlation (boolean)** Returns the correlation coefficient between the label and prediction attributes.

**squared correlation (boolean)** Returns the squared correlation coefficient between the label and prediction attributes.

**cross entropy (boolean)** The cross-entropy of a classifier, defined as the sum over the logarithms of the true label's confidences divided by the number of examples.

**margin (boolean)** The margin of a classifier, defined as the minimal confidence for the correct label.

**soft margin loss (boolean)** The average soft margin loss of a classifier, defined as the average of all 1 - confidences for the correct label.

**logistic loss (boolean)** The logistic loss of a classifier, defined as the average of \( \ln(1+\exp(-[\text{conf(CC)]}) \) where 'conf(CC)' is the confidence of the correct class.

**skip undefined labels (boolean)** If set to true, examples with undefined labels are skipped.

**comparator class (string)** This is an expert parameter. The fully qualified classname of the PerformanceComparator implementation is specified here.
8. Evaluation

use example weights (boolean) This parameter allows example weights to be used for statistical performance calculations if possible. This parameter has no effect if no attribute has weight role. In order to consider weights of examples the ExampleSet should have an attribute with weight role. Several operators are available that assign weights e.g. Generate Weights operator. Study the Set Roles operator for more information regarding weight role.

class weights This is an expert parameter. It specifies the weights 'w' for all classes. The Edit List button opens a new window with two columns. The first column specifies the class name and the second column specifies the weight for that class. If the weight of a class is not specified, that class is assigned weight = 1.

Tutorial Processes

Use of performance port in Performance (Classification)

This Example Process is composed of two Subprocess operators and one Performance (Classification) operator. Double click on the first Subprocess operator and you will see the operators within this subprocess. The first subprocess 'Subprocess (labeled data provider)' loads the 'Golf' data set using the Retrieve operator and then learns a classification model using the k-NN operator. Then the learned model is applied on 'Golf-Testset' data set using the Apply Model operator. Then the Generate Weight operator is used to add an attribute with weight role. Thus, this subprocess provides a labeled ExampleSet with weight attribute. The Breakpoint is inserted after this subprocess to show this ExampleSet. This ExampleSet is provided at the labeled data input port of the Performance (Classification) operator in the main process.

The second Subprocess operator ' Subprocess (performance vector provider) ' loads the' Golf ' data set using the Retrieve operator and then learns a classification model using the k-NN operator. Then the learned model is applied on the' Golf ' data set using the Apply Model operator. Then the Performance (Classifi-
8.2. Performance Measurement

cation) operator is applied on the labeled data to produce a Performance Vector. The Breakpoint is inserted after this subprocess to show this Performance Vector. Note that this model was trained and tested on the same data set (Golf data set), so its accuracy is 100%. Thus this subprocess provides a Performance Vector with 100% accuracy and 0.00% classification error. This Performance Vector is connected to the performance input port of the Performance (Classification) operator in the main process.

When you run the process, first you will see an ExampleSet which is the output of the first Subprocess operator. Press the Run button again and now you will see a Performance Vector. This is the output of the second Subprocess operator. Press the Run button again and you will see various criteria in the criterion selector window in the Results Workspace. These include classification error, accuracy, weighted mean recall and weighted mean precision. Now select the accuracy from the criterion selector window, its value is 71.43%. On the contrary the accuracy of the input Performance Vector provided by the second subprocess was 100%. The accuracy of the final Performance Vector is 71.43% instead of 100% because if the input-performance-vector and the calculated-performance-vector both have same criteria but with different values, the values of the calculated-performance-vector are delivered through the output port. Now, note that the classification error criterion is added to the criteria list because of the Performance Vector provided at the performance input port. Disable the second Subprocess operator and run the same process again, you will see that classification error criterion does not appear now. This is because if a Performance Vector is fed at the performance input port, its criteria are also added to the output-performance-vector.

The accuracy is calculated by taking the percentage of correct predictions over the total number of examples. Correct prediction means the examples where the value of the prediction attribute is equal to the value of label attribute. If you look at the ExampleSet in the Results Workspace, you can see that there are 14 examples in this data set. 10 out of 14 examples are correct predictions i.e. their label and prediction attributes have the same values. This is why accuracy was 71.43% \( \frac{10 \times 100}{14} = 71.43\% \). Now run the same process again but this time set use example weights parameter to true. Check the results again. They have changed now because the weight of each example was taken into account.
8. Evaluation

this time. The accuracy is 68.89% this time. If you take the percentage of weight of correct predictions and the total weight you get the same answer (0.6889 x 100/1 = 68.89%). In this Example Process, using weights reduced the accuracy but this is not always the case.

The weighted mean recall is calculated by taking the average of recall of every class. As you can see in the last row of the resultant matrix in the Results Workspace, class recall for 'true no' is 60% and class recall for 'true yes' is 77.78%. Thus weighted mean recall is calculated by taking the average of these class recall values (((77.78%)+(60%))/2=68.89%).

The weighted mean precision is calculated by taking the average of precision of every class. As you can see in the last column of the resultant matrix in the Results Workspace, class precision for 'pred. no' is 60% and class precision for 'pred. yes' is 77.78%. Thus weighted mean precision is calculated by taking the average of these class precision values (((77.78%)+(60%))/2=68.89%). These values are for the case when the use example weights parameter is set to false.

Note: This Example Process is just for highlighting different perspectives of Performance (Classification) operator. It may not be very useful in real scenarios.
Performance (Binominal Classification)

This operator is used for statistical performance evaluation of binomial classification tasks i.e. classification tasks where the label attribute has a binominal type. This operator delivers a list of performance criteria values of the binominal classification task.

Description

This operator should be used specifically for performance evaluation of only binominal classification tasks i.e. classification tasks where the label attribute has a binominal type. Many other performance evaluation operators are also available in RapidMiner e.g. the Performance operator, the Performance (Classification) operator, the Performance (Regression) operator etc. The Performance (Binominal Classification) operator is used with binominal classification tasks only. On the other hand, the Performance operator automatically determines the learning task type and calculates the most common criteria for that type. You can use the Performance (User-Based) operator if you want to write your own performance measure.

Classification is a technique used to predict group membership for data instances. For example, you may wish to use classification to predict whether the train on a particular day will be 'on time', 'late' or 'very late'. Predicting whether a number of people on a particular event would be 'below-average', 'average' or 'above-average' is another example. For evaluating the statistical performance of a classification model the data set should be labeled i.e. it should have an attribute with label role and an attribute with prediction role. The label attribute stores the actual observed values whereas the prediction attribute stores the values of the label predicted by the classification model under discussion.
8. Evaluation

Input Ports

labeled data (lab) This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has a label attribute and a prediction attribute. See the Set Role operator for more details regarding label and prediction roles of attributes. Also make sure that the label attribute of the ExampleSet is of binominal type i.e. label has only two possible values.

performance (per) This is an optional parameter. It requires a Performance Vector.

Output Ports

performance (per) This port delivers a Performance Vector (we call it output-performance-vector for now). A Performance Vector is a list of performance criteria values. It is calculated on the basis of the label and the prediction attribute of the input ExampleSet. The output-performance-vector contains performance criteria calculated by this Performance operator (we call it calculated-performance-vector here). If a Performance Vector was also fed at the performance input port (we call it input-performance-vector here), the criteria of the input-performance-vector are also added in the output-performance-vector. If the input-performance-vector and the calculated-performance-vector both have the same criteria but with different values, the values of the calculated-performance-vector are delivered through the output port. This concept can be easily understood by studying the attached Example Process.

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

**main criterion** The main criterion is used for comparisons and needs to be specified only for processes where performance vectors are compared, e.g. attribute selection or other meta optimization process setups. If no main criterion is selected, the first criterion in the resulting performance vector will be assumed to be the main criterion.

**accuracy (boolean)** Relative number of correctly classified examples or in other words percentage of correct predictions.

**classification error (boolean)** Relative number of misclassified examples or in other words percentage of incorrect predictions.

**kappa (boolean)** The kappa statistics for the classification. It is generally thought to be a more robust measure than the simple percentage correct prediction calculation since it takes into account the correct prediction occurring by chance.

**AUC(optimistic) (boolean)** AUC is the Area Under the Curve of the Receiver Operating Characteristics (ROC) graph which is a technique for visualizing, organizing and selecting classifiers based on their performance. Given example weights are also considered. Please note that the second class is considered to be positive. AUC(optimistic) is the extreme case when all the positives end up at the beginning of the sequence.

**AUC (boolean)** AUC is the Area Under the Curve of the Receiver Operating Characteristics (ROC) graph which is a technique for visualizing, organizing and selecting classifiers based on their performance. Given example weights are also considered. Please note that the second class is considered to be positive. Usually AUC is the average of AUC(pessimistic) and AUC(optimistic).

**AUC(pessimistic) (boolean)** AUC is the Area Under the Curve of the Receiver Operating Characteristics (ROC) graph which is a technique for visualizing, organizing and selecting classifiers based on their performance. Given example weights are also considered. Please note that the second class is considered to be positive. AUC(pessimistic) is the extreme case when all the negatives end up at the beginning of the sequence.

**precision (boolean)** Relative number of correctly as positive classified examples among all examples classified as positive i.e. precision = (Positives Correctly Class-
8. Evaluation

sified)/(Total Predicted Positives). Note that the Total Predicted Positives is the sum of True Positives and False Positives. This is the same as the positive predictive value.

recall (boolean) This parameter specifies the relative number of correctly as positive classified examples among all positive examples i.e. recall = (Positives Correctly Classified)/(Total Positives). It is also called hit rate or true positive rate. This is the same as sensitivity.

lift (boolean) This parameter specifies the lift of the positive class.

fallout (boolean) The Relative number of incorrectly as positive classified examples among all negative examples i.e. fallout = (Positives Incorrectly Classified)/(Total Negatives)

f measure (boolean) This parameter is a combination of the precision and the recall i.e. f = 2pr/(p+r) where f, r and p are f-measure, recall and precision respectively.

false positive (boolean) This parameter specifies the absolute number of negative examples that were incorrectly classified as positive examples. In other words, if the example is negative and it is classified as positive, it is counted as a false positive.

false negative (boolean) This parameter specifies the absolute number of positive examples that were incorrectly classified as negative examples. In other words, if the example is positive and it is classified as negative, it is counted as a false negative.

true positive (boolean) This parameter specifies the absolute number of positive examples that were correctly classified as positive examples. In other words, if the example is positive and it is classified as positive, it is counted as a true positive.

true negative (boolean) This parameter specifies the absolute number of negative examples that were correctly classified as negative examples. In other words, if the example is negative and it is classified as negative, it is counted as a true negative.

sensitivity (boolean) This parameter specifies the relative number of correctly as positive classified examples among all positive examples i.e. sensitivity = (Positives Correctly Classified)/(Total Positives). It is also called hit rate or true positive rate. This is same as recall.
8.2. Performance Measurement

**specificity (boolean)** The relative number of correctly as negative classified examples among all negative examples i.e. \( \text{specificity} = \frac{\text{Negatives Correctly Classified}}{\text{Total Negatives}} \). Note that Total Negatives is equal to sum of True Negatives and False Positives.

**youden (boolean)** This parameter specifies the sum of sensitivity and specificity minus 1.

**positive predictive value (boolean)** The relative number of correctly as positive classified examples among all examples classified as positive i.e. \( \text{positive predictive value} = \frac{\text{Positives Correctly Classified}}{\text{Total Predicted Positives}} \). Note that the Total Predicted Positives is sum of True Positives and False Positives. This is the same as precision.

**negative predictive value (boolean)** The relative number of correctly as negative classified examples among all examples classified as negative i.e. \( \text{negative predictive value} = \frac{\text{Negatives Correctly Classified}}{\text{Total Predicted Negatives}} \). Note that the Total Predicted Negatives is sum of True Negatives and False Negatives.

**psep (boolean)** This parameter specifies the sum of the positive predictive value and the negative predictive value minus 1. i.e. \( \text{psep} = \text{ppv} + \text{npv} - 1 \) where \( \text{ppv} \) and \( \text{npv} \) are positive predictive value and negative predictive value respectively.

**skip undefined labels (boolean)** If set to true, examples with undefined labels are skipped.

**comparator class (string)** This is an expert parameter. The fully qualified classname of the PerformanceComparator implementation is specified here.

**use example weights (boolean)** This parameter allows example weights to be used for statistical performance calculations if possible. This parameter has no effect if no attribute has weight role. In order to consider weights of examples the ExampleSet should have an attribute with weight role. Several operators are available that assign weights e.g. the Generate Weights operator. Study the Set Roles operator for more information regarding weight role.
Tutorial Processes

Use of parameters of Performance (Binominal Classification)

The focus of this Example Process is to explain different statistical performance criteria of Binominal Classification. To get an understanding of the use of the performance port you can study the Example Process of the Performance operator or the Example Process of the Performance (Classification) operator.

The 'Golf' data set is loaded using the Retrieve operator. The Nominal to Binominal operator is applied on it to convert the label attribute (i.e. Play) from nominal to binominal type. The K-NN operator is applied on it to generate a classification model. This classification model is applied on the 'Golf-Testset' data set using the Apply Model operator. Note that the Nominal to Binominal operator is applied on the 'Golf-Testset' data set as well to convert the label attribute (i.e. Play) from nominal to binominal type to ensure that the training data set ('Golf') and the testing data set ('Golf-Testset') are in the same format. The labels were changed to binominal form because the Performance (Binominal Classification) operator can only handle binominal labels. Run the process and you can see the results in the Results Workspace.

Have a good look at the confusion matrix in the Results Workspace. This matrix will be used to explain all the parameters. The rows 'pred. no' and 'pred. yes' tell about the examples that were classified as 'no' and classified as 'yes' respectively. The columns 'true no' and 'true yes' tell about the examples that were actually labeled 'no' and actually labeled 'yes' respectively. Here is some information that we can get by simply taking a look at the confusion matrix.

- True Negative = the examples that were actually labeled 'no' and were classified as 'no' = 3
- False Negative = the examples that were actually labeled 'yes' and were classified as 'no' = 2
8.2. Performance Measurement

- True Positive = the examples that were actually labeled 'yes' and were classified as 'yes' = 7
- False Positive = the examples that were actually labeled 'no' and were classified as 'yes' = 2
- Total number of examples that were actually labeled 'no' = Total Negatives = 5 (i.e. 3+2)
- Total number of examples that were actually labeled 'yes' = Total Positives = 9 (i.e. 7+2)
- Total number of examples that were classified as 'no' = Total Predicted Negatives = 5 (i.e. 3+2)
- Total number of examples that were classified as 'yes' = Total Predicted Positives = 9 (i.e. 7+2)
- Total number of examples = 14 (i.e. 2+3+2+7)
- Total number of correct classifications = 10 (i.e.3+7)
- Total number of incorrect classifications = 4 (i.e.2+2)

Here is shown how different the statistical performance criteria were calculated. The terms 'Positive' and 'classified as yes' mean the same thing. Same as with other similar terms like 'Correctly Classified Positives' and 'True Positive' which mean the same thing.

- accuracy = (Total Correct Classifications)/(Total number of examples) = (10)/(14) = 71.42%
- classification error = (Total incorrect classifications)/( Total number of examples) = (4)/(14) = 28.57%
- precision = (True Positives)/(Total Predicted Positives) = (7)/(9) = 77.78%
- recall = (True Positive)/(Total Positives) = (7)/(9) = 77.78%
- fallout = (False Positives)/(Total Negatives) = (2)/(5) = 40%
8. Evaluation

- $f$-measure = $\frac{2pr}{p+r}$ where $r$ and $p$ are recall and precision respectively
  =77.78%

- sensitivity = $\frac{\text{True Positive}}{\text{Total Positives}} = \frac{7}{9} = 77.78\%$

- specificity = $\frac{\text{True Negatives}}{\text{Total negatives}} = \frac{3}{5} = 60\%$

- Youden = the sum of sensitivity(0.78) and specificity(0.60) minus 1= 0.378

- positive predicted value = $\frac{\text{True Positives}}{\text{Total Predicted Positives}} = \frac{7}{9} = 77.78\%$

- negative predicted value = $\frac{\text{True Negatives}}{\text{Total Predicted Negatives}} = \frac{3}{5} = 60\%$

- $psep$ = the sum of the positive predictive value (0.78) and the negative predictive value(0.60) minus 1= 0.378
Performance (Regression)

This operator is used for statistical performance evaluation of regression tasks and delivers a list of performance criteria values of the regression task.

Description

This operator should be used for performance evaluation of regression tasks only. Many other performance evaluation operators are also available in RapidMiner e.g. the Performance operator, Performance (Binominal Classification) operator, Performance (Classification) operator etc. The Performance (Regression) operator is used with regression tasks only. On the other hand, the Performance operator automatically determines the learning task type and calculates the most common criteria for that type. You can use the Performance (User-Based) operator if you want to write your own performance measure.

Regression is a technique used for numerical prediction and it is a statistical measure that attempts to determine the strength of the relationship between one dependent variable (i.e. the label attribute) and a series of other changing variables known as independent variables (regular attributes). Just like Classification is used for predicting categorical labels, Regression is used for predicting a continuous value. For example, we may wish to predict the salary of university graduates with 5 years of work experience, or the potential sales of a new product given its price. Regression is often used to determine how much specific factors such as the price of a commodity, interest rates, particular industries or sectors influence the price movement of an asset. For evaluating the statistical performance of a regression model the data set should be labeled i.e. it should have an attribute with label role and an attribute with prediction role. The label attribute stores the actual observed values whereas the prediction attribute stores the values of label predicted by the regression model under discussion.
8. Evaluation

Input Ports

labeled data \((lab)\) This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has the \textit{label} and \textit{prediction} attribute. See the Set Role operator for more details regarding the \textit{label} and \textit{prediction} roles of attributes.

performance \((per)\) This is an optional parameter. It requires a Performance Vector.

Output Ports

performance \((per)\) This port delivers a Performance Vector (we call it \textit{output-performance-vector} for now). The Performance Vector is a list of performance criteria values. The Performance vector is calculated on the basis of the \textit{label} and \textit{prediction} attribute of the input ExampleSet. The \textit{output-performance-vector} contains performance criteria calculated by this Performance operator (we call it \textit{calculated-performance-vector} here). If a Performance Vector was also fed at the \textit{performance} input port (we call it \textit{input-performance-vector} here), the criteria of the \textit{input-performance-vector} are also added in the \textit{output-performance-vector}. If the \textit{input-performance-vector} and the \textit{calculated-performance-vector} both have the same criteria but with different values, the values of the \textit{calculated-performance-vector} are delivered through the output port. This concept can be easily understood by studying the Example Process of the Performance (Classification) operator.

d impeachment set \((exa)\) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
Parameters

**main criterion** The main criterion is used for comparisons and needs to be specified only for processes where performance vectors are compared, e.g. attribute selection or other meta optimization process setups. If no main criterion is selected, the first criterion in the resulting performance vector will be assumed to be the main criterion.

**root mean squared error** (boolean) The averaged root-mean-squared error.

**absolute error** (boolean) The average absolute deviation of the prediction from the actual value. The values of the label attribute are the actual values.

**relative error** (boolean) The average relative error is the average of the absolute deviation of the prediction from the actual value divided by actual value. Values of the label attribute are the actual values.

**relative error lenient** (boolean) The average lenient relative error is the average of the absolute deviation of the prediction from the actual value divided by the maximum of the actual value and the prediction. The values of the label attribute are the actual values.

**relative error strict** (boolean) The average strict relative error is the average of the absolute deviation of the prediction from the actual value divided by the minimum of the actual value and the prediction. The values of the label attribute are the actual values.

**normalized absolute error** (boolean) The absolute error divided by the error made if the average would have been predicted.

**root relative squared error** (boolean) The averaged root-relative-squared error.

**squared error** (boolean) The averaged squared error.

**correlation** (boolean) Returns the correlation coefficient between the label and prediction attributes.

**squared correlation** (boolean) Returns the squared correlation coefficient between the label and prediction attributes.

**prediction average** (boolean) Returns the average of all the predictions. All the predicted values are added and the sum is divided by the total number of predictions.
8. Evaluation

spearman rho (boolean) The rank correlation between the actual and predicted labels, using Spearman's rho. Spearman's rho is a measure of the linear relationship between two variables. The two variables in this case are the label and the prediction attribute.

kendall tau (boolean) The rank correlation between the actual and predicted labels, using Kendall's tau-b. Kendall's tau is a measure of correlation, and so measures the strength of the relationship between two variables. The two variables in this case are the label and the prediction attribute.

skip undefined labels (boolean) If set to true, examples with undefined labels are skipped.

comparator class (string) This is an expert parameter. Fully qualified class-name of the PerformanceComparator implementation is specified here.

use example weights (boolean) This parameter allows example weights to be used for statistical performance calculations if possible. This parameter has no effect if no attribute has the weight role. In order to consider weights of examples the ExampleSet should have an attribute with the weight role. Several operators are available that assign weights e.g. the Generate Weights operator. Study the Set Roles operator for more information regarding the weight role.

Tutorial Processes

Applying the Performance (Regression) operator on the Polynomial data set

The 'Polynomial' data set is loaded using the Retrieve operator. The Filter Example Range operator is applied on it. The first example parameter of the Filter Example Range parameter is set to 1 and the last example parameter is set to 100. Thus the first 100 examples of the 'Polynomial' data set are selected. The Linear Regression operator is applied on it with default values of all parameters. The regression model generated by the Linear Regression operator is applied on the last 100 examples of the 'Polynomial' data set using the Apply Model operator. Labeled data from the Apply Model operator is provided to the Performance
(Regression) operator. The absolute error and prediction average parameters are set to true. Thus the Performance Vector generated by the Performance (Regression) operator has information regarding the absolute error and prediction average in the labeled data set. The absolute error is calculated by adding the difference of all the predicted values from actual values of the label attribute, and dividing this sum by the total number of predictions. The prediction average is calculated by adding all the actual label values and dividing this sum by the total number of examples. You can verify this from the results in the Results Workspace.

**Performance (Costs)**

This operator provides the ability to evaluate misclassification costs for performance evaluation of classification tasks.
8. Evaluation

Description

The Performance (Costs) operator provides the ability to evaluate misclassification costs. A cost matrix should be specified through the cost matrix parameter. The cost matrix is similar in structure to a confusion matrix because it has predicted classes in one dimension and actual classes on the other dimension. Therefore the cost matrix can denote the costs for every possible classification outcome: predicted label vs. actual label. Actually this matrix is a matrix of misclassification costs because you can specify different weights for certain classes misclassified as other classes. Weights can also be assigned to correct classifications but they are not taken into account for evaluating misclassification costs. The classes in the matrix are labeled as Class 1, Class 2 etc where classes are numbered according to their order in the internal mapping. The class order definition parameter allows you to specify the class order for the matrix in which case classes are ordered according to the order specified in this parameter (instead of internal mappings). For a better understanding of this operator please study the attached Example Process.

Input Ports

example set (exa) This input port expects a labeled ExampleSet. The Apply Model operator is a good example of such operators that provide labeled data. Make sure that the ExampleSet has label and prediction attributes. Please see the Set Role operator for more details about attribute roles.

Output Ports

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
performance (*per*) This port delivers a Performance Vector which has information about the misclassification costs.

Parameters

cost matrix (*string*) This parameter is used for specifying the cost matrix. The cost matrix is similar in structure to a confusion matrix because it has predicted classes in one dimension and actual classes on the other dimension. Therefore the cost matrix can denote the costs for every possible classification outcome: predicted label vs. actual label. Actually this matrix is a matrix of misclassification costs because you can specify different weights for certain classes misclassified as other classes. Weights can also be assigned to correct classifications but they are not taken into account for evaluating misclassification costs. The classes in the matrix are labeled as Class 1, Class 2 etc where classes are numbered according to their order in the internal mapping. The *class order definition* parameter can be used for specifying the class order for the matrix (instead of internal mappings).

class order definition (*enumeration*) The *class order definition* parameter allows you to specify the class order for the cost matrix in which case classes are ordered according to the order specified in this parameter (instead of internal mappings).

Tutorial Processes

Measuring Misclassification costs of a classifier

The 'Golf' data set is loaded using the Retrieve operator. The Split Validation operator is applied on it for training and testing a classification model. The Naive Bayes operator is applied in the training subprocess of the Split Validation operator. The Naive Bayes operator trains a classification model. The Apply Model operator is used in the testing subprocess to apply this model. A break-
8. Evaluation

point is inserted here so that you can have a look at the labeled ExampleSet. As you can see, out of 4 examples of the testing data set only 1 has been misclassified. The misclassified example was classified as 'class = yes' while actually it was 'class = no'.

The resultant labeled ExampleSet is used by the Performance (Costs) operator for measuring the misclassification costs of the model. Have a look at the parameters of the Performance (Costs) operator. The class order definition parameter specifies the order of classes in the cost matrix. The classes 'yes' and 'no' are placed in first and second rows respectively. Thus class 'yes' is Class 1 and class 'no' is Class 2 in the cost matrix. Now have a look at the cost matrix in the cost matrix parameter. The case where Class 2 (i.e. class = no) is misclassified as Class 1 (i.e. class = yes) has been given weight 2.0. The case where Class 1 (i.e. class = yes) is misclassified as Class 2 (i.e. class = no) has been given weight 1.0.

Now let us see how this cost matrix is used for evaluating misclassification costs of the labeled ExampleSet. As 1 of the 4 classifications was wrong, one should expect the classification cost to be 1/4 or 0.250. But as this misclassification has weight 2.0 (because class = no is misclassified as class = yes) instead of 1.0 the cost for this misclassification is doubled. Therefore the cost in this case is 0.500. The misclassification cost can be seen in the Results Workspace.

Now set the sampling type parameter of the Split Validation operator to 'linear sampling' and run the process again. Have a look at the labeled ExampleSet generated by the Apply Model operator. 2 out of 4 examples have been misclassified. One example with class = no has been misclassified as class = yes (i.e. weight = 2.0) and one example with class = yes has been misclassified as class = no (i.e. weight = 1.0). The resultant misclassification cost is ((1 x 1.0)+(1 x 2.0))/4 which results to 0.750. The misclassification cost can be seen in the Results Workspace.
Cluster Distance Performance

This operator is used for performance evaluation of centroid based clustering methods. This operator delivers a list of performance criteria values based on cluster centroids.

Description

The centroid based clustering operators like the K-Means and K-Medoids produce a centroid cluster model and a clustered set. The centroid cluster model has information regarding the clustering performed. It tells which examples are parts of which cluster. It also has information regarding centroids of each cluster. The Cluster Distance Performance operator takes this centroid cluster model and clustered set as input and evaluates the performance of the model based on the cluster centroids. Two performance measures are supported: Average within cluster distance and Davies-Bouldin index. These performance measures are explained in the parameters.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding
clusters of customers with similar buying behavior.

### Input Ports

**example set (exa)** This input port expects an ExampleSet. It is output of the K-Medoids operator in the attached Example Process.

**cluster model (clu)** This input port expects a centroid cluster model. It is output of the K-Medoids operator in the attached Example Process. The centroid cluster model has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

**performance (per)** This input port expects a Performance Vector.

### Output Ports

**performance (per)** The performance of the cluster model is evaluated and the resultant Performance Vector is delivered through this port. The Performance Vector is a list of performance criteria values.

**example set (exa)** The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**cluster model (clu)** The centroid cluster model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same centroid cluster model in further operators or to view it in the Results Workspace.
Parameters

**main criterion** *(selection)* This parameter specifies the main criterion to use for performance evaluation.

- **avg._within_centroid_distance** The average within cluster distance is calculated by averaging the distance between the centroid and all examples of a cluster.

- **davies_bouldin** The algorithms that produce clusters with low intra-cluster distances (high intra-cluster similarity) and high inter-cluster distances (low inter-cluster similarity) will have a low Davies–Bouldin index, the clustering algorithm that produces a collection of clusters with the smallest Davies–Bouldin index is considered the best algorithm based on this criterion.

**main criterion only** *(boolean)* This parameter specifies if only the main criterion should be delivered by the performance vector. The main criterion is specified by the **main criterion** parameter

**normalize** *(boolean)* This parameter specifies if the results should be normalized. If set to true, the criterion is divide by the number of features.

**maximize** *(boolean)* This parameter specifies if the results should be maximized. If set to true, the result is not multiplied by minus one.

**Tutorial Processes**

**Evaluating the performance of the K-Medoids clustering model**

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the *label* is loaded too, but it is only used for visualization and comparison and not for building the clusters itself. A *breakpoint* is inserted at this step so that you
8. Evaluation

can have a look at the ExampleSet before application of the K-Medoids operator. The 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Medoids operator is applied on this data set with default values for all parameters. A \textit{breakpoint} is inserted at this step so that you can have a look at the results of the K-Medoids operator. You can see that two new attributes are created by the K-Medoids operator. The \textit{id} attribute is created to distinguish examples clearly. The \textit{cluster} attribute is created to show which cluster the examples belong to. As parameter \textit{k} was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Also note the \textit{Plot View} of this data. You can clearly see how the algorithm has created two separate groups in the \textit{Plot View}. A cluster model is also delivered through the \textit{cluster model} output port. It has information regarding the clustering performed. Under the \textit{Folder View} you can see members of each cluster in folder format. You can see information regarding centroids under the \textit{Centroid Table} and \textit{Centroid Plot View} tabs.

The Cluster Distance Performance operator is applied to measure the performance of this clustering model. The \textit{cluster model} and \textit{clustered set} produced by the K-Medoids operator are provided as input to the Cluster Distance Performance operator which evaluates the performance of this model and delivers a performance vector that has performance criteria values. The resultant performance vector can be seen in the results workspace.
Cluster Density Performance

This operator is used for performance evaluation of the centroid based clustering methods. This operator delivers a list of performance criteria values based on cluster densities.

Description

The centroid based clustering operators like the K-Means and K-Medoids produce a centroid cluster model and a clustered set. The centroid cluster model has information regarding the clustering performed. It tells which examples are parts of which cluster. It also has information regarding centroids of each cluster. The Cluster Density Performance operator takes this centroid cluster model and clustered set as input and evaluates the performance of the model based on the cluster densities. It is important to note that this operator also requires a SimilarityMeasure object as input. This operator is used for evaluation of non-hierarchical cluster models based on the average within cluster similarity/distance. It is computed by averaging all similarities / distances between each pair of examples of a cluster.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Input Ports

**example set** *(exa)* This input port expects an ExampleSet. It is output of the Data to Similarity operator in the attached Example Process.

**distance measure** *(dis)* This input port expects a SimilarityMeasure object. It is output of the Data to Similarity operator in the attached Example Process.
8. Evaluation

performance vector (per) This optional input port expects a performance vector. A performance vector is a list of performance criteria values.

cluster model (clu) This input port expects a centroid cluster model. It is output of the K-Means operator in the attached Example Process. The centroid cluster model has information regarding the clustering performed. It tells which examples are part of which cluster. It also has information regarding centroids of each cluster.

Output Ports

differential equation (exa) The ExampleSet that was given as input is passed without any modifications to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

performance vector (per) The performance of the cluster model is evaluated and the resultant performance vector is delivered through this port. A performance vector is a list of performance criteria values.

Tutorial Processes

Evaluating the performance of the K-Means clustering model

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters. A breakpoint is inserted at this step so that you can have a look at the ExampleSet before application of the K-Means operator. The 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Means operator is applied on this data set with default values for all parameters. A breakpoint is inserted at this step so that you can have a look at the results of the K-Means operator. You can see that two new attributes are created by the K-Means op-
8.2. Performance Measurement

operator. The id attribute is created to distinguish examples clearly. The cluster attribute is created to show which cluster the examples belong to. As parameter $k$ was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'.

The Data to Similarity operator is applied on the resultant ExampleSet. This generates a SimilarityMeasure object. The clustered ExampleSet, cluster model and the Similarity Measure object are provided as input to the Cluster Density Performance operator. The Cluster Density Performance operator evaluates the performance of this model and delivers a performance vector that has performance criteria values. The resultant performance vector can be seen in the results workspace.

Item Distribution Performance

This operator is used for performance evaluation of flat clustering methods. It evaluates a cluster model based on the distribution of examples.

Description

The clustering operators like the K-Means and K-Medoids produce a flat cluster model and a clustered set. The cluster model has information regarding the clustering performed. It tells which examples are parts of which cluster. The
8. Evaluation

Item Distribution Performance operator takes this cluster model as input and evaluates the performance of the model based on the distribution of examples i.e. how well the examples are distributed over the clusters. Two distribution measures are supported: Sum of Squares and Gini Coefficient. These distribution measures are explained in the parameters. Flat clustering creates a flat set of clusters without any explicit structure that would relate clusters to each other. Hierarchical clustering, on the other hand, creates a hierarchy of clusters. This operator can only be applied on models produced by operators that produce flat cluster models e.g. K-Means or K-Medoids operators. It cannot be applied on models created by the operators that produce a hierarchy of clusters e.g. the Agglomerative Clustering operator.

Clustering is concerned with grouping together objects that are similar to each other and dissimilar to the objects belonging to other clusters. It is a technique for extracting information from unlabeled data and can be very useful in many different scenarios e.g. in a marketing application we may be interested in finding clusters of customers with similar buying behavior.

Input Ports

**cluster model** *(clu)* This input port expects a flat cluster model. It is output of the K-Medoids operator in the attached Example Process. The cluster model has information regarding the clustering performed. It tells which examples are part of which cluster.

**performance vector** *(per)* This input port expects a Performance Vector.

Output Ports

**cluster model** *(clu)* The cluster model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same cluster model in further operators or to view it in the Results Workspace.

**performance vector** *(per)* The performance of the cluster model is evaluated
and the resultant Performance Vector is delivered through this port. It is a list of performance criteria values.

**Parameters**

**measure** *(selection)* This parameter specifies the item distribution measure to apply. It has two options:

- **sumofquares** If this option is selected, the sum of squares is used as the item distribution measure.

- **ginicoefficient** The Gini coefficient (also known as the Gini index or Gini ratio) is a measure of statistical dispersion. It measures the inequality among values of a frequency distribution. A low Gini coefficient indicates a more equal distribution, with 0 corresponding to complete equality, while higher Gini coefficients indicate a more unequal distribution, with 1 corresponding to complete inequality.

**Tutorial Processes**

**Evaluating the performance of the K-Medoids clustering model**

The 'Ripley-Set' data set is loaded using the Retrieve operator. Note that the label is loaded too, but it is only used for visualization and comparison and not for building the clusters themselves. A **breakpoint** is inserted at this step so that you can have a look at the ExampleSet before the application of the K-Medoids operator. The 'Ripley-Set' has two real attributes; 'att1' and 'att2'. The K-Medoids operator is applied on this data set with default values for all parameters. A **breakpoint** is inserted at this step so that you can have a look at the results of the K-Medoids operator. You can see that two new attributes are created by the
K-Medoids operator. The \textit{id} attribute is created to distinguish examples clearly. The \textit{cluster} attribute is created to show which cluster the examples belong to. As parameter \( k \) was set to 2, only two clusters are possible. That is why each example is assigned to either 'cluster_0' or 'cluster_1'. Also note the \textit{Plot View} of this data. You can clearly see how the algorithm has created two separate groups in the \textit{Plot View}. A cluster model is also delivered through the \textit{cluster model} output port. It has information regarding the clustering performed. Under the \textit{Folder View} you can see members of each cluster in folder format and under the \textit{Centroid Table} and \textit{Centroid Plot View} tabs information regarding centroids.

The Item Distribution Performance operator is applied to measure the performance of this clustering model on the basis of how well the examples are distributed over the clusters. The \textit{cluster model} produced by the K-Medoids operator is provided as input to the Item Distribution Performance operator which evaluates the performance of this model and delivers a performance vector that has performance measured on the basis of example distribution. The resultant performance vector can be seen in the results workspace.

\textbf{T-Test}

This operator is used for comparison of performance vectors. This operator performs a t-test to determine the probability for the null hypothesis i.e. 'the actual means are the same'.
8.3. Significance

Description

The T-Test operator determines if the null hypothesis (i.e. all actual mean values are the same) holds for the given performance vectors. This operator uses a simple paired t-test to determine the probability that the null hypothesis is wrong. Since a t-test can only be applied on two performance vectors this test will be applied to all possible pairs. The result is a significance matrix.

Paired t-test is a test of the null hypothesis that the difference between two responses measured on the same statistical unit has a mean value of zero. For example, suppose we measure the size of a cancer patient's tumor before and after a treatment. If the treatment is effective, we expect the tumor size for many of the patients to be smaller following the treatment. This is often referred to as the 'paired' or 'repeated measures' t-test.

In case of this operator the dependent samples (or 'paired') t-tests consist of a pair of performance vectors. Doing multiple paired t-tests would result in an increased chance of committing a type I error. 'False positive' or Type I error is defined as the probability that a decision to reject the null hypothesis will be made when it is in fact true and should not have been rejected. It is recommended to apply an additional ANOVA test to determine if the null hypothesis is wrong at all. Please use the ANOVA operator for performing the ANOVA test.

Differentiation

ANOVA Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. See page 954 for details.
8. Evaluation

Input Ports

**performance** *(per)* This operator expects performance vectors as input and can have multiple inputs. When one input is connected, another *performance* input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The performance vector supplied at the first *input* port of this operator is available at the first *performance* output port of the operator.

Output Ports

**significance** *(sig)* The given performance vectors are compared and the result of the significance test is delivered through this port.

**performance** *(per)* This operator can have multiple *performance* output ports. When one output is connected, another *performance* output port becomes available which is ready to deliver another output (if any). The order of outputs remains the same. The performance vector delivered at the first *performance* input port of this operator is delivered at the first *performance* output port of the operator.

Parameters

**alpha** *(real)* This parameter specifies the probability threshold which determines if differences are considered as significant. If a test of significance gives a p-value lower than the significance level *alpha*, the null hypothesis is rejected. It is important to understand that the null hypothesis can never be proven. A set of data can only reject a null hypothesis or fail to reject it. For example, if comparison of two groups reveals no statistically significant difference between the two, it does not mean that there is no difference in reality. It only means that there is not enough evidence to reject the null hypothesis (in other words, the experiment fails to reject the null hypothesis).
8.3. Significance

Related Documents

ANOVA (954)

Tutorial Processes

Comparison of performance vectors using statistical significance tests

Many RapidMiner operators can be used to estimate the performance of a learner or a preprocessing step etc. The result of these validation operators is a performance vector which collects the values of a set of performance criteria. For each criterion, the mean value and standard deviation are given. The question is how can these performance vectors be compared? Statistical significance tests like ANOVA or T-Test can be used to calculate the probability that the actual mean values are different. This Example Process performs exactly the same task.

This Example Process starts with a Subprocess operator which provides two performance vectors as output. Have a look at the inner operators of the Subprocess operator. The Generate Data operator is used for generating an ExampleSet. The Multiply operator is used for producing multiple copies of this ExampleSet. X-Validation operators are applied on both copies of the ExampleSet. The first X-Validation operator uses the Support Vector Machine (LibSVM) operator whereas the second X-Validation operator uses the Linear Regression operator in the training subprocess. The resultant performance vectors are the output of the Subprocess operator.

These performance vectors are compared using the T-Test and ANOVA operators respectively. The performance vectors and the results of the significance tests are connected to the result ports of the process and they can be viewed in the Results Workspace. Run the process and compare the results. The probabilities for a significant difference are equal since only two performance vectors were created.
8. Evaluation

In this case the SVM is probably better suited for the data set at hand since the actual mean values are probably different. SVM is considered better because its p-value is smaller than alpha which indicates a probably significant difference between the actual mean values.

ANOVA

This operator is used for comparison of performance vectors. It performs an analysis of variance (ANOVA) test to determine the probability for the null hypothesis i.e. 'the actual means are the same'.

Description

ANalysis Of VAriance (ANOVA) is a statistical model in which the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether or not the means of several groups are all equal, and therefore generalizes t-test to more than two groups. Doing multiple two-sample t-tests
would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. 'False positive' or Type I error is defined as the probability that a decision to reject the null hypothesis will be made when it is in fact true and should not have been rejected. RapidMiner provides the T-Test operator for performing the t-test. Paired t-test is a test of the null hypothesis that the difference between two responses measured on the same statistical unit has a mean value of zero.

**Differentiation**

**T-Test** Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. For this reason, ANOVA is useful in comparing two, three, or more means. See page 950 for details.

**Input Ports**

**performance** *(per)* This operator expects performance vectors as input it can have multiple inputs. When one input is connected, another performance input port becomes available which is ready to accept another input (if any). The order of inputs remains the same. The performance vector supplied at the first input port of this operator is available at the first performance output port of the operator.

**Output Ports**

**significance** *(sig)* The given performance vectors are compared and the result of the significance test is delivered through this port.

**performance** *(per)* This operator can have multiple performance output ports. When one output is connected, another performance output port becomes available which is ready to deliver another output (if any). The order of outputs
remains the same. The performance vector delivered at first performance input port of this operator is delivered at the first performance output port of the operator.

Parameters

alpha (real) This parameter specifies the probability threshold which determines if differences are considered as significant. If a test of significance gives a p-value lower than the significance level alpha, the null hypothesis is rejected. It is important to understand that the null hypothesis can never be proven. A set of data can only reject a null hypothesis or fail to reject it. For example, if comparison of two groups reveals no statistically significant difference between the two, it does not mean that there is no difference in reality. It only means that there is not enough evidence to reject the null hypothesis (in other words, the experiment fails to reject the null hypothesis).

Related Documents

T-Test (950)

Tutorial Processes

Comparison of performance vectors using statistical significance tests

Many RapidMiner operators can be used to estimate the performance of a learner or a preprocessing step etc. The result of these validation operators is a performance vector which collects the values of a set of performance criteria. For each criterion, the mean value and standard deviation are given. The question is how
8.3. Significance

these performance vectors can be compared? Statistical significance tests like ANOVA or T-Test can be used to calculate the probability that the actual mean values are different. This Example Process performs exactly the same task.

This Example Process starts with a Subprocess operator which provides two performance vectors as output. Have a look at the inner operators of the Subprocess operator. The Generate Data operator is used for generating an ExampleSet. The Multiply operator is used for producing multiple copies of this ExampleSet. X-Validation operators are applied on both copies of the ExampleSet. The first X-Validation operator uses the Support Vector Machine (LibSVM) operator whereas the second X-Validation operator uses the Linear Regression operator in the training subprocess. The resultant performance vectors are the output of the Subprocess operator.

These performance vectors are compared using the T-Test and ANOVA operators respectively. The performance vectors and the results of the significance tests are connected to the result ports of the process and they can be viewed in the Results Workspace. Run the process and compare the results. The probabilities for a significant difference are equal since only two performance vectors were created. In this case the SVM is probably better suited for the data set at hand since the actual mean values are probably different. The SVM is considered better because its p-values is smaller than alpha which indicates a probably significant difference between the actual mean values.
Create Lift Chart

This operator generates a lift chart for the given model and Example-Set based on the discretized confidences and a Pareto chart.

Description

The Create Lift Chart operator creates a lift chart based on a Pareto plot for the discretized confidence values of the given ExampleSet and model. The model is applied on the ExampleSet and a lift chart is produced afterwards. Please note that any predicted label of the given ExampleSet will be removed during the application of this operator. In order to produce reliable results, this operator must be applied on data that has not been used to build the model, otherwise the resulting plot will be too optimistic.

The lift chart measures the effectiveness of models by calculating the ratio be-
tween the result obtained with a model and the result obtained without a model. The result obtained without a model is based on randomly selected records.

Input Ports

**example set** (*exa*) This input port expects an ExampleSet. It is the output of the Generate Direct Mailing Data operator in the attached Example Process. The output of other operators can also be used as input.

**model** (*mod*) This input port expects a model. It is the output of the Naive Bayes operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

**example set** (*exa*) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.

**model** (*mod*) The model that was given as input is passed without changing to the output through this port. This is usually used to reuse the same model in further operators or to view the model in the Results Workspace.

**lift pareto chart** (*lif*) For the given model and ExampleSet a lift chart is generated based on the discretized confidences and a Pareto chart. This lift chart is delivered through this port.

Parameters

**target class** (*string*) This parameter indicates the target class for which the lift chart should be produced.

**binning type** (*selection*) This parameter indicates the binning type of the con-
8. Evaluation

Number of Fiducial Digits

- **Number of bins** (integer) This parameter specifies the number of bins the confidence should be discretized into. This parameter is only available when the *binning type* parameter is set to 'simple' or 'frequency'.

- **Size of bins** (integer) This parameter specifies the number of examples that each bin should contain when the confidence is discretized. This parameter is only available when the *binning type* parameter is set to 'absolute'.

- **Automatic number of digits** (boolean) This parameter indicates if the number of digits should be automatically determined for the range names.

- **Number of digits** (integer) This parameter specifies the minimum number of digits to be used for the interval names. If this parameter is set to -1 then the minimal number is determined automatically. This parameter is only available when the *automatic number of digits* parameter is set to false.

- **Show bar labels** (boolean) This parameter indicates if the bars should display the size of the bin together with the amount of the target class in the corresponding bin.

- **Show cumulative labels** (boolean) This parameter indicates if the cumulative line plot should display the cumulative sizes of the bins together with the cumulative amount of the target class in the corresponding bins.

- **Rotate labels** (boolean) This parameter indicates if the labels of the bins should be rotated.

**Tutorial Processes**

**Creating lift chart for direct mailing data**

The Direct Mailing Data operator is used for generating an ExampleSet with 10000 examples. The Split Validation operator is applied on this ExampleSet. The *split ratio* parameter is set to 0.7 and the *sampling type* parameter is set to 'shuffled sampling'. Here is an explanation of what happens inside the Split Validation operator.
8.4. Visual Evaluation

1. The Split Validation operator provides a training data set through the training port of the training subprocess. This training data set is used as input for the Naive Bayes operator. Thus the Naive Bayes classification model is trained on this training data set.

2. The Naive Bayes operator provides the Naive Bayes classification model as its output. This model is connected to the model port of the training subprocess.

3. The Naive Bayes model that was provided at the model port of the training subprocess is delivered by the Split Validation operator at the model port of the testing subprocess. This model is provided as input at the model port of the Create Lift Chart operator.

4. The Split validation operator provides the testing data set through the test set port of the testing subprocess. This testing data set is provided as input to the Create Lift Chart operator.

5. The Create Lift Chart operator generates a lift chart for the given model and ExampleSet based on the discretized confidences and a Pareto chart. The lift chart is provided to the Remember operator to store it in the object store.

6. The Apply Model operator is provided with the testing data set and the model. The Apply Model operator applies the model on the testing data set and the resultant labeled data set is delivered as output. This labeled data set is provided as input to the Performance operator.

7. The Performance operator evaluates the statistical performance of the model through the given labeled data set and generates a performance vector which holds information about various performance criteria.

Outside the Split Validation operator, the Recall operator is used for fetching the lift chart from the object store. The lift chart is delivered to the output and it can be seen in the Results Workspace.
8. Evaluation

Compare ROCs

This operator generates ROC charts for the models created by the learners in its subprocess and plots all the charts in the same plotter for comparison.

Description

The Compare ROCs operator is a nested operator i.e. it has a subprocess. The operators in the subprocess must produce a model. This operator calculates ROC curves for all these models. All the ROC curves are plotted together in the same plotter.

The comparison is based on the average values of a k-fold cross validation. Please study the documentation of the X-Validation operator for more information about cross validation. Alternatively, this operator can use an internal split into a test and a training set from the given data set in this case the operator behaves like the Split Validation operator. Please note that any former predicted label of the given ExampleSet will be removed during the application of this operator.

ROC curve is a graphical plot of the sensitivity, or true positive rate, vs. false positive rate (one minus the specificity or true negative rate), for a binary classifier system as its discrimination threshold is varied. The ROC can also be represented equivalently by plotting the fraction of true positives out of the positives (TPR
8.4. Visual Evaluation

= true positive rate) vs. the fraction of false positives out of the negatives (FPR = false positive rate).

ROC curves are calculated by first ordering the classified examples by confidence. Afterwards all the examples are taken into account with decreasing confidence to plot the false positive rate on the x-axis and the true positive rate on the y-axis. With optimistic, neutral and pessimistic there are three possibilities to calculate ROC curves. If there is more than one example for a confidence with optimistic ROC calculation the correct classified examples are taken into account before looking at the false classification. With pessimistic calculation it is the other way round: wrong classifications are taken into account before looking at correct classifications. Neutral calculation is a mix of both calculation methods described above. Here correct and false classifications are taken into account alternately. If there are no examples with equal confidence or all examples with equal confidence are assigned to the same class the optimistic, neutral and pessimistic ROC curves will be the same.

Input Ports

example set (exa) This input port expects an ExampleSet with binominal label. It is the output of the Retrieve operator in the attached Example Process. The output of other operators can also be used as input.

Output Ports

example set (exa) The ExampleSet that was given as input is passed without changing to the output through this port. This is usually used to reuse the same ExampleSet in further operators or to view the ExampleSet in the Results Workspace.
rocComparison (roc) The ROC curves for all the models are delivered from this port. All the ROC curves are plotted together in the same plotter.
Parameters

**number of folds (integer)** This parameter specifies the number of folds to use for the cross validation evaluation. If this parameter is set to -1 this operator uses split ratio and behaves like the Split Validation operator.

**split ratio (real)** This parameter specifies the relative size of the training set. It should be between 1 and 0, where 1 means that the entire ExampleSet will be used as training set.

**sampling type (selection)** Several types of sampling can be used for building the subsets. Following options are available:

- **Linear sampling** Linear sampling simply divides the ExampleSet into partitions without changing the order of the examples i.e. subsets with consecutive examples are created.

- **Shuffled sampling** Shuffled sampling builds random subsets of the ExampleSet. Examples are chosen randomly for making subsets.

- **Stratified sampling** Stratified sampling builds random subsets and ensures that the class distribution in the subsets is the same as in the whole ExampleSet. For example in the case of a binominal classification, Stratified sampling builds random subsets so that each subset contains roughly the same proportions of the two values of class labels.

**use local random seed (boolean)** This parameter indicates if a local random seed should be used for randomizing examples of a subset. Using the same value of local random seed will produce the same subsets. Changing the value of this parameter changes the way examples are randomized, thus subsets will have a different set of examples. This parameter is only available if Shuffled or Stratified sampling is selected. It is not available for Linear sampling because it requires no randomization, examples are selected in sequence.

**local random seed (integer)** This parameter specifies the local random seed. This parameter is only available if the use local random seed parameter is set to true.

**use example weights (boolean)** This parameter indicates if example weights...
should be considered. If this parameter is not set to true then weight 1 is used for each example.

**roc bias (selection)** This parameter determines how the ROC are evaluated i.e. correct predictions are counted first, last, or alternately. ROC curves are calculated by first ordering the classified examples by confidence. Afterwards all the examples are taken into account with decreasing confidence to plot the false positive rate on the x-axis and the true positive rate on the y-axis. With optimistic, neutral and pessimistic there are three possibilities to calculate ROC curves. If there are no examples with equal confidence or all examples with equal confidence are assigned to the same class the optimistic, neutral and pessimistic ROC curves will be the same.

- **optimistic** If there is more than one example for a confidence with optimistic ROC calculation the correct classified examples are taken into account before looking at the false classification.

- **pessimistic** With pessimistic calculation wrong classifications are taken into account before looking at correct classifications.

- **neutral** Neutral calculation is a mix of both optimistic and pessimistic calculation methods. Here correct and false classifications are taken into account alternately.

**Tutorial Processes**

Comparing different classifiers graphically by ROC curves

This process shows how several different classifiers could be graphically compared by means of multiple ROC curves. The 'Ripley-Set' data set is loaded using the Retrieve operator. The Compare ROCs operator is applied on it. Have a look at the subprocess of the Compare ROCs operator. You can see that three different learners are applied i.e. Naive Bayes, Rule Induction and Decision Tree. The resultant models are connected to the outputs of the subprocess. The Com-
8. Evaluation

pare ROCs operator calculates ROC curves for all these models. All the ROC curves are plotted together in the same plotter which can be seen in the Results Workspace.
8 Index

AdaBoost, 754
Add Noise, 171
Agglomerative Clustering, 832
Aggregate, 615
ANOVA, 954
Append, 623
Apply Model, 870
Apply Threshold, 883
Backward Elimination, 488
Bagging, 757
Bayesian Boosting, 749
Bootstrapping Validation, 903
Branch, 71
CHAID, 669
Classification by Regression, 747
Cluster Density Performance, 945
Cluster Distance Performance, 941
Collect, 79
Combine Performances, 915
Compare ROCs, 962
Copy File, 151
Correlation Matrix, 856
Create Association Rules, 849
Create Directory, 161
Create Lift Chart, 958
Create Threshold, 880
Cross Distances, 865
Cut, 532
Data to Similarity, 860
Data to Weights, 791
Date to Nominal, 337
Date to Numerical, 332
DBSCAN, 817
De-Pivot, 608
Decision Stump, 673
Decision Tree, 659
Declare Missing Value, 507
Default Model, 641
Delete File, 156
Detect Outlier (COF), 569
Detect Outlier (Densities), 563
Detect Outlier (Distances), 560
Detect Outlier (LOF), 566
Discretize by Binning, 368
Discretize by Entropy, 389
Discretize by Frequency, 375
Discretize by Size, 361
Discretize by User Specification, 382
Drop Uncertain Predictions, 886
Index

Exchange Roles, 277
Execute Process, 132
Execute Program, 145
Execute Script, 135
Execute SQL, 141
Expectation Maximization Clustering, 823
Extract Cluster Prototypes, 841
Extract Macro, 110
Extract Performance, 912
Fill Data Gaps, 558
Filter Example Range, 585
Filter Examples, 575
Find Threshold, 876
Format Numbers, 349
Forward Selection, 483
FP-Growth, 844
Free Memory, 179

Generalized Hebbian Algorithm, 445
Generalized Sequential Patterns, 852
Generate Aggregation, 419
Generate Attributes, 402
Generate Concatenation, 417
Generate Copy, 400
Generate Data, 163
Generate Direct Mailing Data, 167
Generate Empty Attribute, 398
Generate Function Set, 425
Generate ID, 396
Generate Macro, 102
Generate Nominal Data, 165
Generate Sales Data, 169
Group Models, 873

Guess Types, 356
Handle Exception, 12
ID3, 665
Independent Component Analysis, 442
Item Distribution Performance, 947
Join, 625
Join Paths, 10
K-Means, 801
K-Means (Kernel), 805
K-Medoids, 810
K-NN, 644
Linear Discriminant Analysis, 738
Linear Regression, 701
Log, 118
Log to Data, 129
Logistic Regression, 709
Loop, 32
Loop and Average, 54
Loop Attributes, 36
Loop Clusters, 50
Loop Collection, 85
Loop Data Sets, 52
Loop Examples, 46
Loop Files, 62
Loop Parameters, 57
Loop Values, 42
Map, 521
Materialize Data, 177
Merge, 544
MetaCost, 765
Move File, 158
Multiply, 7

Naive Bayes, 651
Naive Bayes (Kernel), 655
Neural Net, 697
Nominal to Binominal, 300
Nominal to Date, 319
Nominal to Numerical, 311
Nominal to Text, 306
Normalize, 512
Numerical to Binominal, 279
Numerical to Polynominal, 285
Numerical to Real, 290
Optimize by Generation (YAGGA), 429
Optimize Parameters (Evolutionary), 26
Optimize Parameters (Grid), 21
Optimize Selection, 492
Optimize Selection (Evolutionary), 499
Optimize Weights (Evolutionary), 795
Parse Numbers, 344
Perceptron, 694
Performance, 908
Performance (Binominal Classification), 925
Performance (Classification), 918
Performance (Costs), 937
Performance (Regression), 933
Pivot, 604
Polynomial by Binominal Classification, 744
Polynomial Regression, 705
Principal Component Analysis, 434
Principal Component Analysis (Kernel), 437
Provide Macro as Log Value, 126
Random Forest, 680
Random Tree, 676
Read Access, 199
Read AML, 202
Read ARFF, 206
Read csv, 189
Read Database, 210
Read Excel, 194
Read Model, 222
Read SAS, 197
Read SPSS, 219
Read Weights, 224
Real to Integer, 295
Recall, 4
Remap Binominals, 546
Remember, 1
Remove Attribute Range, 465
Remove Correlated Attributes, 472
Remove Duplicates, 580
Remove Useless Attributes, 468
Rename, 265
Rename by Replacing, 267
Rename File, 153
Replace, 527
Replace Missing Values, 552
Retrieve, 183
Rule Induction, 684
Sample, 588
Index

Sample (Bootstrapping), 595
Sample (Stratified), 591
Scale by Weights, 519
Select, 82
Select Attributes, 455
Select by Weights, 462
Select Subprocess, 75
Self-Organizing Map, 452
Set Data, 504
Set Macro, 93
Set Macros, 98
Set Minus, 630
Set Parameters, 17
Set Role, 273
Singular Value Decomposition, 448
Sort, 601
Split, 537
Split Data, 598
Split Validation, 891
Stacking, 761
Store, 185
Stream Database, 215
Subgroup Discovery, 687
Subprocess, 89
Superset, 637
Support Vector Clustering, 828
Support Vector Machine, 714
Support Vector Machine (Evolutionary), 726
Support Vector Machine (LibSVM), 720
Support Vector Machine (PSO), 733
T-Test, 950
Text to Nominal, 327
Throw Exception, 15
Top Down Clustering, 838
Transpose, 612
Tree to Rules, 692
Union, 633
Update Database, 240
Vote, 742
Weight by Chi Squared Statistic, 777
Weight by Component Model, 788
Weight by Correlation, 774
Weight by Information Gain, 769
Weight by Information Gain Ratio, 771
Weight by PCA, 785
Weight by Relief, 780
Weight by SVM, 783
Weights to Data, 793
Work on Subset, 477
Write, 227
Write AML, 229
Write Arff, 232
Write as Text, 149
Write Clustering, 250
Write Constructions, 255
Write Database, 236
Write Model, 248
Write Parameters, 259
Write Performance, 257
Write Special Format, 244
Write Threshold, 261
Write Weights, 252
X-Prediction, 66
X-Validation, 896