machine learning framework
for Mathematica
Version 2.1.0

Documentation
Getting Started with the *machine learning framework*

machine learning framework for *Mathematica*

Installation of the *machine learning framework*

The *machine learning framework* 2.1.0 is currently available for Microsoft Windows, Linux and Mac OS X. In general it is sufficient to run the provided installer with its default setting. For the actual installation process, see the detailed installation instructions below (also available at www.unisoftwareplus.com/products/mlf/).

- **Installation and license information for Windows**
- **Installation and license information for Linux**
- **Installation and license information for Mac OS X**

What is the *machine learning framework*?

The *machine learning framework* for *Mathematica* is a collection of powerful machine learning algorithms integrated into a framework for the main purpose of data analysis. *Fuzzy logic* is one of its key techniques. The framework allows for combining different machine learning algorithms to solve one single problem. This combination of distinct algorithms may give the user unforeseen insights into its data. The algorithms are highly parametrizable. Given this parametrizability combined with the efficient core engine of the *machine learning framework* for *Mathematica*, the user is able to analyze their data interactively, with short cycles of changing parameter settings and examining the results.

The *machine learning framework* for *Mathematica* combines
- efficiency and performance behind the scenes delivered by an optimized computational kernel - the core engine - realized in C++, and
- the ease of use supplied by the manipulation, descriptive programming, and graphical capabilities of *Mathematica*.

- **Powered by fuzzy logic**

The framework draws its power from integrating *fuzzy logic* wherever possible. This yields results where crisp Boolean yes/no decisions can be replaced with smooth, continuous transitions. In the realm of data analysis, smooth results very often model the underlying correlations within the data more realistically than crisp ones. For instance, the status of belonging to a certain class may be modeled more appropriately by allowing overlaps and degrees of membership instead of only the two possibilities of belonging or not belonging.

- **Modern software architecture**

The software architecture of the *machine learning framework* for *Mathematica* is based on the principles of modern object-oriented and template-based programming. Standard libraries are used to ensure portability of the C++ kernel.

Mathematica front end

The *machine learning framework* for *Mathematica* is used from the *Mathematica* front end. By using the *Mathematica* front end, one has access to all *Mathematica* functions, including the graphical manipulation tools, and, with the *Mathematica* programming language, one has access to an elegant scripting language.

Wide range of machine learning algorithms

The *machine learning framework* for *Mathematica* covers a wide range of machine learning algorithms which can be integrated to work together and therefore yield new results.

Supervised Learning

- FS-ID3 (fuzzy decision trees)
- FS-LiRT (fuzzy regression trees)
- FS-FOIL (fuzzy rule learning)
- FS-MINER (fuzzy rule learning)
- LAPOC (optimization of fuzzy controllers / regression trees)
- RENO (optimization of fuzzy controllers)
- Ridge regression
- Additive regression
- Quadratic regression model
- Neural networks
- Gaussian process regression
- Support vector machines
- Boosting

Unsupervised Learning

- SOM (Kohonen maps)
- Fuzzy c-means (clustering)
- WARD clustering (crisp clustering)
- LVQ (learning vector quantization)

Additional features

- Fuzzy logic (using different types of fuzzy sets and t-norms)
- Fuzzy inference (Mamdani, Sugeno, Takagi-Sugeno-Kang)
- Advanced data visualization and data manipulation
- Statistical methods (including correlation plots, mutual information with plots, etc.)

How to use this manual

The readers are assumed to be familiar with using *Mathematica* notebooks. They get introduced into the realm of data analysis using machine learning algorithms by this manual and by studying the on-line tutorial and the templates provided.

The following resources are available:

- **On-line Help** (contains examples and a command index)
- **Basic tutorial** (describes how *mlf* can be used)
  - this notebook
  - Supervised Data Analysis
  - Approximation of Numerical Functions
  - Unsupervised Data Analysis; including image analysis
  - Object and Data Handling, Fuzzy Logic Operations, Visualization
- **Templates** (predefined templates to facilitate your analysis tasks)
  - Supervised data analysis I: Classification
  - Supervised Data Analysis II: Numerical Approximation
  - Unsupervised Data Analysis
- **Descriptions of theoretical concepts** (see Overview)
- **Other tutorials** (see Overview)

Analysis Examples

Loading the *mlf* package

```
Needs["mlf`"]
```

Remark: After the evaluation of the above command, all *mlf* related functions are loaded, *mlf*`s optimized C++ core (mlf.exe) is started and a *MathLink* connection to mlf.exe is established. Almost all computationally intensive functions are actually evaluated within the *mlf* (C++) core, whereby the necessary data is sent and retrieved via the *MathLink* connection.

```
MLFVersion[]
```

```
Machine learning framework for Mathematica 8 and 9
2.1.0 for MacOSX (x86_64) (Jun  3 2013)
```

Example 1: Iris data set

In this example we will analyze a simple data set which contains data of about 150 iris flowers of different types, or species (this is the well-known iris data set, which can be obtained from the UCI machine learning repository at http://www.ics.uci.edu/~mlearn/MLRepository.html). This data set describes different specimen of the flowers shown below.

iris setosa

iris versicolor

iris virginica

The blossom of each flower is described by four attributes: the length and width of the sepals and the length and width of the petals. The fifth attribute specifies to which class (species) of iris the flower belongs. Hence we have four numerical attributes and one categorical attribute for each flower. The goal of the analysis is to find out how we can decide to which category a new, unclassified flower belongs.

Load the data set

The data is stored in a text file (one row per flower) where the first row contains the attribute names (column headers). The last column contains the categorical attribute. To load the data such that they can be processed by *mlf*, we use the command `LoadData`.

```
irisData = LoadData["iris.txt", "Table"]
```

```
Name`TrainData7
```

Remark: The command `LoadData` first loads the data via `Import` into *Mathematica*, then does some necessary preprocessing and sends the data to the *mlf* core (mlf.exe). The value returned by LoadData is just a symbol which allows to reference the dataset but does not contain the data itself. Warning: multiple evaluation of above statement will fill up the memory used by the *mlf* core as we want to enable working with several (different) data sets at one time; the on-line help for `LoadData` shows how to release memory again.

To get a first impression of how the data look like, we print out every 13th row in tabular form:

```
TableForm[
 GetOriginalData[irisData][[Table[i, {i, 1, 150, 13}]]],
 TableHeadings → {None, GetParamDS[irisData, Labels]}
]
```

sepal_length	sepal_width	petal_length	petal_width	class
5.1	3.5	1.4	0.2	Iris-versicol
4.3	3.	1.1	0.1	Iris-versicol
5.	3.4	1.6	0.4	Iris-versicol
5.1	3.4	1.5	0.2	Iris-versicol
6.9	3.1	4.9	1.5	Iris-setosa
6.7	3.1	4.4	1.4	Iris-setosa
6.	2.9	4.5	1.5	Iris-setosa
6.1	3.	4.6	1.4	Iris-setosa
6.5	3.	5.8	2.2	Iris-virginica
7.7	3.8	6.7	2.2	Iris-virginica
7.4	2.8	6.1	1.9	Iris-virginica
6.8	3.2	5.9	2.3	Iris-virginica

`GetOriginalData` retrieves the data in their original form - that is, text attributes are shown as such and not with their internal integer representation. (To get the data in their internal representation, use `MLFGetData`.) `GetParamDS[` *dataset*`,` *labels*`]` retrieves the column headers (attribute names).

The command `PrintColumnTypeOverview` prints out an overview about the attributes in the dataset thereby giving information about the type of the attribute and the number of numeric, string and missing values:

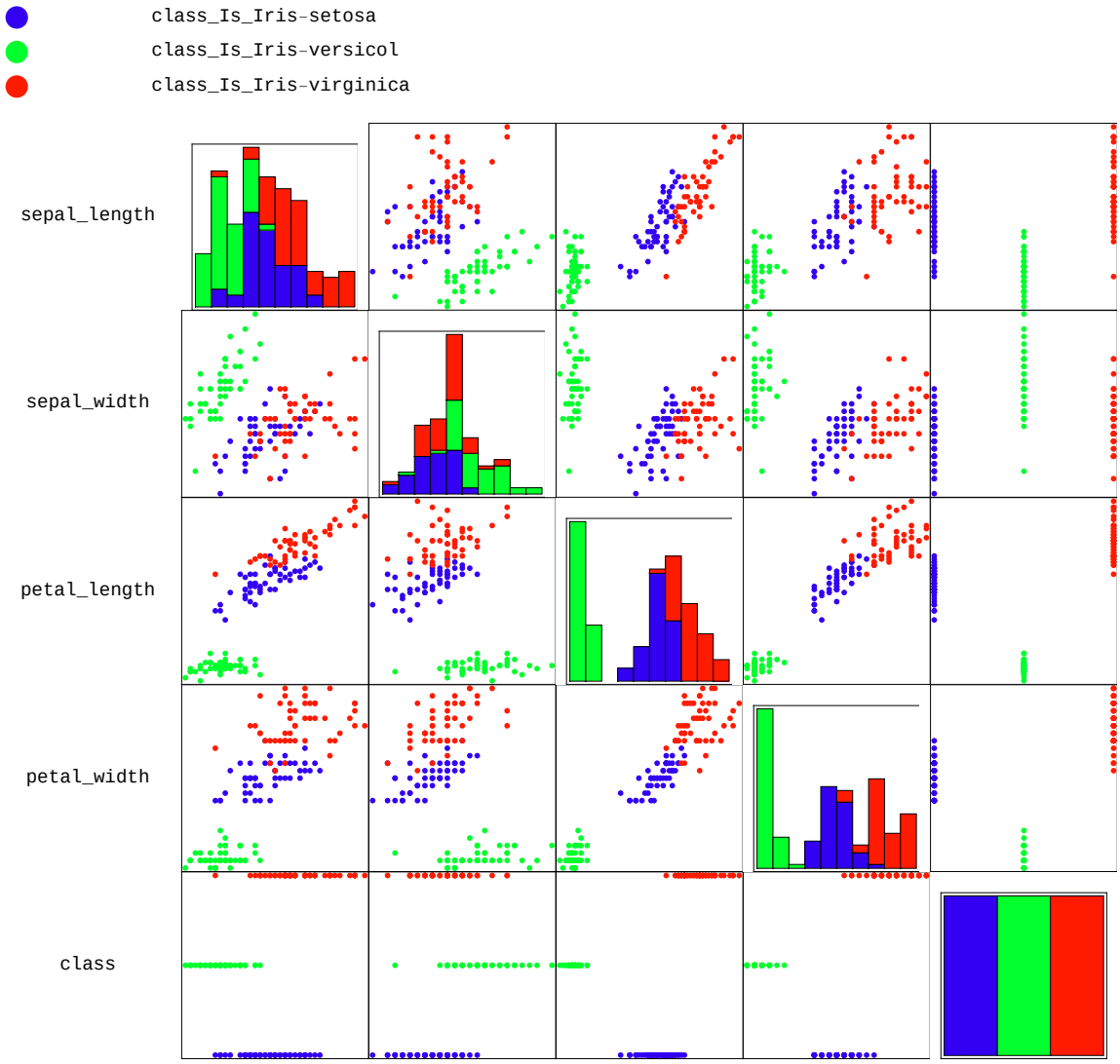
```
PrintColumnTypeOverview[ irisData ]
```

No.	NAME	#NUMERIC	#STRING	#NULL	TYPE	#DIFFVALS	USEFUL
1	sepal_length	150	0	0	Numeric	35	True
2	sepal_width	150	0	0	Numeric	23	True
3	petal_length	150	0	0	Numeric	43	True
4	petal_width	150	0	0	Numeric	22	True
5	class	0	150	0	Categorical	3	True

- **Scatter Plots**

To get a first impression of how the classes are distributed, we can create scatter plots and colorize the output according to the class label. To do so, we use the command `PlotAttributeMatrix` which produces scatter plots for all possible pairs of attributes. As we want to separate the data with respect to the class attribute, we specify a goal for colorizing the sample points by setting the option **`Goal→5`**.

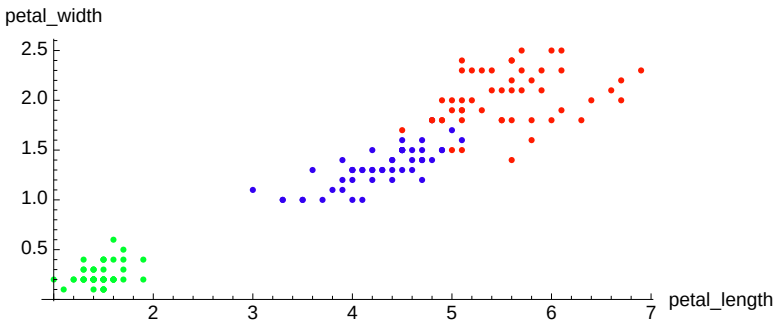
```
PlotAttributeMatrix[irisData, Goal → 5, ClassLegend → True, ImageSize → 600]
```



The main diagonal of the graphics shows histograms of the data w.r.t. the respective attribute and colorized w.r.t. the goal attribute (like the scatter plots).

We can see that petal length vs. petal width separates the data quite well. We will now create this single scatter plot by using the **PlotAttributes** command, where we specify the dimensions to plot with **Dims→{3,4}** and the goal column with **Goal→5**.

```
Show[Graphics[PlotAttributes[irisData, Dims → {3, 4}, Goal → 5, PointSize → 0.01],
  Axes → True, AxesLabel → GetParamDS[irisData, Labels][[{3, 4}]],
  BaseStyle → {FontFamily → "Helvetica", FontSize → 10}, ImageSize → 400]]
```



■ Finding clusters

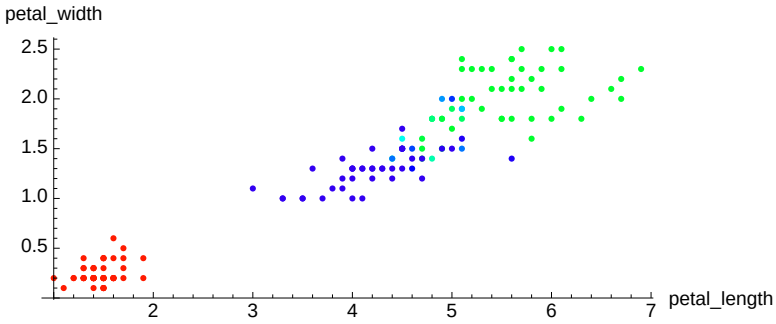
As a next step we use the k-means algorithm to find groups of flowers which are similar, i.e., we look for clusters in the data. We are looking for three groups (**Size**→3) and concentrate on the four input predicates only, using

option **Weights**.

```
clustering = CreateKMeans[irisData, Size → 3, Weights → {1, 1, 1, 1, 0}];
```

We visualize the clustering in a scatter plot of attributes 3 and 4, where the colors of the dots represent cluster membership.

```
Show[
 Graphics[PlotAttributes[irisData, Dims → {3, 4}, Goal → clustering, PointSize → 0.01],
  Axes → True, AxesLabel → GetParamDS[irisData, Labels][[{3, 4}]],
  BaseStyle → {FontFamily → "Helvetica", FontSize → 10}, ImageSize → 400]]
```



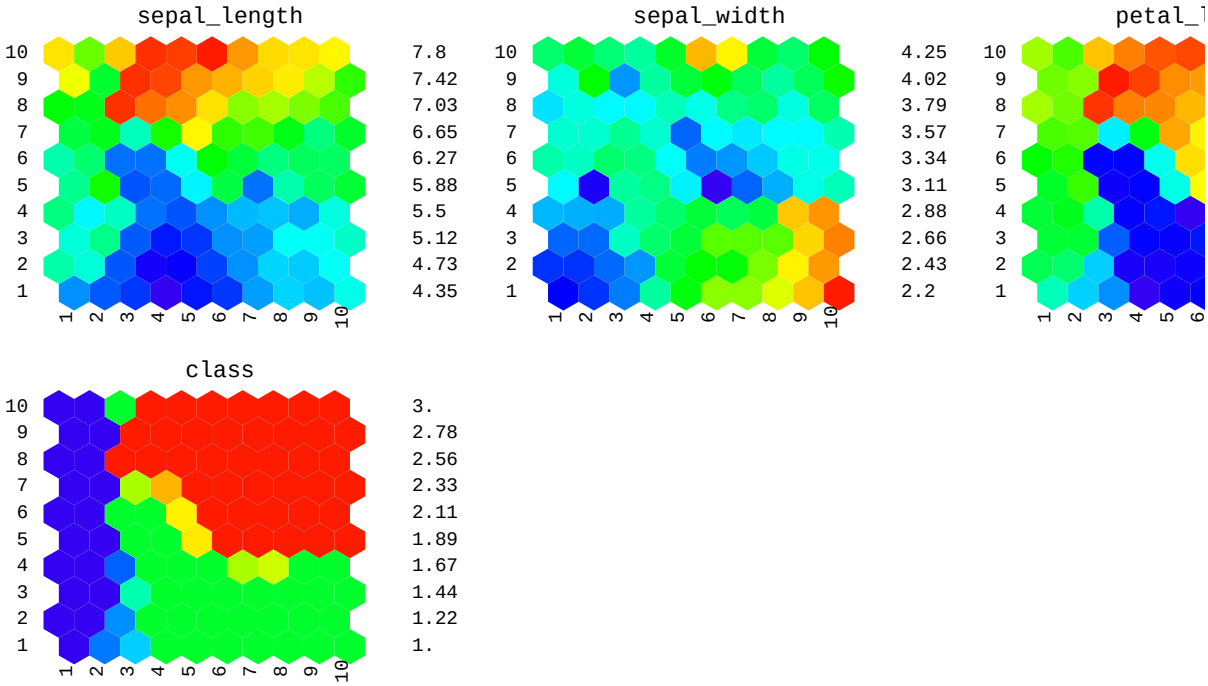
Each color represents a different cluster. We can see that the cluster in the lower left corner is well separated from he other two clusters and that it corresponds directly to one class of flowers. The other two clusters, however, do not exactly correspond to a single class of flowers.

- **Computing a self-organizing map (SOM)**

To visualize further dependencies in the data, we now create a self-organizing map and plot it.

```
somIris = TrainModel[irisData, Algorithm → CreateSOM];
```

```
PlotModel[somIris, ImageSize → 1000]
```

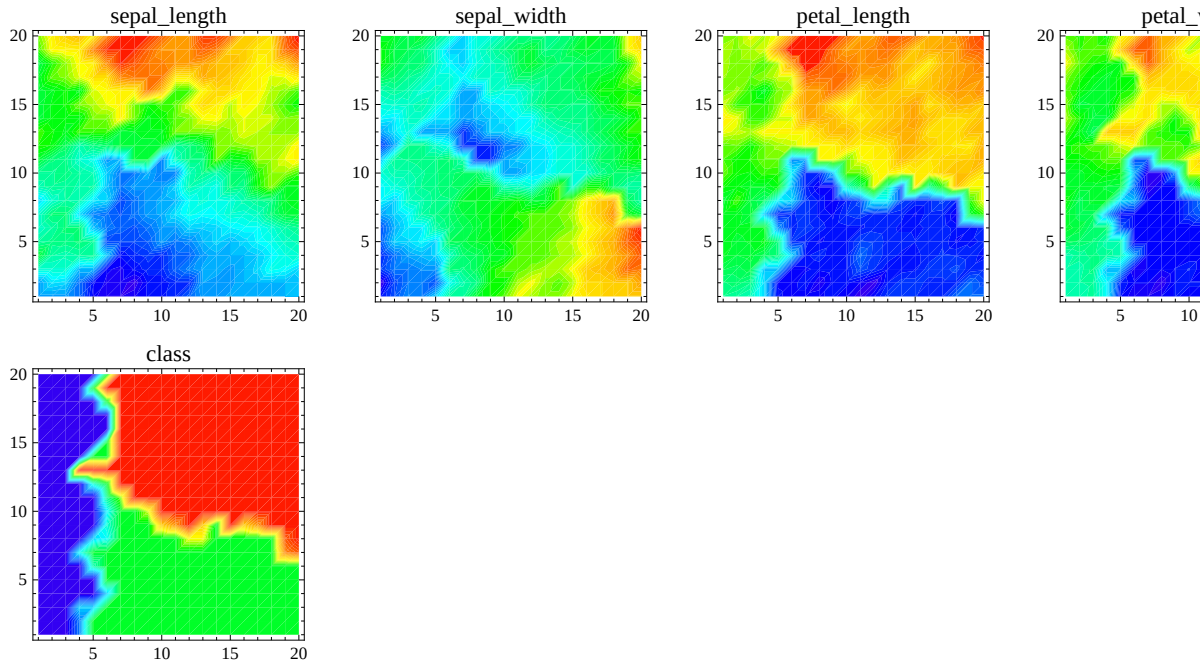


These plots show us that the petal length and petal width are highly correlated as the red/green/blue areas in one plot correspond to the according areas in the other plot. We can also see that these two parameters are a good indicator for the class of flowers (species).

The functions `TrainModel` and `PlotModel` used above represent a set of higher-level functions designed for quick and first-glance analysis. In order to dig deeper, one will sometimes have to use more specific functions which are highly parametrizable. The latter functions will be explained in more detail in later sections of the tutorial. For getting started, we will use the easier, higher-level functions where applicable.

The following is to demonstrate the difference between the two levels of the interface:

```
somIris2 = CreateSOM[irisData, Size -> {20, 20}, Cooldown → 0.1, MaxIter → 50];

PlotSOMContour[somIris2, ImageSize → 700]
```



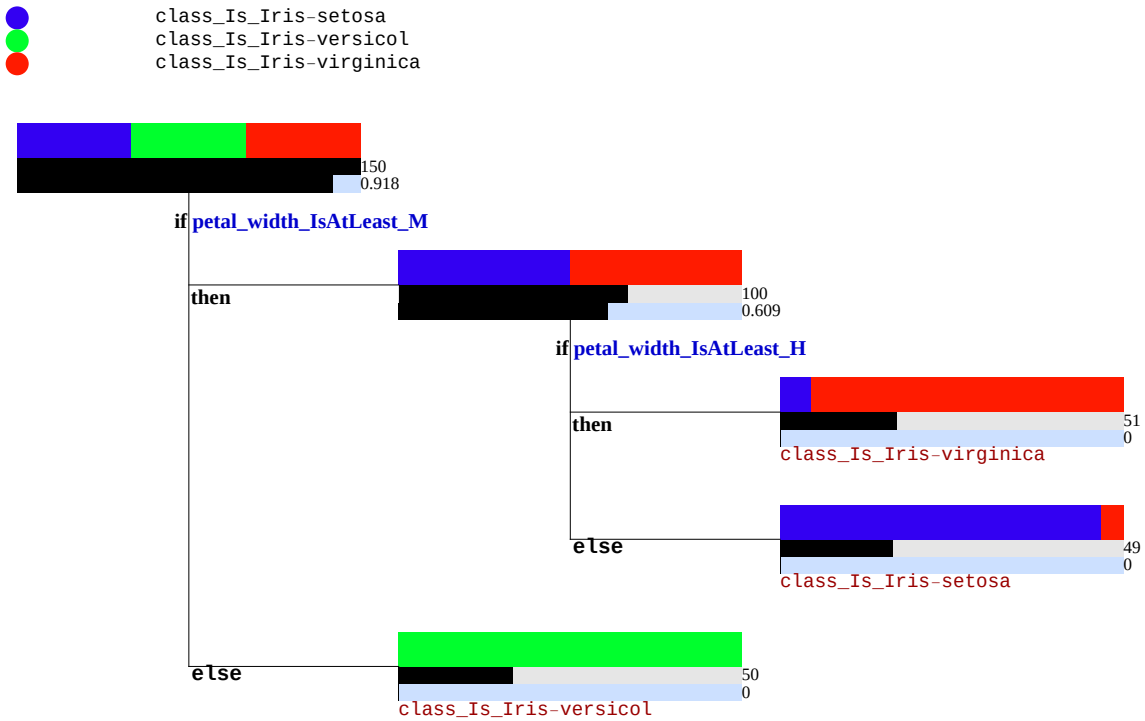
■ Computing a decision tree

Now we will build models of the data suitable for prediction of the flower type based on the four numerical attributes. The key models can also be graphically depicted in a human-readable form suitable for better understand ing the relations between the different attributes.

We first try the decision-tree rule-induction method *FS-ID3*, which is a generalization of Quinlan's ID3 method, adapted for the use of fuzzy sets (which is indicated by the prefix "FS"). A decision tree can be thought of as a set of if-then-else statements and can be used to solve a wide range of real-world classification problems.

In order to generate a decision tree, we have to specify a goal attribute. With the higher-level functions of *mlf*, if not specified otherwise, the last column is automatically taken for the goal attribute. In this case, this is the "class" attribute.

```
id3tree = TrainModel[irisData, Algorithm → CreateID3];

PlotModel[id3tree, ImageSize → 600]
```



The figure above gives a human-readable representation of the computed decision tree. Each leaf of the tree corresponds to a rule. E.g. the leaf labelled `class_Is_Iris-setosa` should be read as the rule, "If petal width is at least medium and petal width is *not* at least high then type is Iris-setosa." In addition, the graphical output provides statistical information at each node. Each node consists of three bars. The first one indicates the class distribution of samples belonging to this node. The second bar indicates the relative support, i.e. the relative number of samples belonging to this node with respect to the total number of samples (the absolute support is given in parentheses after the label of the node). The third bar indicates the entropy gain to the next level. A high entropy gain indicates important decisions. (NB: the notion of *entropy gain* is a technical term used throughout the literature which may be a little confusing; what is meant is actually an information gain.)

We can see that the predicate used in the root noode, `petal_width_IsAtLeast_M`, separates the Iris-versicol flowers in the "`F`" (for `False`) branch. Hence the "`T`" (for `True`) branch contains only samples of the other two classes. These samples are then separated by the predicate `petal_width_IsAtLeast_H` into two sets which contain almost only one species of flower.

Computing a rule base

To explicitly create a set of rules that can be used to predict the classification of a new flower, also a modification of Quinlan's FOIL algorithm, called *FS-FOIL*, is implemented in *mlf*.

We can generate the rule base using the `CreateFOIL` algorithm, which tries to find rules to predict the three classes.

```
ruleBase = TrainModel[irisData, Algorithm → CreateFOIL];
```

```
PlotModel[ruleBase]
```

Class	{}	Condition
class_Is_Iris-setosa	⇐	petal_width_Is_M
class_Is_Iris-versicol	⇐	petal_width_IsAtMost_L
class_Is_Iris-virginica	⇐	petal_width_IsAtLeast_H

Each row of the table above constitutes one rule of the form, "if *condition*, then the flower is in *class*". The conditions can be more complex, of course, as will be shown in later chapters of the tutorial.

Example 2: Numerical Approximation (artificial data set)

This example is intended to illustrate the abilities of the *machine learning framework* for *Mathematica* to create computational models from data for numerical predicates which are easy to interpret (in contrast to e.g. neural networks). For this purpose, we will first create an artificial data set representing a two-dimensional function.

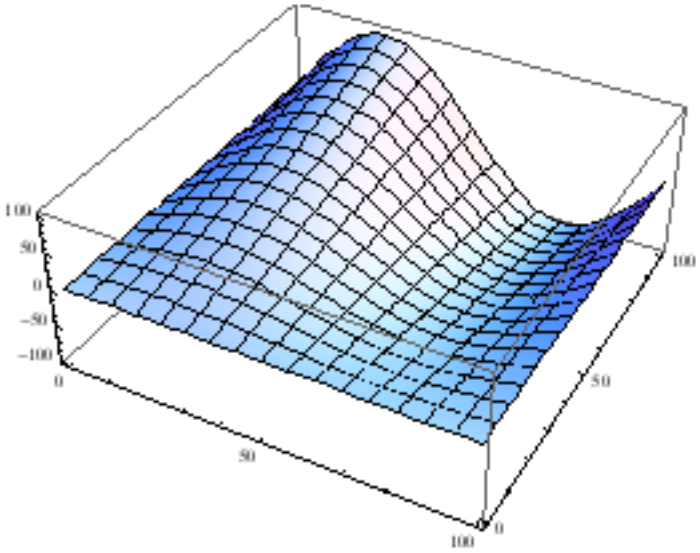
Afterwards, a regression tree which approximates the desired function is computed and compared to linear model.

Although, in this example, only a regression tree and a linear regression are created, it is, however, possible to use any model created with the *machine learning framework* for *Mathematica* (rule bases, self-organizing maps, etc.) to solve numerical approximation problems. Furthermore it is possible to optimize the obtained regression tree or rule base afterwards by applying numerical optimization methods on the underlying fuzzy sets to increase the accuracy of the model.

- **Set up data**

We define a simple two-dimensional set of sample points which we will use to create an approximation function afterwards.

```
range = 100;
f[x_, y_] := N@(Sin[x / range * 2 * Pi] * y);
Plot3D[f[x, y], {x, 0, range}, {y, 0, range}]
```



(Please do not be put off by spell warnings - while they do not constitute error messages, they *can* help to find errors. If they irritate you, use e.g. `Off[General::spell1]` to get rid of them.)

In order to create models using machine-learning algorithms, we take random samples out of this set:

```
samples = 1000;
trainData = Table[{x = Random[] * range, y = Random[] * range, f[x, y]}, {samples}];
headers = {"x", "y", "z"};
```

To be able to use the generated data in *mlf* algorithms, we have to make them available to the *mlf* core. If the data are provided in the form of a file and can be used as is, this is done within the `LoadData` command, as we did in the iris example above. Here, we have to use the commands `DataSet` and `Def` instead since the date is available as a *Mathematica* matrix. The result is the same: a reference to the data set in the *mlf* core is returned:

```
trainDataSet = Def[ "refToDataSetInMlfCore", DataSet[ trainData, headers] ]
```

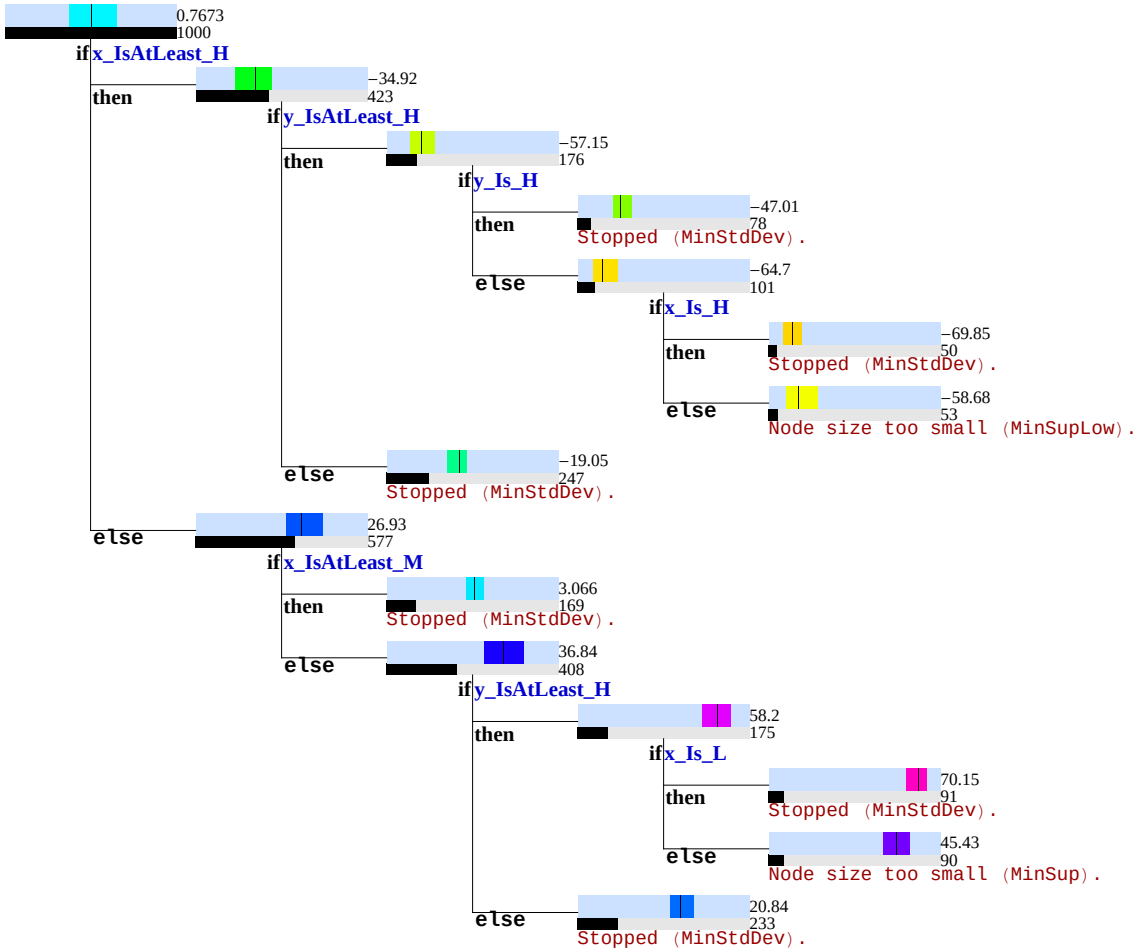
```
Name`refToDataSetInMlfCore
```

- **Create a regression tree**

We now create a regression tree using the training data set, taking *z* (the last column) as the goal parameter.

```
lirt = TrainModel[trainDataSet, Algorithm → CreateLIRT];
```

```
PlotModel[lirt, LeafInfo → "Mean"]
```



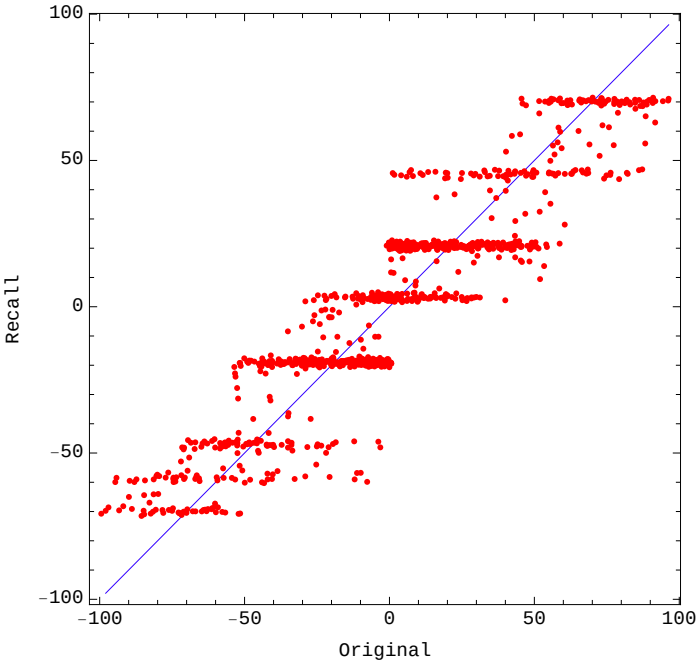
To evaluate the performance of the trained regression tree on the training data we employee the functions `TestModel` and `PrintMLFEvaluations`:

```
PrintMLFEvaluations[ TestModel[ lirt, trainDataSet ] ]
```

```
NumberOfInstances          1000
CorrelationCoefficient     0.923584
MSE                        249.815
RSE                        0.147888
MeanError                  -0.018146
NormalizedMSE              0.00661842
MAE                        12.8866
RAE                        0.400659
NormalizedMAE              0.0663292
StdDev                     15.8134
TargetStdDev               41.1206
TargetMAD                  32.1634
RatioOfNullPredictions     0.
RMSE                       15.8055
RRMSE                      0.384562
ModelSize                  8
```

Using the function `PlotInputRecall` we can easily produce a figure which plots the original values of z against the predicted values. For better comparison, the identity line is shown. Note that MLFGetData[trainDataSet,All,-1] retrieves the values of the last column of the train data set.

```
orig = MLFGetData[trainDataSet, All, -1];
pred = Predict[lirt, trainDataSet];
PlotInputRecall[orig, pred]
```



Obviously the predicted values do deviate largely from the actual values. There are many possible reasons why a machine learning algorithm does not yield satisfactory results with some default settings, and therefore *mlf* algorithms typically come with a great number of options for tuning. We will deal with those options in more detail in later chapters of the tutorial. For now, we simply give an example which yields a better result than the one above. By setting `Optimization→True` we associate with each leaf of the regression tree not only a constant value but rather a linear model of the input attributes.

```
Options[CreateLIRT]
```

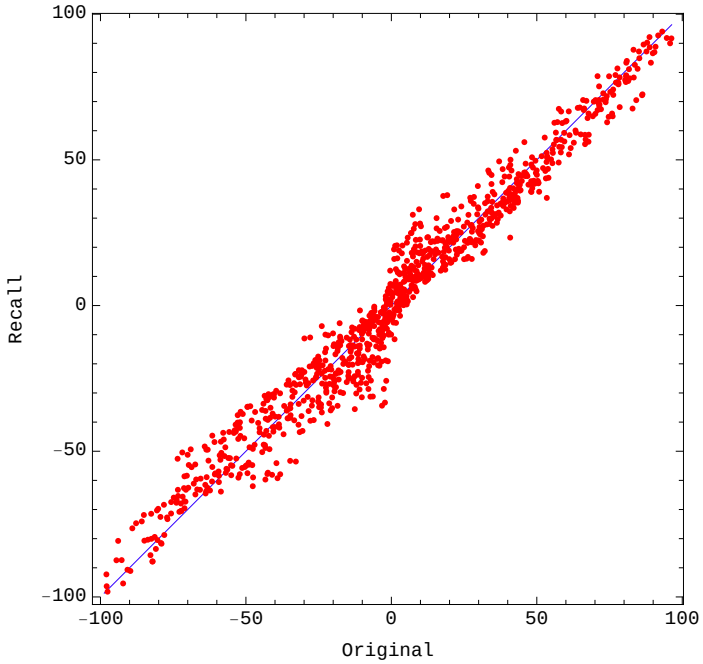
```
{Logic → Automatic, MinSup → 0.1, MinSupLow → 0.01, MinStdDev → 0.1,
 MinIncr → 0.01, MaxLevel → 10, Alternatives → 0, Optimization → False,
 Lambda → 0.1, VarSelection → False, MinApprox → 0.9, MinVars → 0,
 MaxVars → Automatic, Evaluation → Obj`MinSqrErrorEvaluation[],
 PruneDataSet → None, Dims → All, TestPredicates → {}}
```

```
optLirt = TrainModel[trainDataSet,
   Algorithm → CreateLIRT, AlgorithmOpts → {Optimization → True}];
```

```
PrintMLFEvaluations[ TestModel[ optLirt, trainDataSet ] ]
```

```
NumberOfInstances          1000
CorrelationCoefficient     0.982185
MSE                        59.793
RSE                        0.0353969
MeanError                  0.0143251
NormalizedMSE              0.00158411
MAE                        5.97406
RAE                        0.185741
NormalizedMAE              0.0307494
StdDev                     7.73645
TargetStdDev               41.1206
TargetMAD                  32.1634
RatioOfNullPredictions     0.
RMSE                       7.73259
RRMSE                      0.188141
ModelSize                  8
```

```
orig = MLFGetData[trainDataSet, All, -1];
pred = Predict[optLirt, trainDataSet];
PlotInputRecall[orig, pred]
```

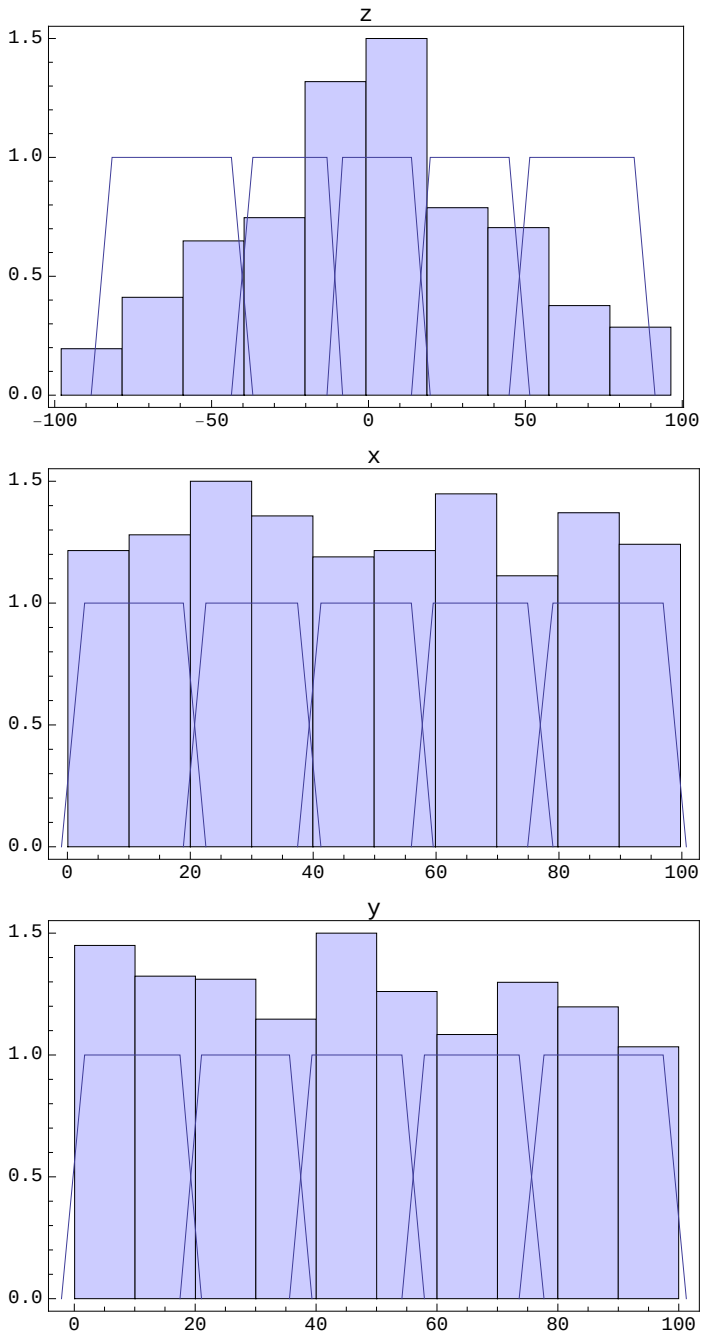


The resulting regression tree now shows much better performance as it can be seen by the evaluation statistics as well as by comparing the scatter plots.

- **Where does fuzzy logic enter the creation of the regression tree?**

  To create the regression tree, the input attributes were first partitioned into fuzzy sets. Let us look at the distribution of the data w.r.t. the single attributes and the according fuzzy partitions which were generated by default (see `CreatePartition` and `CreatePredicates` for more details).

  ```
  PlotPartitions[optLirt, trainDataSet]
  ```



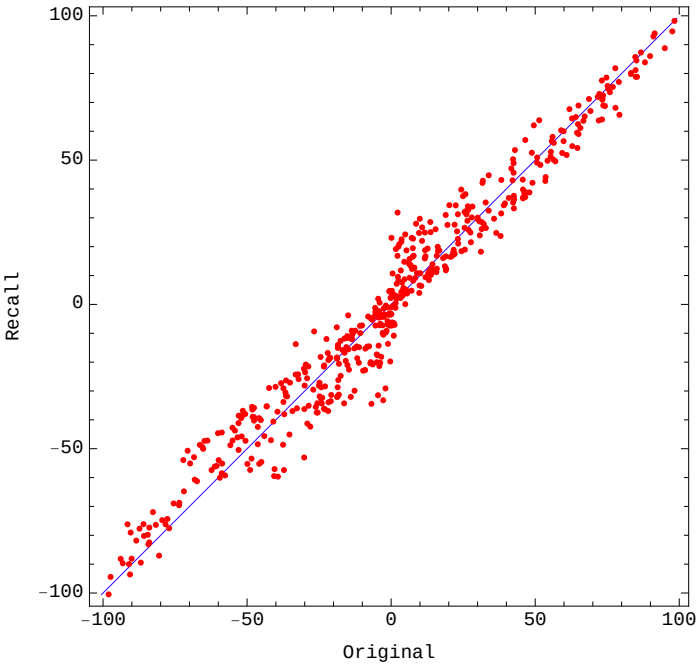
The figure above shows for each attribute the corresponding histogram (red bars) and the five created fuzzy sets which can be described as very low (VL), low (L), medium (M), high (H), and very high (VH) from left to right.

- **Use decision tree to predict new data**

We now use an independent set of 500 sample points to evaluate the regression tree and to compare the results with the original function.

```
samplesTest = 500;
testData = Table[{x = Random[] * range, y = Random[] * range, f[x, y]}, {samplesTest}];
testDataSet = Def[ "testData", DataSet[testData, headers] ];
recalledData = Predict[optLirt, testDataSet];
```

```
PlotInputRecall[MLFGetData[testDataSet, All, -1], recalledData]
```



The figure above plots the original values of *z* against the predicted values. For better comparison, the identity line is shown. We can see that for this example, the decision tree performs quite well on the test data. Hence the trained model does not "overfit" the data.

- **Ridge Regression**

mlf comes with conventional algorithms as well, such as linear regression. We have implemented the *Ridge Regression* algorithm, which is able to compute regularized weights. We will now apply this algorithm on the same data as above to get a comparison for the decision tree.

```
regressionModel = TrainModel[trainDataSet, Algorithm → CreateRidgeRegression];
```

The order of the parameters is according to their normalized weights.The function above is directly applicable to specific input data; however,the following may give a better survey for human eyes:

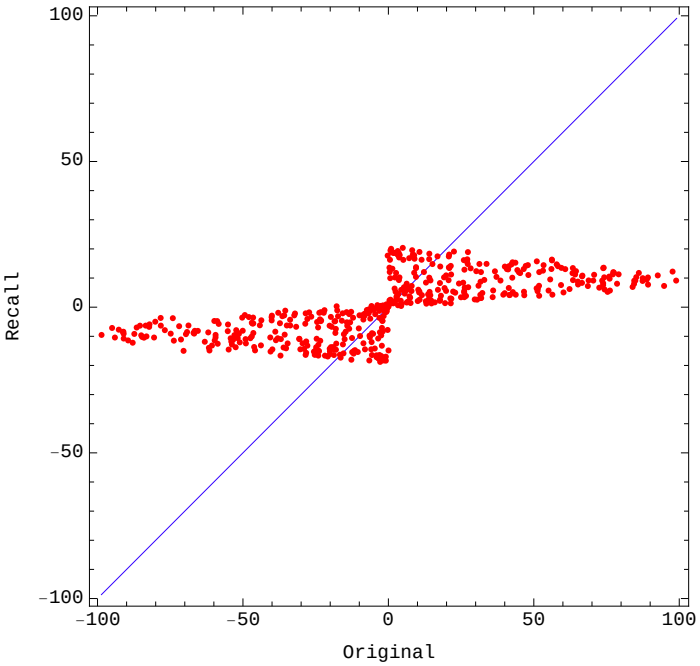
```
PlotModel[regressionModel]
```

```
DOF:        0.380952
Lambda:     1623.38
Noise:      34.6994

Weight      Attribute    Norm.Weight    Mean     Std.dev.     Z-Score     P-Value

19.42
-0.3764     x            -0.2606        49.57    28.47        25.62       0
```

We can now compare the performance of above ID3 tree with that of conventional regression:

```
recalledDataReg = Predict[regressionModel, testDataSet];
```

```
PlotInputRecall[MLFGetData[testDataSet, All, 3], recalledDataReg]
```




```
PrintMLFEvaluations[ TestModel[ regressionModel, trainDataSet ] ]
```

NumberOfInstances	1000
CorrelationCoefficient	0.684089
MSE	1201.64
RSE	0.711361
MeanError	1.25056×10^{-15}
NormalizedMSE	0.0318354
MAE	26.1136
RAE	0.811905
NormalizedMAE	0.134411
StdDev	34.6821
TargetStdDev	41.1206
TargetMAD	32.1634
RatioOfNullPredictions	0.
RMSE	34.6647
RRMSE	0.843422
ModelSize	1

We see that in the given example, the regression tree gives a more accurate model for this prediction task - not to speak of the much better intelligibility. This is what one can expect for non-linear problems.

■ Example 3: Huston housing data set

The goal of this example is to show how one can use mlf`s `CrossValidation` function to find the most suitable algorithm for a given problem and then use this algorithm to come up with the final computational model. We use the well known Houston housing data set (housing.csv) from the UCI machine learning repository (http://www.ics.uci.edu/~mlearn/MLRepository.html) for this purpose which comes with *mlf*.

After loading the data we apply `CrossValidation` to four algorithms (with their default options) and rank them according to the average error:

```
housing = LoadData["housing.csv", "CSV", MinDiffValues → 0];

cvRidge =
  CrossValidation[housing, Algorithm → CreateRidgeRegression, RandomSeed → 1234];

cvLirt = CrossValidation[housing, Algorithm → CreateLIRT, RandomSeed → 1234];

cvId3 = CrossValidation[housing, Algorithm → CreateID3, RandomSeed → 1234];
```

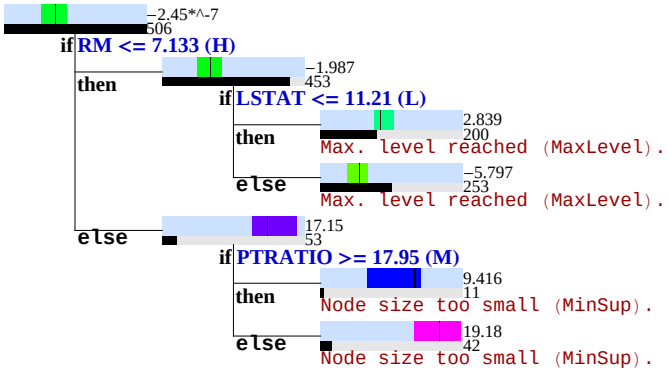
```
Off[CreateAdditiveRegression::alpha];
cvAddReg =
  CrossValidation[housing, Algorithm → CreateAdditiveRegression, RandomSeed → 1234];

Sort[{
   {TotalMAE /. cvRidge, "Ridge regression"},
   {TotalMAE /. cvLirt, "FS-LiRT"},
   {TotalMAE /. cvId3, "FS-ID3"},
   {TotalMAE /. cvAddReg, "Additive Regression"}
  }] // TableForm
```

```
2.84973    Additive Regression
3.45779    FS-ID3
3.49473    FS-LiRT
3.53851    Ridge regression
```

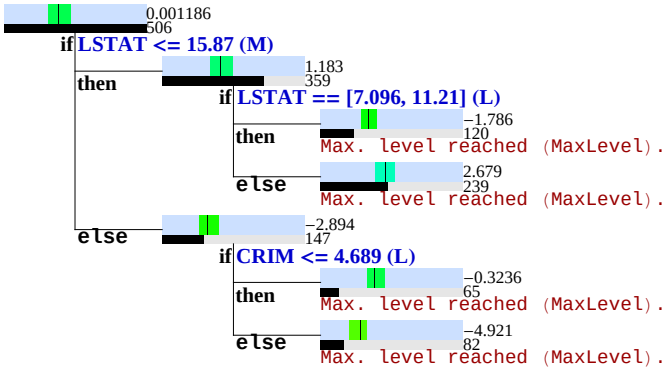
We can see that additive regression performs best (see help for `CreateAdditiveRegression` and the algorithmic background for additive regression). Now we use `TrainModel` and `PlotModel` to train an additive regression model on the whole data set and visualize it:

```
addReg = TrainModel[housing, Algorithm → CreateAdditiveRegression];

PlotModel[addReg, LeafInfo -> "Mean", ImageSize → 350]
```

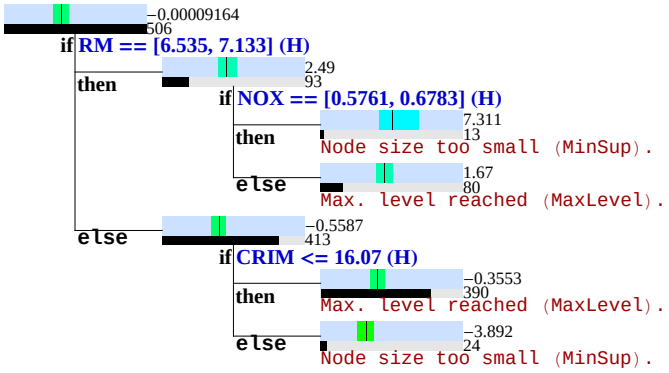
```
++++++++++++ Model 1 (weight=0.9262
 , Mean[SquaredError]={31.2635}, CC={0.7975}) ++++++++++++++
```



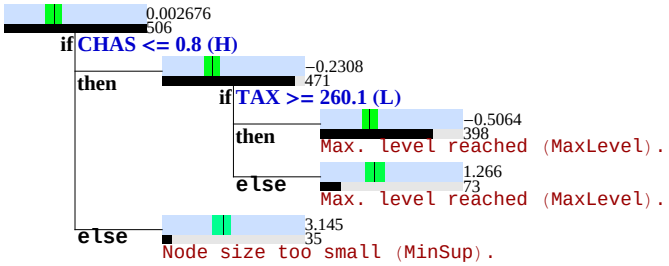

```
++++++++++++ Model 2 (weight=0.9491
 , Mean[SquaredError]={22.823}, CC={0.8543}) ++++++++++++++
```



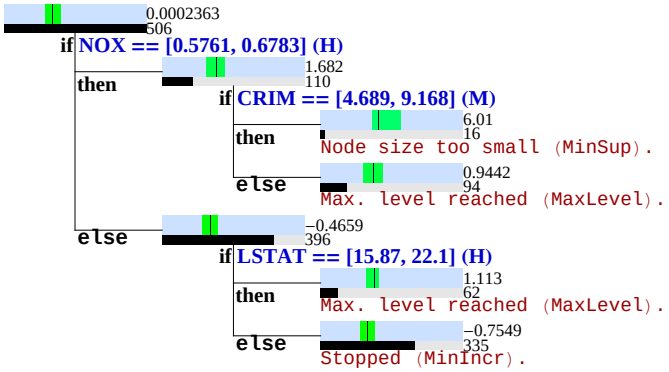

```
++++++++++++ Model 3 (weight=0.9756
 , Mean[SquaredError]={19.9653}, CC={0.8738}) ++++++++++++++
```



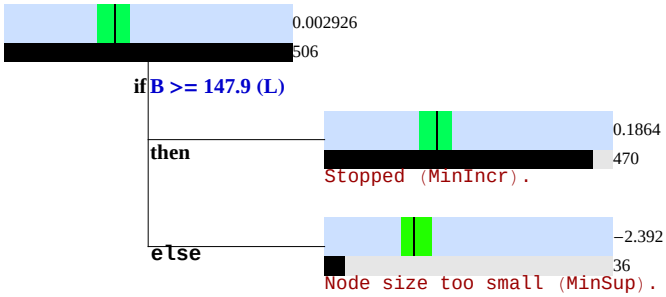
++++++++++++ Model 4 (weight=0.9117
, Mean[SquaredError]={18.8443}, CC={0.8817}) ++++++++++++



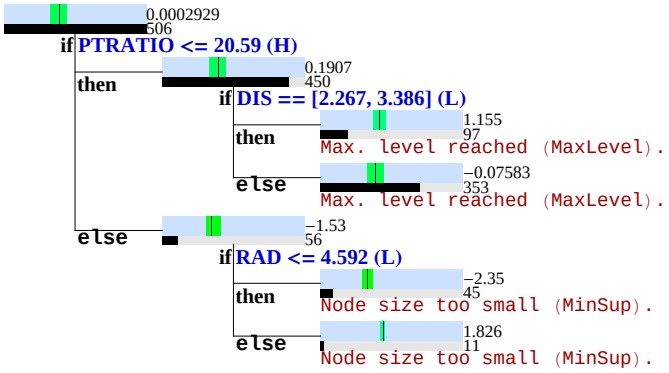
++++++++++++ Model 5 (weight=0.9961
, Mean[SquaredError]={16.8437}, CC={0.8949}) ++++++++++++



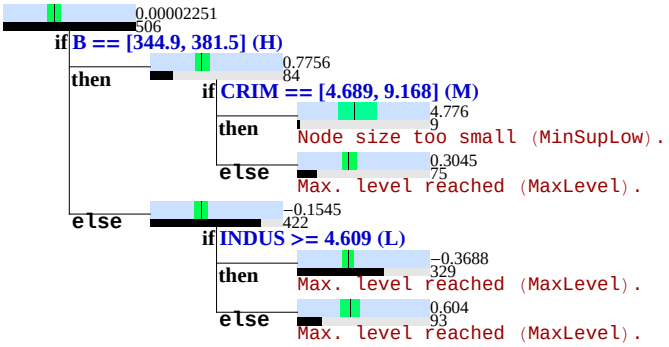
++++++++++++ Model 6 (weight=0.9
, Mean[SquaredError]={16.4088}, CC={0.8976}) ++++++++++++



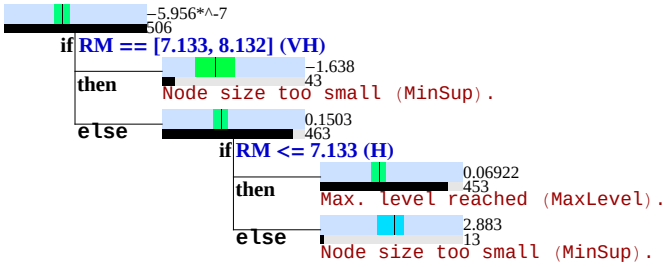
++++++++++++ Model 7 (weight=0.923
, Mean[SquaredError]={15.5717}, CC={0.9031}) ++++++++++++



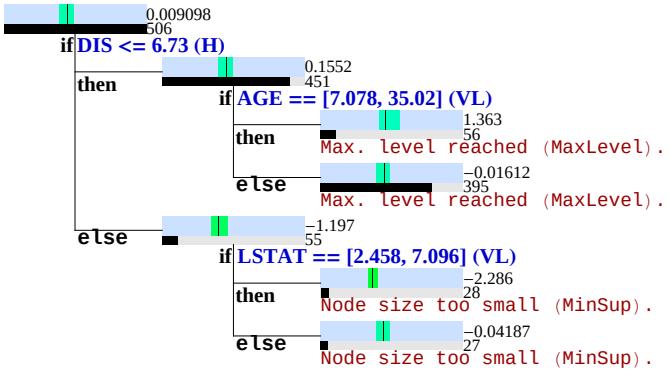
++++++++++++ Model 8 (weight=1.023
, Mean[SquaredError]={14.9318}, CC={0.9076}) ++++++++++++++



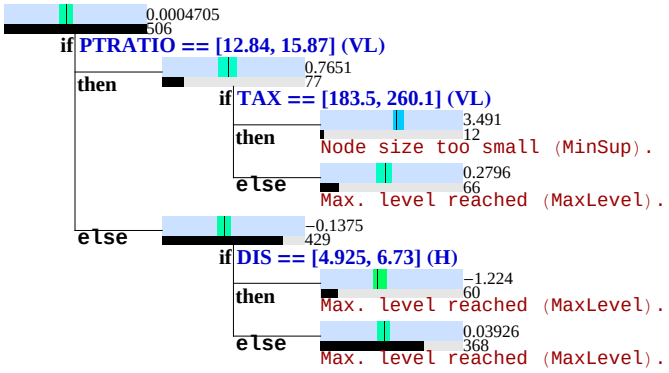
++++++++++++ Model 9 (weight=1.1
, Mean[SquaredError]={14.3854}, CC={0.9108}) ++++++++++++++



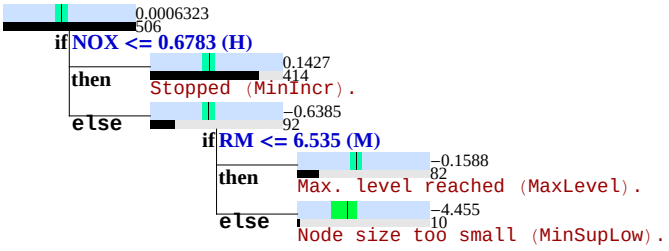
++++++++++++ Model 10 (weight=0.9483
, Mean[SquaredError]={13.8671}, CC={0.9142}) ++++++++++++++



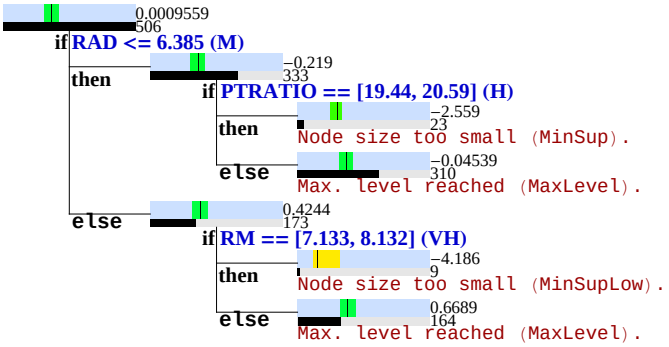
++++++++++++ Model 11 (weight=1.065
, Mean[SquaredError]={13.3158}, CC={0.9179}) ++++++++++++++



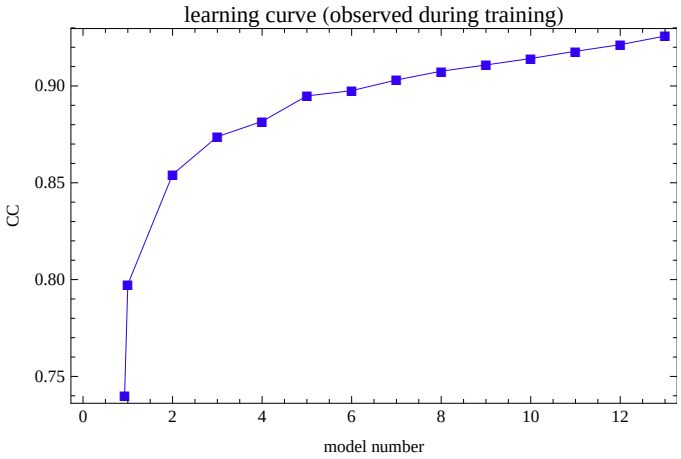
++++++++++++ Model 12 (weight=1.061
, Mean[SquaredError]={12.8165}, CC={0.9213}) ++++++++++++



++++++++++++ Model 13 (weight=0.9117
, Mean[SquaredError]={12.0684}, CC={0.9258}) ++++++++++++



++++++++++++ END OF ADDITIVE REGRESSION MODEL ++++++++++++



More About

XXXX

Related Tutorials

XXXX

Related Wolfram Education Group Courses

XXXX

machine learning framework for *Mathematica*

The most common task in data analysis is to find relations of a set of *input parameters* to one or more *output parameters*. This kind of analysis is called *supervised*, as the goal of the analysis is given by the user. In the case that no explicit goal parameter is available, the analysis is called *unsupervised*. We will describe unsupervised analysis in Chapter 3.

In this chapter we consider the case where we want to predict the value of a categorical variable. This kind of tasks is often called *classification* tasks as we want to assign instances to one class or another. The task of finding a model for predicting a continuous (numeric) parameter will be discussed in Chapter 2.

Loading the *mlf* Package

```
Needs["mlf`"];

Off[DataSet::mindiff];
```

Basics - "iris" Data Set

Problem Statement and Description of the Data Set

In this example we will analyze a simple data file which contains data of about 150 iris flowers of different types. Each flower is descibed by five attributes. The first four of them are sepal length, sepal width, petal length, and petal width, and characterize the blossom of the flower. The fifth attribute specifies to which category (species) of iris the flower belongs.

The goal of our analysis is to find out how we can decide to which category an iris flower belongs based only on the description of the blossom. For that purpose, we will create a decision tree and two different sets of rules, which allow us to classify new flowers by simply asking a few questions about their blossoms.

In our data set, we have three classes of flowers and four attributes describing the blossom of each flower. So we have four numeric attributes and one categorical attribute. The data is stored in a text file where the first row contains the column headers. The categorical attribute is in the last column.

Throughout this first example, we will only evaluate the learned models on the same set of examples which are used for learning the model, i.e., the *training set.* In the examples which follow, we will show better ways to evaluate the quality of a learned model.

Set up data

```
irisData = LoadData["iris.txt", "Table"];
```

`LoadData` returns a reference to a data structure called `DataSet`, which is required for all the algorithms of mlf. You can retrieve the original data, the internal representation of the data, and some meta-information from a data set. To retrieve e.g., the original data and the internal representation of the first data row:

```
GetOriginalData[irisData, 1]
MLFGetData[irisData, 1]
```

```
{5.1, 3.5, 1.4, 0.2, Iris-versicol}
```

```
{5.1, 3.5, 1.4, 0.2, 2.}
```

We see that in the last column, the strings identifying iris classes (species) are replaced by numbers. To see which number represents which class, we can look at the attribute labels:

```
GetParam[irisData, AttrLabels]
```

```
{{}, {}, {}, {}, {Iris-setosa, Iris-versicol, Iris-virginica}}
```

The number 2 represents the second class of the fifth attribute, which is `Iris-versicol`.

The accessible contents of *mlf* data structures can be listed:

```
GetParamList[irisData]
```

```
{Dim, Labels, AttrLabels, DataMatrix}
```

"Labels", for instance, contains the attribute names (headers):

```
headers = GetParam[irisData, Labels]
```

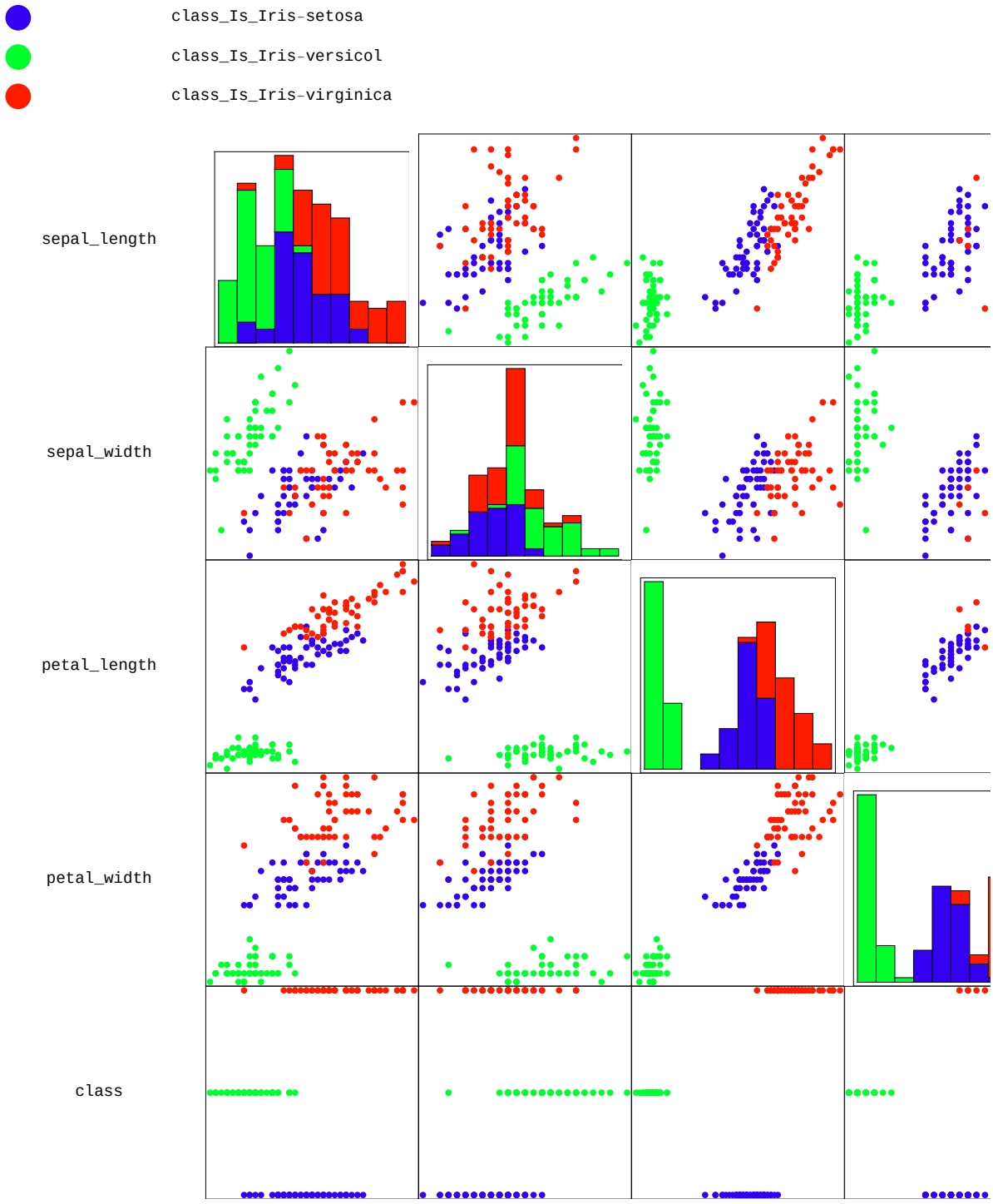
```
{sepal_length, sepal_width, petal_length, petal_width, class}
```

Visualize data

Scatter plots

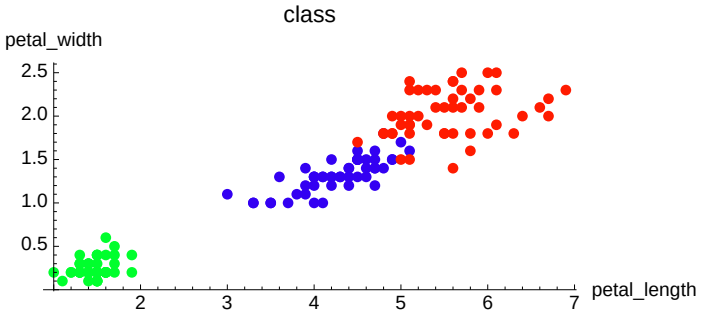
To get a first impression of how the classes are distributed, we can create a scatter plot for two dimensions and colorize the output according to the class labels. To do so, we use the command `PlotAttributeMatrix` which plots all attributes against each other. As we want to separate the data with respect to the class attribute, we specify a goal for colorizing the sample points by setting the option **Goal→5**.

```
PlotAttributeMatrix[irisData, Goal → 5,
 PointSize → 0.03, ClassLegend → True, ImageSize → 800]
```



We can see that petal length vs. petal width separates the data quite well. We will now create this plot by using the `PlotAttributes` command where we specify the dimensions to plot with **`Dims→{3,4}`** and the goal column with **`Goal→5`**.

```
headers = GetParam[irisData, Labels];
Show[Graphics[
  PlotAttributes[
   irisData,
   Dims → {3, 4},
   Goal → 5,
   PointSize → 0.02
  ]],
 Axes → True,
 AxesLabel → headers[[{3, 4}]],
 PlotLabel → headers[[5]],
 TextStyle → {FontFamily → "Helvetica", FontSize → 10}
]
```

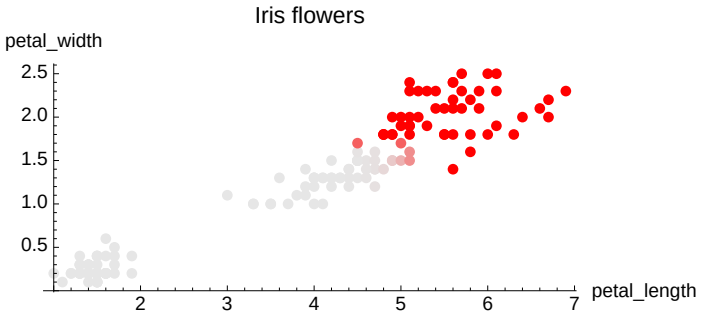


The plot shows that the green points are well separated from the rest, while the red and the blue ones are closer together. When we want to separate the red and the blue samples, we could guess that it might be sufficient to say that the red samples have a petal width larger then 1.75 or a petal length larger then 5.5. To see how this separates the data, we define the according fuzzy predicate.

```
curSelection = FOr[
    FColPredIsAL[3, FuzzySetExp[5.5, 0.3]],
    FColPredIsAL[4, FuzzySetExp[1.75, 0.05]]];
```

To visualize the result, we can now use this predicate as the goal parameter. The results shows all samples that satisfy the predicate in red, while all other samples are shown in gray.

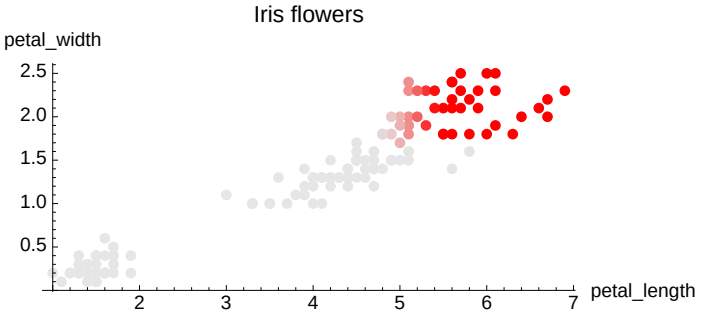
```
Show[Graphics[PlotAttributes[
    irisData,
    Dims → {3, 4},
    Goal → curSelection,
    PointSize → 0.02
   ],
   Axes → True,
   AxesLabel → headers[[{3, 4}]],
   PlotLabel → "Iris flowers"],
  TextStyle → {FontFamily → "Helvetica", FontSize → 10}
]
```



We can also try with `FAnd` instead of `FOr`:

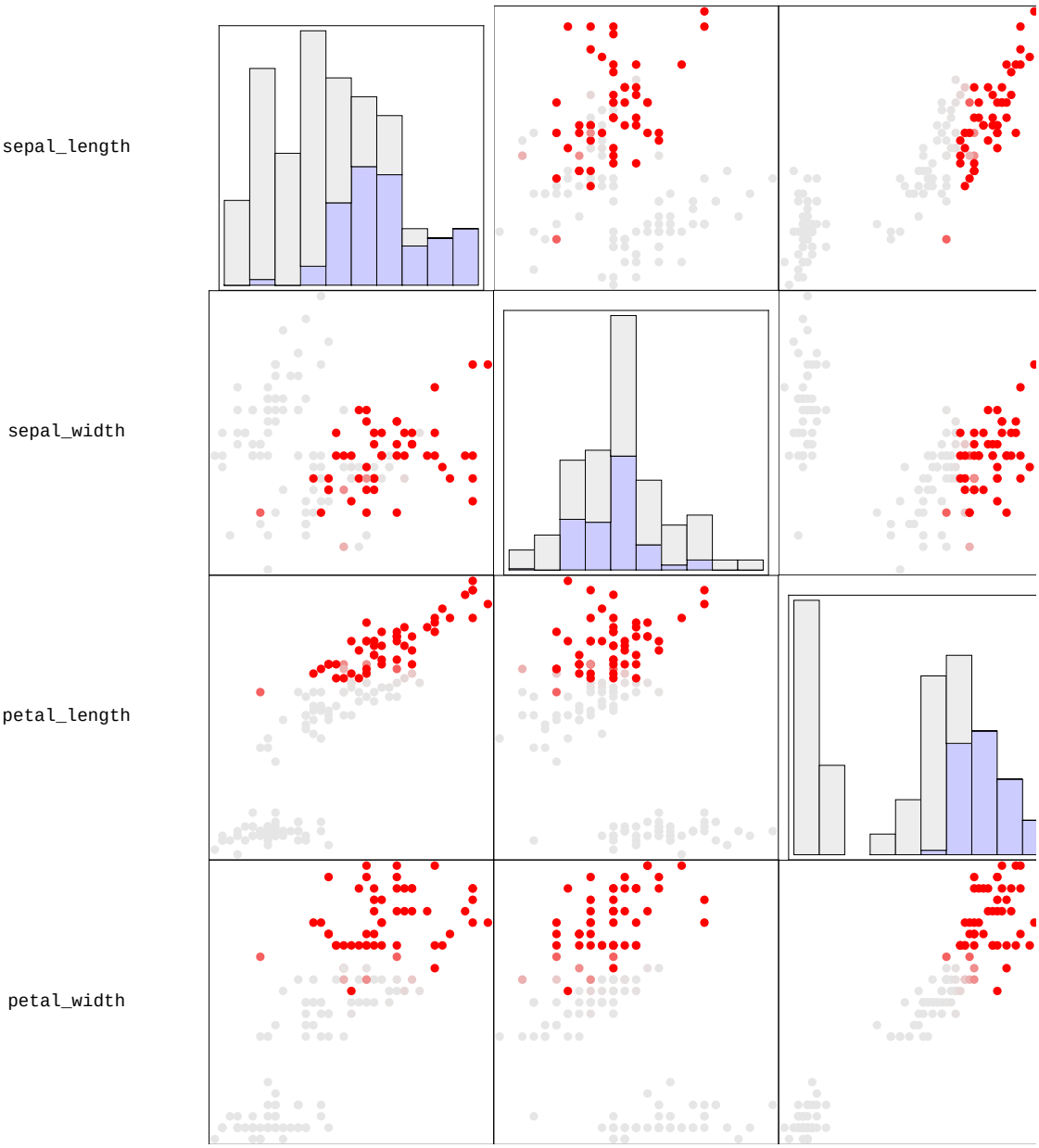
```
curSelection2 = FAnd[
    FColPredIsAL[3, FuzzySetExp[5.5, 0.3]],
    FColPredIsAL[4, FuzzySetExp[1.75, 0.05]]];
```

```
Show[Graphics[PlotAttributes[
    irisData,
    Dims → {3, 4},
    Goal → curSelection2,
    PointSize → 0.02
  ],
   Axes → True,
  AxesLabel → headers[[{3, 4}]],
  PlotLabel → "Iris flowers"],
 TextStyle → {FontFamily → "Helvetica", FontSize → 10}
]
```



We can also use different dimensions for the axes and keep the selection as before (with `For`):

```
PlotAttributeMatrix[
 irisData,
 Dims → Range[4],
 Goal → curSelection,
 PointSize → 0.03,
 ImageSize → 800
]
```



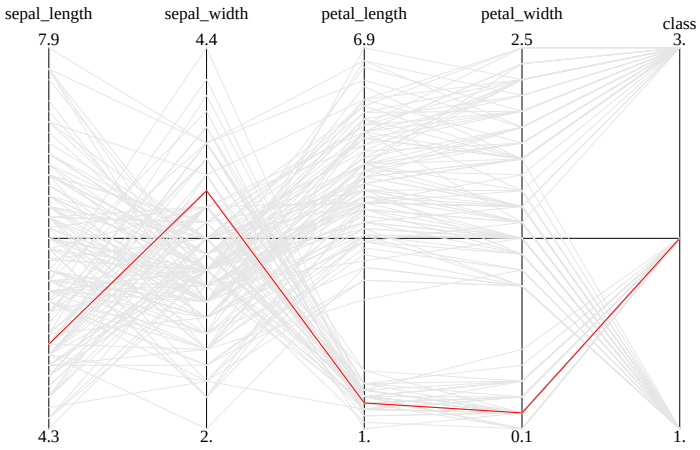
- **Parallel coordinates**

Another possible visualization method is parallel coordinates. While scatter plots are limited to 2 dimensions at a time, parallel coordinates can be used to visualize relations in higher-dimensional data spaces. This is done by plotting all dimensions in parallel to each other. Each dimension is represented as one line covering the whole range of the attribute. One data point is then shown as a line going through all dimensions at the according value. This is best illustrated with an example:

```
curSample = MLFGetData[irisData, 1]
```

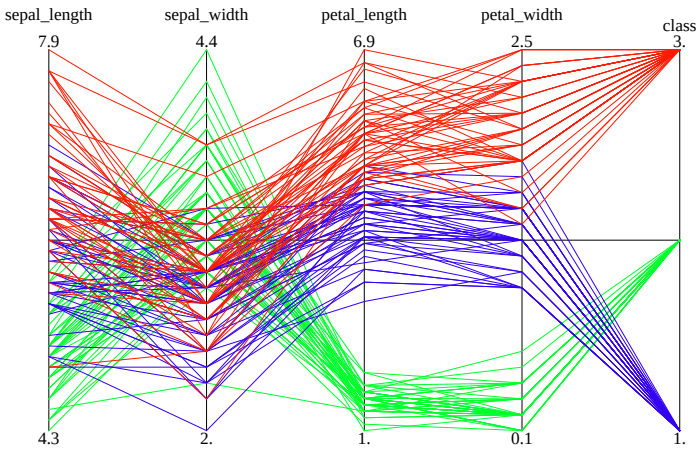
```
{5.1, 3.5, 1.4, 0.2, 2.}
```

```
PlotParallelCoordinates[
 irisData,
 Goal → FAnd[Table[FColPredIsEx[i, curSample[[i]]], {i, Length@curSample}]]
]
```



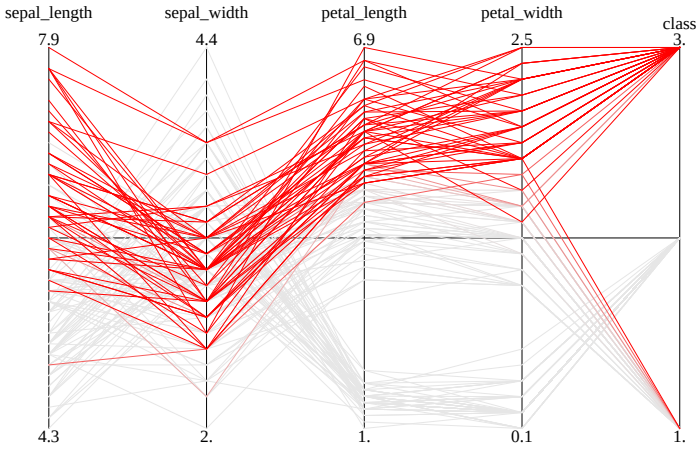
In this plot, one sample is selected and shown with a single red line. All other samples are shown with gray lines, one line for each sample. Of course, it is also possible to use other attributes to colorize the output.

```
PlotParallelCoordinates[irisData, Goal → 5]
```



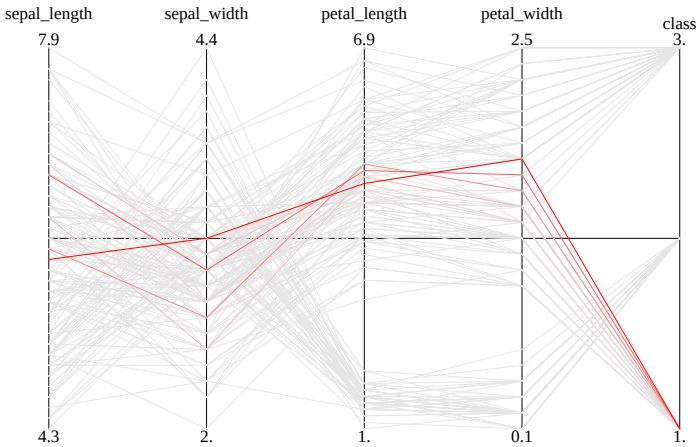
Now we can use the selection defined before to see how it covers the samples of class three.

```
PlotParallelCoordinates[
 irisData,
 Goal → curSelection
]
```



We see that there are some samples of class one covered, too. To get an idea of which samples these are, we combine our selection with a new selection of the first class and visualize the result:

```
PlotParallelCoordinates[
 irisData,
 Goal → FAnd[curSelection, FColPredIsEx [5, 1]]
]
```



Now we can see that these are the samples which have average petal length and width.

- **Prediction Matrix**

Yet another possibility to get a first picture of the data is to look at a `PredictionMatrix` which shows how all the attributes "correlate" with each other, or more exactly, how well one attribute can be predicted by any of the other attributes. (For numeric attributes, this corresponds to correlation).

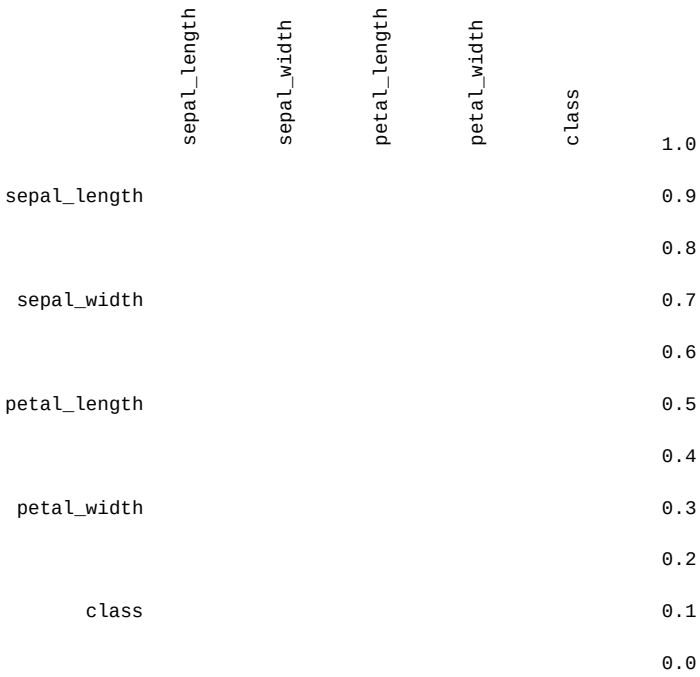
```
predMatrix = PredictionMatrix[irisData];
```

```
predMatrix // MatrixForm
```

1.	0.0119616	0.759955	0.669048	0.426904
0.0119616	1.	0.176834	0.127124	0.207309
0.759955	0.176834	1.	0.926901	0.83577
0.669048	0.127124	0.926901	1.	0.833872
0.618706	0.391881	0.941319	0.928836	1.

This prediction matrix can also be visualized graphically:

```
PlotPredictionMatrix[predMatrix, headers]
```



An entry in this matrix with high values (orange to red) signifies a good predictability - e.g. "class" can be well

predicted from "petal_length" alone (though the reciprocal case does not work as well). Blueish values signify poor predictability.

If all attributes were numeric, we could have used `NumericCorrelationMatrix`. An alternative for data which contain categorical attributes is `MutualInformationMatrix`, with numeric attributes discretized by binning.

Decision tree

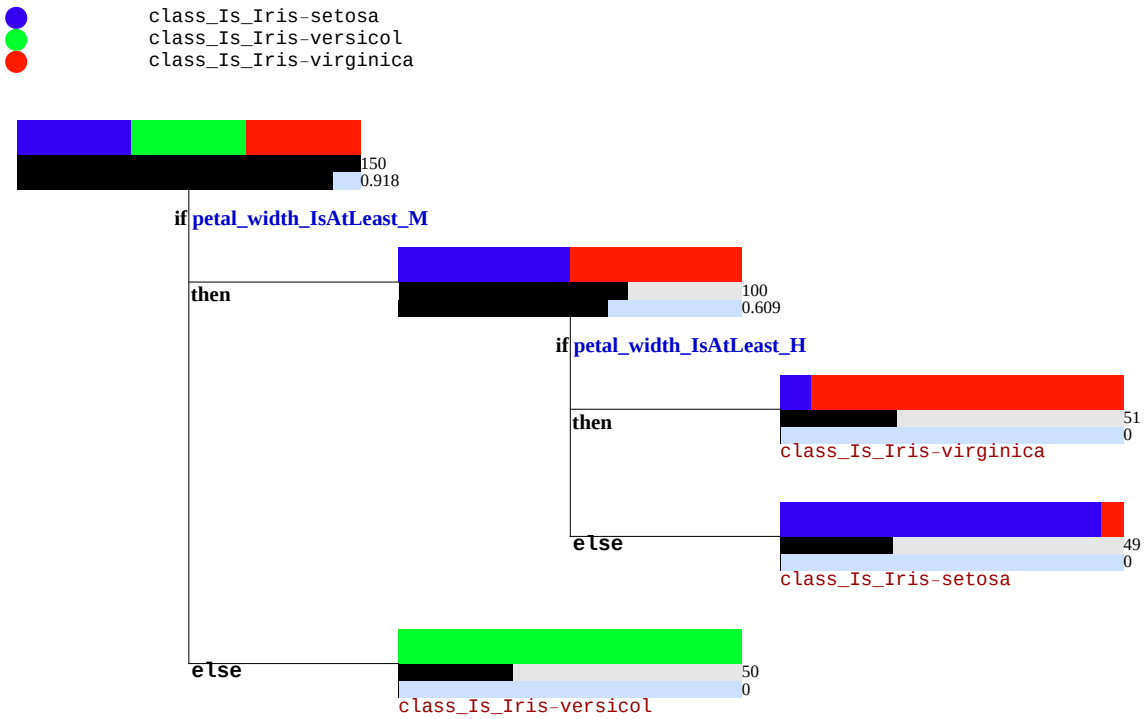
Compute a decision tree

The decision tree rule induction method FS-ID3 is a generalization of Quinlan's ID3 method. A decision tree can be thought of as a set of if-then-else statements and can be used to solve arbitrary classification problems.

To generate a decision tree, we generally have to specify a desired goal attribute. In this example, we will use the "class" attribute, which is the fifth and last column in our input file. As the last column is the default for the goal in our higher-level algorithms, we need not explicitly specify it in the given case.

Now we can generate the decision tree using the `CreateID3` algorithm, which tries to find rules to separate the three classes, and plot the tree.

```
id3tree = TrainModel[irisData, Algorithm → CreateID3];

PlotModel[id3tree]
```



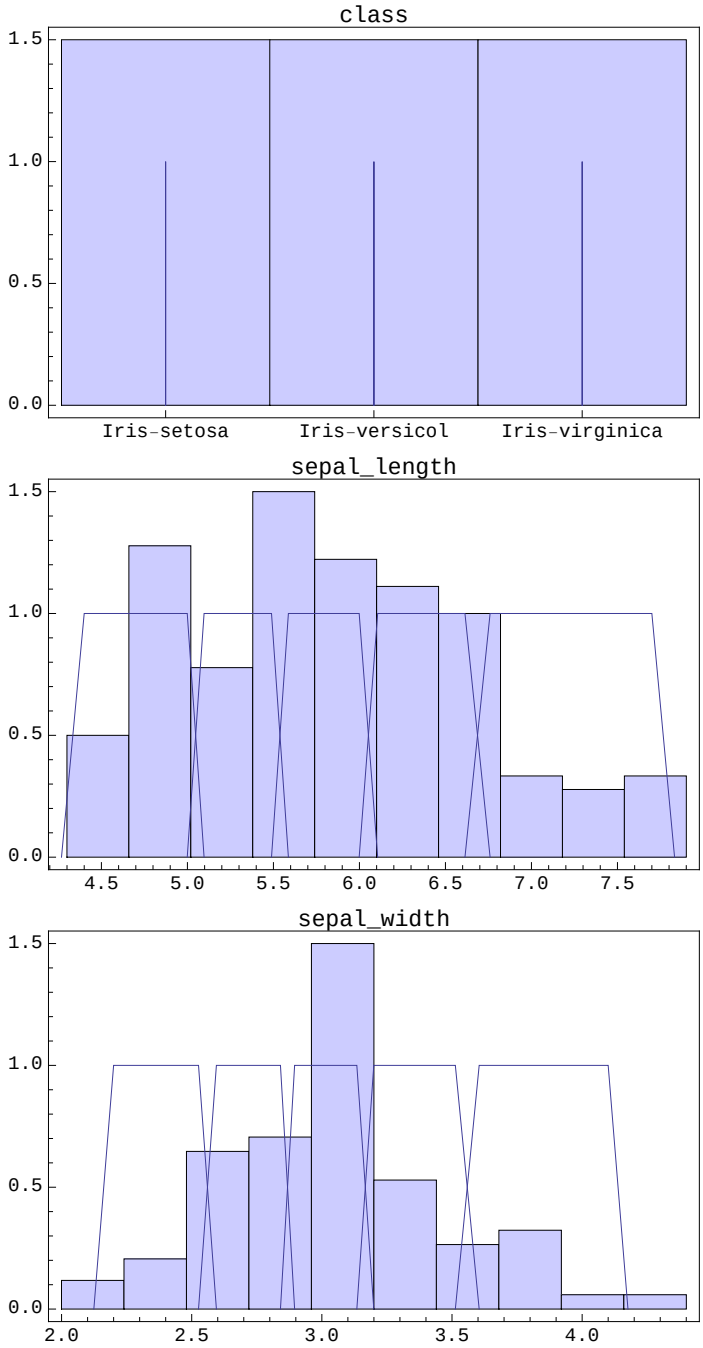
The figure above gives a human-readable representation of the computed decision tree. Each leaf of the tree corresponds to a rule. E.g. the leaf labelled `class_Is_Iris-setosa` should be read as the rule, "If petal width is at least medium and petal width is *not* at least high then type is Iris-setosa." In addition, the graphical output provides statistical information at each node. Each node consists of three bars. The first one indicates the class distribution of samples belonging to this node. The second bar indicates the relative support, i.e. the relative number of samples belonging to this node with respect to the total number of samples (the absolute support is given in parentheses after the label of the node). The third bar indicates the entropy gain to the next level. A high entropy gain indicates important decisions. (NB: the notion of *entropy gain* is a technical term used throughout the literature which may be a little confusing; what is meant is actually an information gain.)

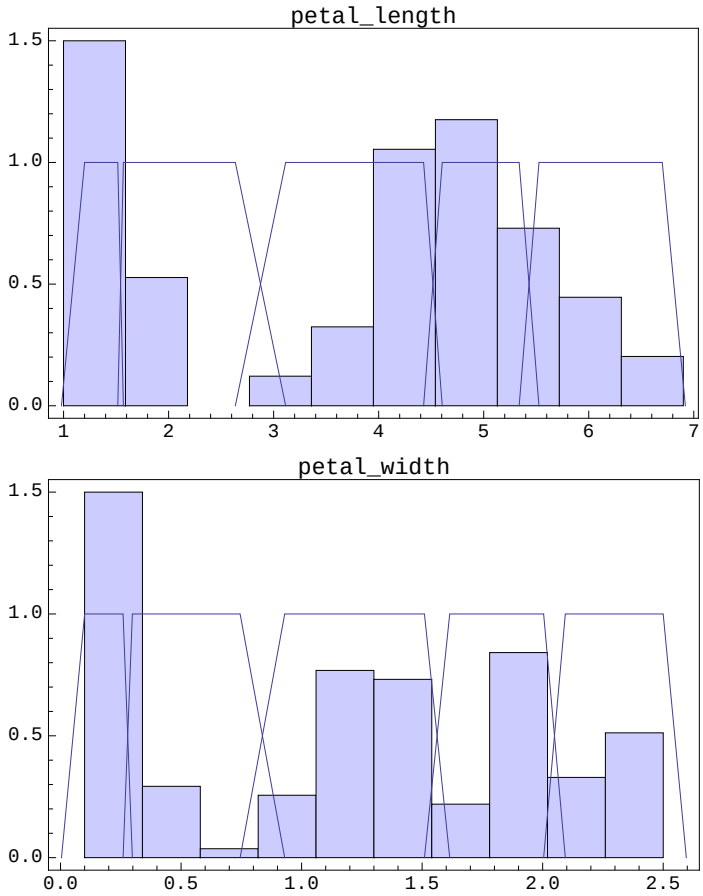
We can see that the predicate used in the root noode, `petal_width_IsAtLeast_M`, separates the Iris-versicol flowers in the "`F`" (`False`) branch. Hence the "`T`" (`True`) branch contains only samples of the other two classes. These samples are then separated by the predicate `petal_width_IsAtLeast_H` into two sets each of which contains almost only one species of flower.

To create the decision tree, the input attributes were first partitioned into fuzzy sets, using the default *piecewise-*

linear (PWL) type. The default number of sets in a partition is five. Let us look at the distribution of the data w.r.t. the single attributes and the according fuzzy partitions (where applicable).

```
PlotPartitions[id3tree, irisData]
```





■ **Evaluating the learned decision tree**

To check if the computed decision tree is able to describe the data, we can now use it to forecast the classification of the flowers and compare the results with the original classification.

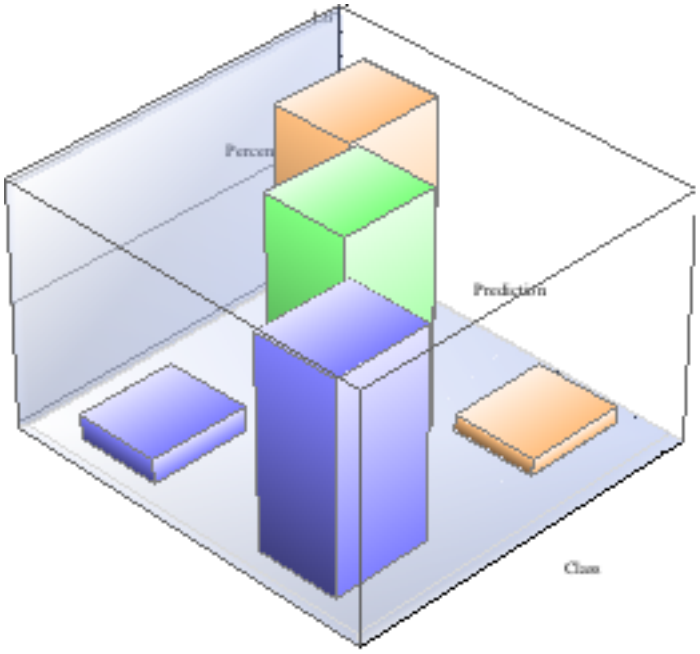
`Predict` applies a model - e.g. a decision tree - to a new data set. Afterwards, the classifications obtained can be compared with the original ones using the `CompEvalMatrix` command. The *evaluation matrix* contains the cross-validation of input and recall in percentages. With the `PlotEvalChart3D` command, this matrix can also be shown graphically.

```
recalledID3 = Predict[id3tree, irisData];
```

```
origData = MLFGetData[irisData, All, 5];
```

```
evalMatrix = CompEvalMatrix[origData, recalledID3];
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} \frac{9}{10} & 0 & \frac{3}{50} \\ 0 & 1 & 0 \\ \frac{1}{10} & 0 & \frac{47}{50} \end{pmatrix}$$



We can see that 90% of the first class have been classified correctly. However, 10% have been predicted to be from the third class. As a single gauge for the accuracy of the model, we can look at the fraction of samples which have been classified correctly:

```
fractionCorrectID3 = MLFFractionEqual[origData, recalledID3]
```

```
0.946667
```

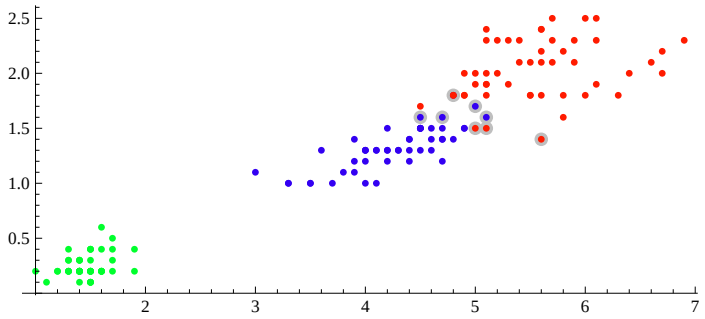
This means that some 95% of the samples have been classified correctly.

To extract those samples that have not been classified correctly, the command `ExtractNonMatchingRecs` can be used.

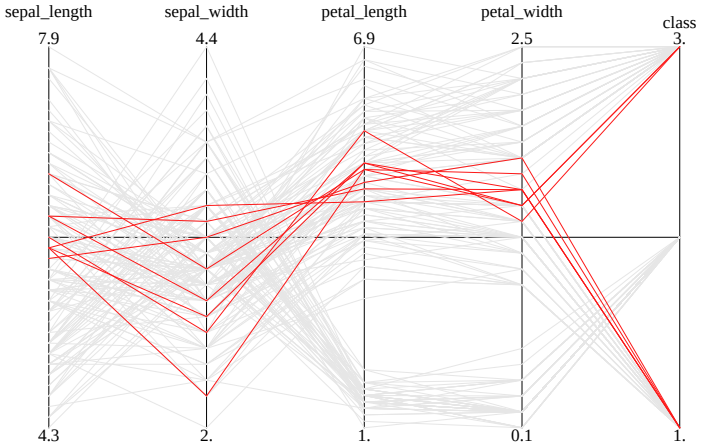
```
failedS =
  Extract[MLFGetData[irisData], ExtractNonMatchingRecs[irisData, 5, recalledID3]];
failedS // TableForm
```

```
6.3   3.3   4.7   1.6   1.
5.9   3.2   4.8   1.8   1.
6.7   3.    5.    1.7   1.
6.    2.7   5.1   1.6   1.
6.    3.4   4.5   1.6   1.
6.    2.2   5.    1.5   3.
6.3   2.8   5.1   1.5   3.
6.1   2.6   5.6   1.4   3.
```

```
Show[Graphics[Join[
    PlotAttributes[failedS, Dims -> {3, 4}, PointSize → 0.02],
    PlotAttributes[irisData, Dims -> {3, 4}, Goal -> 5, PointSize → 0.01]
  ], Axes → True, PlotRange → All]]
```




```
members = Table[0.0, {i, Length@MLFGetData@irisData}];
members[[Flatten[ExtractNonMatchingRecs[irisData, 5, recalledID3]]]] = 0.9;
PlotParallelCoordinates[
 irisData,
 Goal → members
]
```



■ Rule base

■ Compute a set of rules

To create a set of rules that can be used to forecast an attribute like the classification of a new flower, a modification of Quinlan's FOIL algorithm, called FS-FOIL, is implemented in ***mlf***. The respective function is called `CreateFOIL`. Again, the last (default) attribute is our goal.

```
foilRules = TrainModel[irisData, Algorithm → CreateFOIL];
```

```
PlotModel[foilRules]
```

Class	{ }	Condition
class_Is_Iris-setosa	⇐	petal_width_Is_M
class_Is_Iris-versicol	⇐	petal_width_IsAtMost_L
class_Is_Iris-virginica	⇐	petal_width_IsAtLeast_H

In the output, each line corresponds to a rule and its elements are connected my means of conjunction. Rules belonging to the same goal predicate can be interpreted as being disjuncted. In the current example we have only one single-clause rule per class, e.g. *if petal width is medium, then class is Iris-setosa*.

With the option **`Info→True`**, it is possible to display additional information about the accuracy of the rules found.

```
PlotModel[foilRules, Info → True]
```

Class	tt	tf	rt	conf	supp	Condition
class_Is_Iris-setosa	45.41	3.14	4.59	0.94	0.3	petal_width
	45.41	3.14	4.59	0.94	0.3	Total
class_Is_Iris-versicol	50.	0.	0.	1.	0.33	petal_width
	50.	0.	0.	1.	0.33	Total
class_Is_Iris-virginica	46.86	4.59	3.14	0.91	0.31	petal_width
	46.86	4.59	3.14	0.91	0.31	Total

The additional output can be read as follows:
tt is the number of samples fulfilling the rule and belonging to the desired goal class.
tf is the number of samples fulfilling the rule, but which do not belong to the desired goal class.
rt is the number of samples in the goal class which are not covered by the rule.
conf is the confidence of the rule, i.e. the percentage of correctly classified samples.
supp is the support of the rule, i.e. the percentage of samples fulfilling the rule relative to the complete data set.

- **Evaluating the learned set of rules**

To check if the computed rule base is able to describe the data sufficiently we can now use it to forecast the classification of the flowers and compare the results with the original classification.

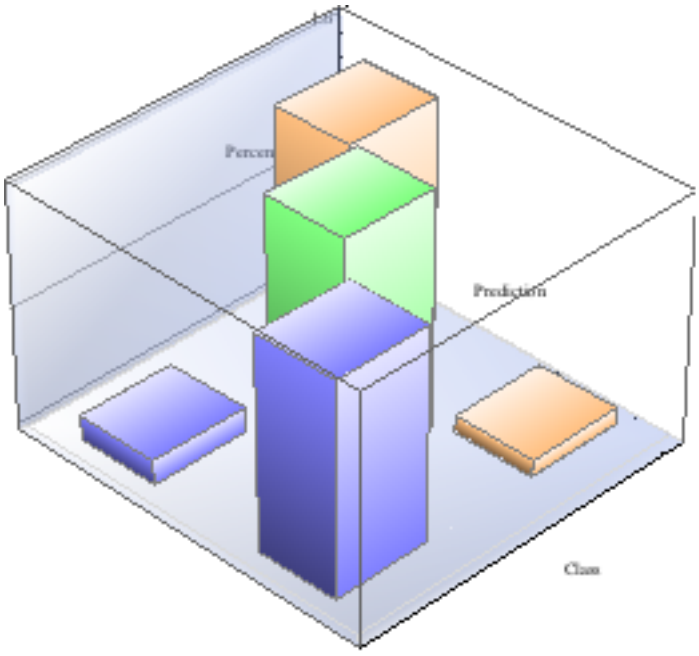
`Predict` applies a model - e.g. a rule base - to a new data set. Afterwards, the classifications obtained can be compared with the original ones using the `CompEvalMatrix` command. The *evaluation matrix* contains the cross validation of input and recall in percent. With the `PlotEvalChart3D` command, this matrix can also be shown graphically.

```
recalledFOIL = Predict[foilRules, irisData];
```

```
origData = MLFGetData[irisData, All, 5];
```

```
evalMatrix = CompEvalMatrix[origData, recalledFOIL];
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} \frac{9}{10} & 0 & \frac{3}{50} \\ 0 & 1 & 0 \\ \frac{1}{10} & 0 & \frac{47}{50} \end{pmatrix}$$




```
fractionCorrectFOIL = MLFFractionEqual[origData, recalledFOIL]
```

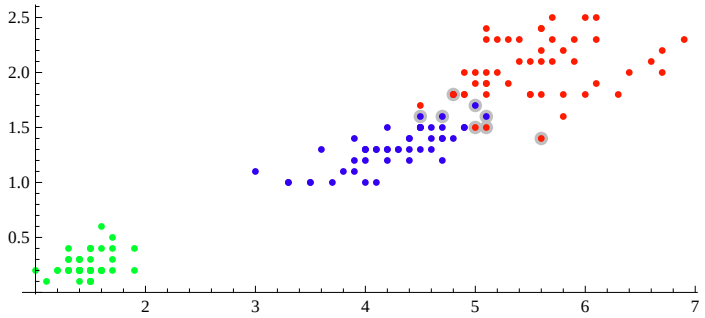
```
0.946667
```

To extract those samples that have not been classified correctly, the command ExtractNonMatchingRecs can be used.

```
failedS =
  Extract[MLFGetData@irisData, ExtractNonMatchingRecs[irisData, 5, recalledFOIL]];
failedS // TableForm
```

```
6.3    3.3    4.7    1.6    1.
5.9    3.2    4.8    1.8    1.
6.7    3.     5.     1.7    1.
6.     2.7    5.1    1.6    1.
6.     3.4    4.5    1.6    1.
6.     2.2    5.     1.5    3.
6.3    2.8    5.1    1.5    3.
6.1    2.6    5.6    1.4    3.
```

```
Show[Graphics[Join[
    PlotAttributes[failedS, Dims -> {3, 4}, PointSize → 0.02],
    PlotAttributes[irisData, Dims -> {3, 4}, Goal -> 5, PointSize → 0.01]
  ], Axes → True, PlotRange → All]]
```



Descriptions

Compute a set of descriptions

To create a set of rules that can be used to forecast the classification of a new flower, also an alternative algorithm called FS-MINER is implemented in *mlf*.

```
descriptions = TrainModel[irisData, Algorithm → CreateMINER];

PlotModel[descriptions, Info → True]
```

Class	tt	tf	rt	conf	supp	Condition
class_Is_Iris-setosa	32.78	0.	17.22	1.	0.22	petal_width
	45.41	2.	4.59	0.96	0.3	petal_width
	45.41	2.	4.59	0.96	0.3	Total
class_Is_Iris-versicol	50.	0.	0.	1.	0.33	petal_width
	50.	0.	0.	1.	0.33	Total
class_Is_Iris-virginica	46.86	4.59	3.14	0.91	0.31	petal_width
	27.34	0.	22.66	1.	0.18	petal_lengt
	38.82	1.89	11.18	0.95	0.26	petal_width
	48.	4.59	2.	0.91	0.32	Total

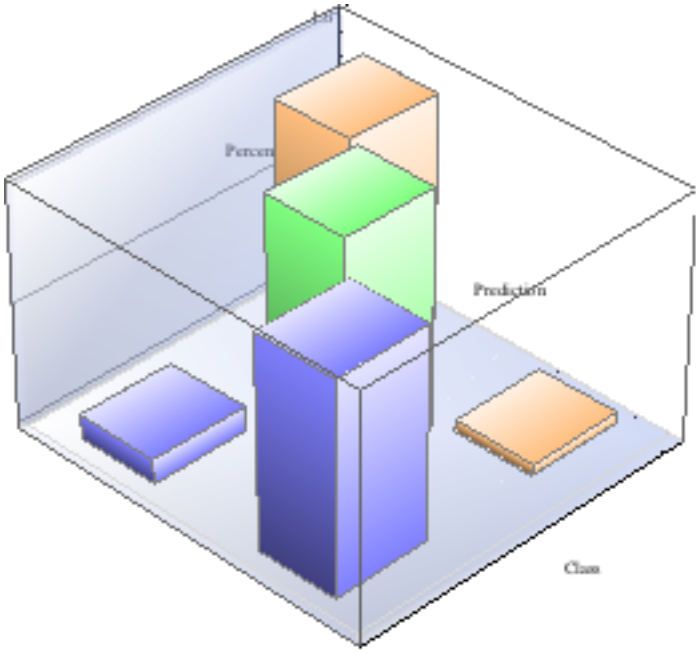
We can see that all rules have very high confidence and support. In contrast to the rules created by FS-FOIL these rules are not disjoint.

Evaluating the learned descriptions

```
recalledMINER = Predict[descriptions, irisData];

origData = MLFGetData[irisData, All, 5];
```

```
evalMatrix = CompEvalMatrix[origData, recalledMINER];
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} \frac{9}{10} & 0 & \frac{1}{25} \\ 0 & 1 & 0 \\ \frac{1}{10} & 0 & \frac{24}{25} \end{pmatrix}$$



```
fractionCorrectMINER = MLFFractionEqual[origData, recalledMINER]
```

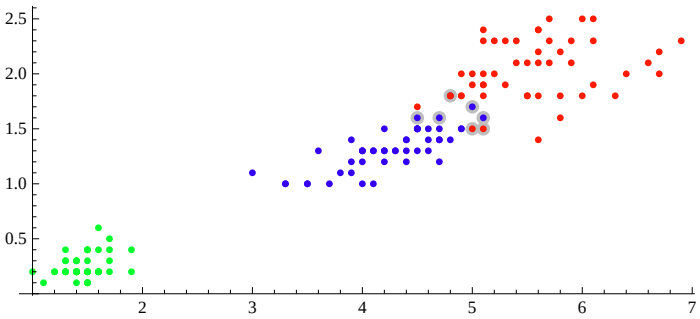
```
0.953333
```

Again, we extract the incorrectly classified examples using `ExtractNonMatchingRecs`.

```
failedS =
  Extract[MLFGetData[irisData], ExtractNonMatchingRecs[irisData, 5, recalledMINER]];
failedS // TableForm
```

```
6.3    3.3    4.7    1.6    1.
5.9    3.2    4.8    1.8    1.
6.7    3.     5.     1.7    1.
6.     2.7    5.1    1.6    1.
6.     3.4    4.5    1.6    1.
6.     2.2    5.     1.5    3.
6.3    2.8    5.1    1.5    3.
```

```
Show[Graphics[Join[
    PlotAttributes[failedS, Dims -> {3, 4}, PointSize → 0.02],
    PlotAttributes[irisData, Dims -> {3, 4}, Goal -> 5, PointSize → 0.01]
  ], Axes → True, PlotRange → All]]
```



We can now compare the predictive performances of ID3 tree, FOIL rules and MINER rules:

```
fractionCorrectID3
fractionCorrectFOIL
fractionCorrectMINER
```

```
0.946667
```

```
0.946667
```

```
0.953333
```

The performance of the ID3 tree and the FOIL rules happens to be equal, while the MINER rules seem to be slightly better. However, we must take into account that the predictions have been made on train data. In the next example, we will split the data into train data for creating the models and separate test data to validate the models.

Simple example - wine data set

Problem statement and description of the data set

This data set is taken from the UCI machine learning repository and contains the results of an analysis of 178 wines grown in the same region in Italy, but coming from three different vineyards. Chemical analysis determined the quantities of constituents found in each of the three types of wines and some optical properties. All together there are 13 input attributes, all of which are numerical. The goal predicate is Boolean categorical - i.e. with a crisp map to classes - with 3 different classes/labels corresponding to the three vineyards. Accordingly, we use fuzzy predicates induced by appropriate fuzzy sets on the domains of the numerical input attributes. Although the goal predicate is Boolean categorical, the use of fuzzy predicates for the input predicates is not meaningless. The reason is that fuzzy sets allow to model regions of overlapping goal classes easier and in a more natural way than by splitting the numerical attributes into Boolean classes.

Set up data

For this example, we will split the available data into a data set used for the training of the models and a second data set for evaluating the models. If we set the option `TrainPart` to a value less than 1 (but greater 0), `LoadData` returns two disjoint data sets instead of one, where the first data set contains that fraction of the data set by the option value, and the second set contains the rest. The split is performed at random, but if we set a `RandomSeed`, the result will always be the same.

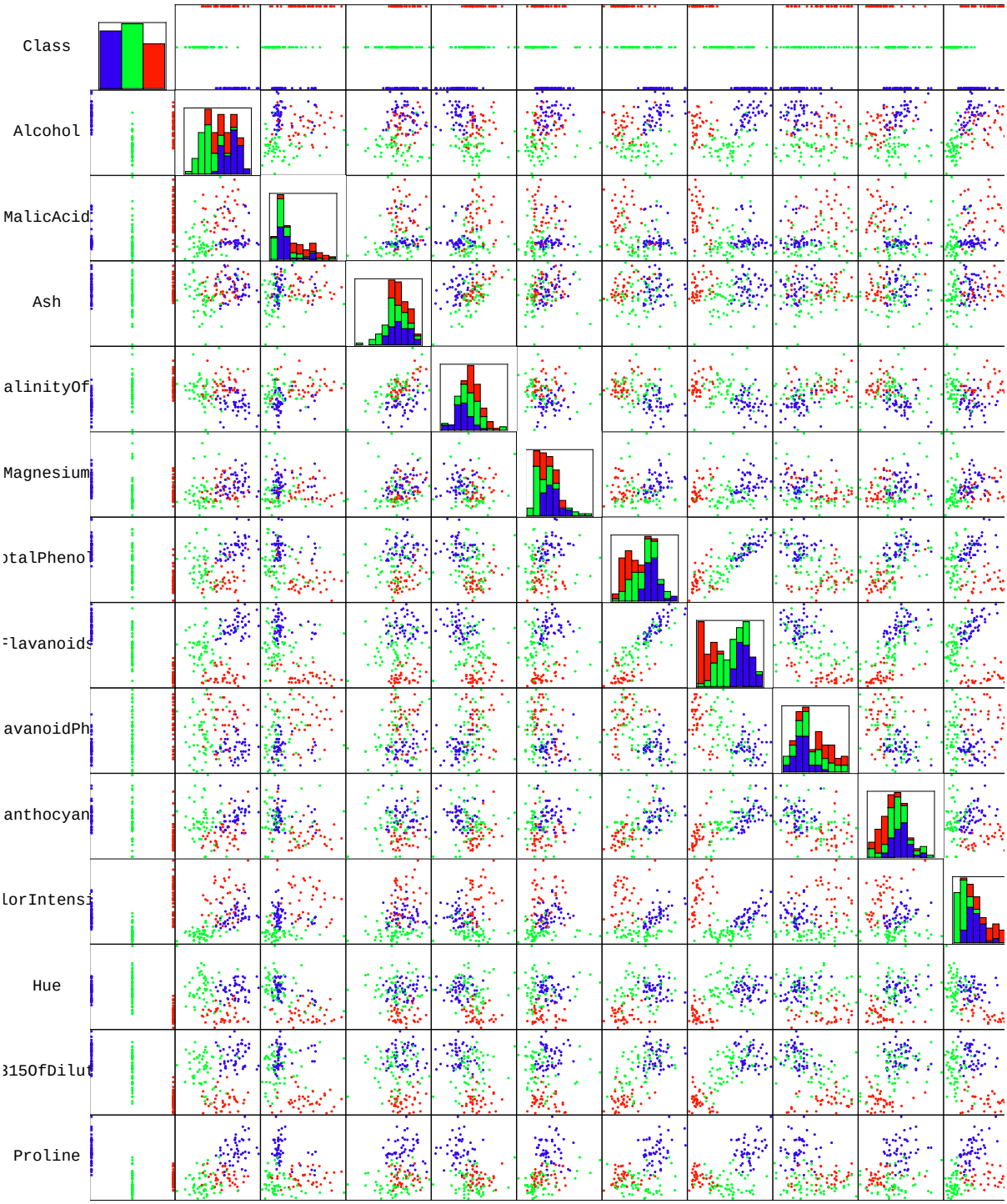
```
{wineData, wineTestData} =
  LoadData["wine.data", "Table", TrainPart → 0.8, RandomSeed → 1];
```

```
GetParam[wineData, Labels]
```

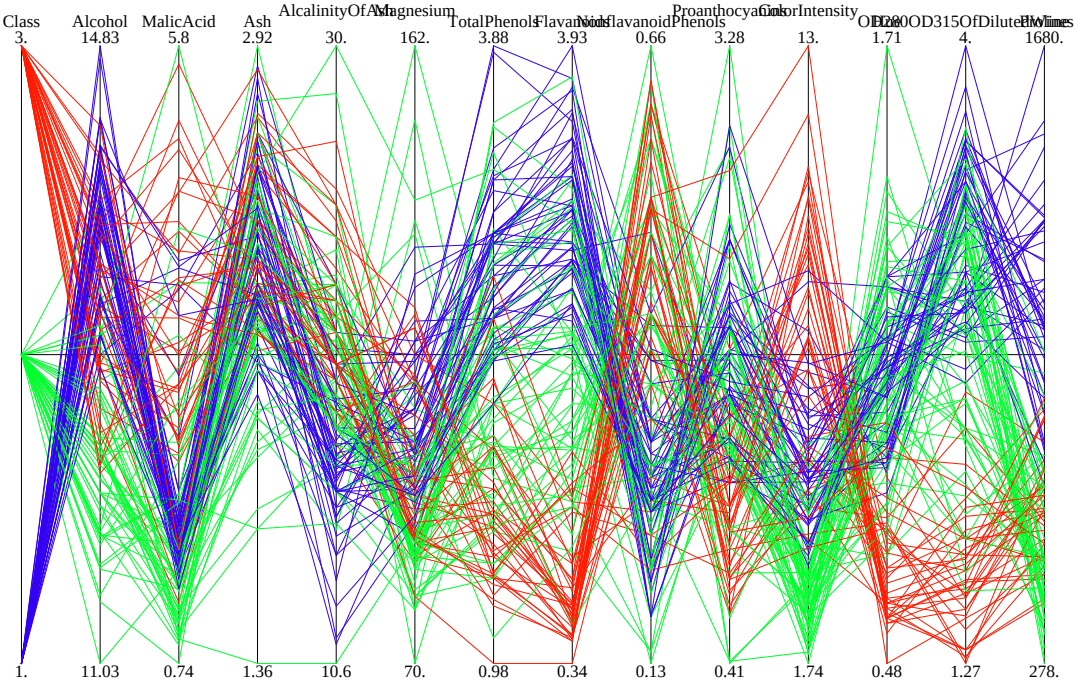
```
{Class, Alcohol, MalicAcid, Ash, AlcalinityOfAsh, Magnesium,
 TotalPhenols, Flavanoids, NonflavanoidPhenols, Proanthocyanins,
 ColorIntensity, Hue, OD280OD315OfDilutedWines, Proline}
```

■ Visualize data

```
PlotAttributeMatrix[wineData, Goal → 1, PointSize → 0.03, ImageSize → 800]
```



```
PlotParallelCoordinates[wineData, Goal → 1]
```

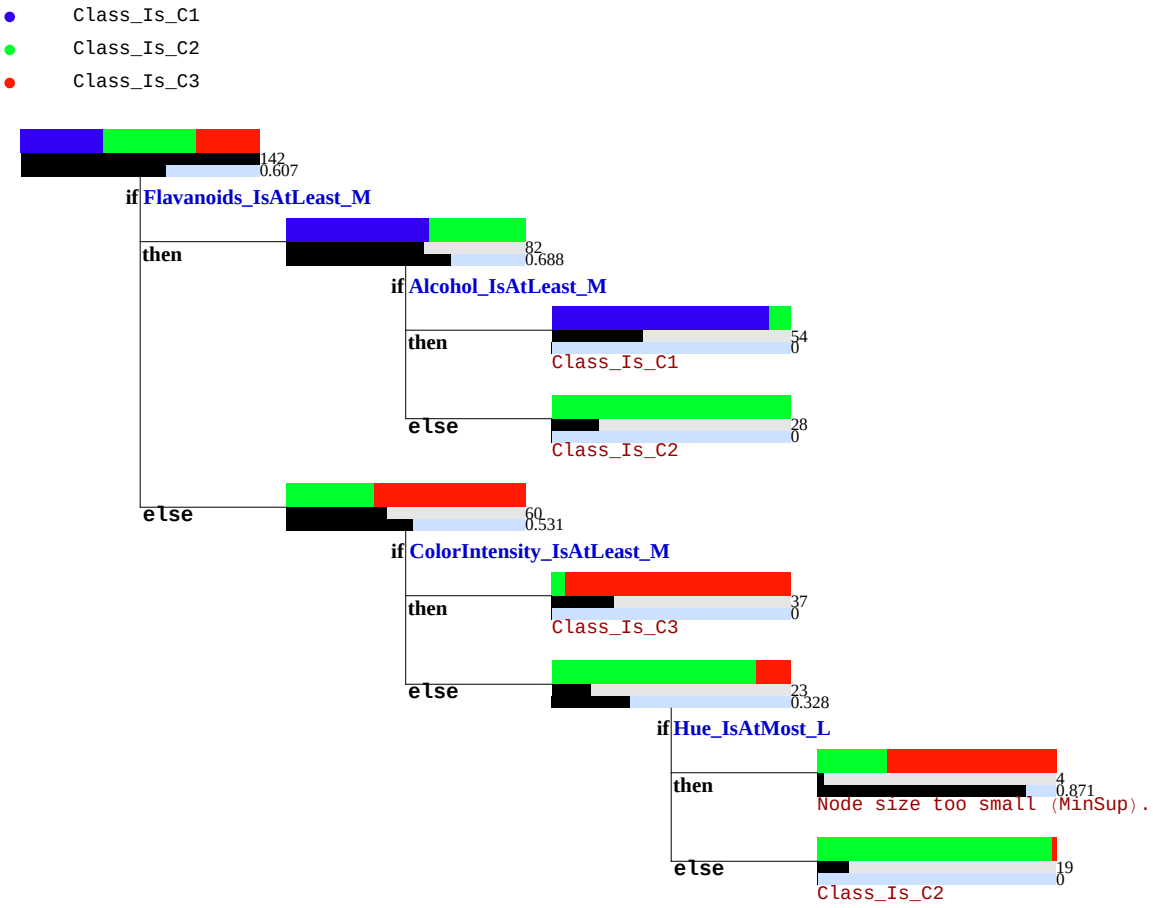


- **Decision tree**
- **Compute a decision tree**

  The goal attribute is the first one; as the default for the goal is the last column, we have to specify it explicitly.

  ```
  id3tree = TrainModel[wineData, 1, Algorithm → CreateID3];
  ```

  ```
  PlotModel[id3tree]
  ```



It is also possible to print the decision tree in a textual form using `PrintID3`. The function `TrainModel` returns a structure called `MLFModel` which not only contains the model - in this case the tree - as such, but also some meta information:

```
GetParamList[id3tree]
```

```
{Model, FuzzySetsOfModel, Algorithm, InputVars, GoalVars, InputCols,
 GoalCol, GoalType, DataSetInfo, DataSetTransformations, ModelSize}
```

`PrintID3`, however, requires the model as such as an argument, so we have to extract this:

```
PrintID3[GetParam[id3tree, Model]] // TableForm
```

```
Flavanoids_IsAtLeast_M (142.)
+-Alcohol_IsAtLeast_M (81.9723)
| +-Class_Is_C1 (53.7911)
| +-Class_Is_C2 (28.4053)
+-ColorIntensity_IsAtLeast_M (60.0277)
  +-Class_Is_C3 (36.6614)
  +-Hue_IsAtMost_L (23.3663)
    +-Node size too small (MinSup). (4.23794)
    +-Class_Is_C2 (19.3663)
```

- **Examine the predicates**

If one is interested in the predicates used in the decision tree, `PredicateStatisticsID3` creates a table containing the predicates, the underlying data dimension, a list of all levels where this predicate occurs, and the total number of occurrences where each leaf is considered as a single rule and counts are on a per-rule basis.

```
PredicateStatisticsID3[GetParam[id3tree, Model],
  GetParam[id3tree, GoalVars]] // TableForm
```

```
Flavanoids_IsAtLeast_M       8     1 1 1 1 1    5
ColorIntensity_IsAtLeast_M   11    2 2 2        3
Hue_IsAtMost_L               12    3 3          2
Alcohol_IsAtLeast_M          2     2 2          2
```

To have a direct look at the predicates, we need to extract the fuzzy variables from the `MLFModel`:

```
vars = GetParam[id3tree, GoalVars]
```

```
{Obj`LogicVar[Class_Is_C1], Obj`LogicVar[Class_Is_C2], Obj`LogicVar[Class_Is_C3]}
```

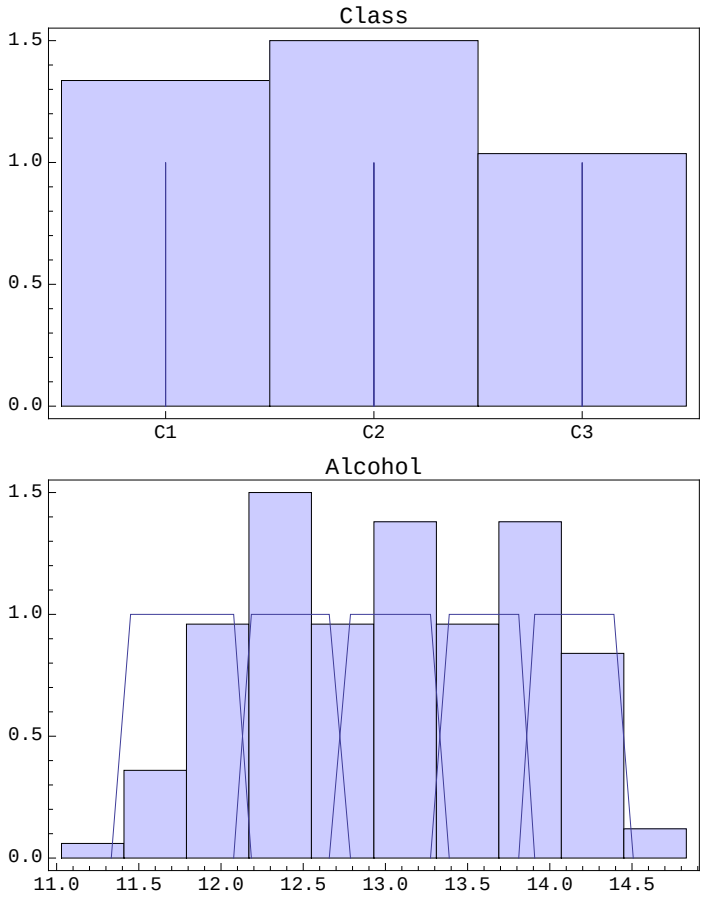
and retrieve the objects that they are referring to using `GetObject` (after discarding the wrappers with `First`).

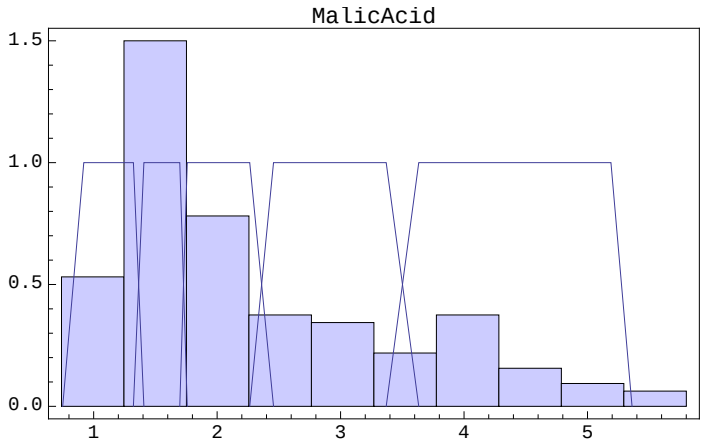
```
Map[GetObject[First[#]] &, vars]
```

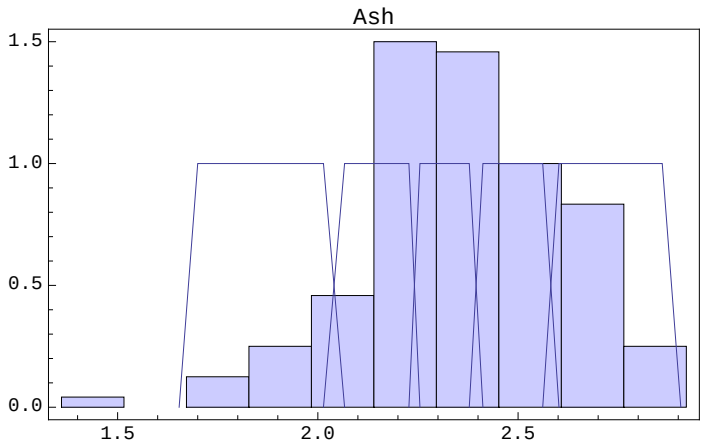
```
{Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[1., 1.]}]],
 Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[2., 1.]}]],
 Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[3., 1.]}]]}
```

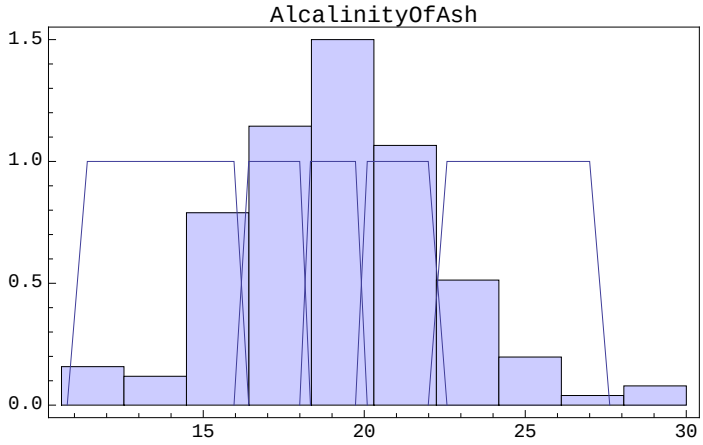
We can easily plot the fuzzy partitions together with histograms of the respective data.

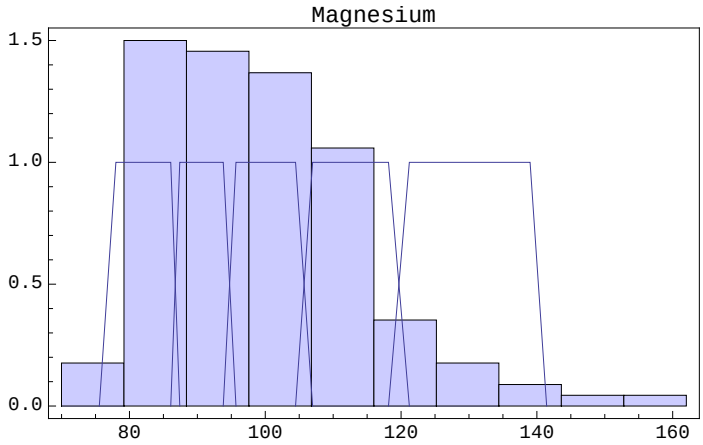
```
PlotPartitions[id3tree, wineData]
```

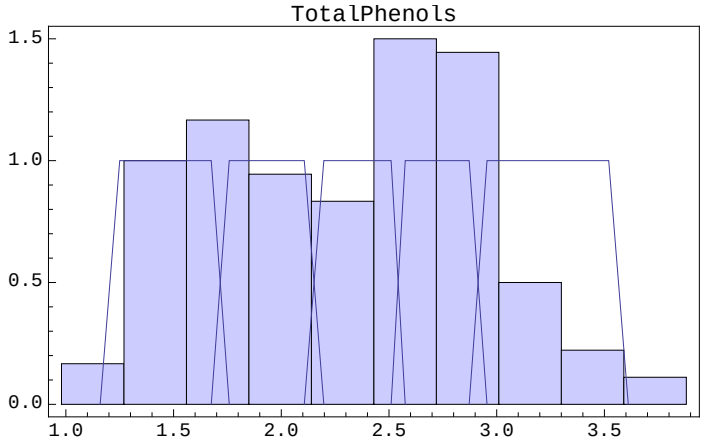


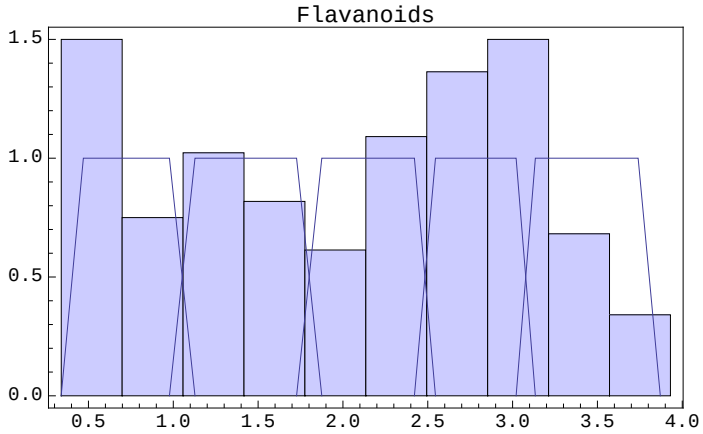

MalicAcid
1.5
1.0
0.5
0.0
1
2
3
4
5

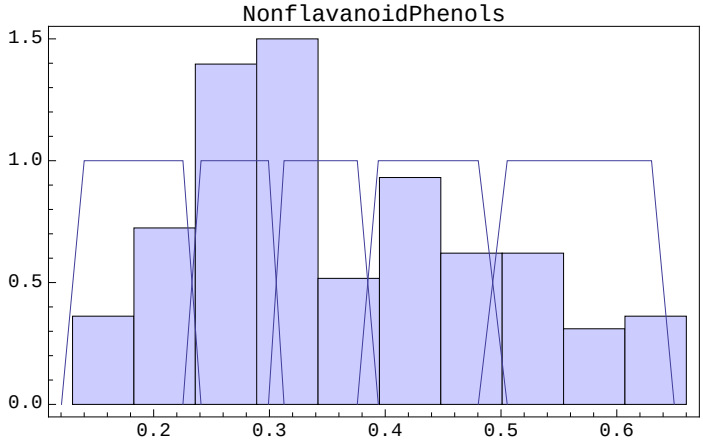

Ash
1.5
1.0
0.5
0.0
1.5
2.0
2.5

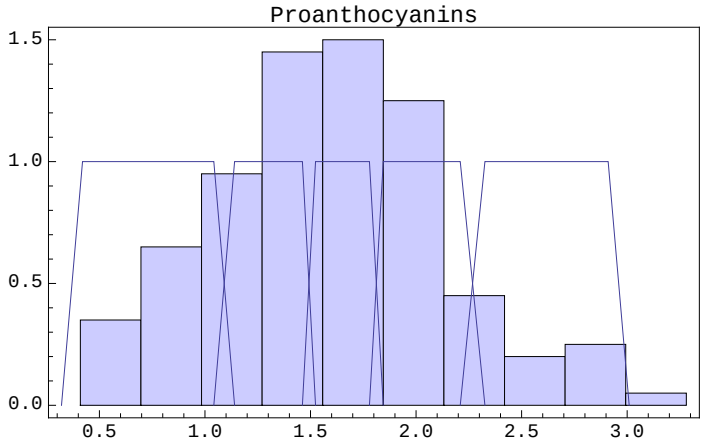

AlcalinityOfAsh
1.5
1.0
0.5
0.0
15
20
25
30

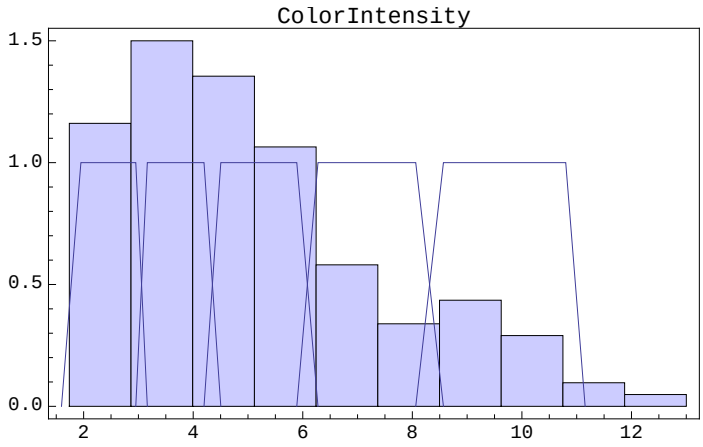

Magnesium
1.5
1.0
0.5
0.0
80
100
120
140
160

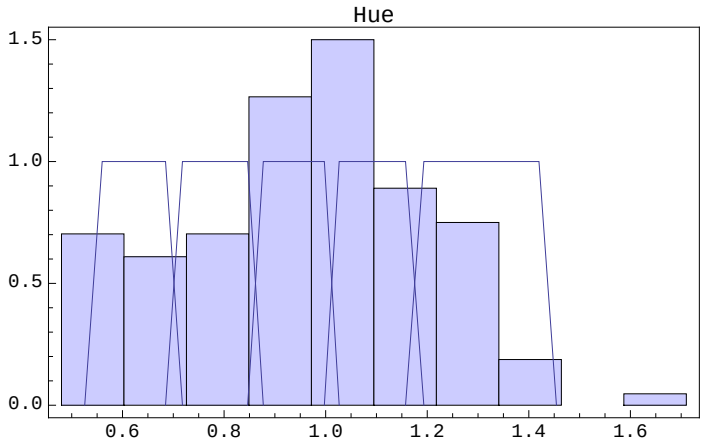

TotalPhenols
1.5
1.0
0.5
0.0
1.0
1.5
2.0
2.5
3.0
3.5

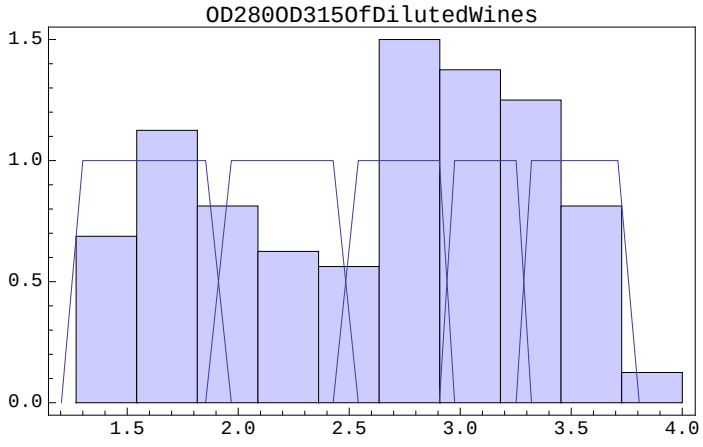

Flavanoids
1.5
1.0
0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

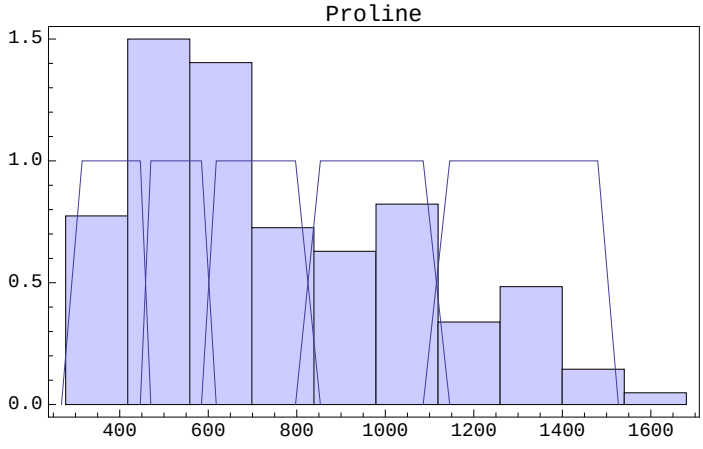

NonflavanoidPhenols
1.5
1.0
0.5
0.0
0.2
0.3
0.4
0.5
0.6


Proanthocyanins
1.5
1.0
0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0


ColorIntensity
1.5
1.0
0.5
0.0
2
4
6
8
10
12


Hue
1.5
1.0
0.5
0.0
0.6
0.8
1.0
1.2
1.4
1.6

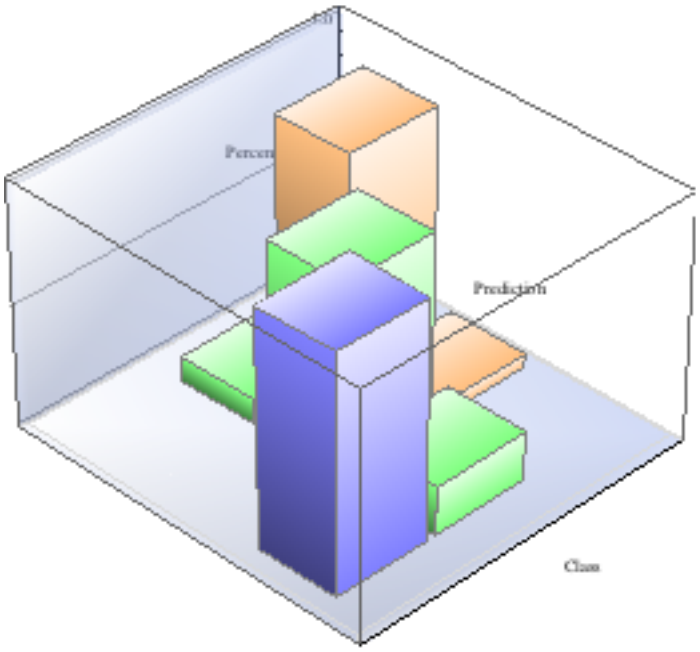

OD280OD315OfDilutedWines
1.5
1.0
0.5
0.0
1.5
2.0
2.5
3.0
3.5
4.0


Proline
1.5
1.0
0.5
0.0
400
600
800
1000
1200
1400
1600

- **Evaluating the learned tree on the test data**

```
recalledID3 = Predict[id3tree, wineTestData];

targetData = MLFGetData[wineTestData, All, 1];

evalMatrix = CompEvalMatrix[targetData, recalledID3];
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} 1 & \frac{1}{5} & 0 \\ 0 & \frac{13}{16} & \frac{1}{16} \\ 0 & \frac{1}{10} & \frac{9}{10} \end{pmatrix}$$



```
fractionCorrectID3 = MLFFractionEqual[targetData, recalledID3]
```

```
0.888889
```

```
failedS = Extract[MLFGetData@wineTestData,
   ExtractNonMatchingRecs[wineTestData, 1, recalledID3]];
failedS // TableForm
```

```
2.    12.04    4.3     2.38    22.    80.    2.1    1.75    0.42    1.35    2.6     0.79
2.    13.03    0.9     1.71    16.    86.    1.95   2.03    0.24    1.46    4.6     1.19
3.    13.52    3.17    2.72    23.5   97.    1.55   0.52    0.5     0.55    4.35    0.89
2.    13.49    1.66    2.24    24.    87.    1.88   1.84    0.27    1.03    3.74    0.98
```

- **Rule base**

- **Compute a set of rules**

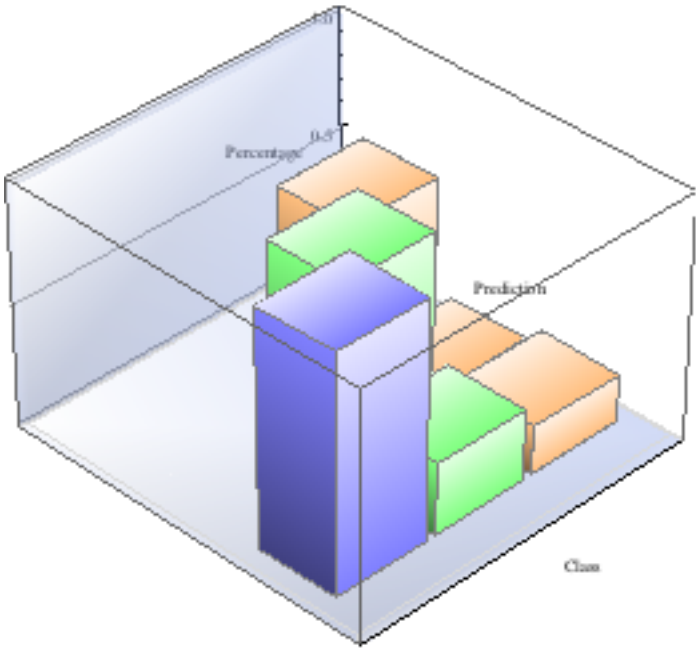
```
foilRules = TrainModel[wineData, 1, Algorithm → CreateFOIL];

PlotModel[foilRules, Info → True]
```

Class	tt	tf	rt	conf	supp	Condition
Class_Is_C1	43.7	8.02	5.3	0.84	0.31	Proline_IsAtLeast_H
	43.7	8.02	5.3	0.84	0.31	Total
Class_Is_C2	46.85	7.14	8.15	0.87	0.33	Alcohol_IsAtMost_L
	46.85	7.14	8.15	0.87	0.33	Total
Class_Is_C3	31.03	4.34	6.97	0.88	0.22	OD280OD315OfDilutedWine
	31.03	4.34	6.97	0.88	0.22	Total

■ **Evaluating the learned set of rules on the test data**

```
recalledFOIL = Predict[foilRules, wineTestData];

targetData = MLFGetData[wineTestData, All, 1];

evalMatrix = CompEvalMatrix[targetData, recalledFOIL];
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} 1 & \frac{3}{10} & \frac{1}{5} \\ 0 & \frac{13}{16} & \frac{1}{8} \\ 0 & 0 & \frac{3}{5} \end{pmatrix}$$




```
fractionCorrectFOIL = MLFFractionEqual[targetData, recalledFOIL]
```

```
0.805556
```

```
failedS = Extract[MLFGetData@wineTestData,
    ExtractNonMatchingRecs[wineTestData, 1, recalledFOIL]];
failedS // TableForm
```

3.	12.53	5.51	2.64	25.	96.	1.79	0.6	0.63	1.1	5.	0.82
3.	12.58	1.29	2.1	20.	103.	1.48	0.58	0.53	1.4	7.6	0.58
2.	13.67	1.25	1.92	18.	94.	2.1	1.79	0.32	0.73	3.8	1.23
3.	12.85	3.27	2.58	22.	106.	1.65	0.6	0.6	0.96	5.58	0.87
2.	13.03	0.9	1.71	16.	86.	1.95	2.03	0.24	1.46	4.6	1.19
3.	13.52	3.17	2.72	23.5	97.	1.55	0.52	0.5	0.55	4.35	0.89
2.	13.49	1.66	2.24	24.	87.	1.88	1.84	0.27	1.03	3.74	0.98

■ Descriptions

■ **Compute a set of descriptions**

```
descriptions = TrainModel[wineData, 1, Algorithm → CreateMINER];
```

```
PlotModel[descriptions]
```

Class	{}	Condition
Class_Is_C1	⇐	Flavanoids_IsAtLeast_M && Alcohol_IsAtLeast_M && Magnesium_IsAtLeast_L
		Flavanoids_IsAtLeast_M && Alcohol_IsAtLeast_M && Proline_IsAtLeast_M
		Flavanoids_IsAtLeast_M && Alcohol_IsAtLeast_H && Magnesium_IsAtLeast_L
		Flavanoids_IsAtLeast_M && Alcohol_IsAtLeast_H && Proline_IsAtLeast_L
		Flavanoids_IsAtLeast_M && Alcohol_IsAtLeast_H && Proline_IsAtLeast_M
Class_Is_C2	⇐	Alcohol_IsAtMost_L && Flavanoids_IsAtLeast_L
		Alcohol_IsAtMost_L && ColorIntensity_IsAtMost_M
Class_Is_C3	⇐	Hue_IsAtMost_L && OD280OD315OfDilutedWines_IsAtMost_L
		OD280OD315OfDilutedWines_IsAtMost_L && Hue_IsAtMost_M
		OD280OD315OfDilutedWines_Is_VL && Hue_IsAtMost_L
		OD280OD315OfDilutedWines_Is_VL && MalicAcid_IsAtLeast_M
		OD280OD315OfDilutedWines_Is_VL && MalicAcid_IsAtLeast_H

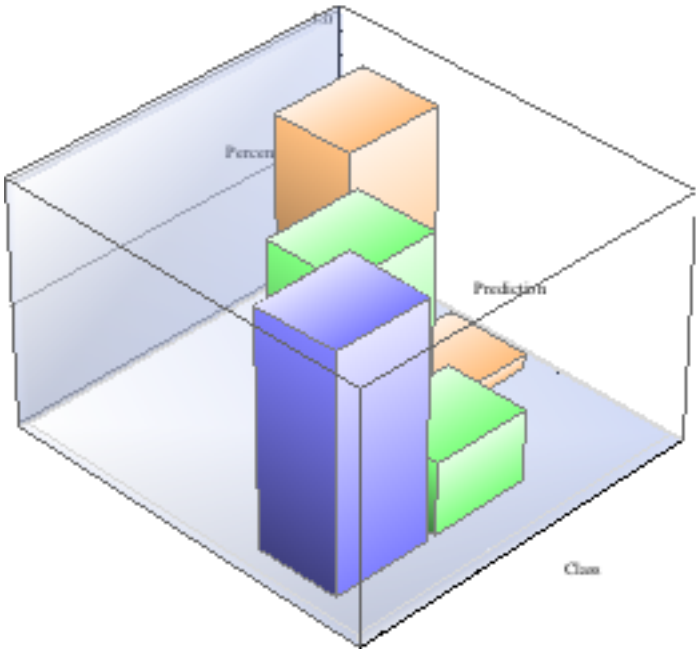
- **Evaluating the learned descriptions on the test data**

```
recalledMINER = Predict[descriptions, wineTestData, Logic → LogicL];
(* using Lukasiewicz logic *)
```

```
targetData = MLFGetData[wineTestData, All, 1];
```

```
evalMatrix = CompEvalMatrix[targetData, recalledMINER];
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} 1 & \frac{3}{10} & 0 \\ 0 & \frac{13}{16} & \frac{1}{16} \\ 0 & 0 & \frac{9}{10} \end{pmatrix}$$



```
fractionCorrectMINER = MLFFractionEqual[targetData, recalledMINER]
```

```
0.888889
```

Now we can compare the performance of ID3, FOIL and MINER:

```
fractionCorrectID3
fractionCorrectFOIL
fractionCorrectMINER
```

```
0.888889
```

```
0.805556
```

```
0.888889
```

Again, the MINER rule base performs best. However, a different split into training and test data may produce a different result. Hence it is a good idea to evaluate several such splits, as it is done in the next subsection via cross-validation.

■ Cross-validation

So far, we have learned and evaluated a model on a single split of the data into training and test model. As we will see shortly, the resulting models can vary considerably if this split is changed. One method to assess the performance of a given learning algorithm is to evaluate it on several splits of the data into training and test sets. One popular method is cross-validation, which is described below.

`CrossValidation` splits the given data set into n disjoint parts and applies the given algorithm to any of the n possible combinations of n-1 subsets each. The models are then tested using the rest 1/n of the data which were not used to train the respective model. The result contains statistical information about the performance of the n models and about overall performance, as well as - by default - the models themselves and predictions.

```
wineData = LoadData["wine.data", "Table"];
```

```
id3models = CrossValidation[wineData, 1, Algorithm → CreateID3, RandomSeed → 1];
```

NB: The data are split randomly. However, by setting a `RandomSeed`, one can ensure that the results will always be the same.

`GetParamList` gives an overview over the quantities returned by `CrossValidation`:

```
GetParamList[id3models]
```

```
{PartialResults, TotalNumberOfInstances, TotalFractionCorrect,
 TotalNormalizedMutualInformation, TotalConfusionMatrix, TotalChiSquared, TotalPLevel,
 TotalMutualInformation, TotalWeightedClassError, TotalRatioOfNullPredictions,
 TotalClassCondAccuracy, TotalChanceLevel, TotalMeanClassCondAccuracy,
 MaxChanceLevel, MaxChiSquared, MaxFractionCorrect, MaxMeanClassCondAccuracy,
 MaxModelSize, MaxMutualInformation, MaxNormalizedMutualInformation,
 MaxNumberOfInstances, MaxPLevel, MaxRatioOfNullPredictions, MaxWeightedClassError,
 MinChanceLevel, MinChiSquared, MinFractionCorrect, MinMeanClassCondAccuracy,
 MinModelSize, MinMutualInformation, MinNormalizedMutualInformation,
 MinNumberOfInstances, MinPLevel, MinRatioOfNullPredictions, MinWeightedClassError,
 QuantileChanceLevel, QuantileChiSquared, QuantileFractionCorrect,
 QuantileMeanClassCondAccuracy, QuantileModelSize, QuantileMutualInformation,
 QuantileNormalizedMutualInformation, QuantileNumberOfInstances, QuantilePLevel,
 QuantileRatioOfNullPredictions, QuantileWeightedClassError, AverageModelSize}
```

The statistical measures that are returned depend on the type of the goal attribute: for numeric goals, different measures are used.

The parameter `PartialResults` contains 10 lists, each containing statistical information on a particular model (trained on a particular subset of the given data), as well as - by default - predictions, the particular subset of the original test data that was used, and the model; to list the contents relating to the first subset:

```
GetParamList[GetParam[id3models, PartialResults][[1]]]
```

```
{NumberOfInstances, FractionCorrect, NormalizedMutualInformation,
 ConfusionMatrix, ChiSquared, PLevel, MutualInformation, WeightedClassError,
 RatioOfNullPredictions, ClassCondAccuracy, ChanceLevel,
 MeanClassCondAccuracy, ModelSize, Predictions, OriginalData, Model}
```

By default, 10-fold cross-validation is performed, so we have 10 partial results. To obtain a particular result, we do not have to explicitly extract the i-th part of `PartialResults` and the parameter in question therefrom, but we can use `GetParam` with the index of the model as third parameter:

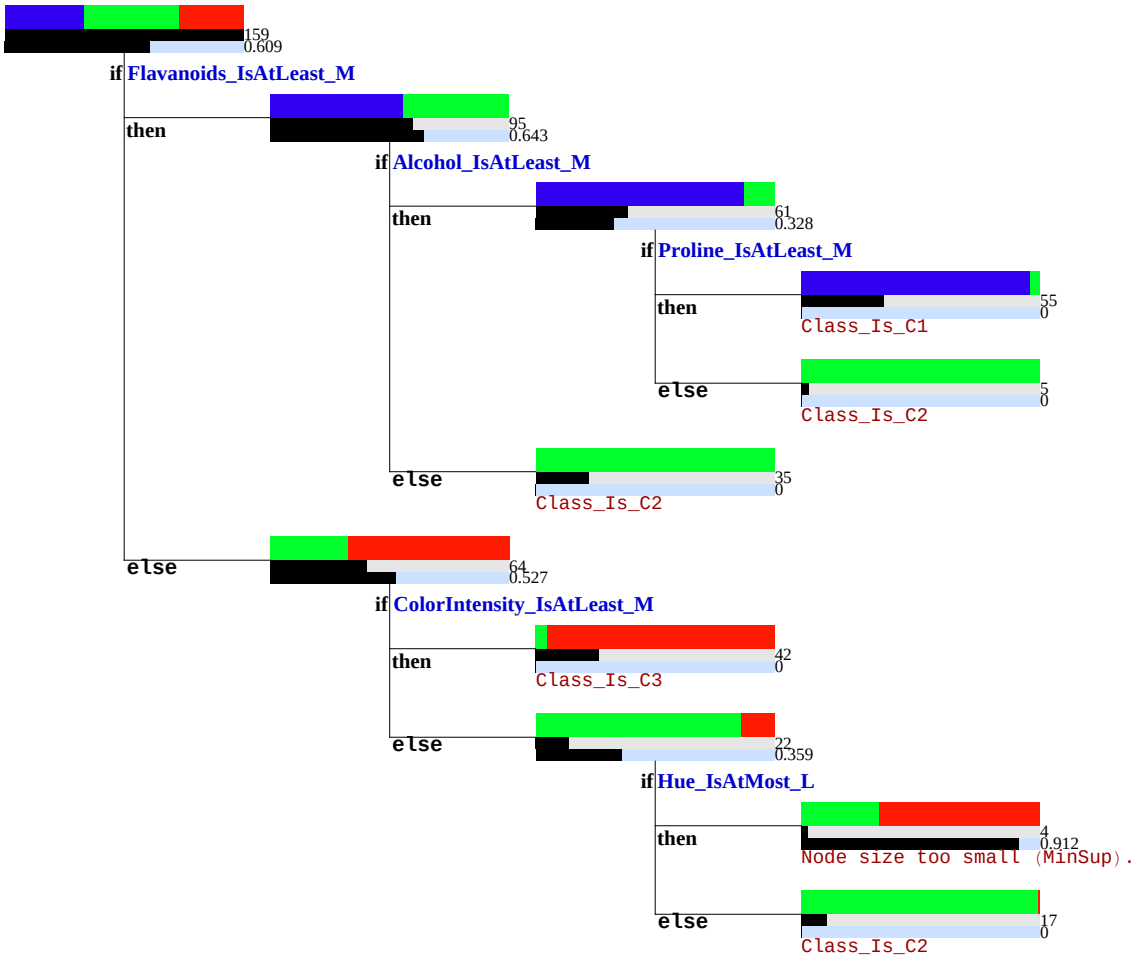
```
GetParam[id3models, FractionCorrect, 1]
```

```
1.
```

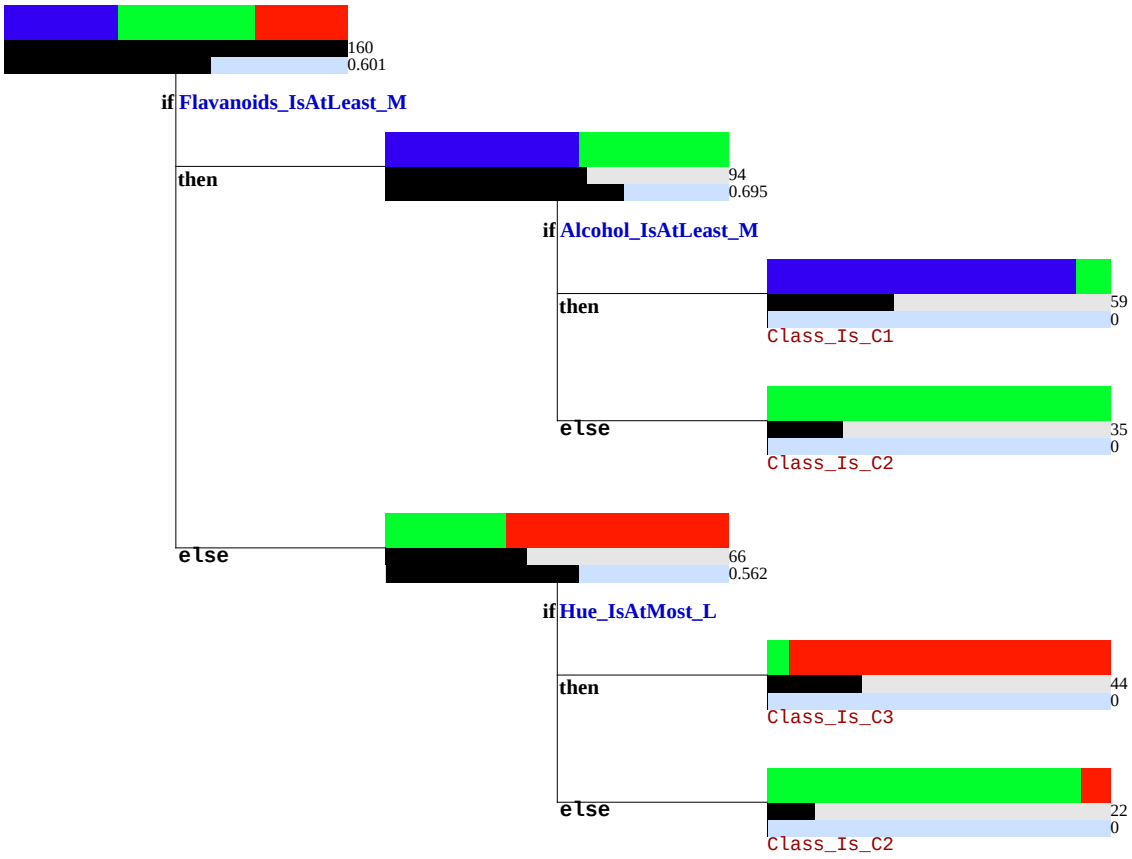
The models are each of data structure `MLFModel`, thus they contain certain meta-information which makes it easy to use them further. To plot the first and the third model:

```
PlotModel[GetParam[id3models, Model, 1]]
Null
PlotModel[GetParam[id3models, Model, 3]]
```

- Class_Is_C1
- Class_Is_C2
- Class_Is_C3



- Class_Is_C1
- Class_Is_C2
- Class_Is_C3



As you can see, these trees are quite different - due to the different sets of instances used by `CrossValidation` to train the trees.

Now let us see how well these two models predicted the classes of the respective test data set by inspecting the corresponding confusion matrices:

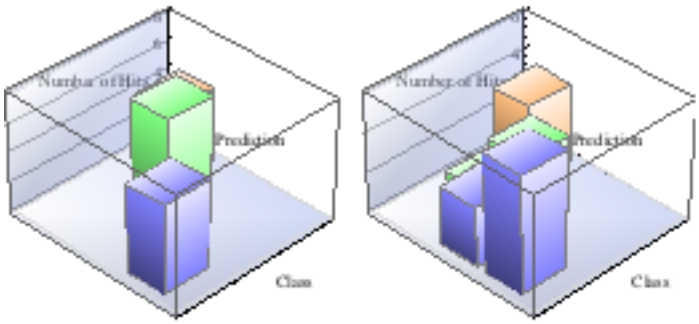
```
cMatrix1 = GetParam[id3models, ConfusionMatrix, 1];
cMatrix3 = GetParam[id3models, ConfusionMatrix, 3];
cMatrix1 // MatrixForm
cMatrix3 // MatrixForm
```

$$\begin{pmatrix} 6 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 6 & 0 & 0 \\ 3 & 4 & 0 \\ 0 & 1 & 4 \end{pmatrix}$$

In graphical form:

```
Show[GraphicsGrid[
  {{PlotEvalChart3D[cMatrix1, DisplayFunction → Identity, Percentage → False],
    PlotEvalChart3D[cMatrix3, DisplayFunction → Identity, Percentage → False]}}]]
```



Also `FractionCorrect` shows a difference in performance:

```
GetParam[id3models, FractionCorrect, 1]
GetParam[id3models, FractionCorrect, 3]
```

```
1.
```

```
0.777778
```

To get an overall picture of the 10 confusion matrices:

```
cMatrices = GetParam[id3models, TotalConfusionMatrix];
```

```
cMatrices // N // MatrixForm
PlotEvalChart3D[cMatrices, Percentage → False];
```

$$\begin{pmatrix} 57. & 2. & 0. \\ 6. & 61. & 4. \\ 0. & 3. & 45. \end{pmatrix}$$

```
GetParam[id3models, TotalFractionCorrect]
```

```
0.91573
```

This means that about 94% of instances were classified correctly.

`CrossValidation` can be used to compare different algorithms with different parameters which is demonstrated at the end of the next example.

Application example - credit grants

- **Problem statement and description of the data set**

  In this example, we will analyze data of about 404 customers of a bank to find out which customers shall be granted a credit. The data set contains information about profession, income, and other properties of the customers.

- **Set up Data**

```
{trainData, testData} =
  LoadData["credits.txt", "Table", TrainPart → 0.66, RandomSeed → 1];
```

```
GetParam[trainData, Labels]
```

```
{Name, Sex, FamiliyStatus, Childs,
 Profession, Properties, Income, Savings, CreditRequest}
```

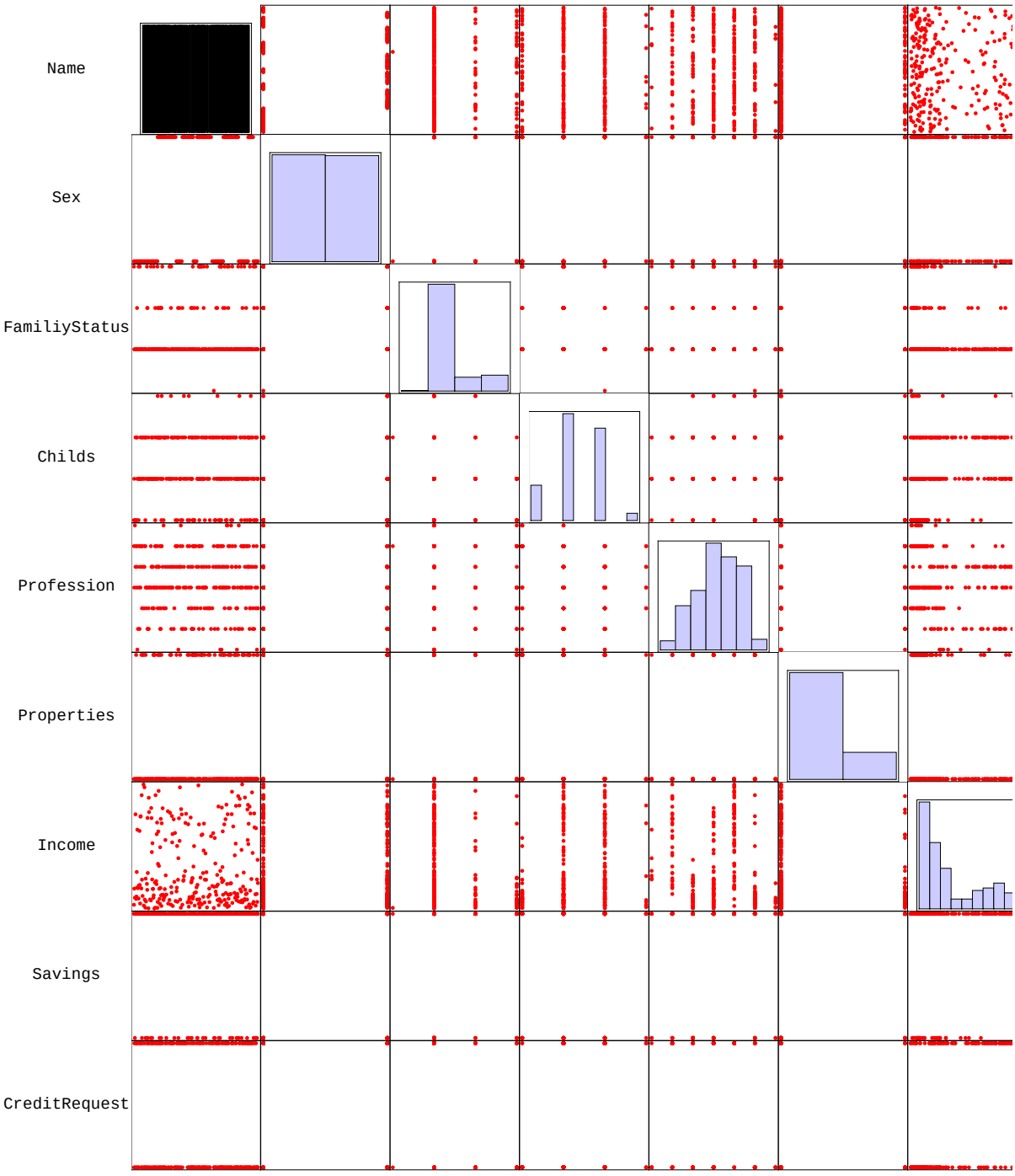
We have been informed that the 4th attribute, "Childs", will be treated as categorical by default, because it contains only 4 different values. If we decide that it would make more sense to treat this attribute as numeric, we can reduce the value of `MinDiffValues`:

```
{trainData, testData} =
  LoadData["credits.txt", "Table", TrainPart → 0.66, RandomSeed → 1, MinDiffValues → 0];
```

With this setting, all attributes which do not contain any non-numeric value will be treated as numeric. Another possibility would be to explicitly set the attribute types with option `AttrTypes`.

Visualize Data

```
PlotAttributeMatrix[trainData, ImageSize → 800]
```



Because there is a lot of names all happening just once, the "Name" attribute histogram is full of lines and thus all black.

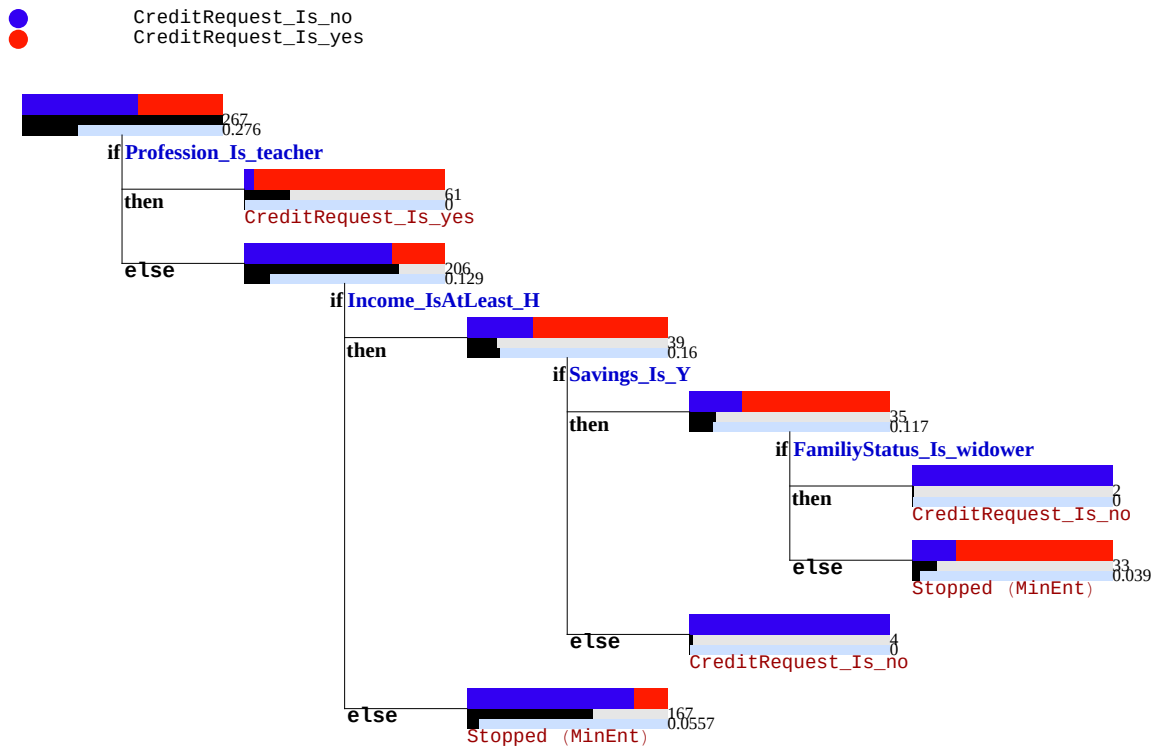
Decision Tree

Compute a decision tree

The goal attribute is the last one, which is the default. The first attribute, *Name*, is obviously irrelevant; to exclude it from the model, we must explicitly state the input dimensions.

```
id3tree = TrainModel[trainData, {2, 3, 4, 5, 6, 7, 8}, Algorithm → CreateID3];
```

PlotModel[id3tree]



It is obvious that this tree is unsatisfactory: instead of clear statements of classification at the leaves, we are notified in two cases that the algorithm stopped for some reason; in particular, it says that the entropy gain in the next possible branching would be smaller than what is set (by default) as the *minimum entropy gain*.

A remark to the notion of *entropy gain*: This technical term has been adopted from the literature. What is actually meant is a *gain in purity* (or *ordering*, with respect to the target attribute) obtained by a particular rule or branching, consequently a *reduction of entropy* as seen from the root towards the leaves of a decision tree.

There are many possible reasons why a machine learning algorithm does not yield satisfactory results with some default settings, and therefore *mlf* algorithms typically come with a great number of options for tuning. We will deal with those options in more detail in later chapters of the tutorial. For now, we simply give an example which yields a better result than the one above.

First let us look at all the parameters we could modify in order to obtain a better result. `TrainModel` accepts five types of options:

Options[TrainModel]

```
{Algorithm → None, InputPredOpts → {}, GoalPredOpts → {}, InputPredVars → {},
 GoalPredVars → {}, AlgorithmOpts → {}, RandomSeed → Automatic,
 ExcludeConstantColumns → False, FrequencyEqualization → None,
 Resampling → None, FeatureSelection → None, DataSetTransformations → {}}
```

With `InputPredOpts` and `GoalPredOpts`, the creation of (fuzzy) predicates can be controlled. With `InputPredVars` and `GoalPredVars`, existing predicates can be used. With `AlgorithmOpts`, the model-creating algorithm itself can be tuned.

The options for the creation of predicates - separately for goal predicates and for all input predicates - are the options for the `CreatePartition` function (with which such fuzzy partitions can be created manually):

Options[CreatePartition]

```
{NoOfSets → 5, Overlap → 0.2, Width → 1., PredType → ALL,
 SetType → PWL, MinSup → 0.025, Border → False, Linguistic → True,
 SeparateSets → False, SetNames → Automatic, CutPoints → Automatic,
 Minimum → Automatic, Maximum → Automatic, Confidence → 0.02, Labels → {},
 MaxLengthOfCombinations → 1, CascadingCombinations → False, OutputColumn → 1}
```

And these are the options for the particular algorithm we want to use:

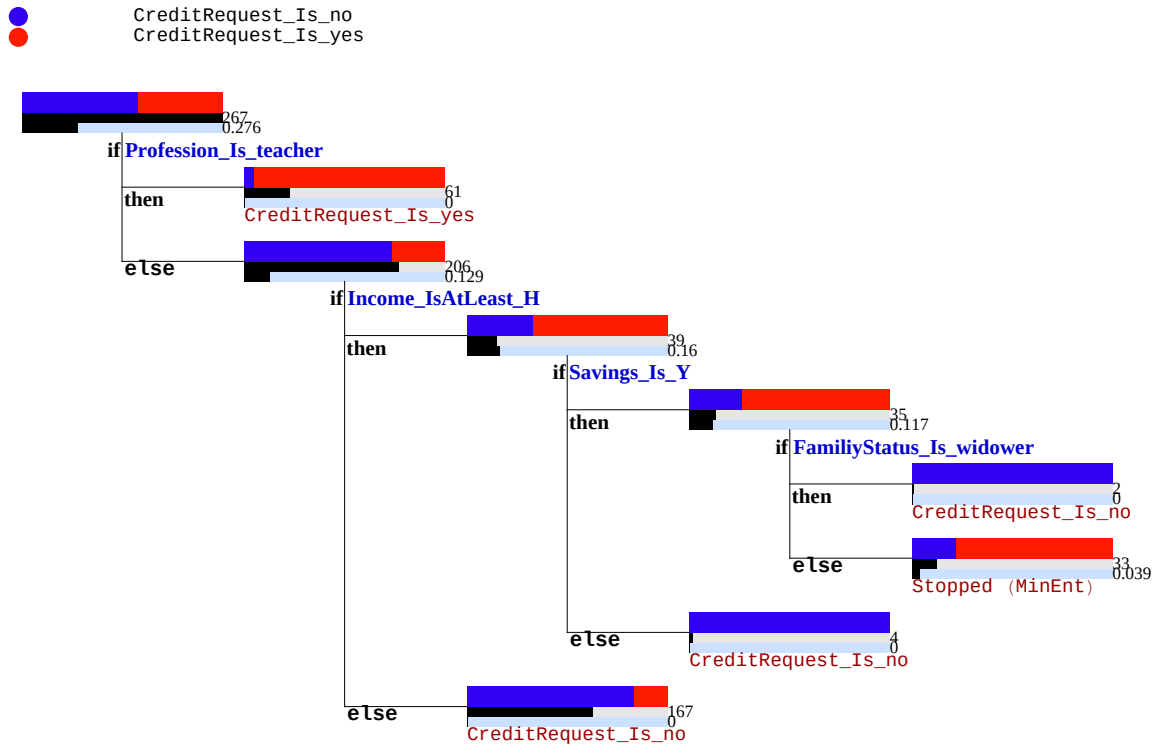
```
Options[CreateID3]
```

```
{Logic → Automatic, MinConf → 0.9, MinSup → 0.1, MinEnt → 0.1, MinIncr → 0.1,
 MaxLevel → 10, Alternatives → 0, Evaluation → Obj`EntropyEvaluation[], Goal → Automatic}
```

For the given problem, we will considerably reduce the minimum entropy gain (as suggested by the message at the second and fourth leaves above) as well as reduce the minimum confidence for each branching:

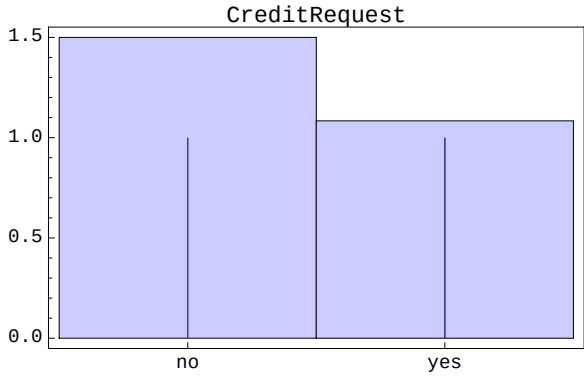
```
id3tree = TrainModel[trainData, {2, 3, 4, 5, 6, 7, 8},
   Algorithm → CreateID3, AlgorithmOpts → {MinEnt → 0.05, MinConf → 0.8}];
```

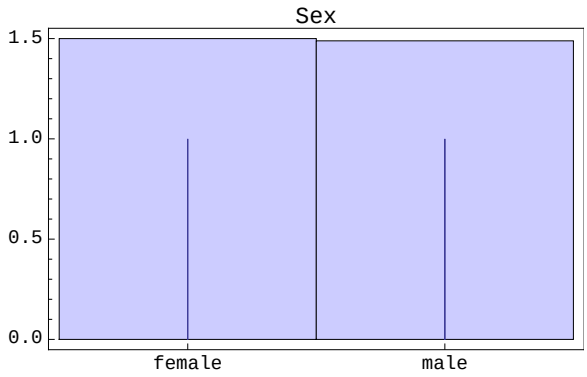
```
PlotModel[id3tree]
```

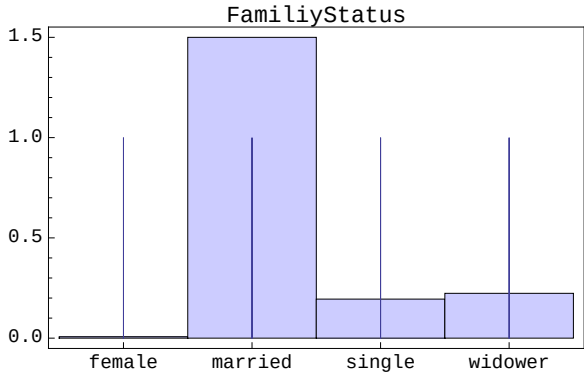


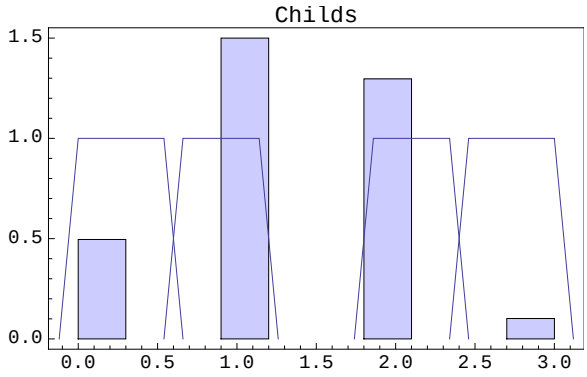
■ Show partitions defined

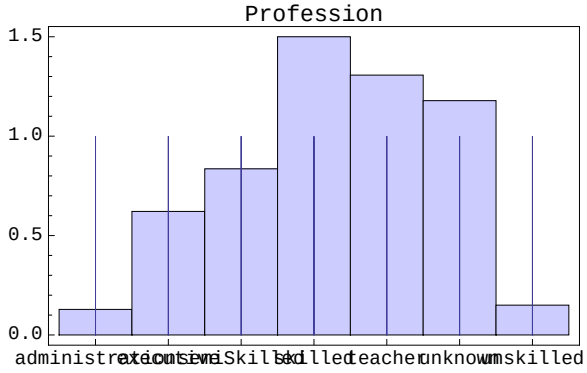
```
PlotPartitions[id3tree, trainData, ImageSize → 300]
```

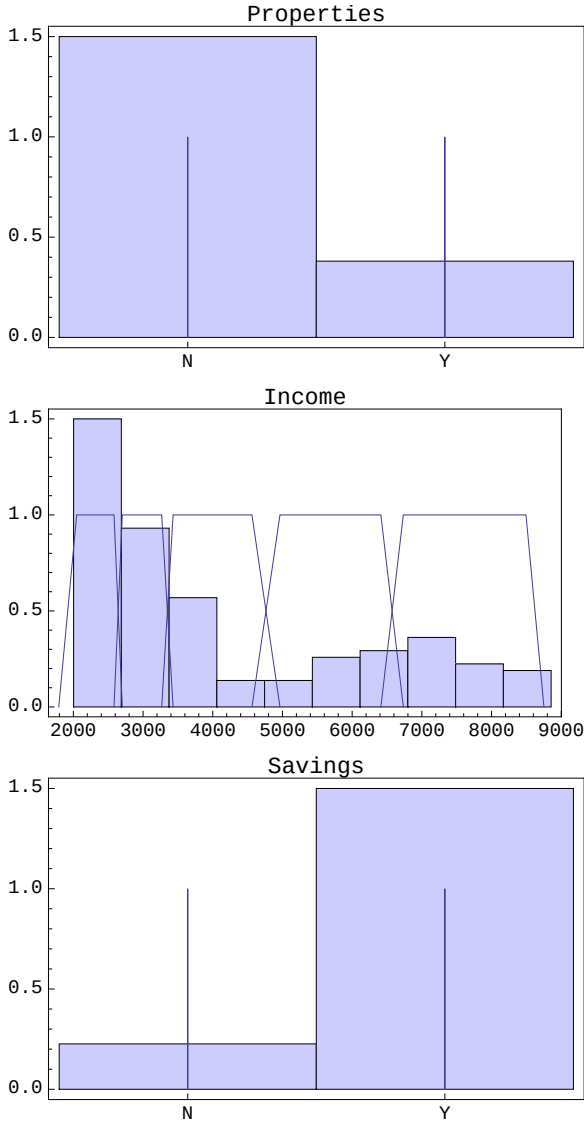



Sex
1.5
1.0
0.5
0.0
female
male


FamiliyStatus
1.5
1.0
0.5
0.0
female
married
single
widower


Childs
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


Profession
1.5
1.0
0.5
0.0
administrator
unskilled
teacher
unknown
unskilled

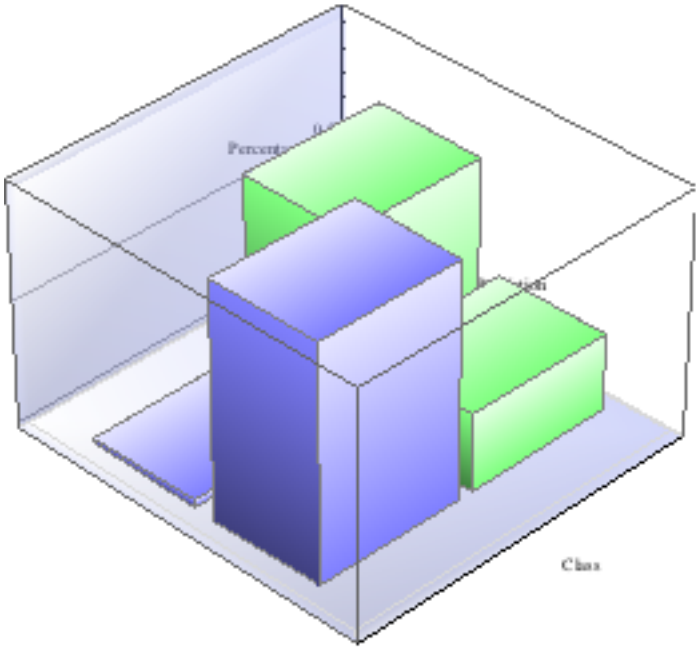


- **Evaluate the trained decision tree on the test data set**

```
recalledID3 = Predict[id3tree, testData, Type -> "PROB"];

domIndex = ComputeDomIndex[testData, GetParam[id3tree, GoalVars]];
evalMatrix = CompEvalMatrix[domIndex, recalledID3];
```

```
evalMatrix // MatrixForm
PlotEvalChart3D [evalMatrix]
```

$$\begin{pmatrix} \frac{29}{30} & \frac{19}{60} \\ \frac{2}{77} & \frac{58}{77} \end{pmatrix}$$



```
fractionCorrectID3 = MLFFractionEqual[domIndex, recalledID3]
```

```
0.846715
```

Rule base

Compute a rule set

```
foilRules = TrainModel[trainData, Algorithm → CreateFOIL];

PlotModel[foilRules]
```

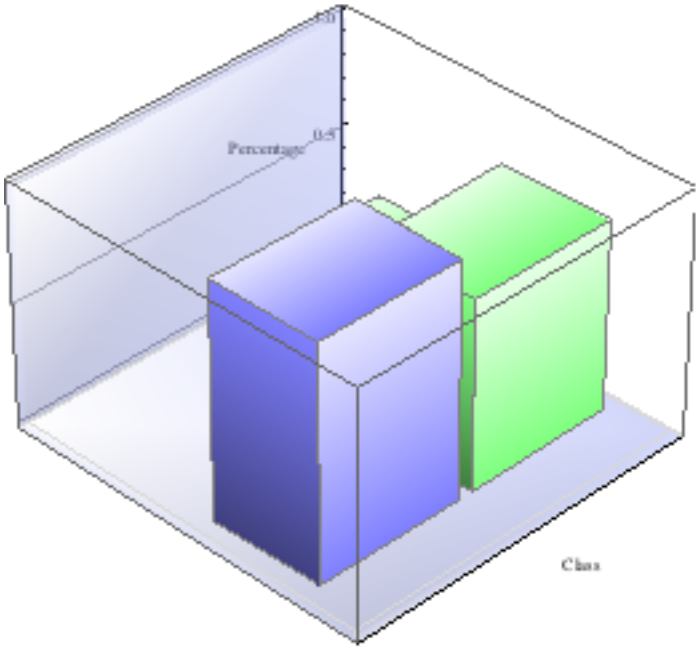
Class	{}	Condition
CreditRequest_Is_no	⇐	Income_IsAtMost_L
CreditRequest_Is_yes	⇐	Profession_Is_teacher

Evaluate the learned rule set on the test data set

```
recalledFOIL = Predict[foilRules, testData];

domIndex = ComputeDomIndex[testData, GetParam[foilRules, GoalVars]];
evalMatrix = CompEvalMatrix[domIndex, recalledFOIL];
```

```
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} 1 & \frac{4}{5} \\ 0 & \frac{29}{77} \end{pmatrix}$$




```
fractionCorrectFOIL = MLFFractionEqual[domIndex, recalledFOIL]
```

```
0.649635
```

Descriptions

Compute descriptions

```
minerRules = TrainModel[trainData, Algorithm → CreateMINER];

PlotModel[minerRules]
```

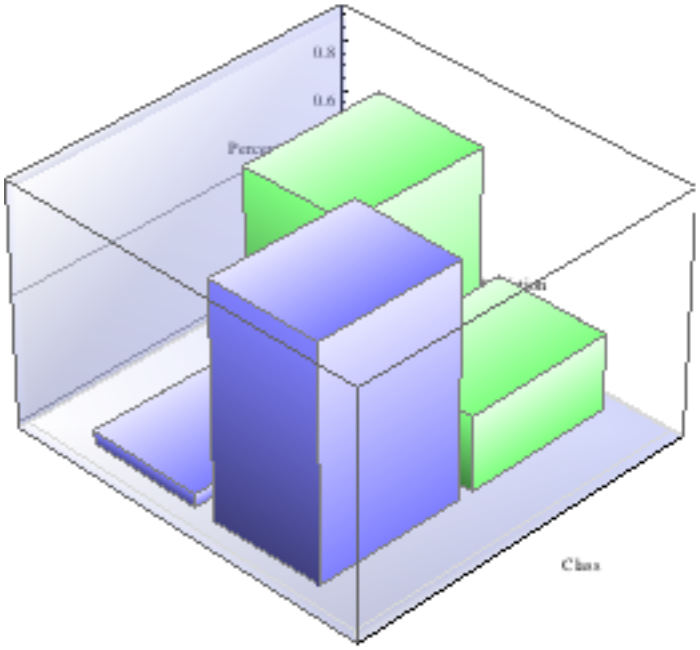
Class	{}	Condition
CreditRequest_Is_no	⇐	Income_IsAtMost_L
		Income_Is_VL
		Savings_Is_N
		Income_IsAtMost_M && Childs_Is_VL
		Income_IsAtMost_L && Profession_Is_unknown
		Income_IsAtMost_L && Childs_IsAtMost_L
		Income_Is_VL && Childs_IsAtMost_L
CreditRequest_Is_yes	⇐	Income_IsAtLeast_H && Savings_Is_Y && FamiliyStatus_Is_married
		Income_IsAtLeast_H && Savings_Is_Y
		Profession_Is_teacher && Properties_Is_N

Evaluate learned descriptions on the test set

```
recalledMINER = Predict[minerRules, testData];

domIndex = ComputeDomIndex[testData, GetParam[minerRules, GoalVars]];
evalMatrix = CompEvalMatrix[domIndex, recalledMINER];
```

```
evalMatrix // MatrixForm
PlotEvalChart3D@evalMatrix
```

$$\begin{pmatrix} \frac{14}{15} & \frac{3}{10} \\ \frac{4}{77} & \frac{59}{77} \end{pmatrix}$$




```
fractionCorrectMINER = MLFFractionEqual[domIndex, recalledMINER]
```

```
0.839416
```

Comparison with ID3 and FOIL:

```
fractionCorrectID3
fractionCorrectFOIL
```

```
0.846715
```

```
0.649635
```

So in this example, MINER and ID3 appears to outperform FOIL with respect to predictive power. However, as mentioned before, results may differ with a different data split, i.e., with different training and test data. Cross-validation gives a better comparison.

■ Comparing Different Algorithms via Cross-validation

In this subsection we use `CrossValidation` to assess the performance of different learning algorithms on the credit data set:

```
creditsData = LoadData["credits.txt", "Table", RandomSeed → 1, MinDiffValues → 0];

somModels = CrossValidation[creditsData,
    {2, 3, 4, 5, 6, 7, 8}, Algorithm → CreateSOM, RandomSeed → 1];

id3DefModels = CrossValidation[creditsData,
    {2, 3, 4, 5, 6, 7, 8}, Algorithm → CreateID3, RandomSeed → 1];

id3Models = CrossValidation[creditsData, {2, 3, 4, 5, 6, 7, 8}, Algorithm → CreateID3,
    AlgorithmOpts → {MinEnt → 0.05, MinConf → 0.8}, RandomSeed → 1];

foilModels = CrossValidation[creditsData,
    {2, 3, 4, 5, 6, 7, 8}, Algorithm → CreateFOIL, RandomSeed → 1];

minerModels = CrossValidation[creditsData,
    {2, 3, 4, 5, 6, 7, 8}, Algorithm → CreateMINER, RandomSeed → 1];
```

Below, we print the overall fraction of correctly classified instances for each algorithm, starting with the algorithm showing the best performance:

```
comparisonTable = {
    {"SOM", GetParam[somModels, TotalFractionCorrect]},
    {"FS-ID3 with default parameters",
     GetParam[id3DefModels, TotalFractionCorrect]},
    {"FS-ID3 with user settings", GetParam[id3Models, TotalFractionCorrect]},
    {"FS-FOIL", GetParam[foilModels, TotalFractionCorrect]},
    {"FS-MINER", GetParam[minerModels, TotalFractionCorrect]}
   };
TableForm[
 Sort[comparisonTable, #1[[2]] > #2[[2]] &],
 TableHeadings → {None, {StyleForm["ALGORITHM", FontWeight -> "Bold"], StyleForm[
     "PERCENTAGE CORRECT\n(10-fold crossvalidation)", FontWeight → "Bold"]}}
]
```

ALGORITHM	PERCENTAGE CORRECT (10-fold crossvalidation)
FS-MINER	0.834158
FS-ID3 with user settings	0.811881
FS-ID3 with default parameters	0.811881
SOM	0.782178
FS-FOIL	0.717822

This result indicates that for the given problem, FS-MINER seems to be the best choice with respect to predictive performance.

More About

Template for Supervised data analysis I: Classification

Related Tutorials

Supervised Data Analysis (this tutorial)

Approximation of Numerical Functions

Unsupervised Data Analysis; including image analysis

Object and Data Handling, Fuzzy Logic Operations, Visualization

Related Wolfram Education Group Courses

XXXX

machine learning framework for *Mathematica*

While we have presented the general task of *supervised classification* in Chapter 1, we will now show how supervised methods can be used to solve numeric approximation problems (also often called *regression* problems). While traditional analytical solutions might provide a higher accuracy, the models obtained from the methods shown in this chapter are very easy to interpret, also by non-experts.

- **Loading the *mlf* package**

```
Needs["mlf`"];
```

```
Off[DataSet::mindiff];
```

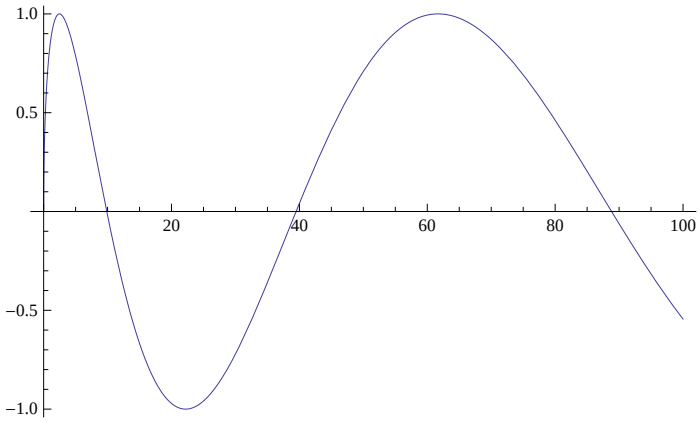
A Simple, One-dimensional Example

In this example we will show how a one-dimenional function can be approximated using various methods.

- **Set up Data**

We will use the following function f in this first example.

```
f[x_] := N[Sin[Sqrt[x]]];
range = 100;
Plot[f[x], {x, 0, range}]
```



To obtain sample data, we define the first coordinate to contain the *x* and the second coordinate to contain the *y* = f(*x*) values. If the data are given in a file and can be used as-is, `LoadData` provides the easiest way to obtain a data set. In the present example, however, we have to create a data set by hand. We will use 50 samples for training and 50 different samples for evaluating the different approximations we will consider.

```
headers = {"x", "y"};
```

```
trainData = Sort[Table[{x, f[x]}, {x, 0, 100, 1}]];
trainDataSet = Def["tagTrainDataSet1", DataSet[trainData, headers]];
```

```
testData = Sort[Table[{x, f[x]}, {x, 0.5, 100, 1}]];
testDataSet = Def["tagTestDataSet1", DataSet[testData, headers]];
```

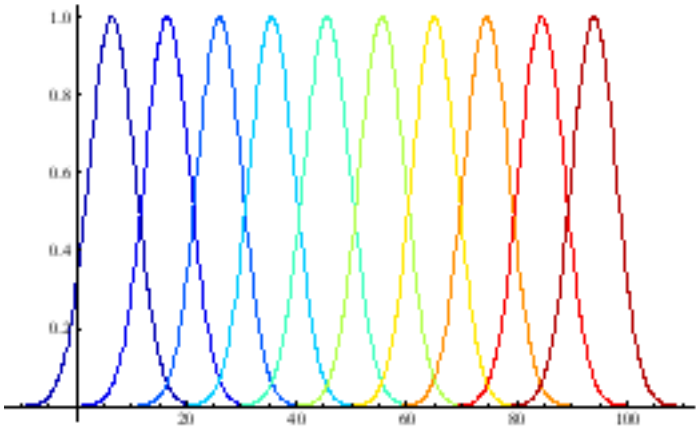
■ A Simple Fuzzy Controller

As a first step we will define a simple Sugeno controller by hand which will serve as a kind of benchmark to compare more advanced algorithms with.

Therefore we have to define a set of predicates (and predicate variables) for the input attribute *x* which is accomplished by the functions `CreatePartition` and `DefPredicateVars`. `PlotFuzzySet` is used to visualize the 10 underlying membership functions.

```
testParts = CreatePartition[trainDataSet, 1, 10,
    Overlap → 1.0,
    PredType -> "ISEX",
    SetType -> "EXP",
    Border → False,
    Linguistic → False
  ];
testVars = DefPredicateVars[testParts[[2]]];
```

```
PlotFuzzySet[testParts, ColorFunction → (CFJet[#] &)]
```



Now we will define a rule of the form

$p_i \to f\,(\mathrm{CoG}\,(p_i))$

for each predicate where $\mathrm{CoG}(p_i)$ is the center of gravity of the i-th predicate. This is done via the function call FRule[pi, f[CoG[pi]]]

```
testParts[[1, 1]]
```

```
{Set0 → Obj`FuzzySetExp[6.5, 4.23098], Set1 → Obj`FuzzySetExp[16.5, 4.1252],
 Set2 → Obj`FuzzySetExp[26., 4.01943], Set3 → Obj`FuzzySetExp[35.5, 4.1252],
 Set4 → Obj`FuzzySetExp[45.5, 4.23098], Set5 → Obj`FuzzySetExp[55.5, 4.1252],
 Set6 → Obj`FuzzySetExp[65., 4.01943], Set7 → Obj`FuzzySetExp[74.5, 4.1252],
 Set8 → Obj`FuzzySetExp[84.5, 4.1252], Set9 → Obj`FuzzySetExp[94., 4.01943]}
```

```
CoG[fuzzySet_] := EvalObject[fuzzySet];
```

```
interpolRules = Table[
    FRule[testVars[[i]], NVal@f[ CoG[ testParts[[1, 1, i, 2]] ] ] ],
    {i, Length[testParts[[1, 1]]]}
  ];
```

```
PrintRules[interpolRules]
```

```
x == 0_6.5 (0) → 0.558091
x == 1_16.5 (1) → -0.79586
x == 2_26 (2) → -0.926185
x == 3_35.5 (3) → -0.319307
x == 4_45.5 (4) → 0.445904
x == 5_55.5 (5) → 0.919437
x == 6_65 (6) → 0.978389
x == 7_74.5 (7) → 0.71277
x == 8_84.5 (8) → 0.230304
x == 9_94 (9) → -0.267292
```

For faster computation, we will define a temporary object SimpleControlSug containing this rule set in the form of an FRuleSetSugeno.

```
simpleControlSug = Def["tagSimpleControlSug", FRuleSetSugeno[interpolRules]];
```

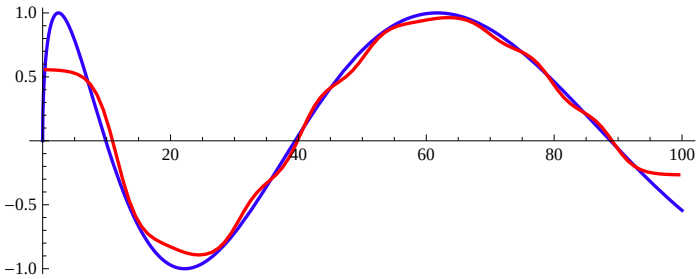
We can now define the controller as the result of the evaluation of the inference system defined by the rule set (see FuzzyInference).

```
inferenceSystem[x_] := EvalObject[FuzzyInference[simpleControlSug, x]];
```

To see how this controller approximates the desired goal function, we will evaluate it on the given test data set and plot the result.

```
recallData = Map[{x = #[[1]], inferenceSystem[{x}]} &, testData];
```

```
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```



We can see that - except for the borders - the controller performs very well.

To assess the result, we will finally compute the mean square error (MSE) of the approximation.

```
mseSimple = MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]];
Print["Mean Squared Error: ", mseSimple];
```

```
Mean Squared Error: 0.0117274
```

■ Using Self-organizing Maps (SOMs)

We will now use a SOM to create a prediction function. We want to create the SOM (of size 1×10) only in the input space of the problem (that is the single attribute x in this example). Hence we set the weight of the target value (y in our case) to zero (option Weights→{1,0}; you might want to try out different weights and see what happens).

As the SOM algorithm normalises the variances of all dimensions to 1, we have to use a higher weight for the first parameter.
(Again, you might want to try out different weights and see what happens.)

```
som = CreateSOM[trainData, Size -> {1, 10}, Weights → {1, 0}, InitType → Random];
```

The SOM interpolates the goal function with 10 supporting points in a non-linear way which can be printed like this:

```
Sort[MLFGetData[som]] // TableForm
```

```
5.      0.550432
15.5    -0.656045
25.     -0.928879
34.     -0.429938
43.     0.262846
52.     0.786293
61.     0.985363
71.     0.827086
82.     0.357095
94.     -0.260548
```

We can recall the function values using the SOM in a straight-forward way with a command of the form

```
som[{{29, Null}}, NSize → 3]
```

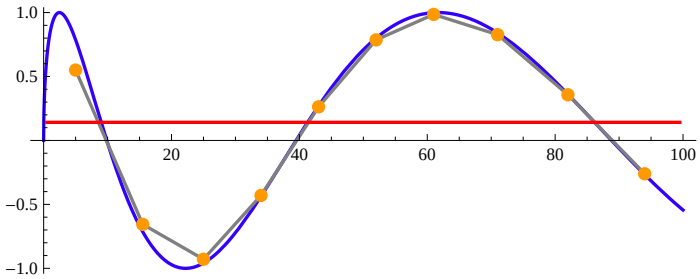
```
{{8.92425, 0.141636}}
```

where we simply place `Null` at the coordinate where we would like the SOM to fill in appropriate values. (The DataSet::mindiff warnings can be ignored in this case.) The option NSize → 3 indicates that the three nearest neighbors should be considered for prediction. Below, this is done for the whole data set:

```
recallData = Map[{x = #[[1]], som[{{x, Null}}, NSize → 3][[1, 2]]} &, testData];
```

To see how good our inital function is approximated, we create a plot which shows the original function (in blue), the location of the ten units of the SOM (orange dots), and the approximation by by the SOM (red).

```
plotOrig = Plot[f[x], {x, 0, range},
   DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotSOM = PlotSOMGrid[som, Dims → {1, 2}, PointSize -> 0.02,
   DefaultColor -> Hue[0.1], Thickness → 0.005];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
   Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotSOM, Graphics@plotRecall,
 DisplayFunction → $DisplayFunction, AspectRatio → 0.4]
```



The average squared error gives us a good measure for the quality of our approximation (you may want to try different values for NSize and Size to see how this influences the performance on the test set):

```
mseSOM = MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]];
Print["Mean Squared Error: ", mseSOM];
```

```
Mean Squared Error: 0.431663
```

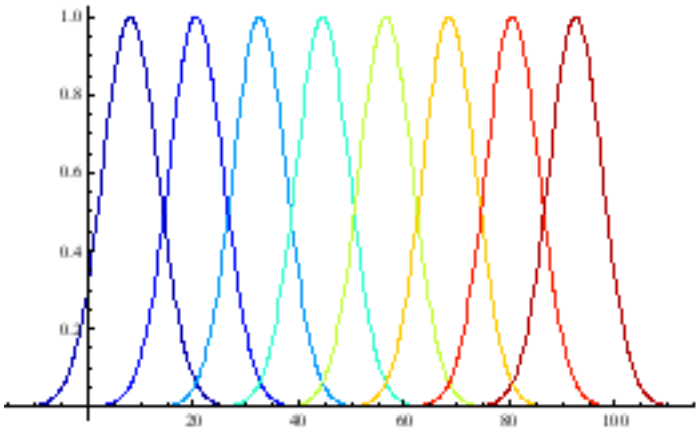
Using a Decision Tree (FS-ID3)

Create a decision tree using ID3

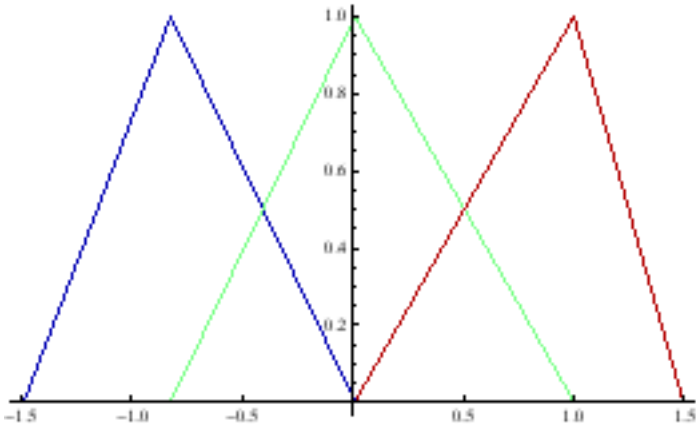
To create a decision tree for the approximation problem, we will first define the predicates.

```
testPreds = CreatePredicates[trainDataSet, 1, 8,
    Overlap → 1.0, SetType -> "EXP", Border → False, Linguistic → False];
goalPreds = CreatePredicates[trainDataSet, 2, 3, Overlap → 1.0,
    Border → True, PredType -> "ISEX"];
testVars = DefPredicateVars[testPreds];
goalVars = DefPredicateVars[goalPreds];
```

```
PlotFuzzySet[testPreds, ColorFunction → (CFJet[#] &)]
```




```
PlotFuzzySet[goalPreds, ColorFunction → (CFJet[#] &)]
```



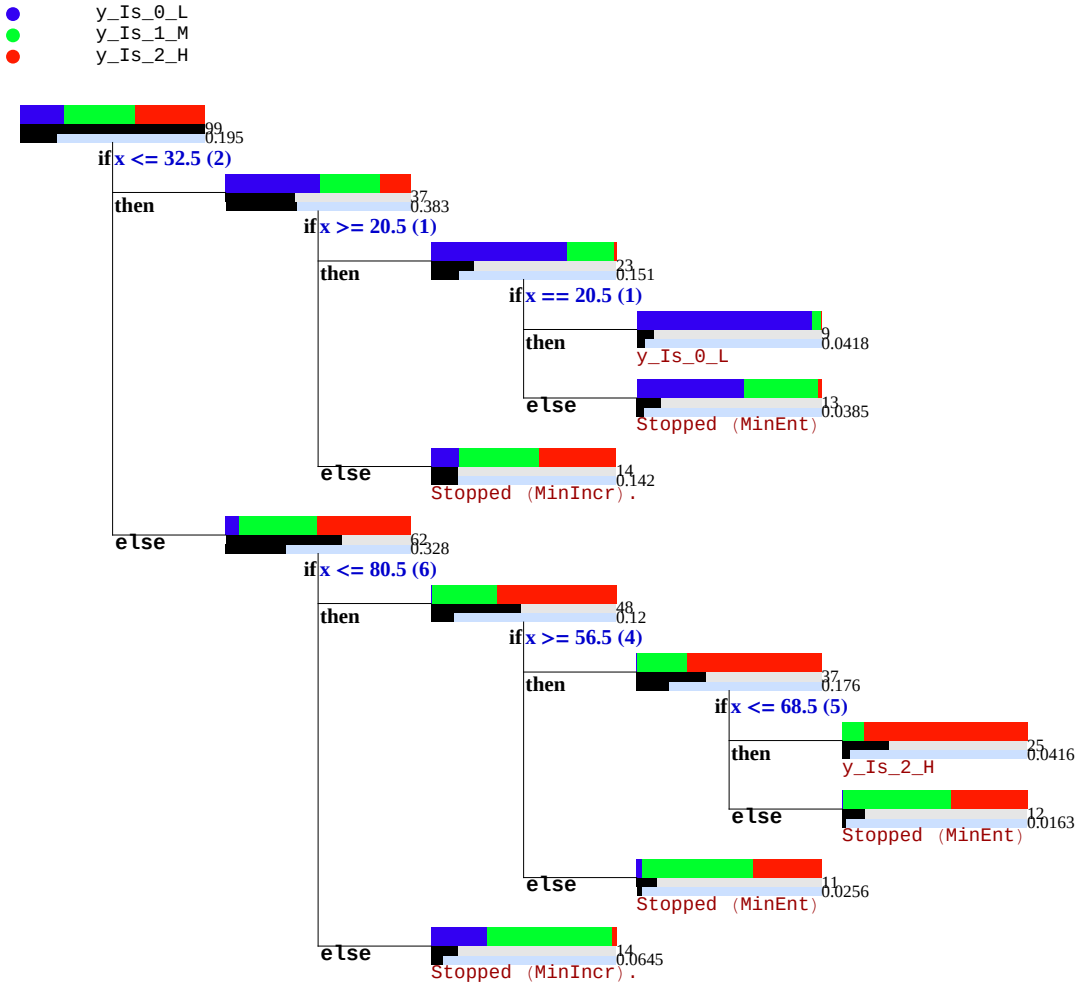
If you use non-PWL membership functions, it might be necessary to increase the minimum entropy gain to 0.05.

```
id3tree = CreateID3[trainDataSet, testVars,
    goalVars, Goal → 2, MinEnt → 0.05, Logic → LogicP, MinConf → 0.8];
```

(Option Goal is needed for the regression-view plot, see below.)

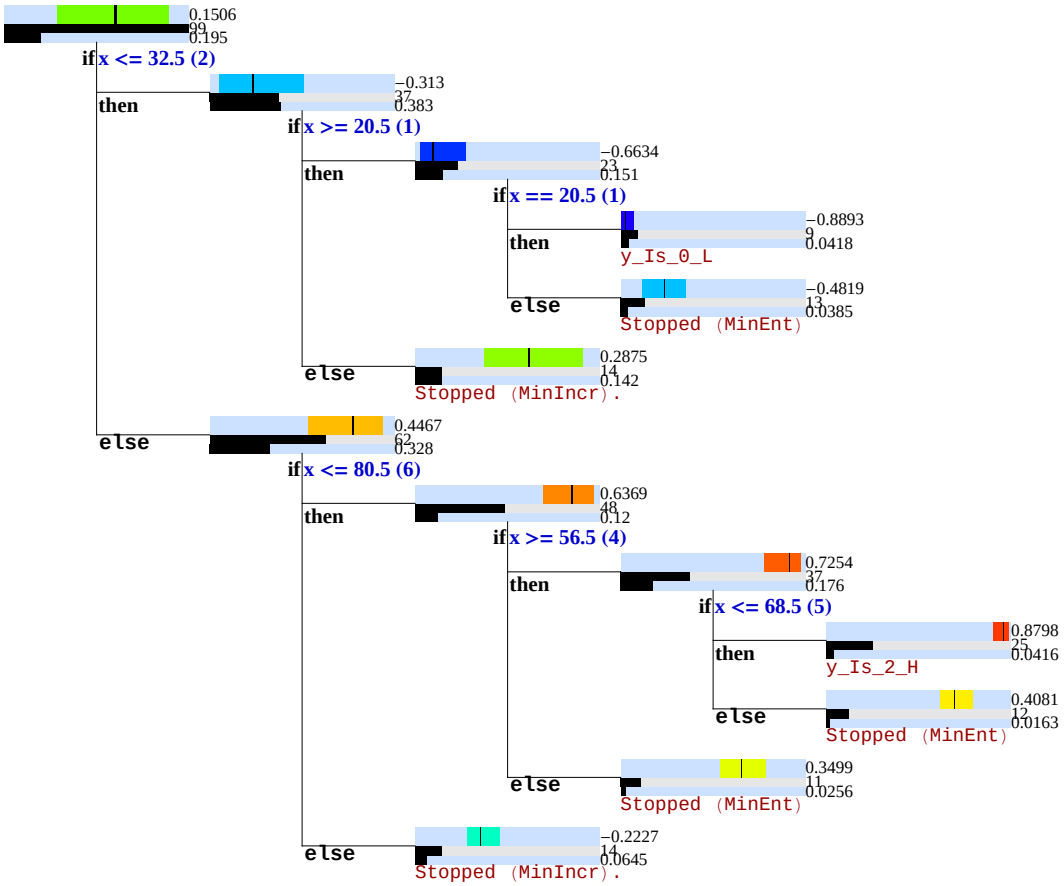
Class view of the tree:

```
Show[Graphics[PlotClassLegend[goalVars], AspectRatio → 0.1], PlotRange → All]
PlotID3[id3tree, ImageSize → 560]
```



Regression view of the tree:

```
PlotID3[id3tree, Regression → True, ImageSize → 560]
```

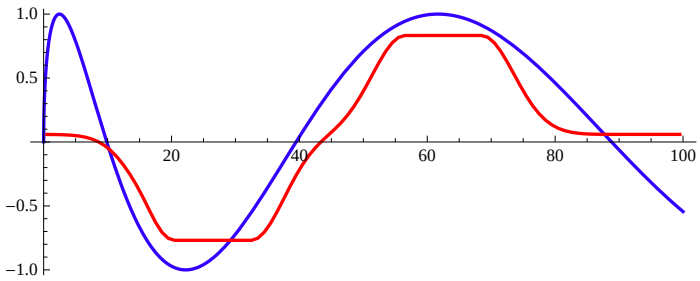


- **Create a Sugeno controller from the decision tree**

  To create a controller using the previously computed decision tree, we will use the `MakeSugenoInference` command, which evaluates the given tree with respect to the specified set of goal predicates.

  You can select different parameters to tune the evaluation process. We will use a majority-based evaluation first, which means that each node has only one goal class assigned.

```
recallData = Transpose[{
    testData[[All, 1]],
    MakeSugenoInference[id3tree, goalPreds,
     testData[[All, {1}]], Type -> "MAJOR", Logic → LogicP]
   }];
plotOrig = Plot[f[x], {x, 0, range},
   DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
   Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```

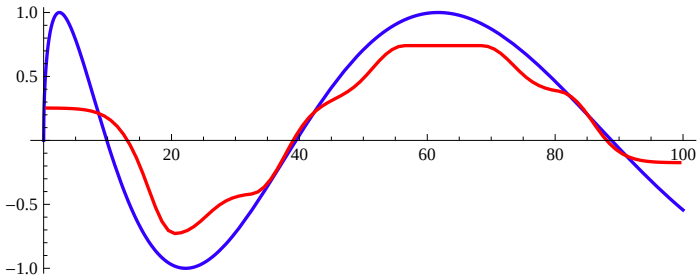


```
mseID3 = MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]];
Print["Average Squared Error: ", mseID3]
```

```
Average Squared Error: 0.100688
```

We will now try a probabilistic evaluation, where a tree node can belong to more goal classes with a certain probability.

```
recallData = Transpose[{
     testData[[All, 1]],
     MakeSugenoInference[id3tree,
      goalPreds, testData[[All, {1}]], Type -> "PROB", Logic → LogicP]
    }];
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```




```
mseID3 = MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]];
Print["Average Squared Error: ", mseID3]
```

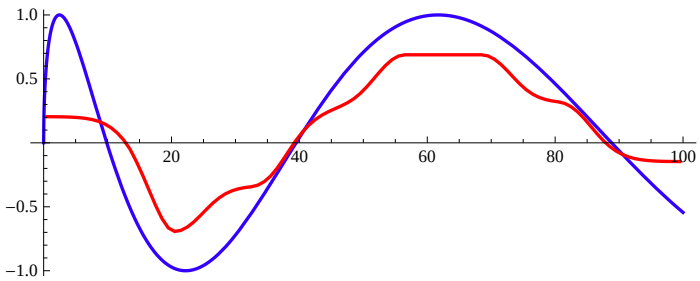
```
Average Squared Error: 0.0681852
```

- **Mamdani controller**

We will now take a look at a so called Mamdani controller. For a Mamdani controller, a rule consists of a condition and a conclusion, where the conclusion is an (arbitrary) fuzzy set. Currently only piecewise-linear fuzzy sets are supported for the right-hand side.

To evaluate a decsion tree using a Mamdani controller, we simply call `MakeMamdaniInference`.

```
recallData = Transpose[{
     testData[[All, 1]],
     MakeMamdaniInference[id3tree,
      goalPreds, testData[[All, {1}]], Type -> "PROB", Logic → LogicP]
    }];
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, plotRecall, DisplayFunction → $DisplayFunction, AspectRatio → 0.4]
```




```
Print["Average Squared Error: ",
 MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]]]
```

```
Average Squared Error: 0.0891123
```

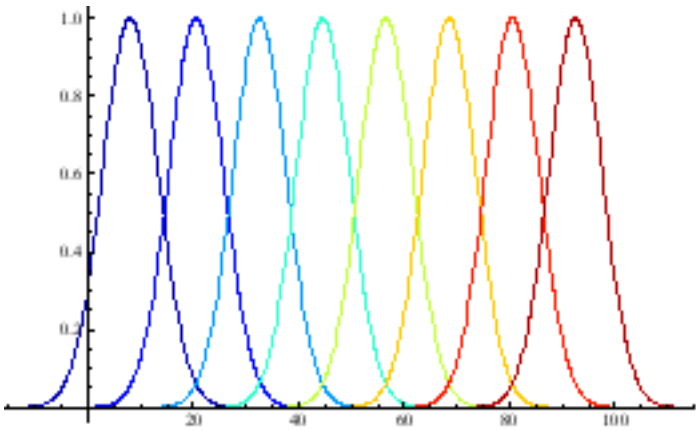
Using a Regression Tree (FS-LIRT)

Create a regression tree using LIRT

To create a regression tree for the approximation problem, we have to create predicates for the input attributes.

```
testPreds = CreatePredicates[trainDataSet, 1, 8,
    Overlap → 1.0, SetType -> "EXP", Border → False, Linguistic → False];
testVars = DefPredicateVars[testPreds];
goalDim = 2;
```

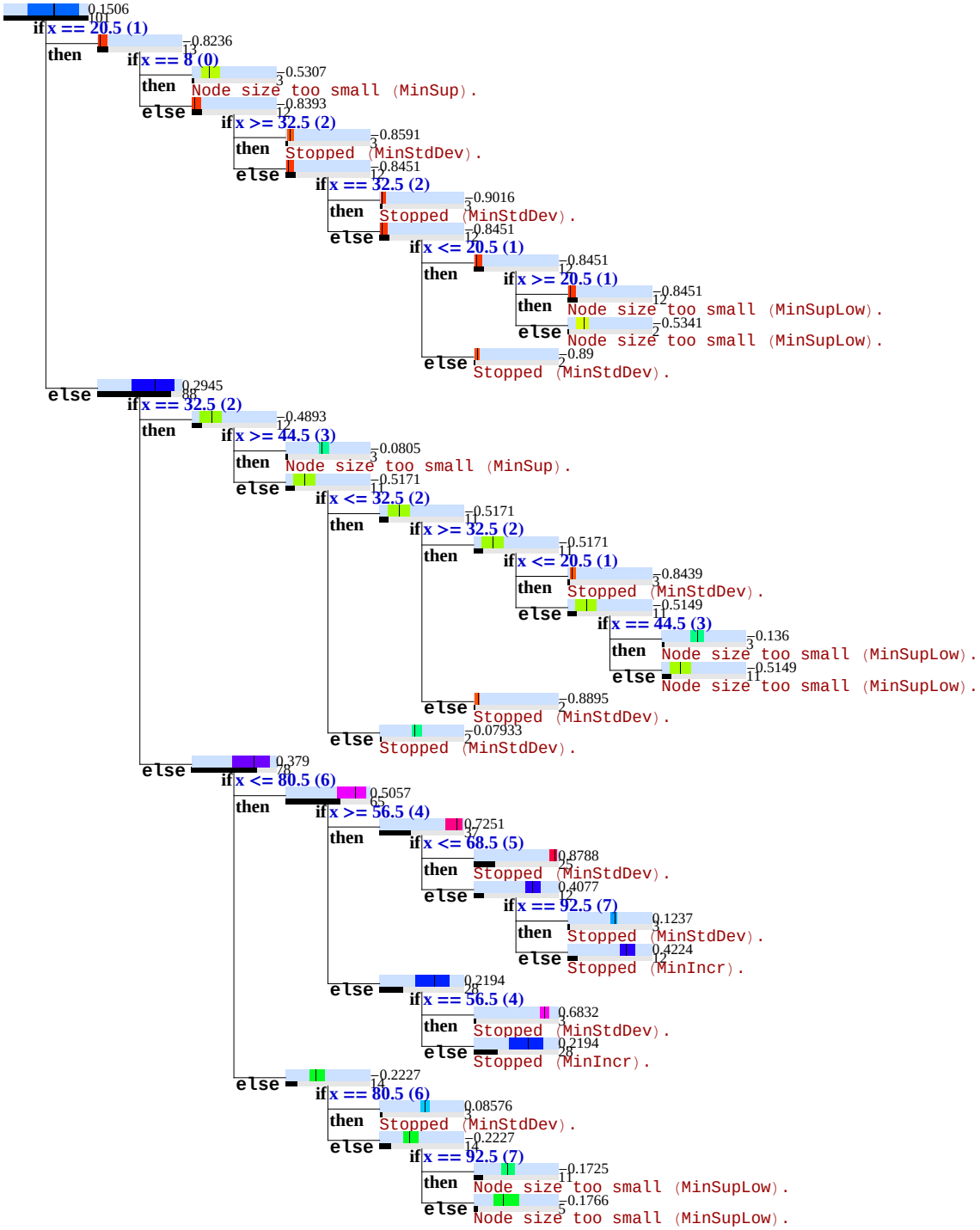
```
PlotFuzzySet[testPreds, ColorFunction → (CFJet[#] &)]
```




```
lirtTree = CreateLIRT[trainDataSet, testVars, goalDim];
```

The tree can be displayed with `PlotLIRT`:

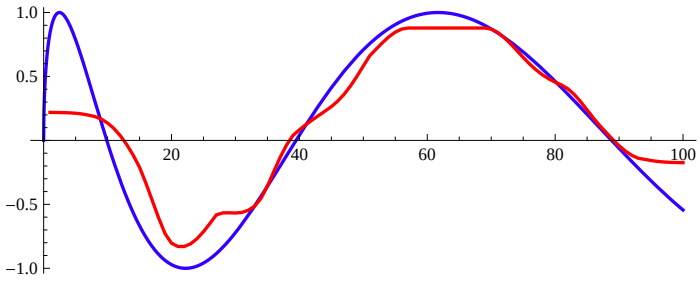
```
PlotLIRT[lirtTree]
```



■ Predictions

```
recallData = RecallLIRT[testDataSet, lirtTree];
```

```
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```




```
MLFMeanSquaredError[recallData, MLFGetData[testDataSet, All, 2]]
```

```
0.0497781
```

■ Using a Rule Base

■ Compute a rule set

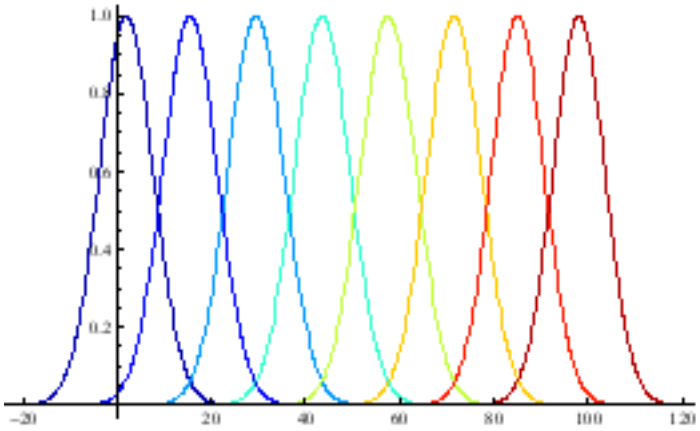
To find a set of independent rules for the given approximation problem, we will now apply the FS-FOIL rule-induction method.

```
Options[CreatePartition]
```

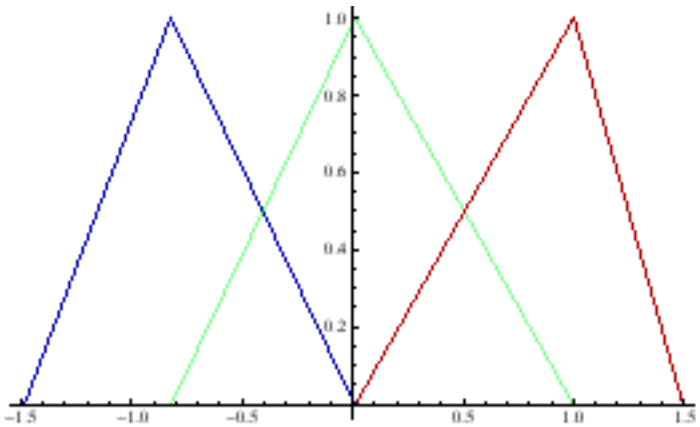
```
{NoOfSets → 5, Overlap → 0.2, Width → 1., PredType → ALL,
 SetType → PWL, MinSup → 0.025, Border → False, Linguistic → True,
 SeparateSets → False, SetNames → Automatic, CutPoints → Automatic,
 Minimum → Automatic, Maximum → Automatic, Confidence → 0.02, Labels → {},
 MaxLengthOfCombinations → 1, CascadingCombinations → False, OutputColumn → 1}
```

```
testPreds = CreatePredicates[trainDataSet, 1, 8,
    Overlap → 1.0, Border → True, SetType -> "EXP", Linguistic -> False];
goalPreds = CreatePredicates[trainDataSet, 2, 3, Overlap → 1.0,
    Border → True, PredType -> "IS"];
testVars = DefPredicateVars[testPreds];
goalVars = DefPredicateVars[goalPreds];
```

```
PlotFuzzySet[testPreds, ColorFunction → (CFJet[#] &)]
```



```
PlotFuzzySet[goalPreds, ColorFunction → (CFJet[#] &)]
```



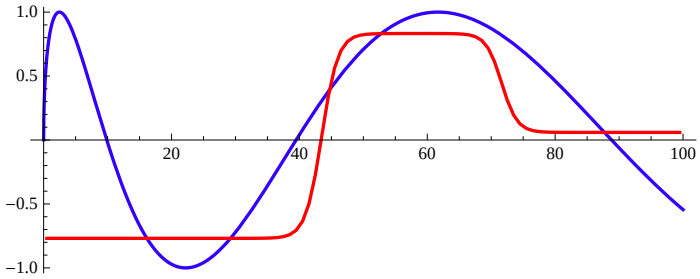

```
foilRules = CreateFOIL[trainDataSet, testVars, goalVars, MinConf → 0.6, Logic → LogicL];

PrintFOILRules[foilRules, Info → False]
```

Class	{}	Condition
y_Is_H	⇐	x == 57.5 (4)
y_Is_L	⇐	x <= 29.5 (2) && x >= 15.5 (1) && x >= 15.5 (1)
y_Is_M	⇐	x >= 85 (6)

- **Create a Sugeno controller from the rule base**

To apply the obtained set of rules to the test data, we can use the `MakeSugenoInference` command again.

```
recallData = Transpose[{
    testData[[All, 1]],
    MakeSugenoInference[foilRules, goalPreds, testData[[All, {1}]], Logic → LogicP]
   }];

plotOrig = Plot[f[x], {x, 0, range},
   DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
   Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```




```
Print["Mean Square Error: ",
  MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]]];
```

```
Mean Square Error: 0.301842
```

Using Numerical Optimized Rule Bases

We will now introduce RENO (short for **Re**gularized **N**umerical **O**ptimization). This is a new method for the generation and optimization of fuzzy systems from sample data. In more detail, RENO is concerned with the solution of the problem of how to identify the parameters describing fuzzy systems by minimizing the distance between the desired and the actually obtained output. It solves the approximation problem by not only combining (fuzzy) rule induction and numerical optimization, but also by applying regularization to stabilize the approximation process for noisy data. RENO takes care of both interpretability and high accuracy of the resulting fuzzy systems.

■ **Training a Sugeno Controller - Default Settings**

This example demonstrates the RENO approximation of f(x) with default option settings.

```
Options[CreateRENO]
```

```
{Dims → {1, 2}, Goal → 3, Controller → Sugeno, Membership → Automatic, NDims → Automatic,
 SetType → BSPLINE, LowerBounds → Automatic, UpperBounds → Automatic,
 Orders → Automatic, Lambda1 → Automatic, Lambda2 → Automatic}
```

We just run CreateRENO with our training data set, where the first dimension is used as data input and the second dimension as the goal to approximate. The type of fuzzy system which will be generated is a Sugeno controller. As the underlying sets of the fuzzy controller are tuned by the system, it is not necessary to initialize predicates in advance. By default, the input range is partitioned into five fuzzy sets of type B-spline of order two (which are equal to triangular membership functions).

```
renoController = CreateRENO[trainDataSet, Dims → {1}, Goal → 2];
```

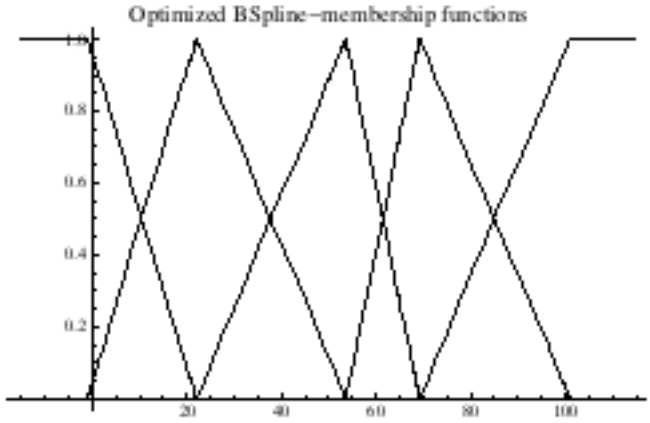
The resulting Sugeno controller consists of five rules, which are printed with PrintRENORules.

```
PrintRENORules[renoController]
```

```
x_Is_VL_R → 1.08073
x_Is_L_R → -1.20166
x_Is_M_R → 0.946268
x_Is_H_R → 0.967409
x_Is_VH_R → -0.598125
```

Next, we plot the (optimized) antecedents of the Sugeno rule base. Due to restrictions on the shape of the fuzzy partitions in the RENO tuning process, these membership functions fit quite well to be linguisticly interpretable.

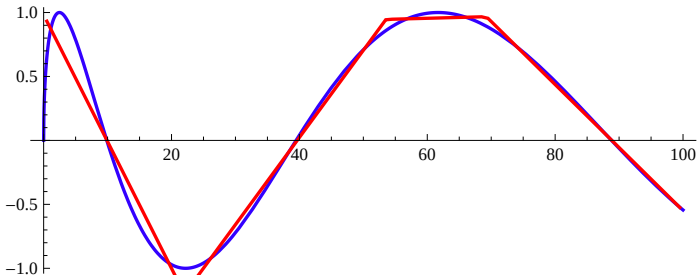
```
PlotRENOFuzzySet[renoController, PlotRange → {{-15, 115}, All},
 PlotLabel -> "Optimized BSpline-membership functions"]
```



To see how this controller approximates the desired goal function f(x), we evaluate it for the given test data set and plot the result. To apply the obtained set of rules on the test data, we use the `MakeSugenoInference` command.

```
recallData = Transpose[{
    testData[[All, 1]],
    MakeSugenoInference[renoController, testData[[All, {1}]]]
   }];
```

```
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```



To compare the result, we will compute the average square error of the approximation.

```
Print["Mean Squared Error: ",
 MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]]]
```

```
Mean Squared Error: 0.00816176
```

■ **Training a Takagi-Sugeno-Kang Controller - Default Settings**

We run our approximation once again, but now we would like to construct another type of controller - a Takagi-Sugeno-Kang (TSK) controller, sometimes also called a Sugeno controller of the first order. The output of the rules is not a constant numerical value, but a linear function of the inputs. We choose to compute a TSK controller by setting the option Controller to 1.

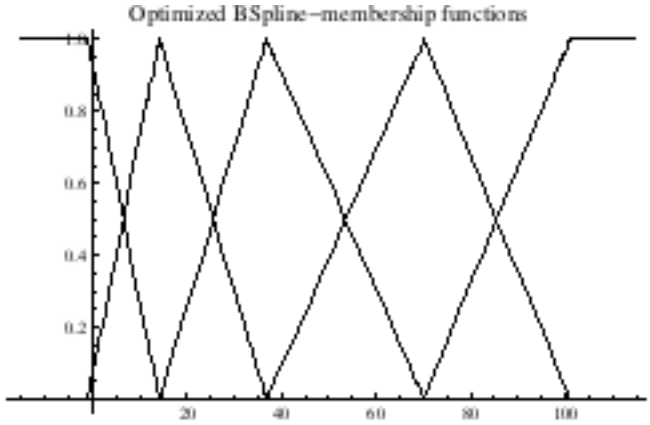
```
renoTSKController = CreateRENO[trainDataSet, Dims → {1}, Goal → 2, Controller → 1];
```

```
PrintRENORules[renoTSKController]
```

```
x_Is_VL_R → 0.674195 + Obj`RealDataScaled[0, 0.166228]
x_Is_L_R → -0.502895 + Obj`RealDataScaled[0, -0.0183125]
x_Is_M_R → -2.35125 + Obj`RealDataScaled[0, 0.058045]
x_Is_H_R → 1.06709 + Obj`RealDataScaled[0, -0.00249624]
x_Is_VH_R → 0.194793 + Obj`RealDataScaled[0, -0.0081632]
```

We plot the optimized fuzzy set partition of the TSK rule base.

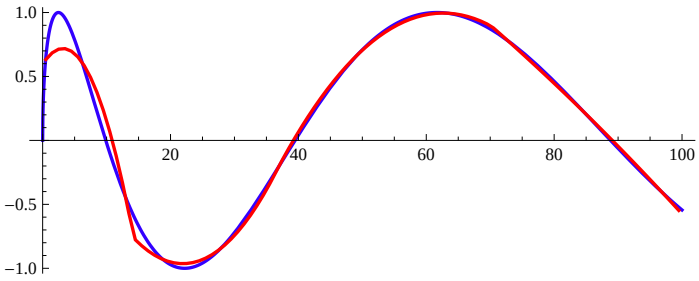
```
PlotRENOFuzzySet[renoTSKController, PlotRange → {{-15, 115}, All},
 PlotLabel -> "Optimized BSpline-membership functions"]
```



To see how this controller approximates the desired goal function f(x), we evaluate it for the given test data set and plot the result.

```
recallData = Transpose[{
    testData[[All, 1]],
     MakeSugenoInference[renoTSKController, testData[[All, {1}]]]
   }];
```

```
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```




```
Print["Mean Squares Error: ",
 MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]] ]
```

```
Mean Squares Error: 0.00415754
```

Compared to the Sugeno controller (see above), the approximation is more accurate due to the additional number of parameters to tune the consequences of the rules.

- **Training a Takagi-Sugeno-Kang Controller - Higher-order B-splines**

We compute a TSK controller using B-splines of order 3 (quadratic splines) as membership functions. We still use 5 fuzzy sets.

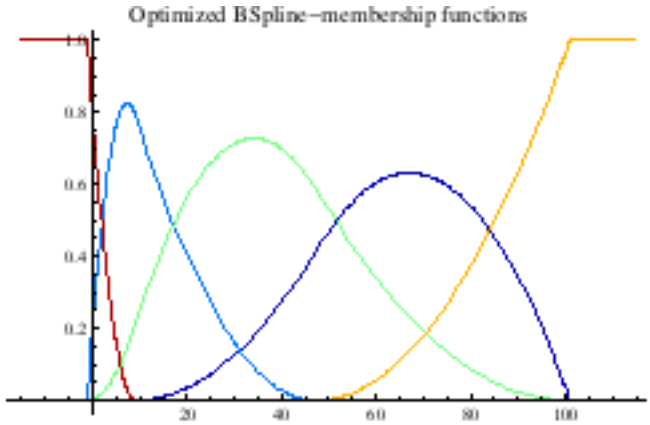
```
renoTSKController = CreateRENO[trainDataSet,
    Dims → {1}, Goal → 2, Controller → 1, NDims → {5}, Orders → {3}];
```

```
PrintRENORules[renoTSKController]
```

```
x_Is_VL_R → -0.107329 + Obj`RealDataScaled[0, 0.377855]
x_Is_L_R → 1.42188 + Obj`RealDataScaled[0, -0.0777228]
x_Is_M_R → -2.86443 + Obj`RealDataScaled[0, 0.0577606]
x_Is_H_R → 2.34242 + Obj`RealDataScaled[0, -0.0162083]
x_Is_VH_R → -0.974685 + Obj`RealDataScaled[0, 0.00360941]
```

We plot the optimized fuzzy set partition of the TSK rule base. As far as interpretability is concerned, higher-order B-splines suffer from the fact that they are not normalized.

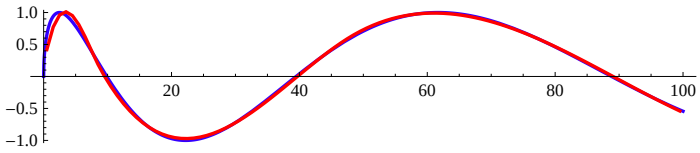
```
PlotRENOFuzzySet[renoTSKController, ColorFunction → (CFJet[#] &),
 PlotRange → {{-15, 115}, All}, PlotLabel -> "Optimized BSpline-membership functions"]
```



To see how this controller approximates the desired goal function f(x), we evaluate it for the given test data set and plot the result.

```
recallData = Transpose[{
    testData[[All, 1]],
     MakeSugenoInference[renoTSKController, testData[[All, {1}]]]
   }];
```

```
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.2]
```




```
Print["Mean Squared Error: ",
 MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]] ]
```

```
Mean Squared Error: 0.00137972
```

The quality of approximation is considerably improved compared to lower-order splines.

- **Training a Sugeno Controller - Piecewise Linear Membership Functions**

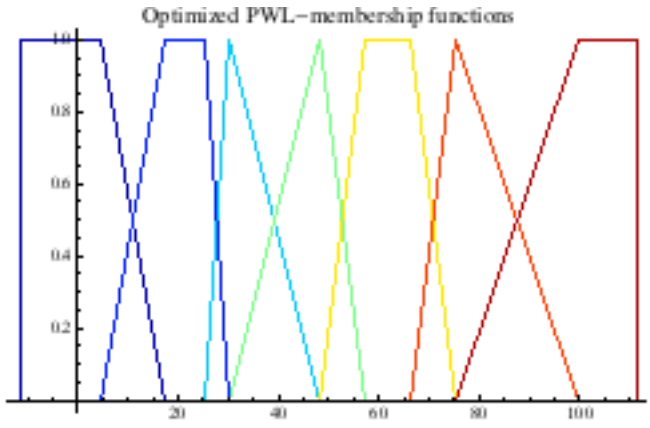
We compute a TSK controller using piecewise linear (trapezoidal) membership functions. We use 7 fuzzy sets.

```
renoControllerPWL = CreateRENO[trainDataSet,
    Dims → {1}, Goal → 2, Controller → 0, NDims → {7}, SetType -> "PWL"];
PrintRENORules[renoControllerPWL]
```

```
x_Is_VVL_R → 0.744963
x_Is_VL_R → -0.9662
x_Is_L_R → -0.717182
x_Is_M_R → 0.643141
x_Is_H_R → 0.986327
x_Is_VH_R → 0.689576
x_Is_VVH_R → -0.543462
```

We plot the optimized fuzzy antecedents.

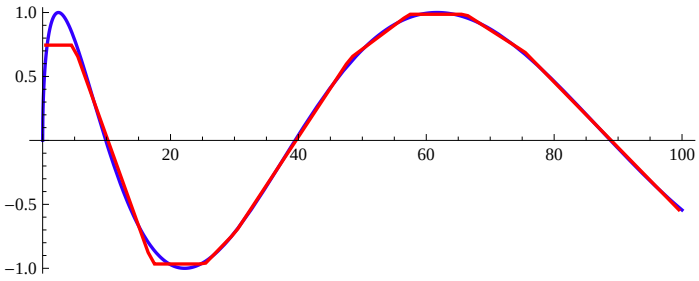
```
PlotRENOFuzzySet[renoControllerPWL,
 ColorFunction → (CFJet[#] &),
 PlotRange → {{-15, 115}, All},
 PlotLabel -> "Optimized PWL-membership functions"
]
```



To see how this controller approximates the desired goal function f(x), we evaluate it for the given test data set and plot the result.

```
recallData = Transpose[{
    testData[[All, 1]],
     MakeSugenoInference[renoControllerPWL, testData[[All, {1}]]]
   }];
```

```
plotOrig = Plot[f[x], {x, 0, range},
    DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
    Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```




```
Print["Average Squared Error: ",
  MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]]];
```

```
Average Squared Error: 0.00221156
```

The quality of the approximation is still very good.

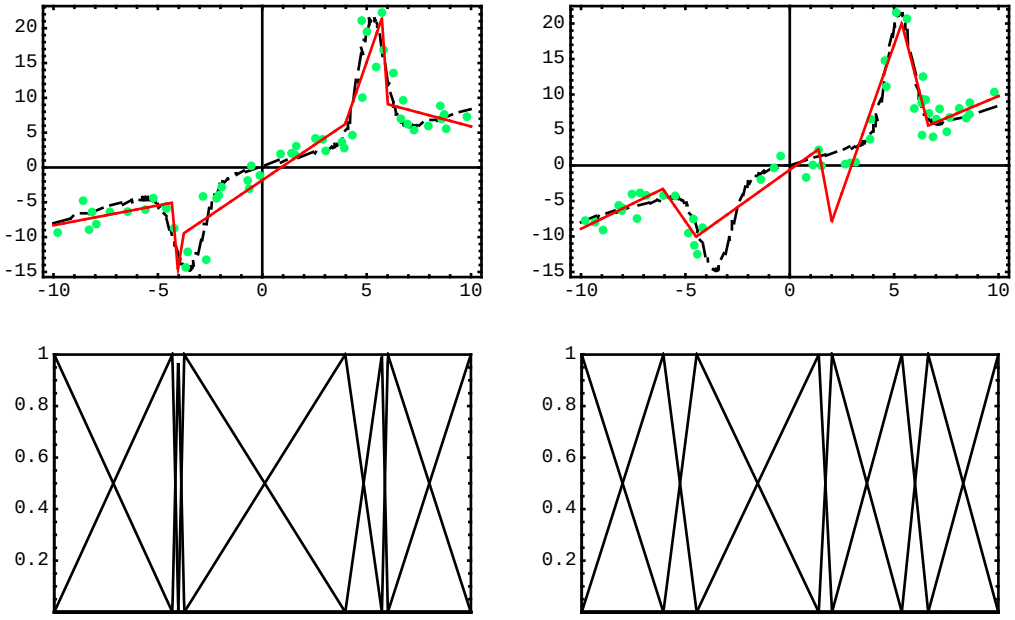
■ **A Short Excursion into Regularization**

RENO is concerned with the solution of the problem of how to identify the parameters describing a Sugeno or Takagi-Sugeno-Kang controller by minimizing the distance between the desired and the actually obtained output. On the one hand, the consequent parameters can be determined by solving a linear least-squares problem. On the other hand, the dependence of the functional to optimize the parameters describing the antecedent fuzzy sets is highly nonlinear. Assuming appropriate restrictions in terms of interpretability, there is a considerable number of constraints that have to be taken into account. So we have to solve a nonlinear constrained least-squares problem, which we denote by

$$F(\alpha, t) \rightarrow \min_{(\alpha, t)}$$

where α denotes the vector of consequent values and t the vector of parameters determining the antecedent fuzzy sets.

It turns out that solving the nonlinear least-squares problem is not stable, i.e., that small (measurement) errors in the data may lead to totally wrong results. The following figure demonstrates that for two sets of noisy data (green dots) generated from a spectral data function (dashed line). The red line represents the approximation as obtained by minimizing the functional F. The lower parts of the figure show the corresponding fuzzy sets.



Both the approximation and the optimized membership functions vary considerably for the two noisy data sets. The approximation shows undesirable peaks in the region of sparse data. These peaks are characteristic for the instabilities both in calculating the consequences as well as the parameters describing the fuzzy sets.

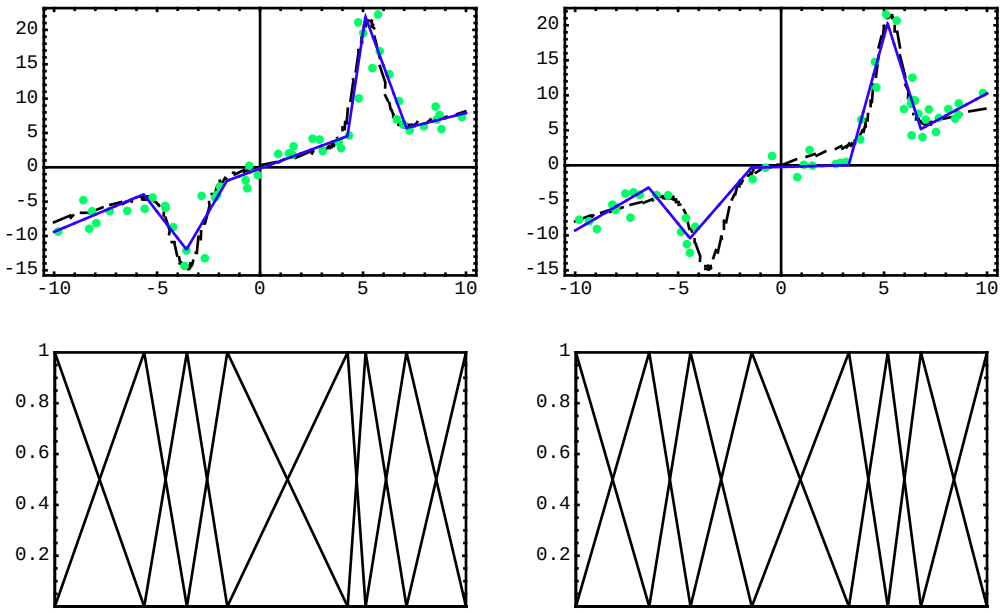
These difficulties are removed when using regularization, which stabilizes the optimization process. One method for regularizing is the classical Tikhonov regularization. It consists of adding a stabilization term to the functional F resulting in minimizing a functional F_T:

$$F_T(\alpha, t) \to \min_{(\alpha,t)} \qquad \text{where } F_T(\alpha, t) := F_T(\alpha, t) + \beta_1 T_1(\alpha) + \beta_2 T_2(t).$$

In general, $T_1(\alpha)$ is just the squared sum of the consequences α and $T_2(t)$ the squared sum of the difference of the antecedent parameters t to a given prior parameter sequence $\bar{t}$. RENO just chooses the prior automatically, most often uniform grid points. However, the choice of the regularization parameters β_1 and β_2 is of crucial importance for the solution process. RENO also makes a proposal for these parameters, but the parameters can also be specified by the user.

In general, regularization can be seen as trying to find a balance between stability and accuracy of the approximation. A large value of the regularization parameter produces a stable solution; however it may not adequately satisfy the original data. For a small value, we could expect to approximate the minimum of F well; however, the problem then becomes unstable.

The following figures show the effect of Tikhonov regularization for the two noisy data sets. Although the data sets are quite different, Tikhonov regularization succeeds in constructing stable approximations as well as stable and interpretable membership functions.



RENO also offers a second method for stabilizing the solution process which is called "smoothing". Here the special feature of smoothness of the approximation is enforced. Smoothing is only available for B-spline membership functions.

- **Training a Sugeno Controller - Tikhonov Regularization**

We compute a Sugeno controller using seven piecewise linear (trapezoidal) membership functions and adapted regularization parameters. Currently, only Tikhonov regularization is supported. We choose the regularization parameters adjusted to the problem size.

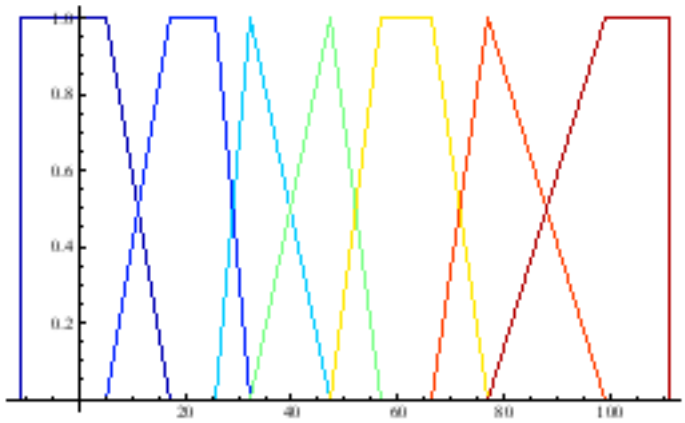
```
renoController = CreateRENO[trainDataSet, Dims → {1}, Goal → 2, Controller → 0,
    NDims → {7}, SetType -> "PWL", Lambda1 → {0.01}, Lambda2 → {0.0001}];
```

```
PrintRENORules[renoController]
```

```
x_Is_VVL_R → 0.74362
x_Is_VL_R → -0.96512
x_Is_L_R → -0.58492
x_Is_M_R → 0.593788
x_Is_H_R → 0.985354
x_Is_VH_R → 0.616017
x_Is_VVH_R → -0.519987
```

We plot the optimized fuzzy antecedents.

```
PlotRENOFuzzySet[renoController,
 ColorFunction → (CFJet[#] &), PlotRange → {{-15, 115}, All}]
```

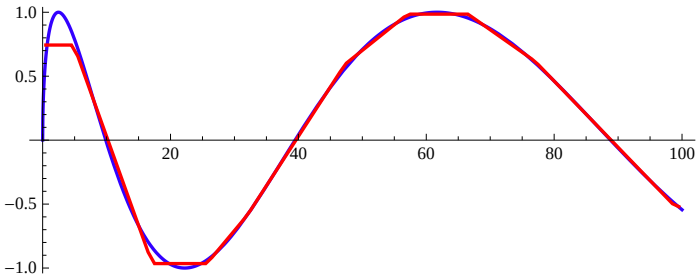


To see how this controller approximates the desired goal function f(x), we evaluate it for the given test data set and plot the result.

```
 MakeSugenoInference[renoController, testData];
```

```
recallData = Transpose[{
    testData[[All, 1]],
     MakeSugenoInference[renoController, testData[[All, {1}]]]
   }];
```

```
plotOrig = Plot[f[x], {x, 0, range},
   DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
   Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.4]
```




```
Print["Average Squared Error: ",
 MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]] ]
```

```
Average Squared Error: 0.00224142
```

We do the calculations once again, but now with regularization parameters which are far too large to demonstrate the effect of inappropriately chosen parameters on the approximation.

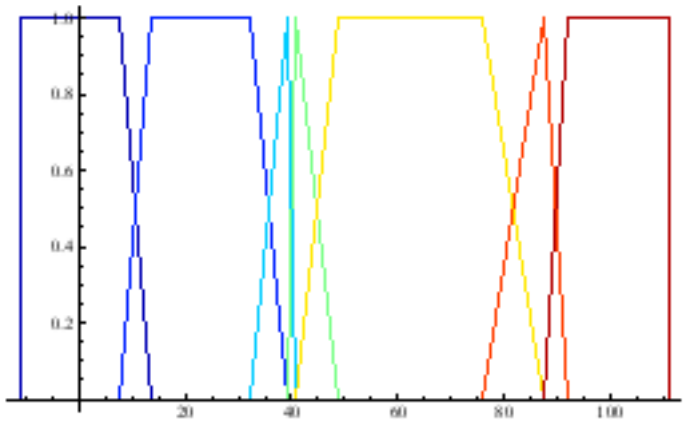
```
renoController = CreateRENO[trainDataSet, Dims → {1}, Goal → 2,
   Controller → 0, NDims → {7}, SetType -> "PWL", Lambda1 → {10}, Lambda2 → {0.1}];
```

```
PrintRENORules[renoController]
```

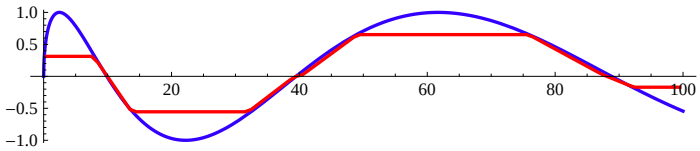
```
x_Is_VVL_R → 0.312327
x_Is_VL_R → -0.554458
x_Is_L_R → -0.00899562
x_Is_M_R → 0.0265113
x_Is_H_R → 0.652361
x_Is_VH_R → 0.0316439
x_Is_VVH_R → -0.17003
```

We plot the optimized fuzzy antecedents.

```
PlotRENOFuzzySet[renoController,
 ColorFunction → (CFJet[#] &), PlotRange → {{-15, 115}, All}]
```




```
recallData = Transpose[{
    testData[[All, 1]],
     MakeSugenoInference[renoController, testData[[All, {1}]]]
   }];
plotOrig = Plot[f[x], {x, 0, range},
   DisplayFunction → Identity, PlotStyle → {Thickness[0.005], Hue[0.7]}];
plotRecall = ListPlot[recallData, DisplayFunction -> Identity,
   Joined → True, PlotStyle → {Thickness[0.005], Hue[0]}];
Show[plotOrig, Graphics@plotRecall, DisplayFunction → $DisplayFunction,
 AspectRatio → 0.2]
```




```
Print["Average Squared Error: ",
 MLFMeanSquaredError[recallData[[All, 2]], testData[[All, 2]]]]
```

```
Average Squared Error: 0.0598127
```

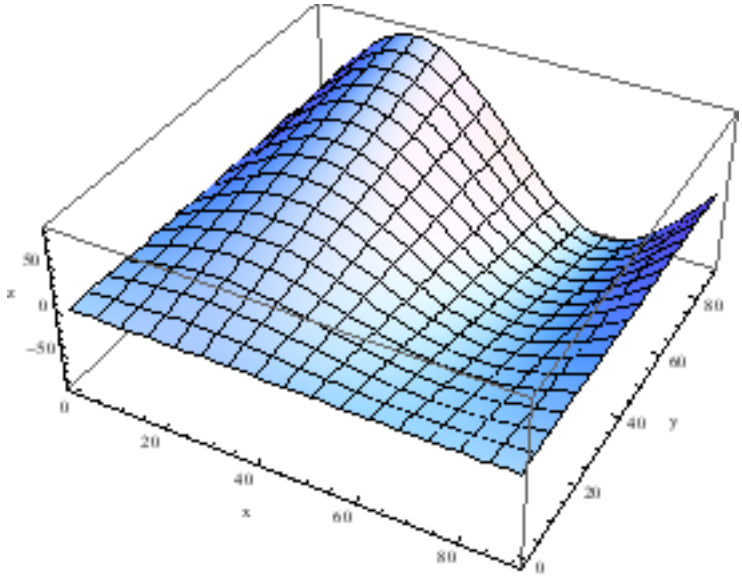
A more Complex, Two-dimensional Example

In this example, we will show how a two-dimensional function can be approximated using various methods. The approach is similar to the one-dimensional example and we will therefore skip the details.

■ Set up Data

First we have to define the goal function f2 and its range.

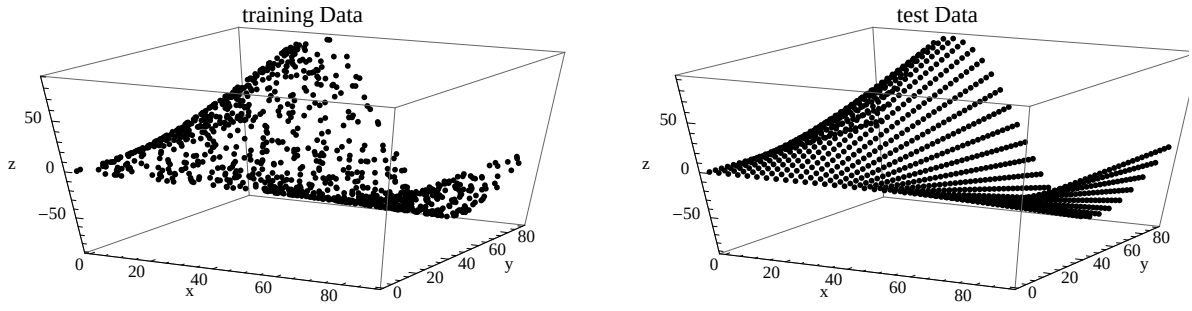
```
range = 90;
f2[x_, y_] := N@(Sin[x / range * 2 * Pi] * y);
Plot3D[f2[x, y], {x, 0, range}, {y, 0, range},
 AxesLabel → {"x", "y", "z"}, ImageSize → 380]
```



Now we can store a number of sample points in a matrix which is later used for training, and another one for testing.

```
samples = 1000;
headers = {"x", "y", "z"};
SeedRandom[1];
trainData = Table[{x = Random[] * range, y = Random[] * range, f2[x, y]}, {samples}];
trainDataSet = Def["tagTrainDataSet2", DataSet[trainData, headers]];
samplesTest = 31;
testData =
  Flatten[Table[Table[{x = i * range / samplesTest, y = j * range / samplesTest, f2[x, y]},
     {i, 0, samplesTest}], {j, 0, samplesTest}], 1];
```

```
GraphicsGrid[
 {{
   Graphics3D[Map[Point, trainData],
    AspectRatio → 0.5,
    Axes -> True,
    AxesEdge → {{-1, -1}, {+1, -1}, {-1, -1}},
    AxesLabel → headers,
    PlotLabel → "training Data"
   ],
   Graphics3D[Map[Point, testData],
    AspectRatio → 0.5,
    Axes → True,
    AxesEdge → {{-1, -1}, {+1, -1}, {-1, -1}},
    AxesLabel → headers,
    PlotLabel → "test Data"
   ]
  }},
 ImageSize → 640
]
```



■ Using a Decision Tree

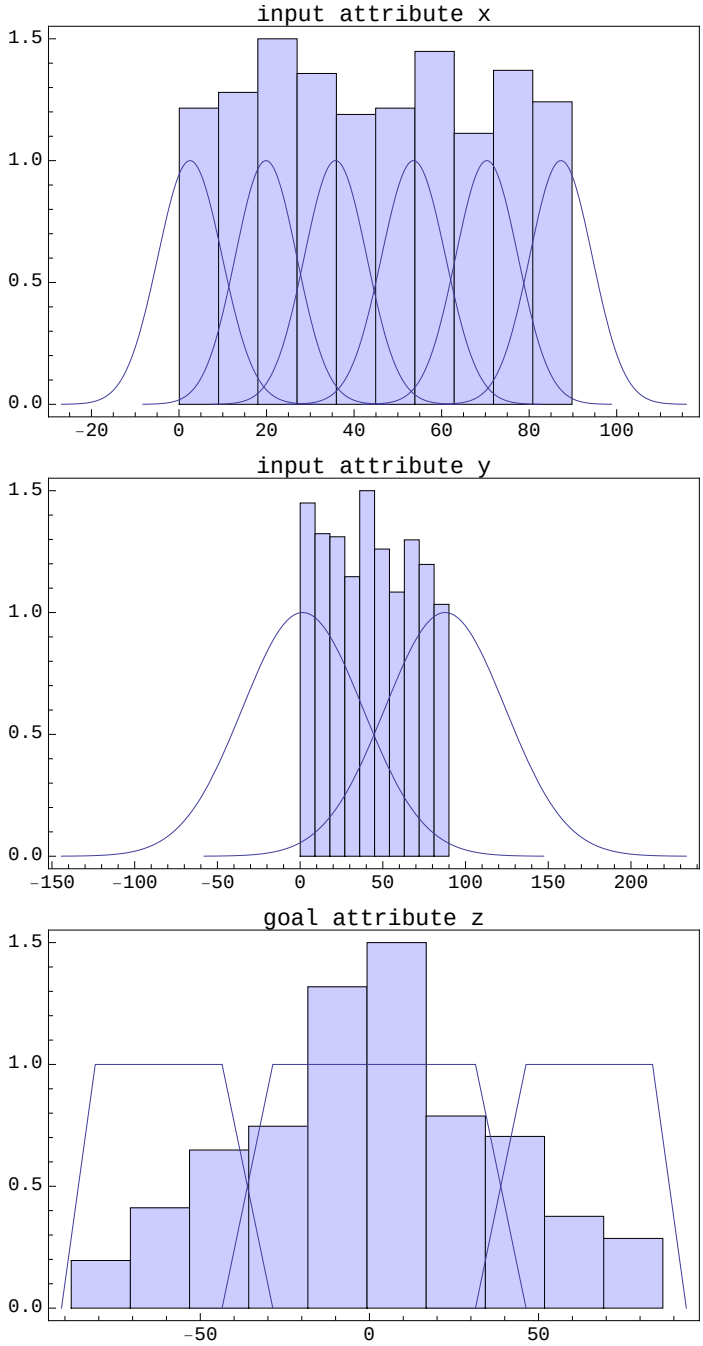
■ Create a decision tree using ID3

Since the function varies more in the *x* direction, we define more predicates for that dimension (1).

```
testPredsSep = {
    CreatePredicates[trainDataSet, 1, 6, Width → 1.0, Border → True, SetType -> "EXP"],
    CreatePredicates[trainDataSet, 2, 2, Overlap → 0.7, Border → True, SetType -> "EXP"]
  };
testPreds = Flatten[testPredsSep, 1];
goalPreds = CreatePredicates[trainDataSet, 3, 3,
    Width → 1.0, Border → True, PredType -> "ISEX", SetType -> "PWL"];
testVars = DefPredicateVars[testPreds];
goalVars = DefPredicateVars[goalPreds];
```

Let us have a look at the partitions which we just created and at the underlying data distributions of each dimension

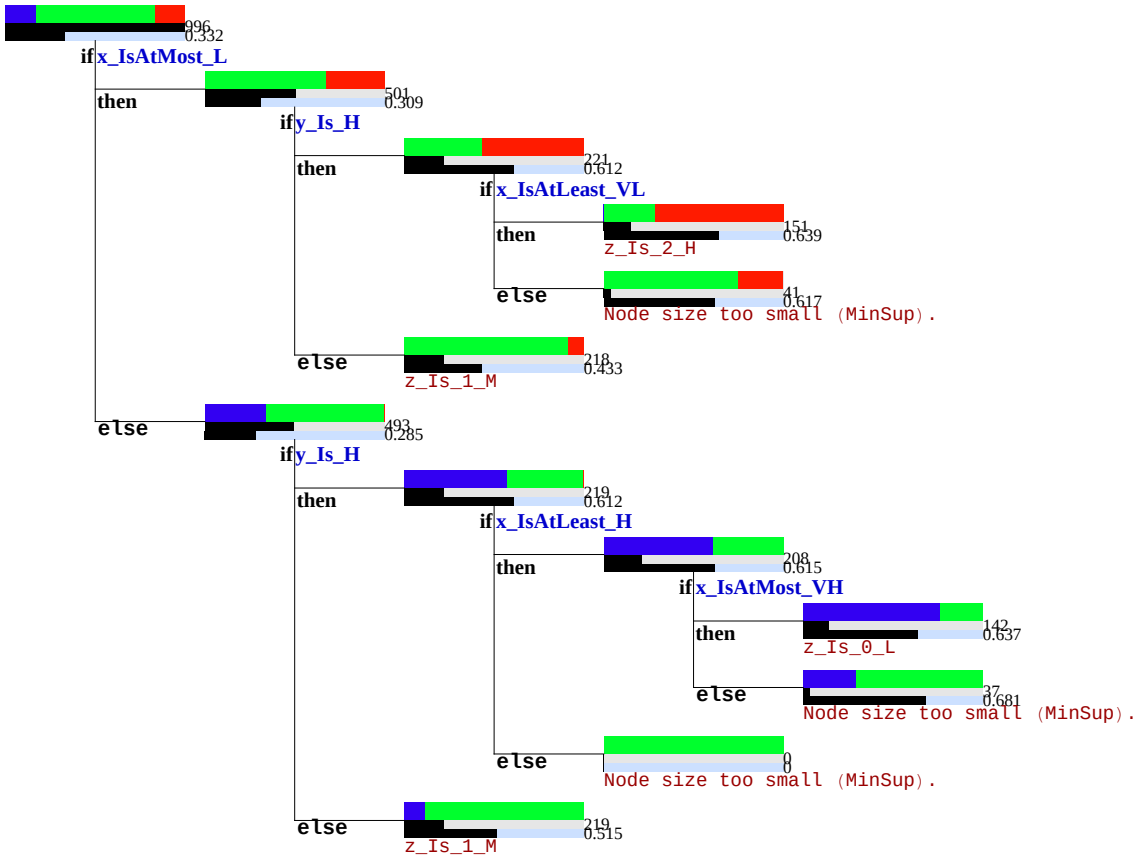
```
PlotFuzzySetHistogram[testPredsSep[[1]],
  trainDataSet, 1, PlotLabel -> "input attribute x"];
PlotFuzzySetHistogram[testPredsSep[[2]], trainDataSet,
  2, PlotLabel -> "input attribute y"];
PlotFuzzySetHistogram[goalPreds, trainDataSet, 3, PlotLabel -> "goal attribute z"];
```



If you use non-PWL membership functions, it might be necessary to increase the minimum entropy gain to 0.05.

```
id3tree = CreateID3[trainDataSet, testVars, goalVars,
    MinConf → 0.8, MinEnt → 0.01, MinIncr → 0.01, Logic → LogicL, MaxLevel → 4];
Show[Graphics[PlotClassLegend[goalVars], AspectRatio → 0.1], PlotRange → All]
```

```
PlotID3[id3tree, ImageSize → 600]
```



- **Create a Sugeno controller from the decision tree**

  Apply the controller to the test data set

```
recallData = MapThread[Append[#1, #2] &,
    {testData[[All, {1, 2}]],
     MakeSugenoInference[id3tree, goalPreds,
      testData[[All, {1, 2}]], Type -> "PROB", Logic → LogicP]
    }];
```

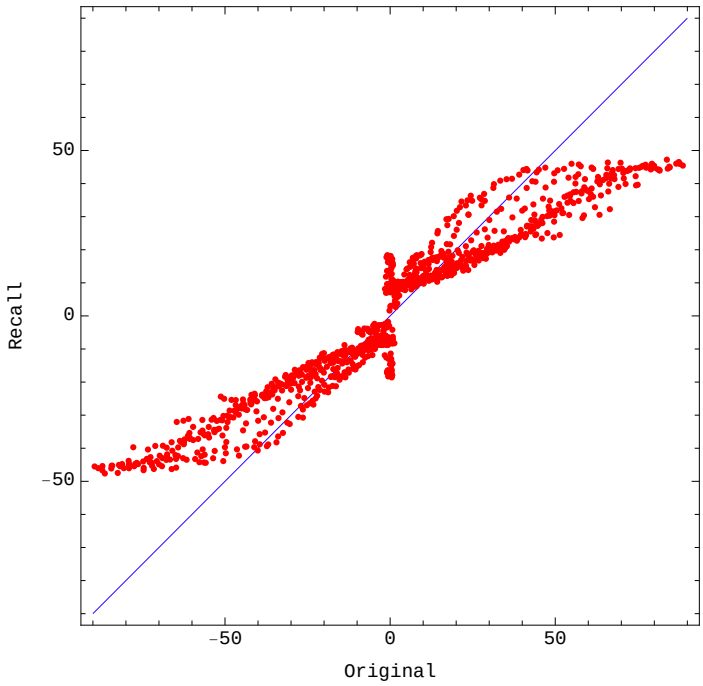
  The correlation coefficient and the mean squared error are good measure for the quality of the approximation

```
Print["Correlation Coefficient: ",
  NumericCorrelation[recallData[[All, 3]], testData[[All, 3]]]];
r = Max[testData[[All, 3]]] - Min[testData[[All, 3]]];
Print["Normalized Mean Squared Error: ",
  MLFNormalizedMeanSquaredError[recallData[[All, 3]], testData[[All, 3]]]
 ];
```

```
Correlation Coefficient: 0.9597

Normalized Mean Squared Error: 0.0243323
```

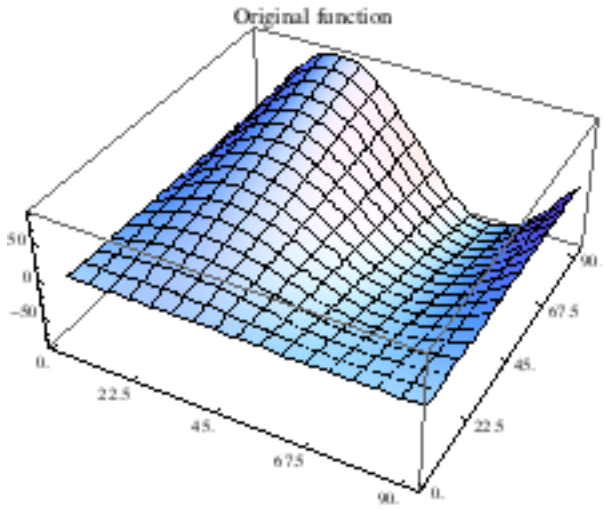
  We can also inspect visually how well the predicted values match the original values

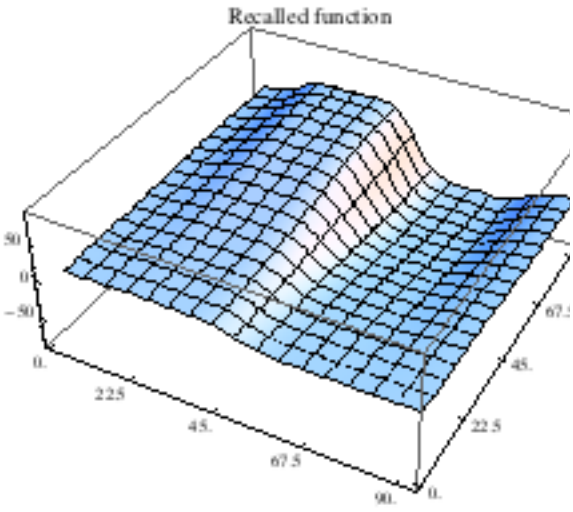
```
PlotInputRecall[testData[[All, 3]], recallData[[All, 3]], PointSize → 0.01]
```

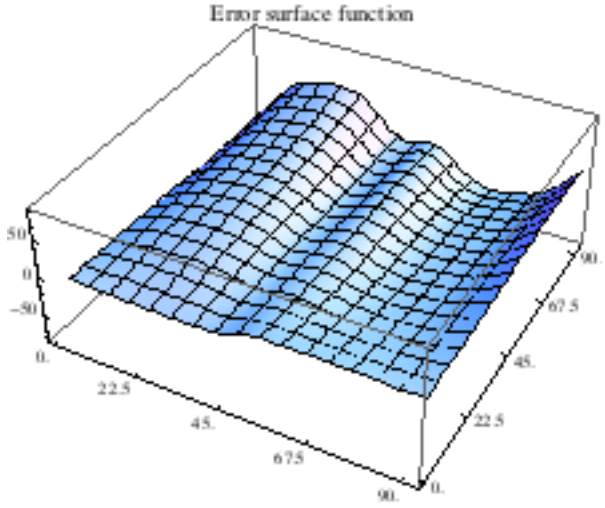


A three-dimensional visualization of the original values and the approximated function:

```
testDataMatrix = Partition[testData[[All, 3]], samplesTest + 1];
recallDataMatrix = Partition[recallData[[All, 3]], samplesTest + 1];
curTicks = {
    Table[{x * samplesTest / range, x}, {x, 0, range, range / 4.0}],
    Table[{x * samplesTest / range, x}, {x, 0, range, range / 4.0}],
    Automatic
   };
GraphicsGrid[
 {
  {
   ListPlot3D[
    testDataMatrix,
    PlotRange → {Automatic, Automatic, {-range, range}},
    Ticks → curTicks, ImageSize → 380,
    PlotLabel → "Original function"
   ],
   ListPlot3D[
    recallDataMatrix,
    PlotRange → {Automatic, Automatic, {-range, range}},
    Ticks → curTicks, ImageSize → 380,
    PlotLabel → "Recalled function"
   ]
  }, {
   ListPlot3D[
    testDataMatrix - recallDataMatrix,
    PlotRange → {Automatic, Automatic, {-range, range}},
    Ticks -> curTicks, Axes → True, ImageSize → 380,
    PlotLabel → "Error surface function"
   ],
   Graphics[{}]
  }
 },
 ImageSize → 640
]
```







The following function summarizes the above evaluation for later use:

```
MyEvaluation[test_, recall_] := Module[
    {testDataMatrix, recallDataMatrix, cc, nmse, curTicks},

    cc = NumericCorrelation[recall[[All, 3]], test[[All, 3]]];
    nmse = MLFNormalizedMeanSquaredError[recall[[All, 3]], test[[All, 3]]];
    testDataMatrix = Partition[test[[All, 3]], samplesTest + 1];
    recallDataMatrix = Partition[recall[[All, 3]], samplesTest + 1];
    curTicks = {
      Table[{x * samplesTest / range, x}, {x, 0, range, range / 4.0}],
      Table[{x * samplesTest / range, x}, {x, 0, range, range / 4.0}],
      Automatic
     };
    Labeled[
     "Correlation Coefficient = " <> ToString[SetPrecision[cc, 3]] <>
      ",   normalized MSE = " <> ToString[SetPrecision[nmse, 3]], GraphicsGrid[
      {
       {
        PlotInputRecall[
         test[[All, 3]],
         recall[[All, 3]],
         PointSize → 0.01
        ],
        ListPlot3D[
         testDataMatrix - recallDataMatrix,
         PlotRange → {Automatic, Automatic, {-range, range}},
         Ticks -> curTicks, Axes → True,
         PlotLabel → "Error surface function"
        ]
       }, {
        ListPlot3D[
         testDataMatrix,
         PlotRange → {Automatic, Automatic, {-range, range}},
         Ticks → curTicks,
         PlotLabel → "Original function"
        ],
        ListPlot3D[
         recallDataMatrix,
         PlotRange → {Automatic, Automatic, {-range, range}},
         Ticks → curTicks,
         PlotLabel → "Recalled function"
        ]
       }
      },
      ImageSize → 640
     ]
    ]
   ];
```

Using a Regression Tree

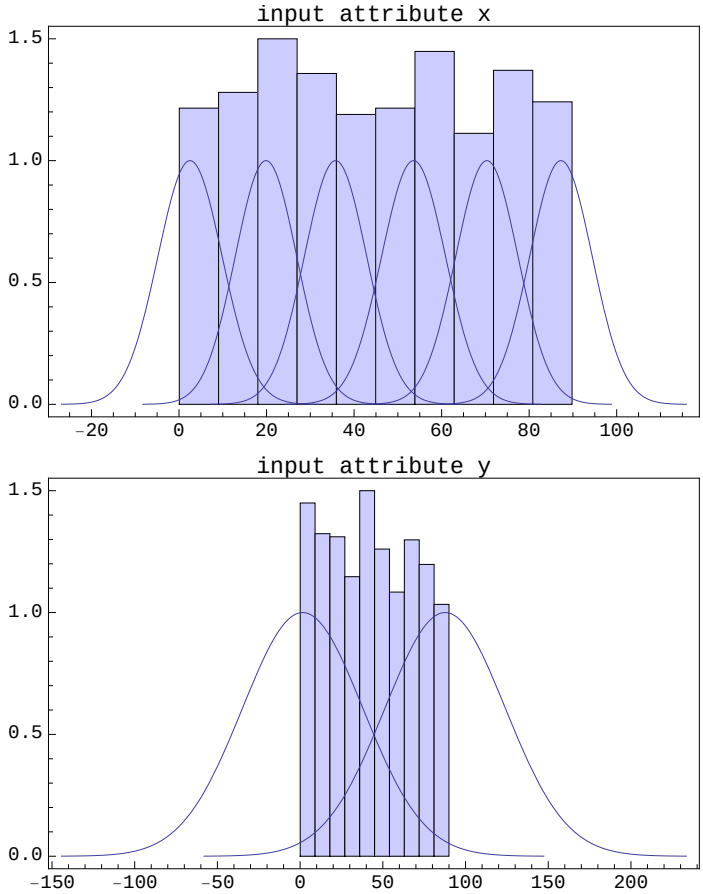
Create a regression tree using FS-LIRT

Since the function varies more in the *x* direction, we define more predicates for that dimension (1).

```
testPredsSep = {
    CreatePredicates[trainDataSet, 1, 6, Width → 1.0, Border → True, SetType -> "EXP"],
    CreatePredicates[trainDataSet, 2, 2, Overlap → 0.7, Border → True, SetType -> "EXP"]
  };
testPreds = Flatten[testPredsSep, 1];
testVars = DefPredicateVars[testPreds];
```

Let us have a look at the partitions which we just created and at the underlying data distributions of each dimension

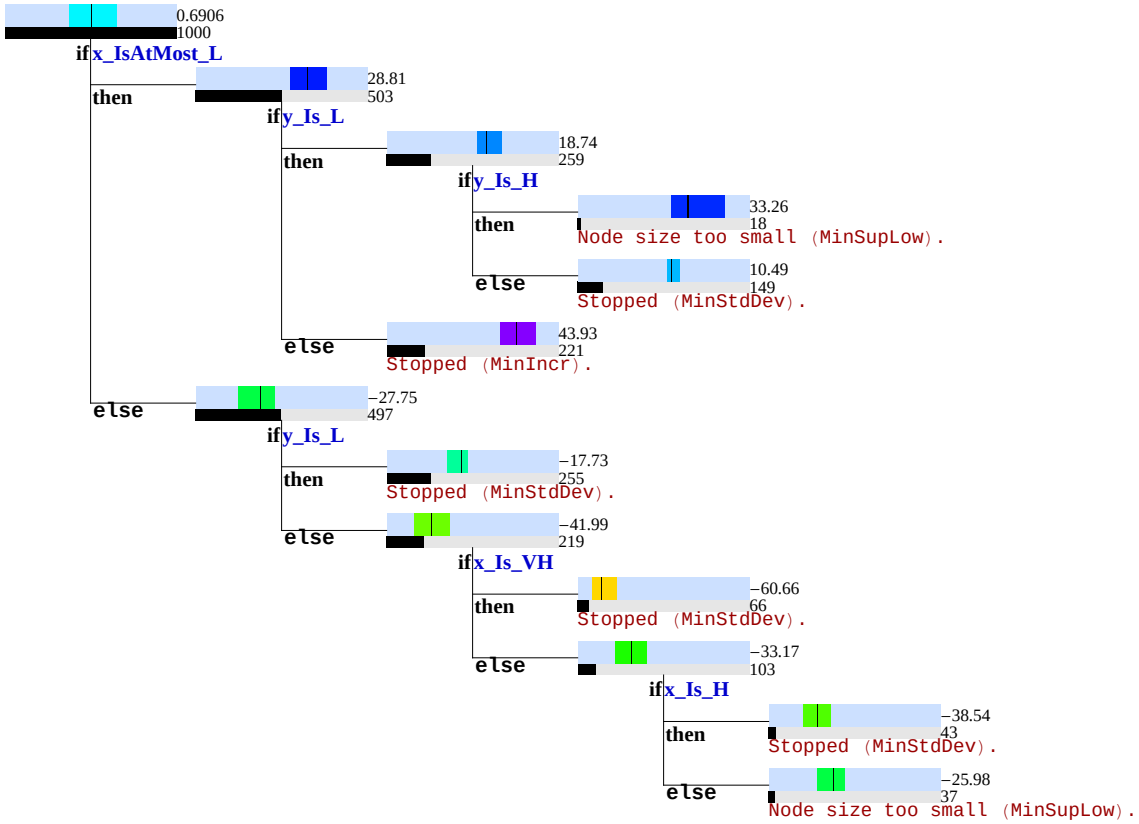
```
PlotFuzzySetHistogram[testPredsSep[[1]],
  trainDataSet, 1, PlotLabel -> "input attribute x"];
PlotFuzzySetHistogram[testPredsSep[[2]], trainDataSet,
  2, PlotLabel -> "input attribute y"];
```



If you use non-PWL membership functions, it might be necessary to adjust the parameters.

```
lirt =
  CreateLIRT[trainDataSet, testVars, 3, MinIncr → 0.01, Logic → LogicL, MaxLevel → 4];
```

```
PlotLIRT[lirt]
```



- **Make predictions**

  Apply the tree to the test data set:

```
recallData = RecallLIRT[testData, lirt];
```

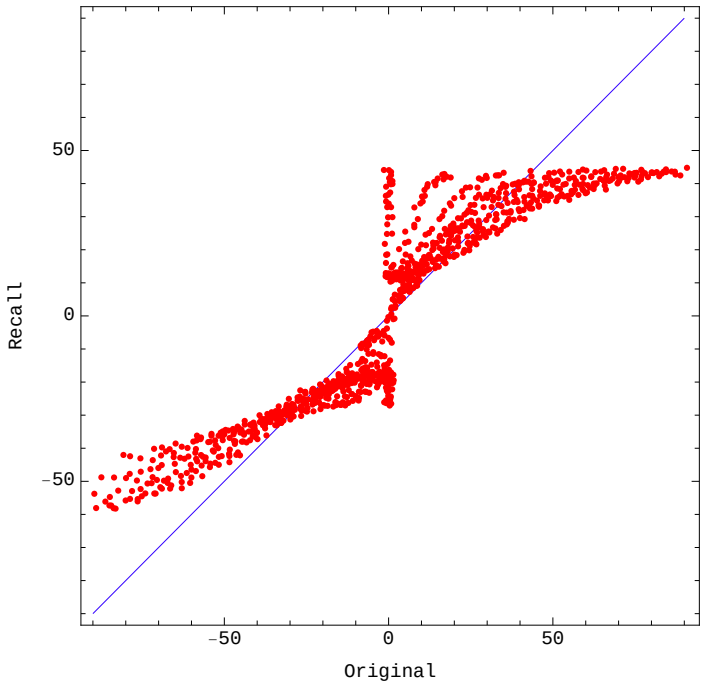
  The correlation coefficient and the mean squared error are good measure for the quality of the approximation:

```
Print["Correlation Coefficient: ",
  NumericCorrelation[recallData, testData[[All, 3]]]];
r = Max[testData[[All, 3]]] - Min[testData[[All, 3]]];
Print["Normalized Mean Squared Error: ",
  MLFNormalizedMeanSquaredError[recallData, testData[[All, 3]]]
 ];
```

```
Correlation Coefficient: 0.917173

Normalized Mean Squared Error: 0.0214802
```

  We can also inspect visually how well the predicted values match the original values

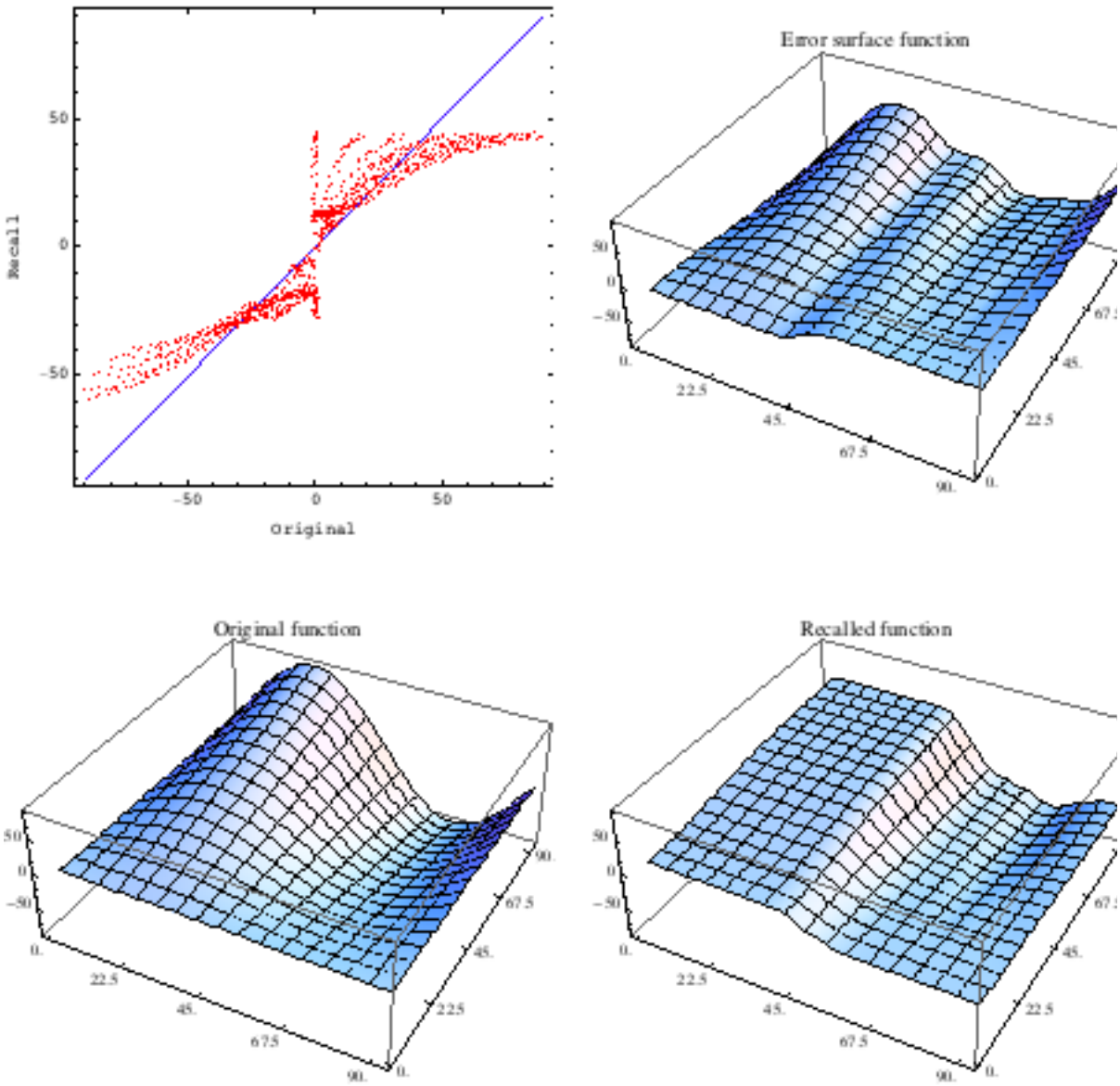
```
PlotInputRecall[testData[[All, 3]], recallData, PointSize → 0.01]
```



A three-dimensional visualization of the original values and the approximated function:

```
MyEvaluation[testData, MapThread[Append[Most[#1], #2] &, {testData, recallData}]]
```

```
Correlation Coefficient = 0.917,   normalized MSE = 0.0215
```



Using a Rule Base

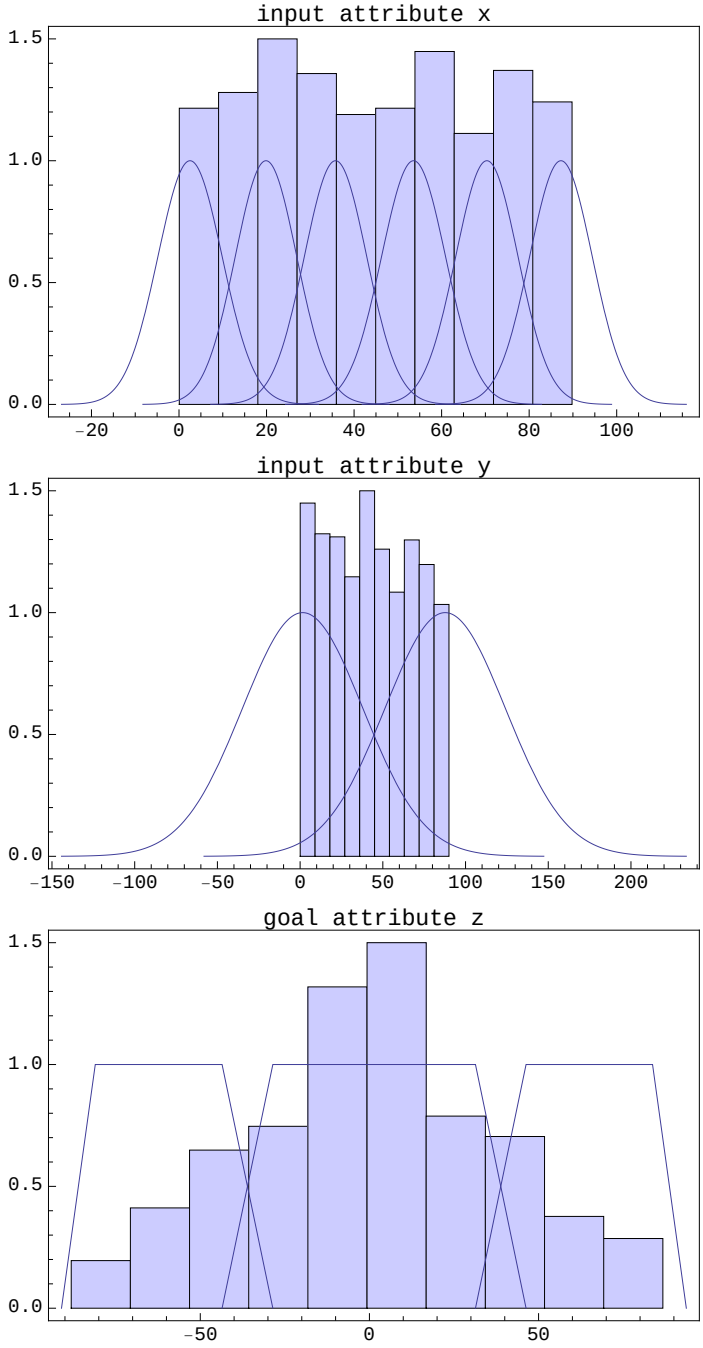
Create a rule base using FS-FOIL

Since the function varies more in the *x* direction, we define more predicates for that dimension (1).

```
testPredsSep = {
    CreatePredicates[trainDataSet, 1, 6, Width → 1.0, Border → True, SetType -> "EXP"],
    CreatePredicates[trainDataSet, 2, 2, Overlap → 0.7, Border → True, SetType -> "EXP"]
   };
testPreds = Flatten[testPredsSep, 1];
goalPreds = CreatePredicates[trainDataSet, 3, 3,
    Width → 1.0, Border → True, PredType -> "ISEX", SetType -> "PWL"];
testVars = DefPredicateVars[testPreds];
goalVars = DefPredicateVars[goalPreds];
```

Let us have a look at the partitions which we just created, and display the underlying data distributions of each dimension

```
PlotFuzzySetHistogram[testPredsSep[[1]],
  trainDataSet, 1, PlotLabel -> "input attribute x"];
PlotFuzzySetHistogram[testPredsSep[[2]], trainDataSet,
  2, PlotLabel -> "input attribute y"];
PlotFuzzySetHistogram[goalPreds, trainDataSet, 3, PlotLabel -> "goal attribute z"];
```




```
Options@CreateFOIL
```

```
{Logic → Automatic, MinDetail → 0.1, MinSup → 0.1, MinConf → 0.8,
 MinConfFallBack → None, MaxLevel → 10, MaxNegIter → 5, MaxIter → 500, ExpLevel → 10}
```

```
foilRules = CreateFOIL[trainDataSet, testVars,
    goalVars, MinConf → 0.6, MinSup → 0.05, Logic → LogicP];
```

```
PrintFOILRules[foilRules, Info → False]
```

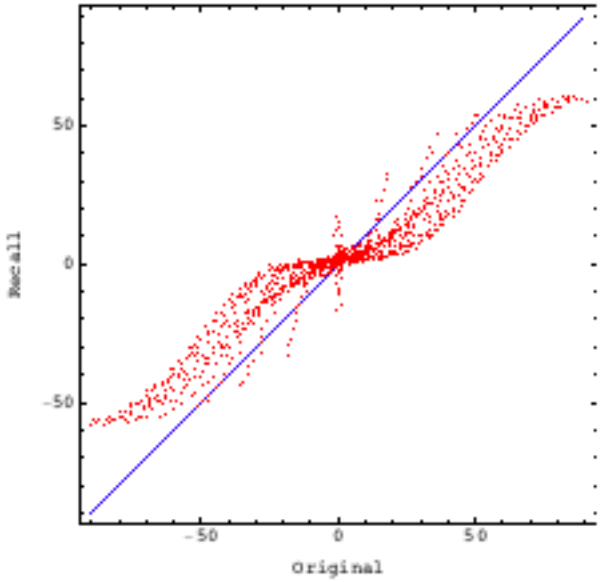
Class	{}	Condition
z_Is_0_L	⇐	x_Is_VH && y_Is_H
z_Is_1_M	⇐	y_Is_L
z_Is_2_H	⇐	x_Is_VL && y_Is_H

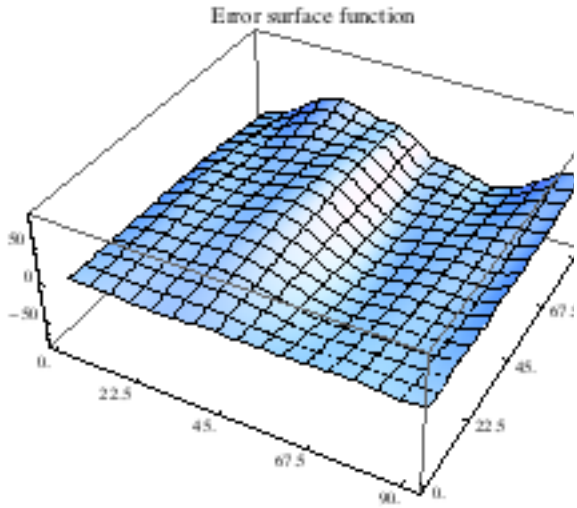
- **Create a Sugeno controller from the rule base**

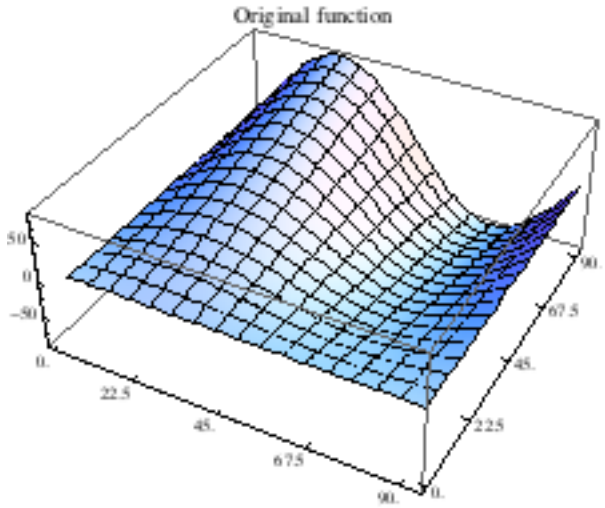
Apply the controller to the test data set:

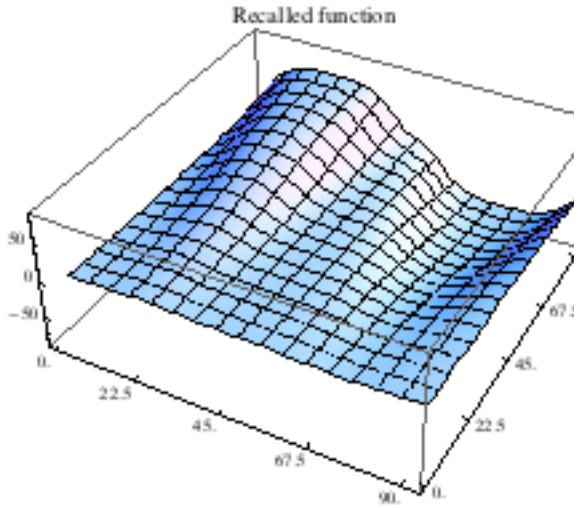
```
recallData = MapThread[Append[#1, #2] &,
   {testData[[All, {1, 2}]],
    MakeSugenoInference[foilRules, goalPreds, testData[[All, {1, 2}]], Logic → LogicP]
   }];
MyEvaluation[testData, recallData]
```

```
Correlation Coefficient = 0.967,   normalized MSE = 0.0128
```









Using Numerical Optimized Rule Bases

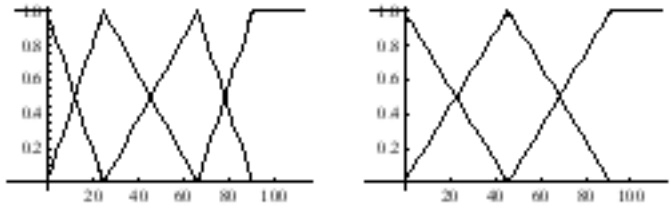
Trainig a Sugeno controller

We decide to use linear (order 2) B-spline membership functions, 4 for input attribute x and 3 for input attribute y.

```
reno = CreateRENO[trainDataSet, Dims → {1, 2}, Goal → 3, NDims → {4, 3},
    SetType -> "BSPLINE", Lambda1 → {0.001, 0.001}, Lambda2 → {0.00001, 0.001}];
PrintRules[reno]
```

```
x_Is_VL_R && y_Is_L_R → -0.168174
x_Is_L_R && y_Is_L_R → -0.745303
x_Is_H_R && y_Is_L_R → 0.891684
x_Is_VH_R && y_Is_L_R → -0.0461526
x_Is_VL_R && y_Is_M_R → 5.95835
x_Is_L_R && y_Is_M_R → 52.0867
x_Is_H_R && y_Is_M_R → -53.5652
x_Is_VH_R && y_Is_M_R → -5.16398
x_Is_VL_R && y_Is_H_R → 13.8075
x_Is_L_R && y_Is_H_R → 105.294
x_Is_H_R && y_Is_H_R → -108.694
x_Is_VH_R && y_Is_H_R → -11.4357
```

```
PlotRENOFuzzySet[reno, PlotRange → {{-15, 115}, All}]
```

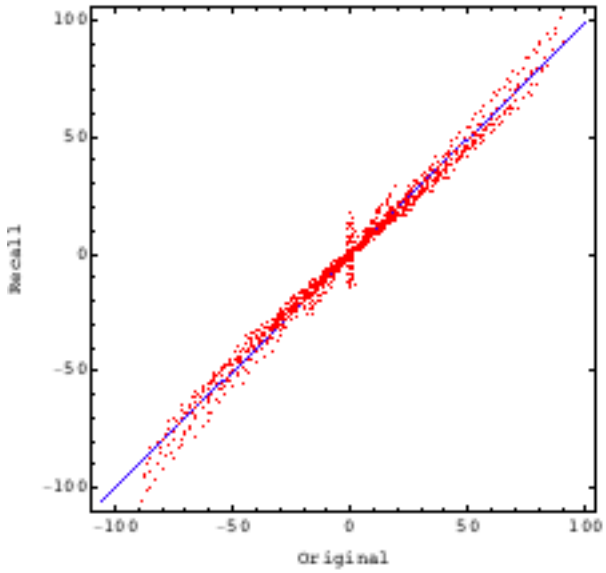


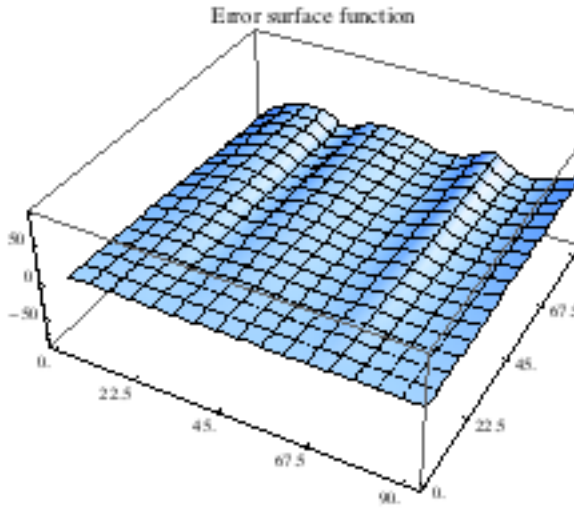
Evaluate model on test data

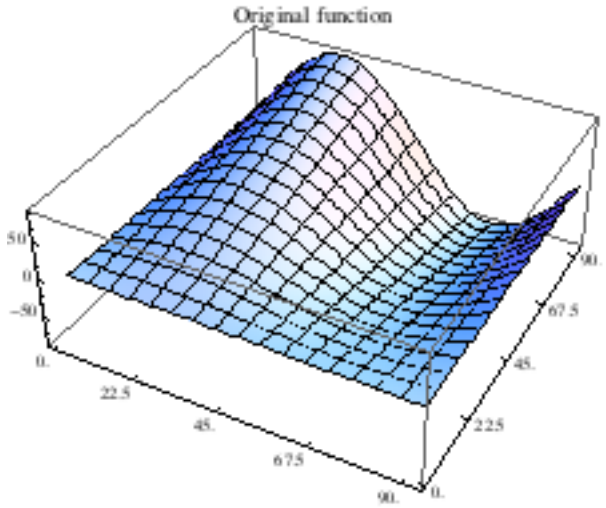
```
recallData = MapThread[Append[#1, #2] &,
    {testData[[All, {1, 2}]],
     MakeSugenoInference[reno, testData[[All, {1, 2}]]]
    }];
```

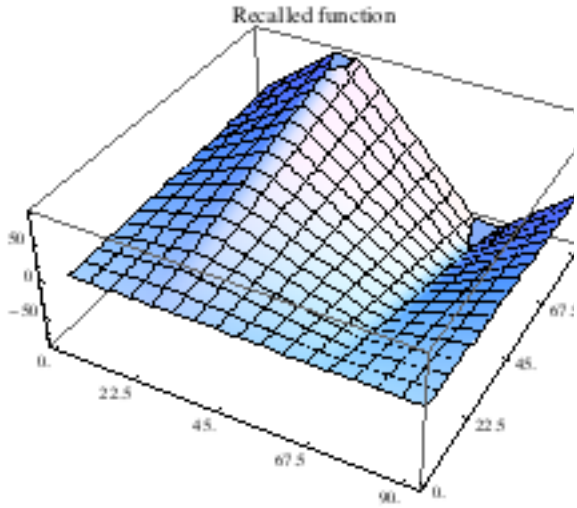
```
MyEvaluation[testData, recallData]
```

```
Correlation Coefficient = 0.994,    normalized MSE = 0.000403
```









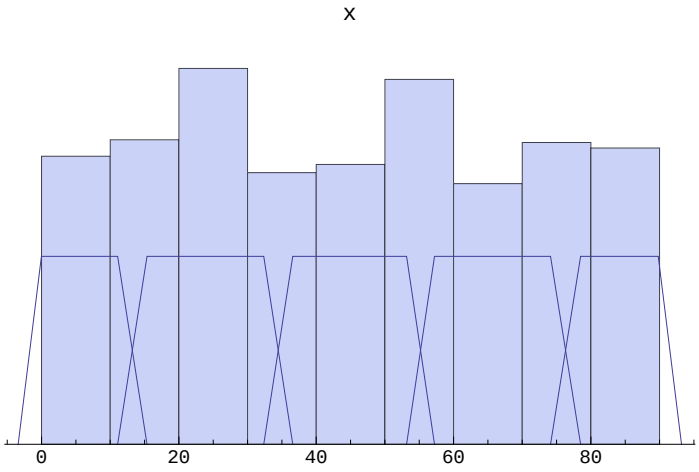
Note the improved predictive quality!

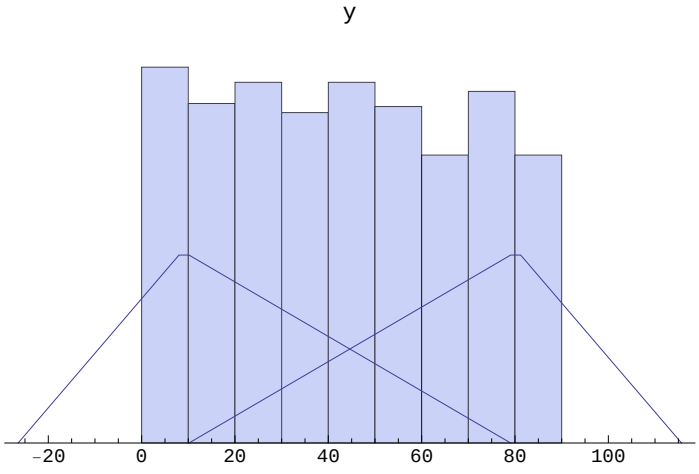
Using Numerical Optimized Decision Trees

Create an initial rule base using ID3 decision trees

We generate initial partitions and predicates. It is important that `CreatePartition`'s option `SeparateSets` is set to `True` to have access to the partitions created later on.

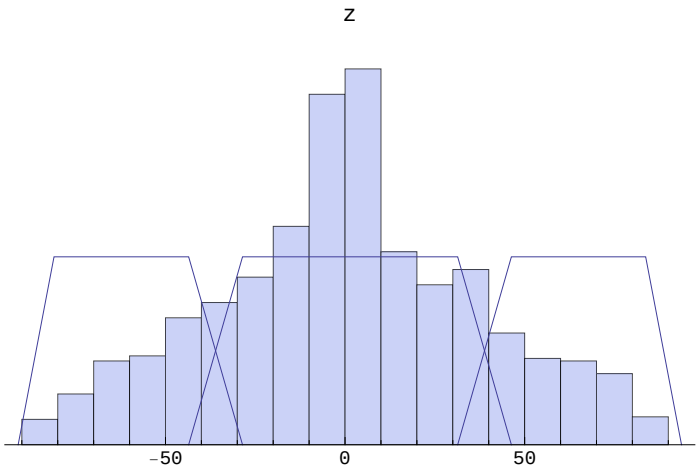
```
trainParams = {1, 2}; goalParam = 3;
testPartitions = {
    CreatePartition[trainDataSet, 1,
     5, Border → True, SetType -> "PWL", SeparateSets → True],
    CreatePartition[trainDataSet, 2, 2, Overlap → 0.8,
     Border → True, SetType -> "PWL", SeparateSets → True]
  };

goalPartitions = CreatePartition[trainDataSet, 3, 3, Width → 1.0,
    Border → True, PredType -> "ISEX", SetType -> "PWL", SeparateSets → True];

testPreds = Flatten@testPartitions[[All, 2]];
goalPreds = goalPartitions[[2]];
testVars = DefPredicateVars[testPreds];
goalVars = DefPredicateVars[goalPreds];

Table[PlotFuzzySetHistogram[
  testPartitions[[i]],
  MLFGetData[trainDataSet, All, trainParams[[i]]],
  PlotLabel → headers[[trainParams[[i]]]]
 ], {i, Length@testPartitions}]
```





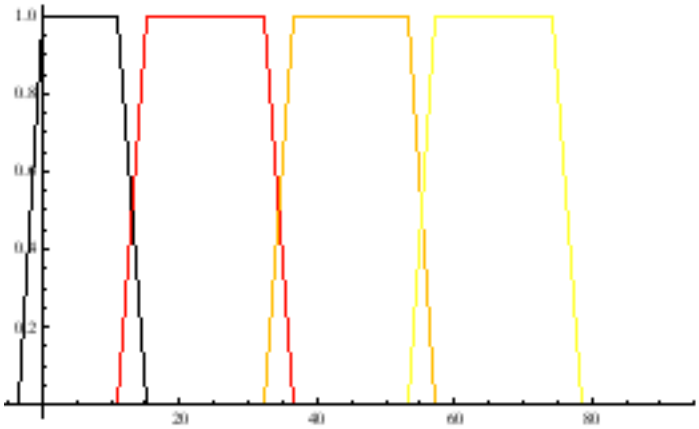

```
{Null, Null}
```

```
PlotFuzzySetHistogram[
 goalPartitions,
 MLFGetData[trainDataSet, All, goalParam],
 PlotLabel → headers[[goalParam]]
]
```



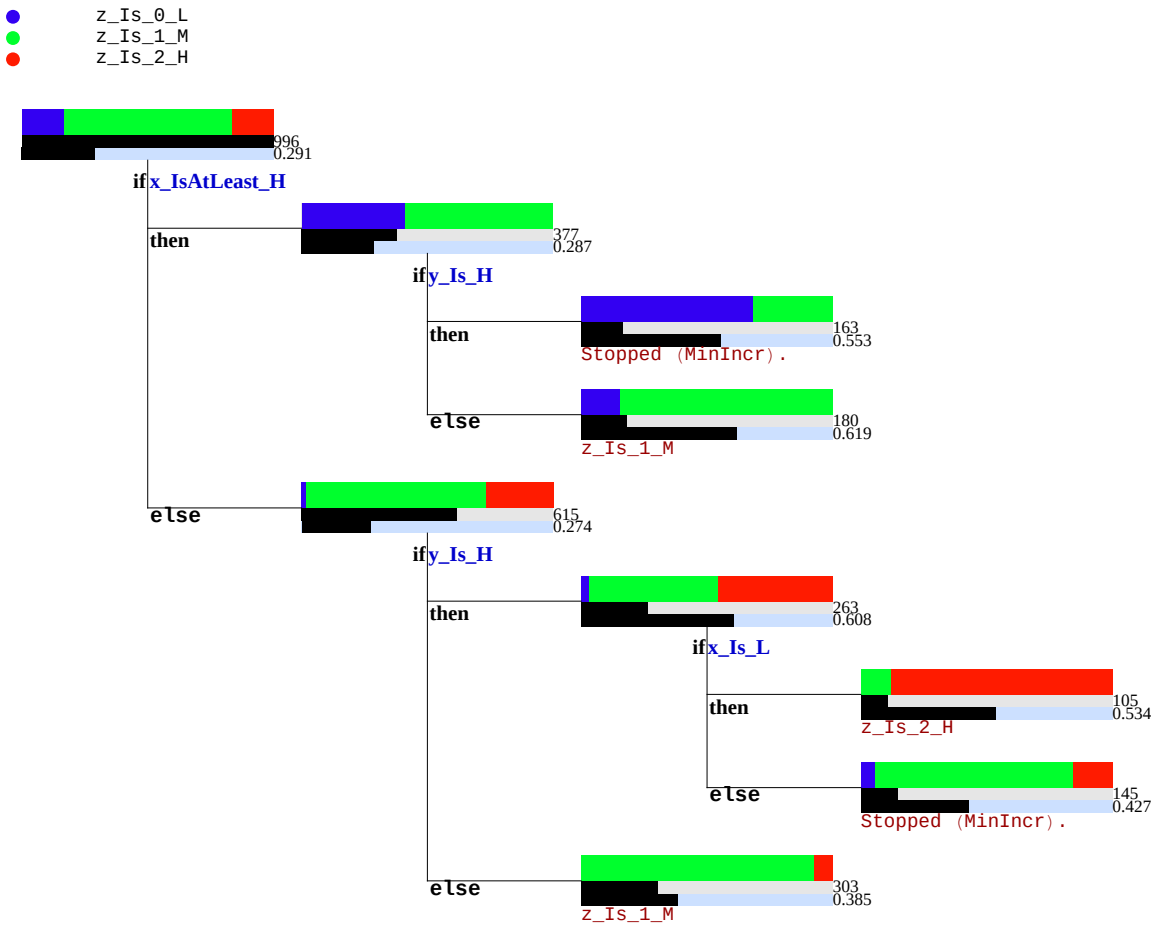
Now we can generate the decision tree with `CreateID3`. The algorithm tries to find rules to separate the three goal classes.

```
PlotFuzzySet[testPartitions[[1]], ColorFunction → (CFHot[#] &)]
```



```
id3tree = CreateID3[trainDataSet, testVars, goalVars,
    MinConf → 0.9, MinEnt → 0.01, MinIncr → 0.01, Logic → LogicL, MaxLevel → 4];
```

```
Show[Graphics[PlotClassLegend[goalVars], AspectRatio → 0.1], PlotRange → All]
Null
PlotID3[id3tree]
```



■ Sugeno controller from ID3

Next, we extract a Sugeno rule base with `ExtractRuleBaseID3` from the decision tree.

```
id3controller = ExtractRuleBaseID3[id3tree, goalPreds];
```

We print the Sugeno rule base. It is much more compact compared to a tensor-product-like rule base as would be created by `CreateRENO`.

```
PrintRules[id3controller]
```

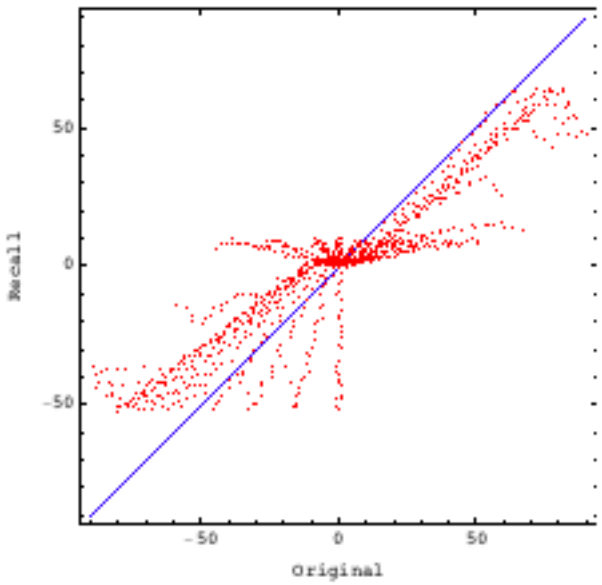
```
x_IsAtLeast_H && y_Is_H → -51.2986
x_IsAtLeast_H && ~[y_Is_H] → 1.36215
~[x_IsAtLeast_H] && y_Is_H && x_Is_L → 63.6505
~[x_IsAtLeast_H] && y_Is_H && ~[x_Is_L] → 9.16609
~[x_IsAtLeast_H] && ~[y_Is_H] → 1.38563
```

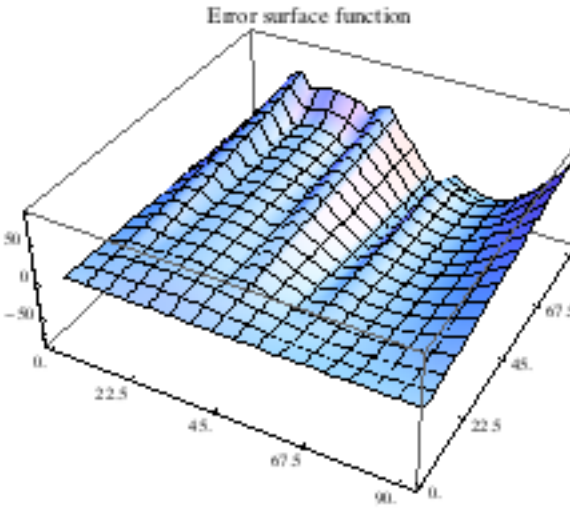
We recall the function and compare it with the original function. We will see that the approximation shows some undesirable effects such as sharp edges.

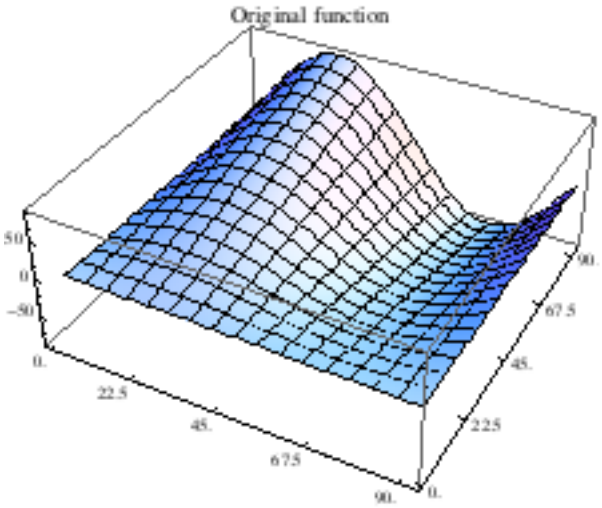
```
recallData = MapThread[Append[#1, #2] &,
    {testData[[All, {1, 2}]],
     MakeSugenoInference[id3controller, testData[[All, {1, 2}]]]
    }];
```

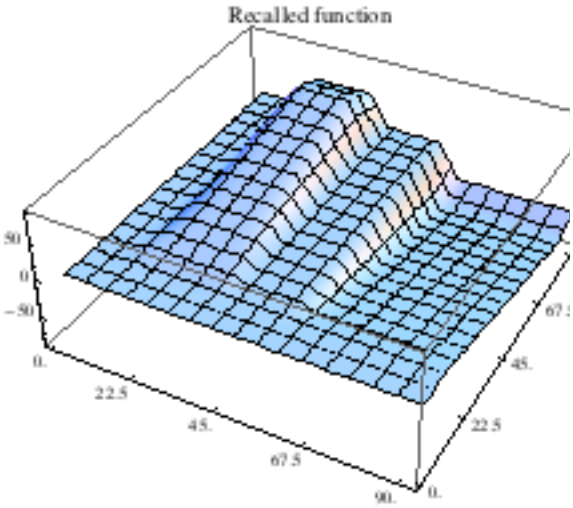
```
MyEvaluation[testData, recallData]
```

```
Correlation Coefficient = 0.899,   normalized MSE = 0.0228
```









■ Tune rule base with OptimizeRENO

In the next step, we want to increase the approximation accuracy of the rule base extracted from the decision tree. We tune both the shape of the antecedent fuzzy sets and the consequence values to fit the output of the rule base as close as possible to the given data. `OptimizeRENO` again uses regularization techniques, namely Tikhonov regularization.

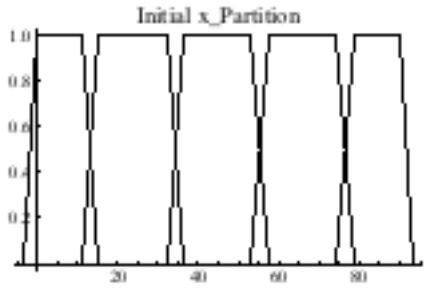
```
Options[OptimizeRENO]
```

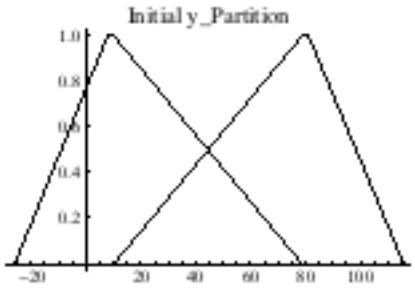
```
{Dims → {1, 2}, Goal → 3, PartitionNames → Automatic,
 Controller → Sugeno, Lambda1 → Automatic, Lambda2 → Automatic}
```

We have to tell `OptimizeRENO` which fuzzy partitions to optimize. We take the partitions generated above by `CreatePartition` (5 trapezoidal fuzzy sets for input x and 2 for input y) and plot them.

```
partitionNames = GetPartitionNames[
   ExtractDimensionsID3[id3tree, goalPreds], headers
  ];
```

```
GraphicsGrid[{{
    PlotFuzzySet[GetObject[partitionNames[[1]]],
     PlotLabel → "Initial " <> partitionNames[[1]]], PlotFuzzySet[
     GetObject[partitionNames[[2]]] , PlotLabel → "Initial " <> partitionNames[[2]]]
  }},
 ImageSize → 500
]
```





Now we invoke the optimization algorithm.

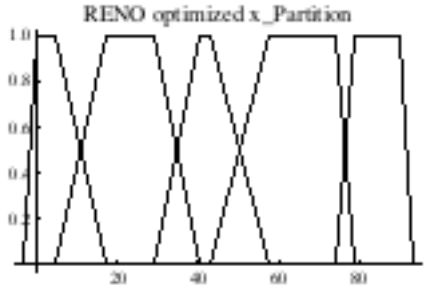
```
reno = OptimizeRENO[trainDataSet, id3controller, PartitionNames → partitionNames,
    Dims → {1, 2}, Goal → 3,  Lambda1 → {0.000001, 0.000001}, Lambda2 → {0.0001, 0.0001}];
```

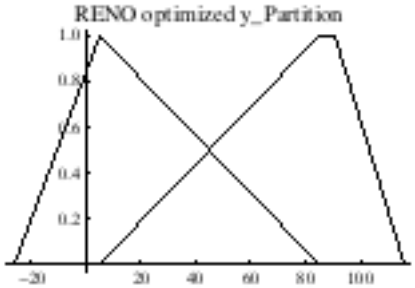
Next, we print the tuned rule base and the adjusted partitions.

```
PrintRules[reno]
```

```
x_IsAtLeast_H && y_Is_H → -64.2286
x_IsAtLeast_H && ~[y_Is_H] → -1.58816
~[x_IsAtLeast_H] && y_Is_H && x_Is_L → 79.2731
~[x_IsAtLeast_H] && y_Is_H && ~[x_Is_L] → 15.1708
~[x_IsAtLeast_H] && ~[y_Is_H] → 0.811893
```

```
GraphicsGrid[{{
    PlotFuzzySet[GetObject[partitionNames[[1]]],
     PlotLabel → "RENO optimized " <> partitionNames[[1]]],
    PlotFuzzySet[GetObject[partitionNames[[2]]],
     PlotLabel → "RENO optimized " <> partitionNames[[2]]]
  }},
 ImageSize → 500
]
```





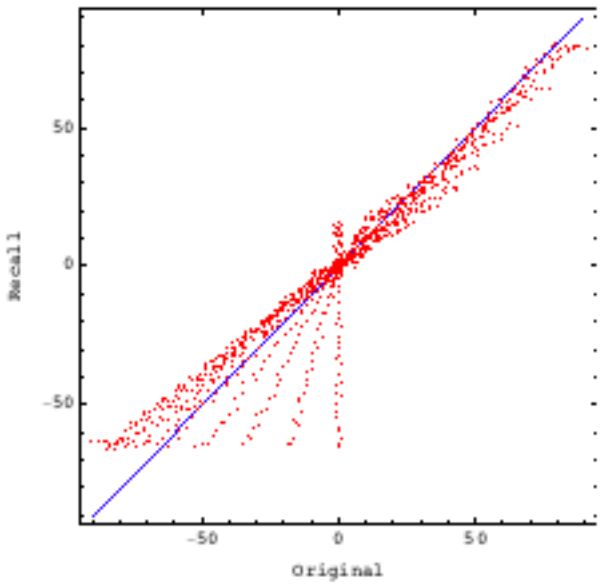
- **Recalled data from the optimized rule base**

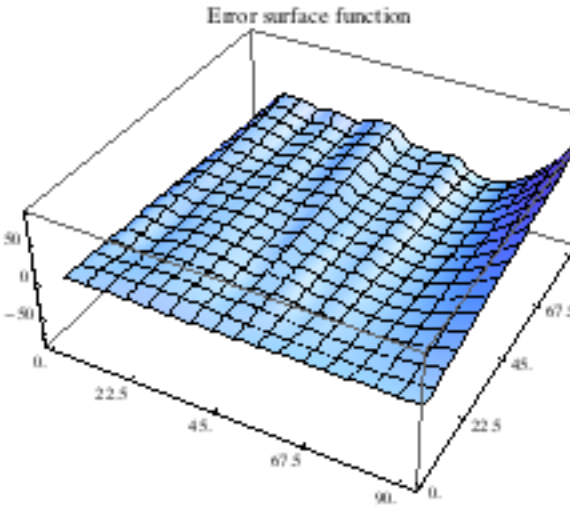
We recall the data from the optimized rule base. The quality of approximation depends heavily on how well the rules describe the behavior of the function to approximate. In general, the output of the optimized rule base is much closer to the data and the approximation error decreases considerably.

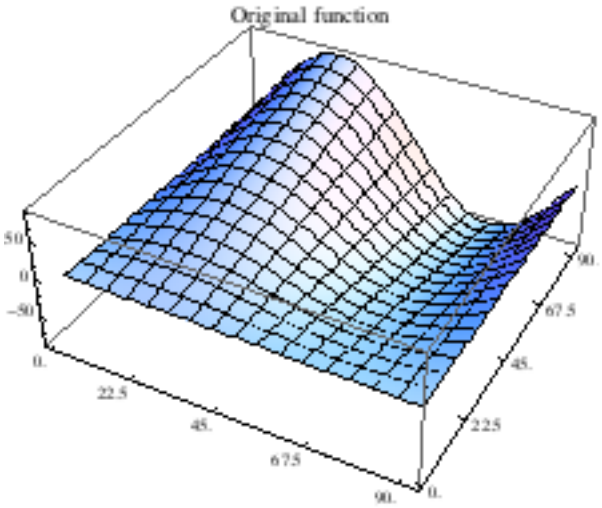
```
recallData = MapThread[Append[#1, #2] &,
    {testData[[All, {1, 2}]],
     MakeSugenoInference[reno, testData[[All, {1, 2}]]]
    }];
```

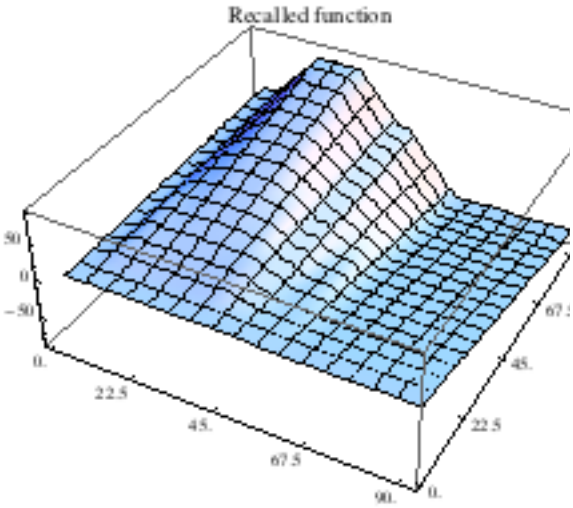
```
MyEvaluation[testData, recallData]
```

```
Correlation Coefficient = 0.953,    normalized MSE = 0.00602
```





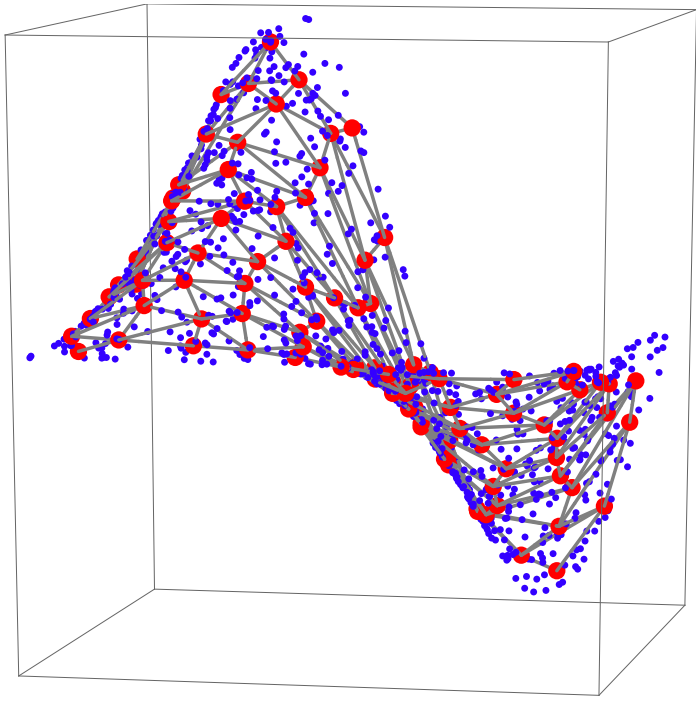




■ Using Self-organizing Maps

```
som = CreateSOM[
    trainDataSet,
    Size → {10, 10},
    Type -> "BATCH",
    InitType -> "Deterministic",
    Errorlevel → 0,
    MaxIter → 10
  ];
```

```
Show[Graphics3D[
  {GrayLevel[0.5],
   PlotAttributes[trainDataSet,
    Dims → {1, 2, 3}, PointSize → 0.01, DefaultColor → Hue[0.7]],
   PlotSOMGrid[som, Dims → {1, 2, 3}, Thickness → 0.005],
   PlotAttributes[som, Dims → {1, 2, 3}, PointSize → 0.025, DefaultColor → Hue[0]]
  }, Axes → False, PlotRange → All, ViewPoint -> {1, -5, 1}, AspectRatio → 1
 ]]
```

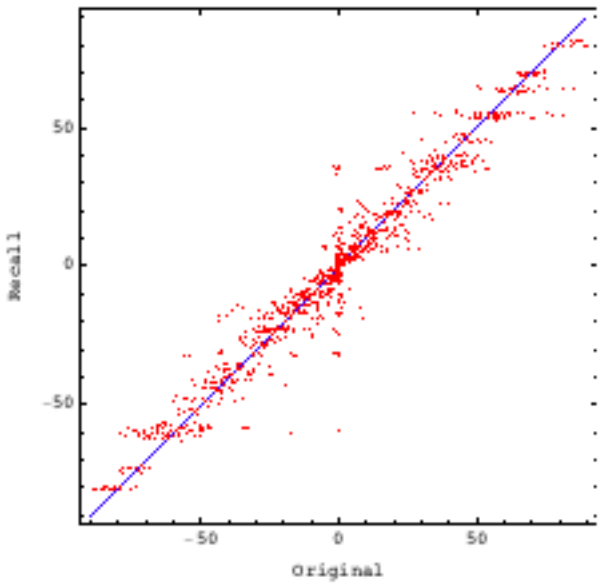


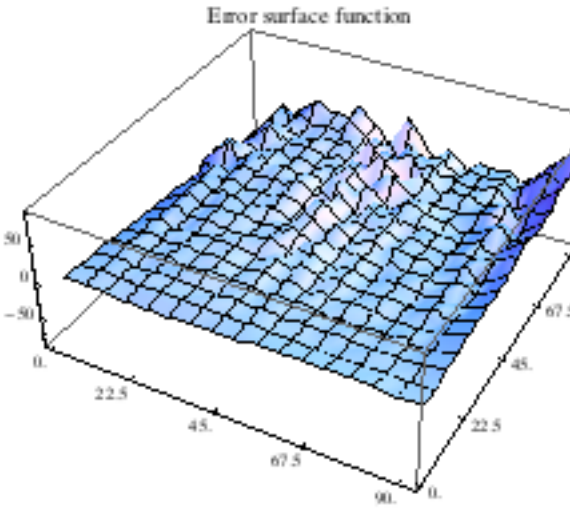
- **Recall data**

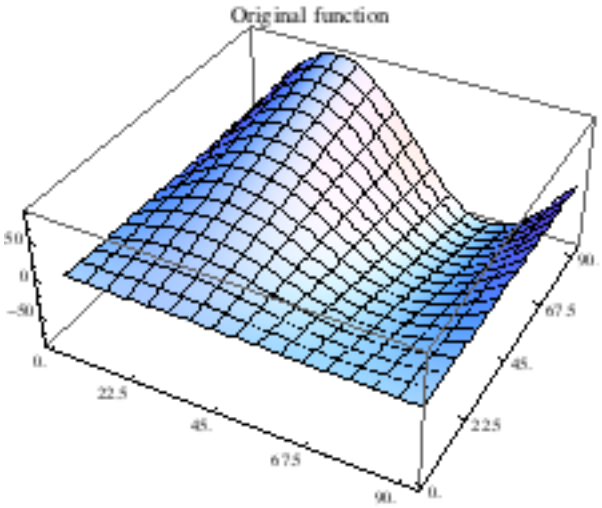
```
recallData = RecallBM[testData[[All, {1, 2}]], som, Weights → {1, 1, 0}];
```

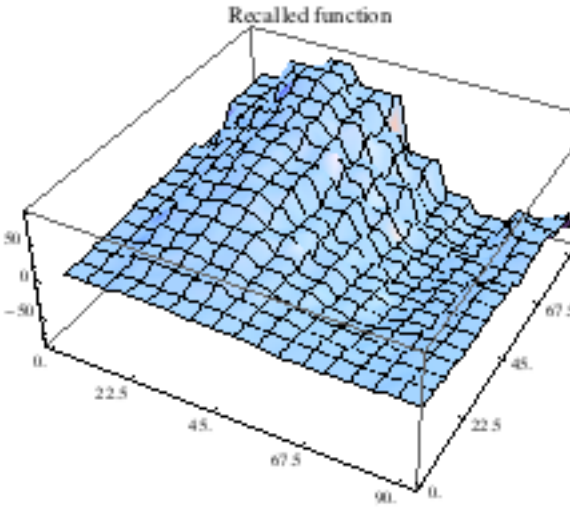
```
MyEvaluation[testData, recallData]
```

```
Correlation Coefficient = 0.972,   normalized MSE = 0.00283
```









■ Comparing Different Models

mlf offers a function for quickly comparing different models with respect to their usefulness for a particular data set and problem. We already introduced `CrossValidation` in the first chapter of the tutorial, but this function is equally well suited for numerical prediction problems. Let us compare the general performance of ID3 decision trees, LIRT regression trees, FOIL rule bases, self-organizing maps and linear (ridge) regression with default settings. The computations may take a few seconds, as the creation of predicates and the model as well as prediction and statistical analysis are performed 10 times. For large data sets, one might want to run `CrossValidation` with different settings overnight, for instance.

In order to make the results repeatable and better comparable, we use option `RandomSeed`.

```
id3Models = CrossValidation[trainDataSet, Algorithm → CreateID3, RandomSeed → 1];
```

```
lirtModels = CrossValidation[trainDataSet, Algorithm → CreateLIRT, RandomSeed → 1];
```

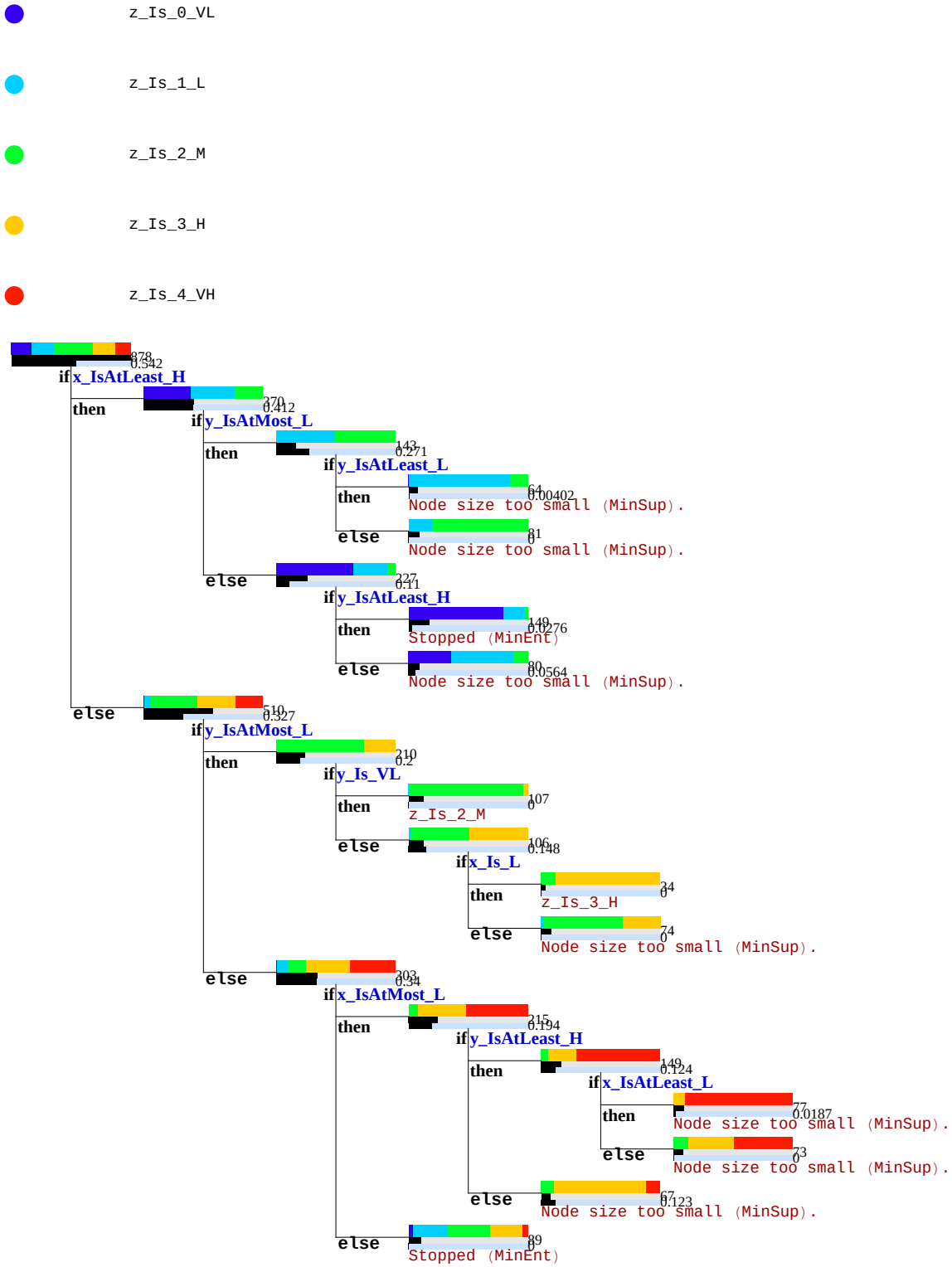
```
foilModels = CrossValidation[trainDataSet, Algorithm → CreateFOIL, RandomSeed → 1];
```

```
somModels = CrossValidation[trainDataSet, Algorithm → CreateSOM, RandomSeed → 1];
```

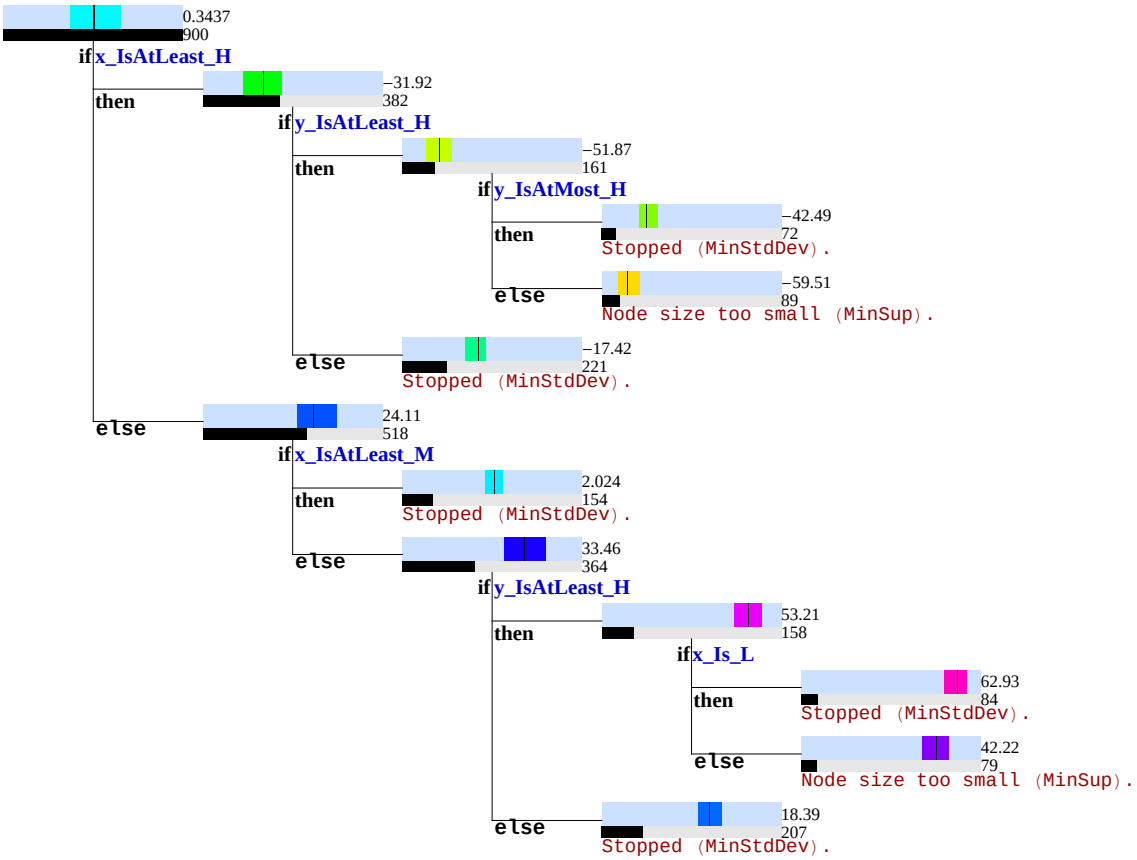
```
regressionModels =
  CrossValidation[trainDataSet, Algorithm → CreateRidgeRegression, RandomSeed → 1];
```

In order to extract the first out of the 10 models generated for each algorithm, we use `GetParam` with an additional parameter indicating the number of the model:

```
PlotModel[GetParam[id3Models, Model, 1], ImageSize → 600]
```



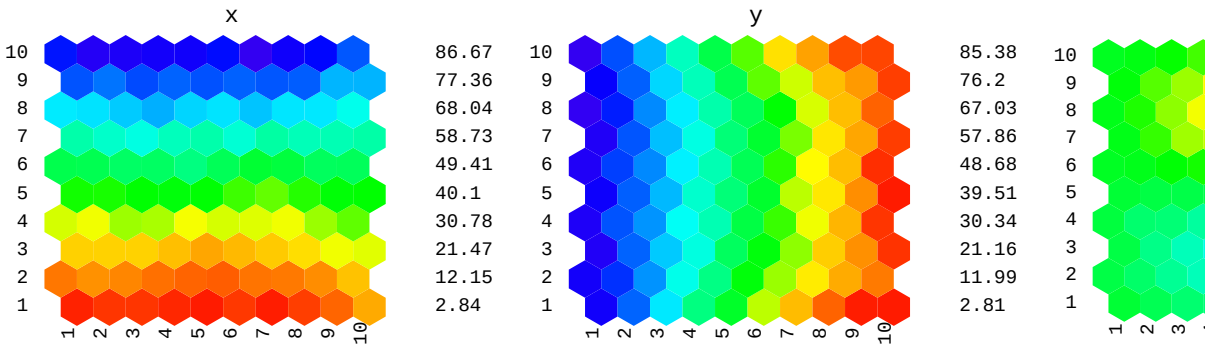
```
PlotModel[GetParam[lirtModels, Model, 1]]
```




```
PlotModel[GetParam[foilModels, Model, 1]]
```

Class	{}	Condition
z_Is_0_VL	⇐	{}
z_Is_1_L	⇐	{}
z_Is_2_M	⇐	y_Is_VL
z_Is_3_H	⇐	{}
z_Is_4_VH	⇐	{}

```
PlotModel[GetParam[somModels, Model, 1], ImageSize → 800]
```



```
PlotModel[GetParam[regressionModels, Model, 1]]
```

```
DOF:       0.380952
Lambda:    1460.88
Noise:     31.4273

Weight     Attribute    Norm.Weight    Mean     Std.dev.     Z-Score     P-Value

17.62
-0.3846    x            -0.262         44.92    25.42        24.48       0
```

List all the available general information of the result:

```
GetParamList[id3Models]
```

```
{PartialResults, TotalNumberOfInstances, TotalCorrelationCoefficient, TotalMSE,
 TotalRSE, TotalMeanError, TotalNormalizedMSE, TotalMAE, TotalRAE, TotalNormalizedMAE,
 TotalStdDev, TotalTargetStdDev, TotalTargetMAD, TotalRatioOfNullPredictions,
 TotalRMSE, TotalRRMSE, MaxCorrelationCoefficient, MaxMAE, MaxMeanError,
 MaxModelSize, MaxMSE, MaxNormalizedMAE, MaxNormalizedMSE, MaxNumberOfInstances,
 MaxRAE, MaxRatioOfNullPredictions, MaxRMSE, MaxRRMSE, MaxRSE, MaxStdDev,
 MaxTargetMAD, MaxTargetStdDev, MinCorrelationCoefficient, MinMAE, MinMeanError,
 MinModelSize, MinMSE, MinNormalizedMAE, MinNormalizedMSE, MinNumberOfInstances,
 MinRAE, MinRatioOfNullPredictions, MinRMSE, MinRRMSE, MinRSE, MinStdDev,
 MinTargetMAD, MinTargetStdDev, QuantileCorrelationCoefficient, QuantileMAE,
 QuantileMeanError, QuantileModelSize, QuantileMSE, QuantileNormalizedMAE,
 QuantileNormalizedMSE, QuantileNumberOfInstances, QuantileRAE,
 QuantileRatioOfNullPredictions, QuantileRMSE, QuantileRRMSE, QuantileRSE,
 QuantileStdDev, QuantileTargetMAD, QuantileTargetStdDev, AverageModelSize}
```

The available statistics depend on the type of the goal; for a categorical goal, the list looks different.

`PartialResults` contains information about each of the 10 models that were generated:

```
GetParamList[GetParam[id3Models, PartialResults][[1]]]
```

```
{NumberOfInstances, CorrelationCoefficient, MSE, RSE, MeanError,
 NormalizedMSE, MAE, RAE, NormalizedMAE, StdDev, TargetStdDev, TargetMAD,
 RatioOfNullPredictions, RMSE, RRMSE, ModelSize, Predictions, OriginalData, Model}
```

The average mean absolute error of all 10 models, respectively:

```
GetParam[id3Models, TotalMAE]
GetParam[lirtModels, TotalMAE]
GetParam[foilModels, TotalMAE]
GetParam[somModels, TotalMAE]
GetParam[regressionModels, TotalMAE]
```

```
10.6519

12.0157

6.23621

4.83627

23.5309
```

The mean absolute error of one particular subset:

```
GetParam[foilModels, MAE, 1]
```

```
6.31903
```

The relative square error of all models, etc:

```
GetParam[id3Models, TotalRSE]
GetParam[lirtModels, TotalRSE]
GetParam[foilModels, TotalRSE]
GetParam[somModels, TotalRSE]
GetParam[regressionModels, TotalRSE]
```

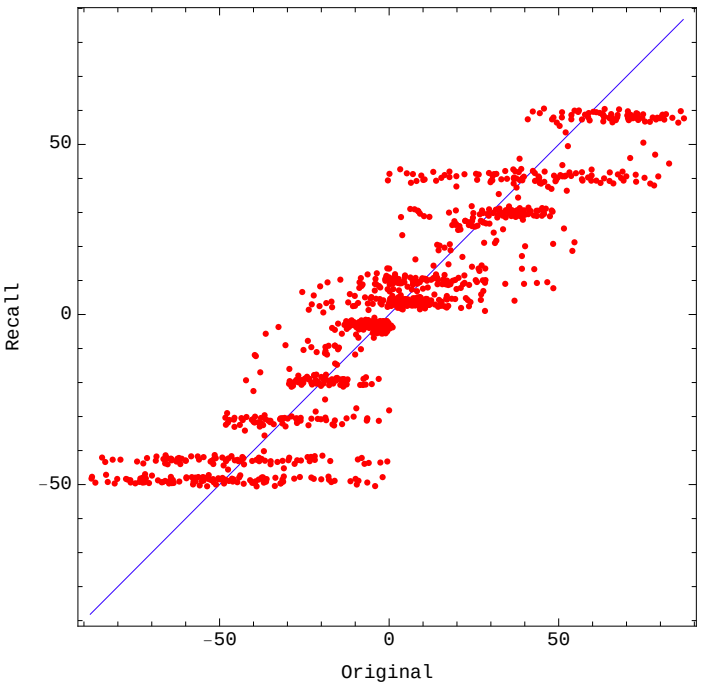
```
0.150503

0.16713

1.19632

0.0356836

0.713579
```

```
GetParam[id3Models, TotalStdDev]
GetParam[lirtModels, TotalStdDev]
GetParam[foilModels, TotalStdDev]
GetParam[somModels, TotalStdDev]
GetParam[regressionModels, TotalStdDev]
```

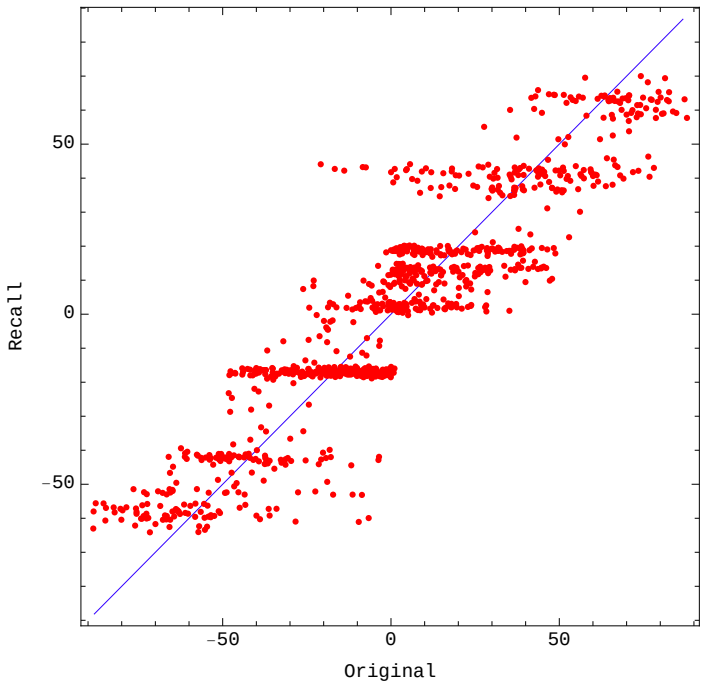
```
14.355

15.1294

7.24295

6.99045

31.2625
```

For a graphical comparison of original data and recalled data, we have to extract those data from `PartialResults` for each model separately and combine them. `PartialResults` need not be accessed explicitly - single elements can be extracted from the total result with an additional integer parameter to `GetParam` (as above, where the models were plotted):

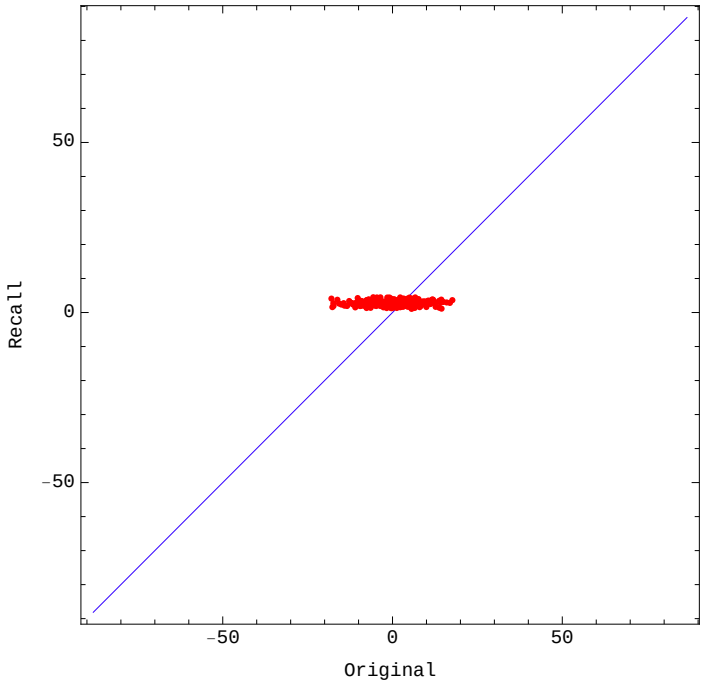
```
PlotInputRecall[Flatten[Table[GetParam[id3Models, OriginalData, i], {i, 1, 10}]],
 Flatten[Table[GetParam[id3Models, Predictions, i], {i, 1, 10}]]]
```



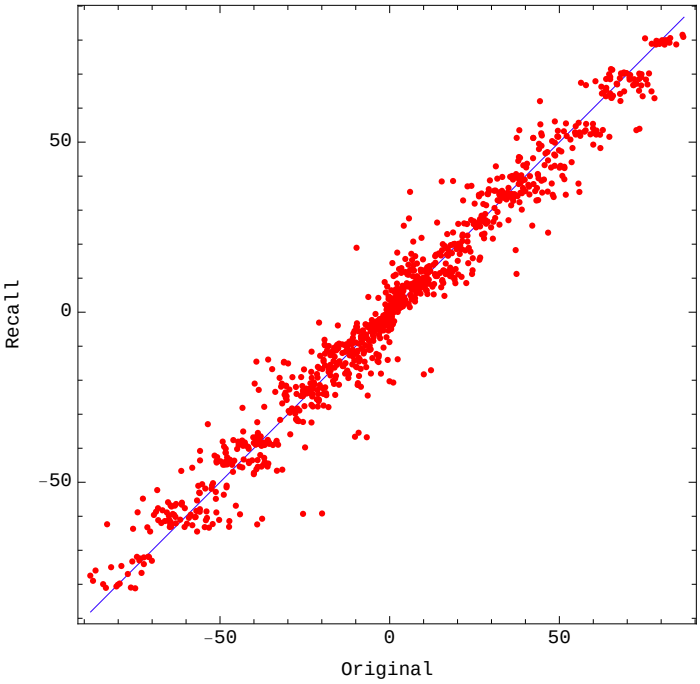
```
PlotInputRecall[Flatten[Table[GetParam[lirtModels, OriginalData, i], {i, 1, 10}]],
 Flatten[Table[GetParam[lirtModels, Predictions, i], {i, 1, 10}]]]
```



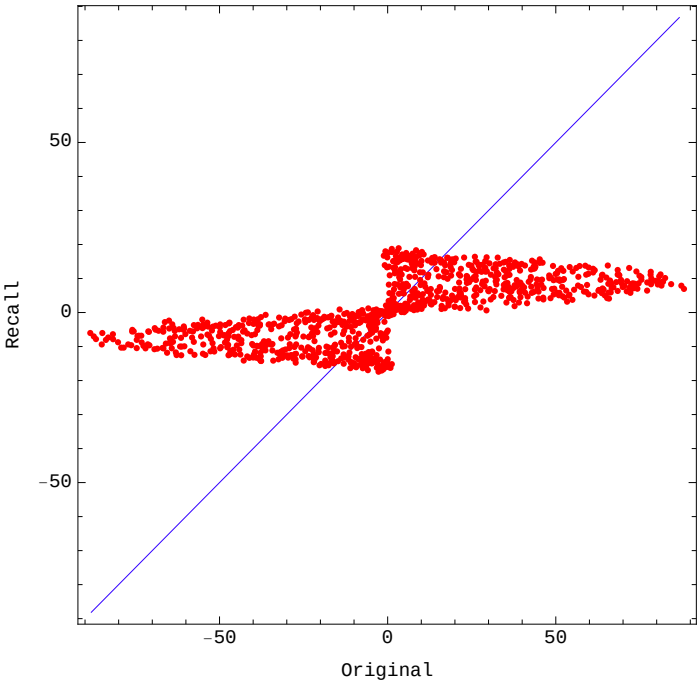

```
PlotInputRecall[Flatten[Table[GetParam[foilModels, OriginalData, i], {i, 1, 10}]],
 Flatten[Table[GetParam[foilModels, Predictions, i], {i, 1, 10}]]]
```



```
PlotInputRecall[Flatten[Table[GetParam[somModels, OriginalData, i], {i, 1, 10}]],
 Flatten[Table[GetParam[somModels, Predictions, i], {i, 1, 10}]]]
```




```
PlotInputRecall[Flatten[Table[GetParam[regressionModels, OriginalData, i], {i, 1, 10}]],
 Flatten[Table[GetParam[regressionModels, Predictions, i], {i, 1, 10}]]]
```



It is obvious that the SOM performs best - at least with default parameters. We can now try to alter some parameters for model generation as well as for predicate generation via options of `CrossValidation`. However, for fine-tuning models for high predictive accuracy, the use of highly parametrizable lower-level functions as described further above is recommended. We will now give a short demonstration of tuning models using CrossVal idation.

```
Options[CrossValidation]
```

```
{Algorithm → None, InputPredOpts → {}, GoalPredOpts → {}, InputPredVars → {},
 GoalPredVars → {}, AlgorithmOpts → {}, RandomSeed → Automatic,
 ExcludeConstantColumns → False, FrequencyEqualization → None, Resampling → None,
 FeatureSelection → None, DataSetTransformations → {}, Folds → 10,
 RandomShuffling → True, ReturnModels → True, ReturnPredictions → True,
 ReturnDistributions → False, ReturnFoldPermutation → False, TrainRowSelection → None,
 TestRowSelection → None, PerformanceFunction → ComputePerformanceMeasures,
 PerformanceQuantile → 0.5, ReInstallWeka → False, Verbose → False,
 ParallelEvaluation → False, FailWhenSlavesDead → True}
```

```
Options[CreateID3]
```

```
{Logic → Automatic, MinConf → 0.9, MinSup → 0.1, MinEnt → 0.1, MinIncr → 0.1,
 MaxLevel → 10, Alternatives → 0, Evaluation → Obj`EntropyEvaluation[], Goal → Automatic}
```

```
Options[CreatePartition]
```

```
{NoOfSets → 5, Overlap → 0.2, Width → 1., PredType → ALL,
 SetType → PWL, MinSup → 0.025, Border → False, Linguistic → True,
 SeparateSets → False, SetNames → Automatic, CutPoints → Automatic,
 Minimum → Automatic, Maximum → Automatic, Confidence → 0.02, Labels → {},
 MaxLengthOfCombinations → 1, CascadingCombinations → False, OutputColumn → 1}
```

```
id3Models2 = CrossValidation[trainDataSet, Algorithm → CreateID3,
   RandomSeed → 1, InputPredOpts → {NoOfSets → 7, Border → True},
   AlgorithmOpts → {MinEnt → 0.01, MinConf → 0.8, MaxLevel → 7}];
```

```
PlotModel[GetParam[id3Models2, Model, 1], ImageSize → 600]
```

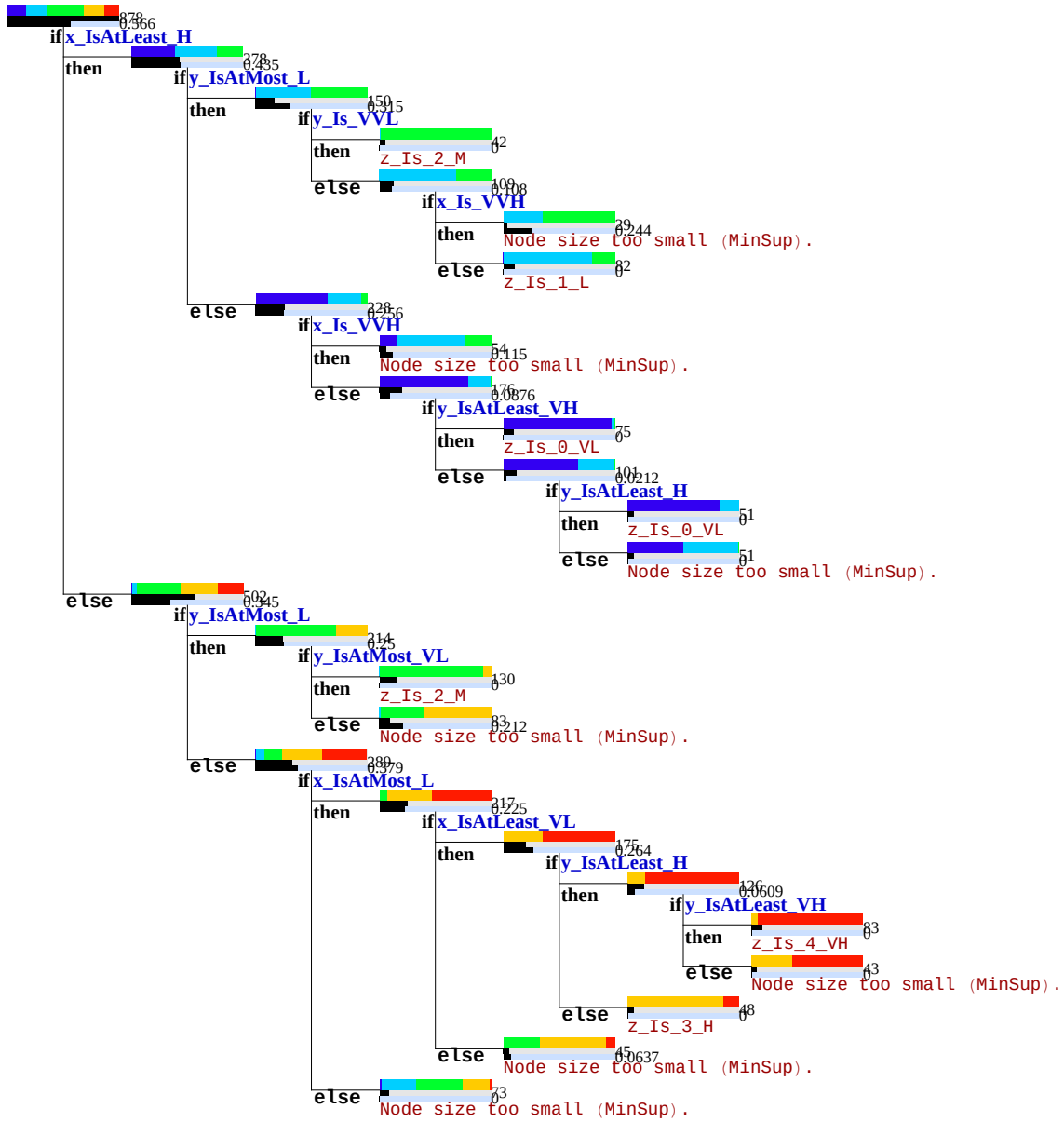
z_Is_0_VL

z_Is_1_L

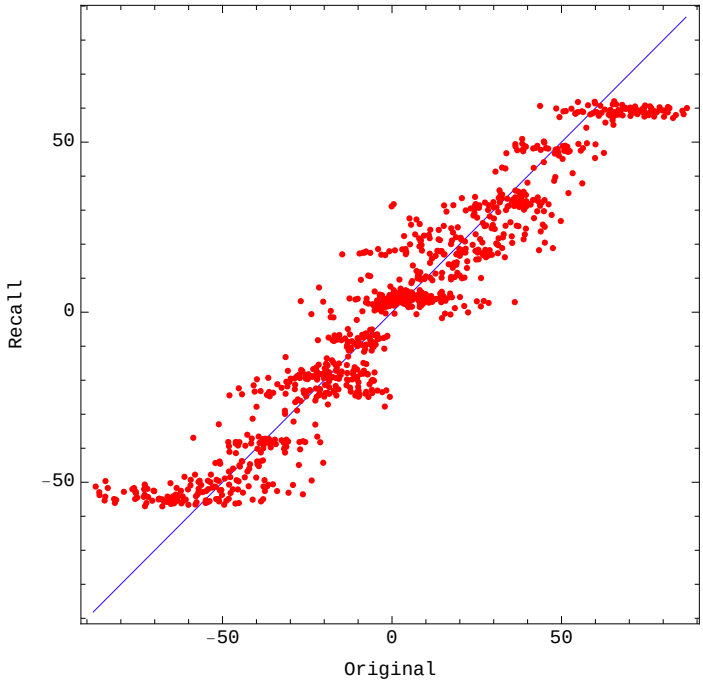
z_Is_2_M

z_Is_3_H

z_Is_4_VH



```
PlotInputRecall[Flatten[Table[GetParam[id3Models2, OriginalData, i], {i, 1, 10}]],
 Flatten[Table[GetParam[id3Models2, Predictions, i], {i, 1, 10}]]]
```



The result is much better than with default settings, but still does not beat the performance of the SOMs that were created with default settings:

```
GetParam[id3Models2, TotalRSE]
```

```
0.0803913
```

```
GetParam[id3Models, TotalRSE]
```

```
0.150503
```

```
GetParam[somModels, TotalRSE]
```

```
0.0356836
```

More About

Template for Supervised Data Analysis II: Numerical Approximation

Related Tutorials

Supervised Data Analysis

Approximation of Numerical Functions (this tutorial)

Unsupervised Data Analysis; including image analysis

Object and Data Handling, Fuzzy Logic Operations, Visualization

Related Wolfram Education Group Courses

XXXX

machine learning framework for *Mathematica*

Unsupervised analysis is concerned with the analysis of data without specifying any explicit goal parameter. It covers various tasks like *clustering* and *finding general relations within data*.

Unsupervised analysis is often performed prior to *supervised analysis* in order to identify issues of interest which are analyzed afterwards, to get rid of noise in the data, or simply to reduce the amount of data.

In this chapter, we will describe several methods and scenarios where unsupervised analysis can help to understand the underlying structure of the data. First, we will present a special kind of neural net, the so-called self-

organizing map (SOM), which is used to both visualize high-dimensional data and cleanse and reduce data. Afterwards, we will describe different clustering algorithms which are applicable to identify regions of similarity within the data. We will also show how supervised methods can be used to create cluster descriptions.

Initialization of the system

```
Needs["mlf`"];

Off[DataSet::mindiff];
```

Basics: an Artificial Data Set

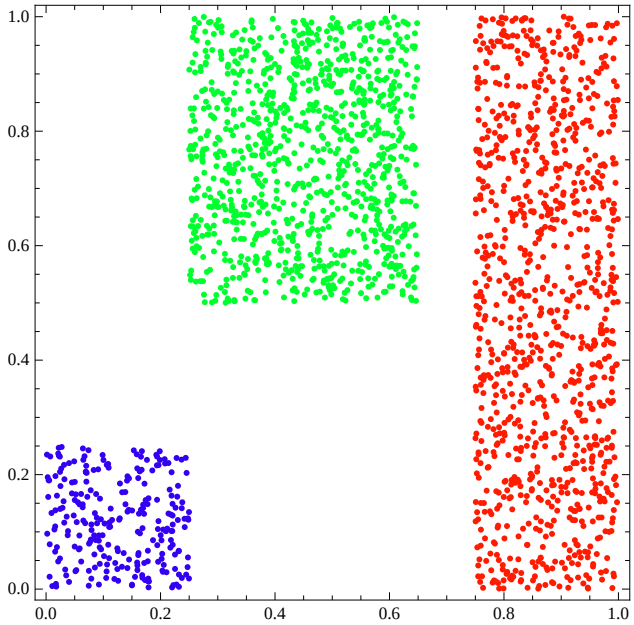
In this section, we will briefly describe the methods available for unsupervised learning. First we explain how self-organizing maps (SOMs) can be computed. Then we show how different clustering methods are employed to find regions of similarity within the data. Finally, we apply supervised learning methods to create descriptions for the clusters that have been found.

Set up Data

First we create some random data with three distinct regions and visualize it.

```
SeedRandom[10];
samples = 1000;
data1 = Table[{RandomReal[0.25], RandomReal[0.25], 1}, {i, samples * 0.25}];
data2 = Table[{0.25 + RandomReal[0.4], 0.5 + RandomReal[0.5], 2}, {i, samples}];
data3 = Table[{0.75 + RandomReal[0.25], RandomReal[], 3}, {i, samples}];
sampleData = DataSet[Join[data1, data2, data3], {"x", "y", "class"}];
```

```
Show[Graphics[
  PlotAttributes[sampleData, Dims → {1, 2}, Goal → 3, PointSize → 0.01],
  Frame → True,
  AspectRatio → 1,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}}
  (* this option ensures that all tick labels are visible *)
 ]]
```



Compute a SOM

Self-organizing maps (SOMs) have been introduced by Teuvo Kohonen in 1982. His idea was to create a neural network to represent the input space using the topological structure of a grid to store neighborhood relationships. Most neural network methods use the desired result to compute the weights of the network; in contrast, SOMs do not need any reference output to tune their parameters, but they are able to organize themselves autonomously

(*unsupervised learning*).

A SOM defines a mapping of an n-dimensional input space R to an m-dimensional output space C (we use m=2). The output space consists of N neurons. They are embedded in the topological structure of C, which may be an m-dimensional grid or any other graph structure (e.g a hexagonal grid). To each neuron s of output space C, a parametric weight vector in the input space is associated. If a new sample is presented to the map, it is associated with the node which is closest to it.

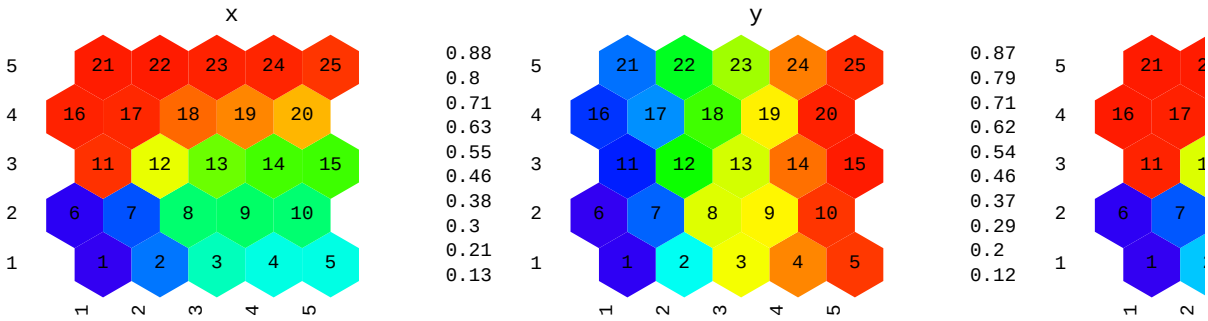
We now compute a 5x5 SOM using the command `CreateSOM`, considering only the two input dimensions (the first two dimensions) by setting option Weights→{1,1,0}:

```
som = CreateSOM[
    sampleData,
    Weights → {1, 1, 0},
    Size → {5, 5},
    Type → "BATCH"
  ];
```

Now we have obtained a SOM with 5*5=25 node vectors. The net has been trained with the so-called BATCH algorithm and weights of {1,1,0}, i.e., the third attribute was not taken into account for training. Note that we have trained the SOM implicitly with the class information. This means that each node contains the class information; however, it was not used to compute the neighborhood relations.

We can visualize the SOM in grid coordinates using the command `PlotSOMHexRaster`. In this plot, each node is color-coded with respect to the corresponding attribute (given as plot label). We use the option `SOMLabels` to enumerate the nodes.

```
PlotSOMHexRaster[som, {1, 2, 3}, SOMLabels → Range[25], ImageSize → 800]
```



The command `GetDataSOM` is now applied to inspect the calculated weight vectors of each node. GetDataSOM[som,i,j] returns the weight vector associated with the node in row i and column j of the SOM.

```
GetDataSOM[som, 1, 2]
```

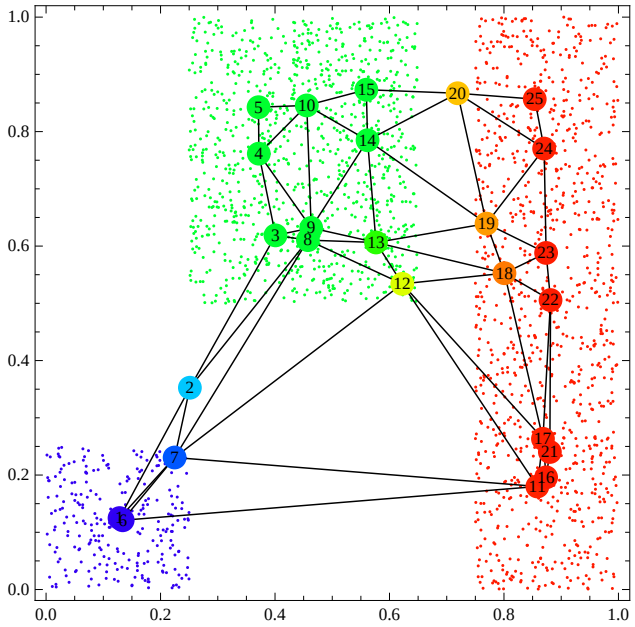
```
{0.251258, 0.352556, 1.4929}
```

```
allVectors =
  Table[{i, j, {Map[SetPrecision[#, 3] &, GetDataSOM[som, i, j]]}}, {i, 5}, {j, 5}];
TableForm[Flatten[allVectors, 1], TableHeadings →
  {Automatic, {"row", "column", "weight vector"}}]
```

	row	column	weight vector
1	1	1	0.128 0.125 1.01
2	1	2	0.251 0.353 1.49
3	1	3	0.401 0.619 2.00
4	1	4	0.372 0.761 2.00
5	1	5	0.371 0.843 2.00
6	2	1	0.134 0.121 1.02
7	2	2	0.225 0.231 1.28
8	2	3	0.457 0.610 2.00
9	2	4	0.463 0.632 2.00
10	2	5	0.456 0.846 2.00
11	3	1	0.858 0.180 2.98
12	3	2	0.622 0.534 2.30
13	3	3	0.576 0.606 2.13
14	3	4	0.562 0.785 2.00
15	3	5	0.560 0.873 2.00
16	4	1	0.874 0.196 3.00
17	4	2	0.868 0.263 2.98
18	4	3	0.801 0.553 2.74
19	4	4	0.770 0.639 2.65
20	4	5	0.719 0.867 2.53
21	5	1	0.880 0.240 3.00
22	5	2	0.881 0.506 3.00
23	5	3	0.874 0.588 3.00
24	5	4	0.870 0.770 2.99
25	5	5	0.854 0.857 2.98

Another possibility is to plot the (labeled) nodes with the command `PlotSOMGrid` together with the data in the first two dimensions:

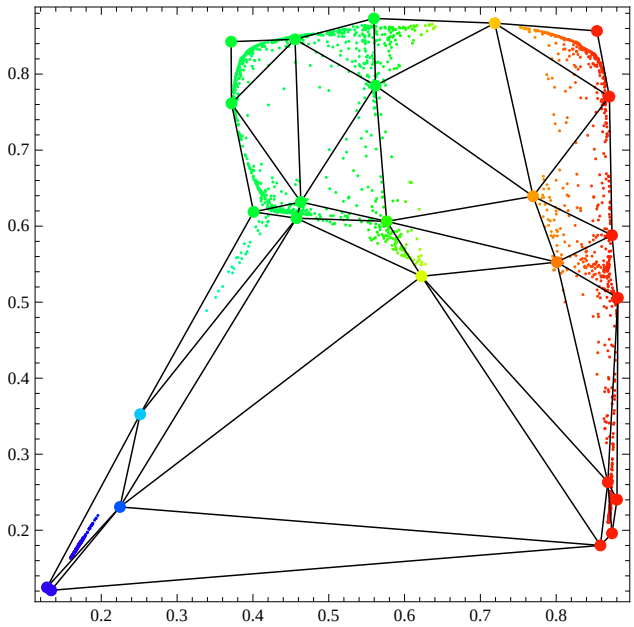
```
Show[Graphics[Join[
    PlotAttributes[sampleData, Dims → {1, 2}, Goal → 3, PointSize → 0.005],
    PlotSOMGrid[som, Dims → {1, 2},
     Goal → 3,
     PointSize → 0.04,
     Thickness → 0.0025,
     Labels → Range[25],
     GridColor → GrayLevel[0]]
  ],
  Frame → True,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}},
  AspectRatio → 1
 ]]
```



We can recall the best-matching reference vector(s) using the command `RecallBM`. When more than one best-matching node is used (NSize > 1), an interpolation between these two nodes is performed.

```
recallData = RecallBM[sampleData, som, NSize → 4];
```

```
Show[Graphics[Join[
    PlotAttributes[recallData, Dims → {1, 2}, Goal → 3, PointSize → 0.005],
    PlotSOMGrid[som, Dims → {1, 2}, Goal → 3,
     PointSize → 0.02, Thickness → 0.0025, GridColor → GrayLevel[0]]
  ],
  Frame → True,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}},
  AspectRatio → 1
 ]]
```



We can see how the SOM covers the data space. Since the recall mechanism includes interpolation, the samples are distributed uniformly within the boundaries of the net. On the edges, however, we can see that samples lying outside are mapped into it.

■ Compute Clusters with WARD

We now compute clusters within the data by using the so-called WARD method. Since we do not know the optimal number of clusters in advance, we forecast 1 and pass this prediction to algorithm `CreateWARD` via option Size (note that this option is only available for WARD clustering).

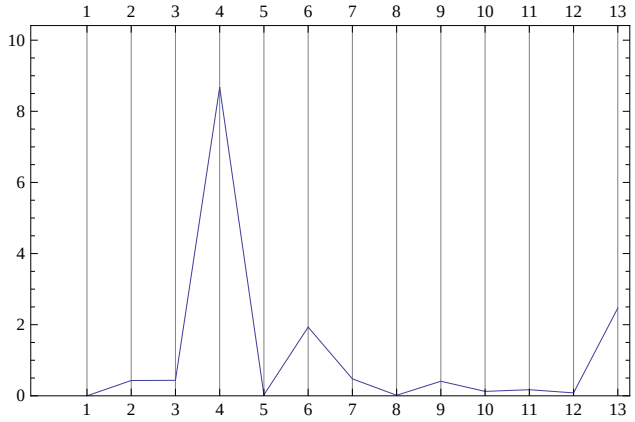
As WARD clustering is computationally very expensive, we suggest to *apply it to the weight vectors of SOMs only*! SOMs are of course much smaller than the original data sets from which they were created, and are therefore manageable by this method. Again, we are only interested in the first two dimensions:

```
centersWARDinit = CreateWARD[
    som,
    Weights → {1, 1, 0},
    Size → 1
  ];
```

The WARD clustering algorithm just applied has computed a validation index called WIndex, which provides 13 validation values. Each of these validation values rates the likelihood that the data set contains a certain number of clusters. In *mlf* 's implementation of WARD, the validation values are computed for clusters 1 to 13, since this turned out to produce the most reasonable results.

To plot the validation index, we first have to extract the WIndex parameter from the clustering result. Afterwards, we can plot the validation values with the command `PlotLineGraph` command.

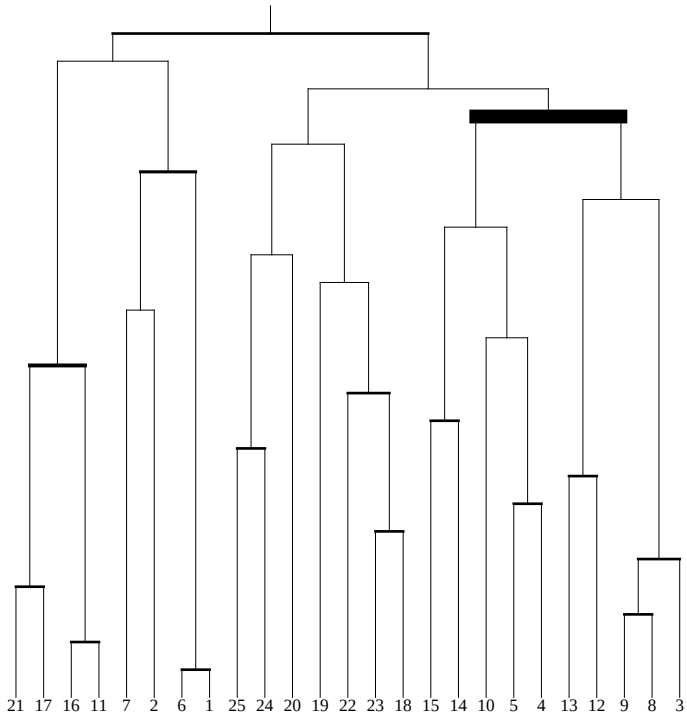
```
wardIndex = GetParam[centersWARDinit, "WIndex"][[1]];
PlotLineGraph[wardIndex, PlotRegion → {{0.05, 0.95}, {0.05, 0.95}}]
```



Examining this plot, we can now identify the numbers of clusters which are most likely the most appropriate for the given data set. The numbers of clusters (ranging from 1 to 13) are enumerated on the abscissa and the respective validation value is given on the ordinate. The higher the validation value, the more likely is the respective number of clusters. We can now experiment with the best-validated numbers of clusters to obtain the optimal number of clusters for our purpose, i.e., we simply pick the peaks of the graph.

Applying a dentogram is an alternative way of displaying the cluster index. A dentogram describes how clusters are joined together. This feature is only available for WARD clustering.

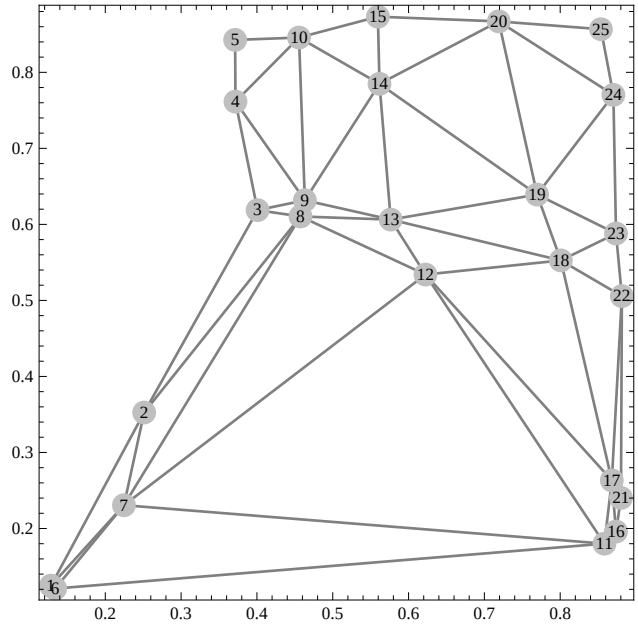
```
Show[Graphics[PlotClusteringTree[centersWARDinit]], PlotRange → All]
```



The numbers used in the dentogram correspond to the numbering of the nodes. We see that nodes 1 and 6 were grouped into a cluster in the very first step, followed by nodes 9 and 8, which were put into a second cluster (note the height of the horizontal lines which signifies the succession of steps bottom-up). The very fat line marks the step where 6 clusters remain (the most promising number of clusters). The lesser fat line marks the step where the second-most promising number of clusters is reached (4 clusters).

We can plot the SOM with the corresponding labels using the Labels option of `PlotSOMGrid`.

```
Show[Graphics[
  PlotSOMGrid[som, Dims → {1, 2}, Thickness → 0., Labels → Range[25] , PointSize → 0.04],
  Frame → True,
  AspectRatio → 1,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}}
 ]]
```



After having fixed the number of clusters for our data set, we can proceed to the final clustering step. This final clustering step can be a WARD clustering again, but we may also apply *fuzzy k-means clustering* now. In both cases, the final number of clusters chosen has to be given via option Size.

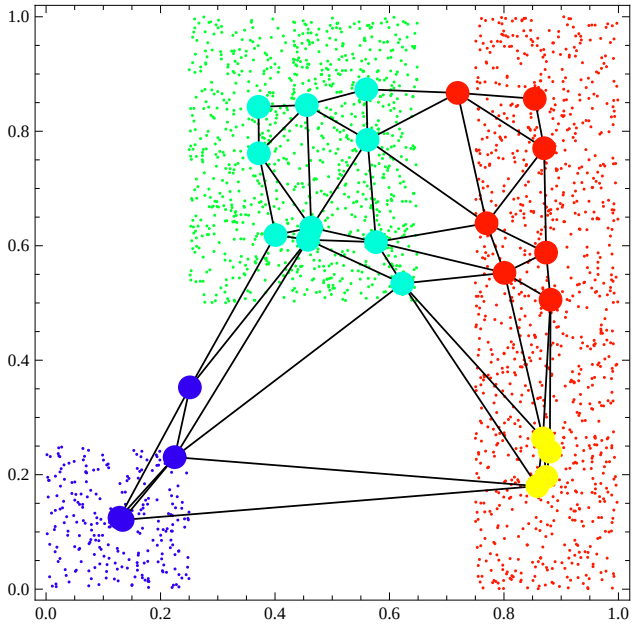
We will now compute the final clustering of the training data for the optimal number of clusters:

```
optSize = Position[wardIndex, Max@wardIndex][[1, 1]];
Print["Computing ", optSize, " clusters."];
centersWARD = CreateWARD[
    som,
    Weights -> {1, 1, 0},
    Size → optSize
  ];
```

```
Computing 4 clusters.
```

Now we can plot the new clusters by specifying a proper value for option Goal of `PlotSOMGrid`.

```
Show[Graphics[Join[
    PlotAttributes[sampleData, Goal → 3, Dims → {1, 2}],
    PlotSOMGrid[som, Dims -> {1, 2},
     Goal → centersWARD,
     PointSize → 0.04, GridColor → GrayLevel[0], Thickness → 0.003
    ]
  ],
  Frame → True,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}},
  AspectRatio → 1
 ]]
```



■ Computing Clusters with k-Means

When looking at the final clustering found by the WARD method, we can see that some nodes are on the border of the scope of their corresponding clusters. As the WARD clustering method assigns each node to only one cluster, this node can cause misinterpretations. For such cases, it is much more appropriate to use fuzzy clusters, where each node can belong to multiple clusters.

To compute a fuzzy clustering, we use the fuzzy k-means method. It is called by the `CreateKMeans` command. Note that the k-means clustering method is sensitive to initialization. Therefore, running the algorithm multiple times might give different results. See the next section on how to overcome this problem.

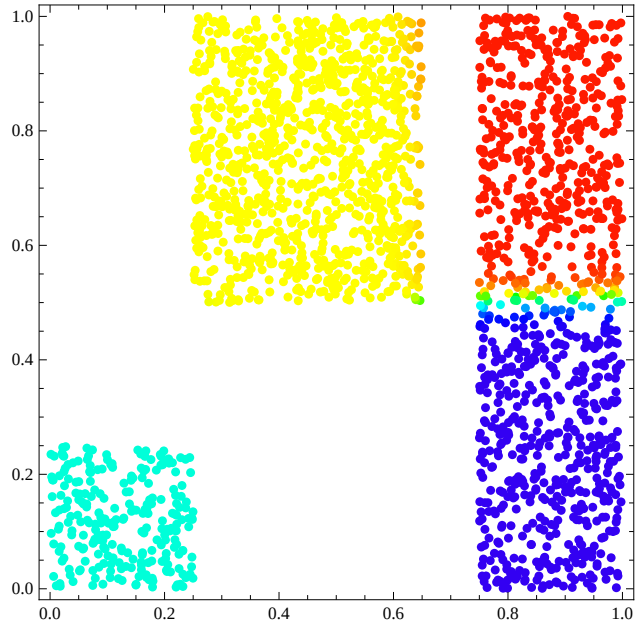
```
centersKMeans = CreateKMeans[
    sampleData,
    Weights -> {1, 1, 0},
    Size → 4,
    Fuzziness → 50.0,
    SomFactor → 10.0
  ];
```

As the k-means clusters are described by their cluster centers, we can print out the representative of each cluster or plot the data colorized according to their cluster membership (using Goal→centersKMeans).

```
GetParam[GetParam[centersKMeans, Clusters], "DataMatrix"][[1]] // TableForm
```

```
0.873547    0.245778    2.99927
0.126925    0.121132    1.00209
0.449323    0.755056    2.00016
0.868842    0.769593    2.98909
```

```
Show[Graphics[
  PlotAttributes[sampleData, Dims -> {1, 2}, Goal → centersKMeans, PointSize → 0.015],
  Frame → True,
  AspectRatio → 1,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}}
 ]]
```



■ Computing Clusters by Combining Different Methods

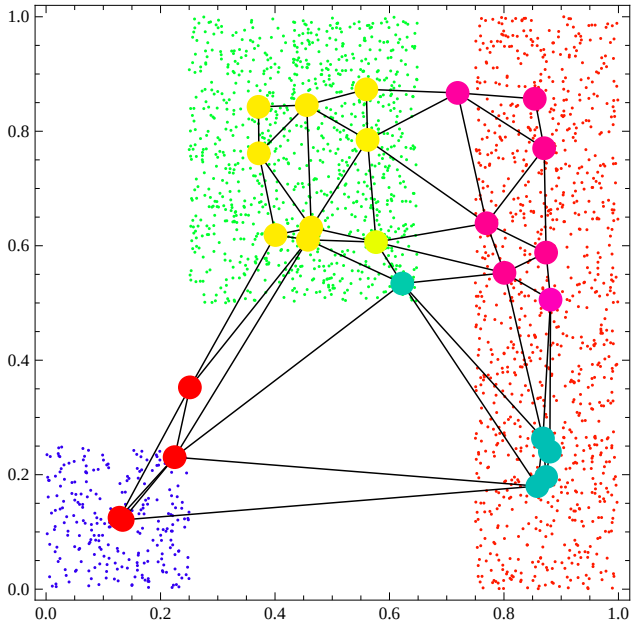
As it is often a critical task to identify the number of clusters and the initial cluster centers, it is useful to combine different methods to accomplish this.

In the following example, we use the WARD method to initialize the k-means clustering to overcome its sensitivity to initialization. This is done by appropriately setting option InitClusters to the centers gained by WARD when calling the command `CreateKMeans`.

```
centersWARD = CreateWARD[
    som,
    Weights -> {1, 1, 0},
    Size → 4
  ];
centersKMeans = CreateKMeans[
    som,
    InitClusters → GetParam[centersWARD, Clusters],
    Weights -> {1, 1, 0},
    Size → 4
  ];
```

Now we can plot the new clusters.

```
Show[Graphics[Join[
    PlotAttributes[sampleData, Dims -> {1, 2}, Goal -> 3, PointSize → 0.005],
    PlotSOMGrid[som, Dims → {1, 2},
     Goal → centersKMeans,
     ColorFunction → CFPrism,
     PointSize → 0.04, Thickness → 0.0025, GridColor → GrayLevel[0]
    ]
   ],
   Frame → True,
   PlotRegion → {{0.05, 0.95}, {0.05, 0.95}},
   AspectRatio → 1]]
```



■ Create descriptions

In this section, we will create a description of the clusters generated by the k-means algorithm. Creating the descriptions amounts to a supervised learning problem where we want to predict to which cluster a given data point belongs. Typically, a rule-base learning algorithm like FS-MINER is used for that purpose since we want an interpretable description of the clusters.

```
centersKMeans = CreateKMeans[
    sampleData,
    InitClusters → GetParam[centersWARD, Clusters],
    Weights -> {1, 1, 0},
    Size → 4
   ];
```

```
testPreds = CreatePredicates[sampleData, {1, 2}, 3];
testVars = DefPredicateVars[testPreds];
```

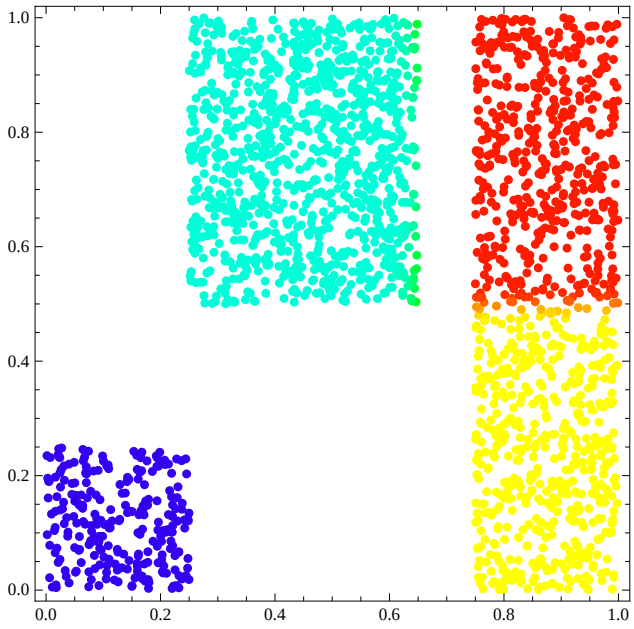
```
descr = CreateClusterDescriptions[sampleData, centersKMeans, testVars];
PrintDescriptions[descr, Info → False]
```

Class	{}	Condition
Is_Class1	⇐	x_Is_L && y_Is_L
Is_Class2	⇐	x_IsAtMost_M && y_IsAtLeast_M
Is_Class3	⇐	y_Is_L && x_IsAtLeast_M
Is_Class4	⇐	x_Is_H && y_Is_H
		x_Is_H && y_IsAtLeast_M

By setting the maximum number of rules to 1, we get quite a good description of the clusters which can be confirmed by looking at the data points colored according to their class description.

```
descr = CreateClusterDescriptions[sampleData, centersKMeans, testVars, MaxRules → 1];
PrintDescriptions[descr, Info → False]
Show[Graphics[
  PlotAttributes[sampleData, Dims -> {1, 2}, Goal → centersKMeans, PointSize → 0.015],
  Frame → True,
  PlotRegion → {{0.05, 0.95}, {0.05, 0.95}},
  AspectRatio → 1
 ]]
```

Class	{}	Condition
Is_Class1	⇐	x_Is_L && y_Is_L
Is_Class2	⇐	x_Is_M
Is_Class3	⇐	y_Is_L && x_IsAtLeast_M
Is_Class4	⇐	x_Is_H && y_Is_H
		x_Is_H && y_IsAtLeast_M



Simple Analysis: the "iris" Data Set

In this example, we will analyze a simple data file which contains data of about 150 iris flowers of different types (species). Each flower is described by five attributes. The first four attributes (sepal length, sepal width, petal length, and petal width) characterize the blossom of the flower, while the last attribute specifies to which category (or species) of iris the flower belongs.

Set up Data

First, we load the data set from a file.

```
irisData = LoadData["iris.txt", "Table"];
```

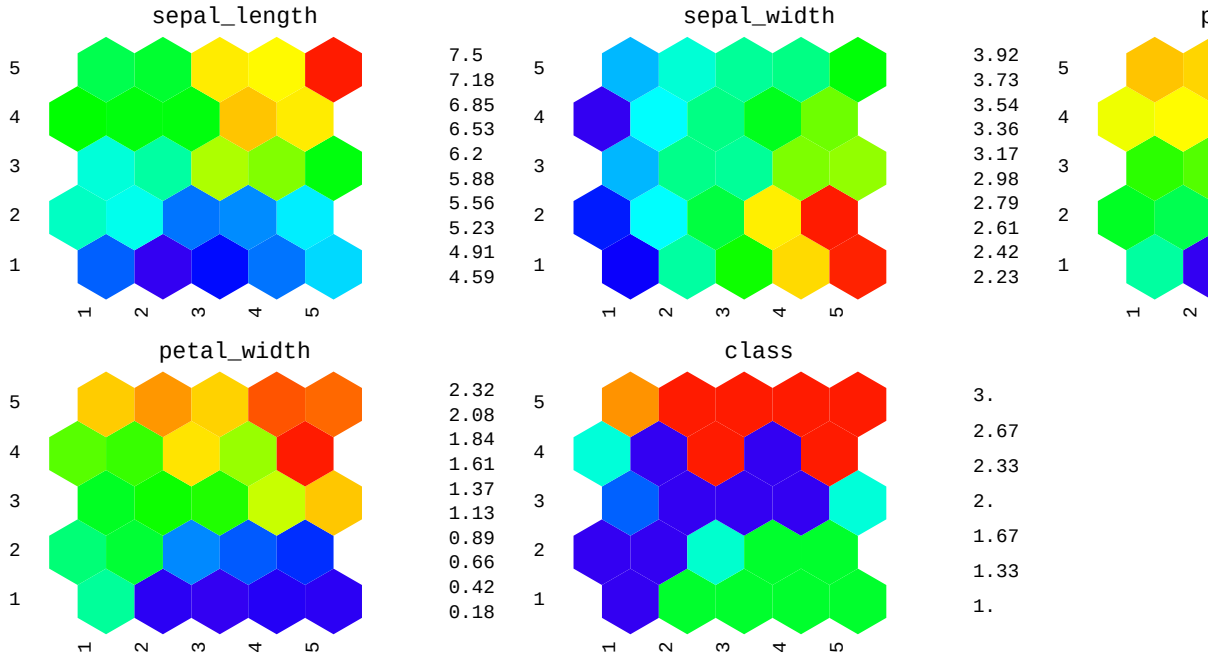
Computing a SOM

Next, we will compute a SOM using `CreateSOM` and train it with the iris data. For training purposes, we take only the first four attributes into account. The fifth attribute, however, is also present in the map and can be used for visualization or for recall.

```
somIris = CreateSOM[
    irisData,
    Size -> {5, 5},
    Cooldown → 0.5,
    Type -> "BATCH",
    Weights → {1, 1, 1, 1, 0}
  ];
```

To visualize the distribution of the data, we create a simple plot using the command `PlotSOMHexRaster`.

```
PlotSOMHexRaster[somIris, DisplayColumns → 3, ImageSize → 800]
```



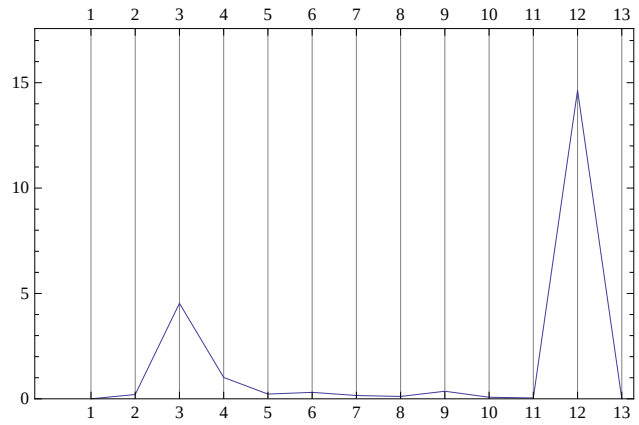
Each grid point in these plots represents a node which can be thought of as a typical sample of the original data. Neighboring points are more similar than far distant ones.
A grid point always belongs to all dimensions, i.e., the top left corner is the same grid point in all plots. This fact allows us to draw conclusions about the coherence of attributes by simply comparing their SOM projections. For example, looking at the SOM projections on sepal_length and petal_length, we see that the bottom right corner in both plots is colored blue. This allows us to draw the conclusion, "If the sepal length is small, then the petal length is also likely to be small". Furthermore, we can see that the class attribute is well clustered and behaves similar to the petal attributes. This indicates that there is a good chance to predict the class membership of an iris flower by inspecting its petal length and width only.

■ Computing Clusters with WARD

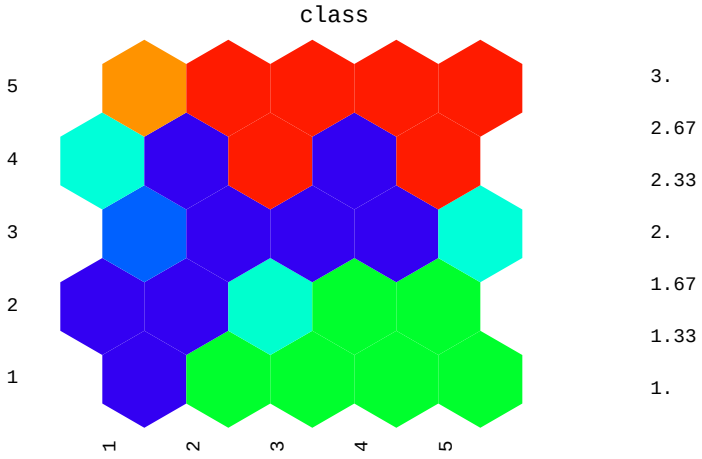
To identify groups of similarity in the data, we will first use the WARD method. This method also allows us to identify the optimal number of clusters.

```
centersWARD = CreateWARD[
    somIris,
    Size → 1,
    Weights → {1, 1, 1, 1, 1}
  ];
```

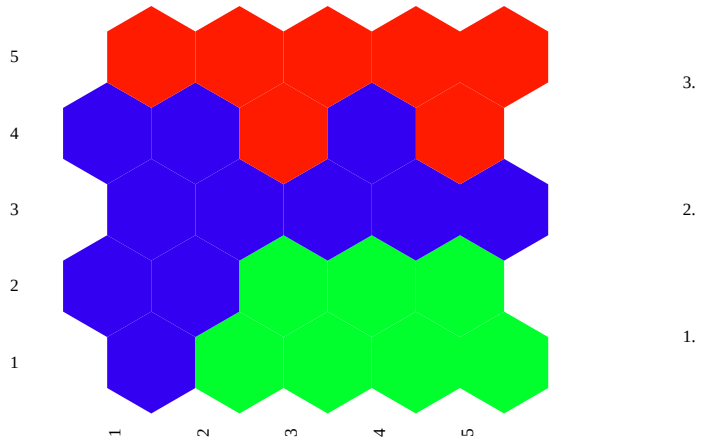
```
wardIndex = ("WIndex" /. centersWARD)[[1]];
PlotLineGraph[wardIndex, PlotRegion → {{0.05, 0.95}, {0.05, 0.95}}]
```




```
centersWARD = CreateWARD[
    somIris,
    Size → 3,
    Weights → {1, 1, 1, 1, 1}
  ];
PlotSOMHexRaster[somIris, 5]
Null
PlotSOMHexRaster[somIris, centersWARD]
```



Clustering with 3 clusters



We can see that the clustering correlates very well with the class attribute.

Create descriptions

If we now create a rule base which describes the clusters, we already get a pretty good description of the data with regard to the class attribute.

```
testPreds = CreatePredicates[irisData, {1, 2, 3, 4}, 3];
testVars = DefPredicateVars[testPreds];

descr = CreateClusterDescriptions[somIris, centersWARD, testVars, MaxRules → 3];
PrintDescriptions[descr, Info → False]
```

Class	{}	Condition
Is_Class1	⇐	petal_length_Is_M
		petal_width_Is_M
		petal_length_Is_M && sepal_length_Is_H
		petal_width_Is_M && sepal_width_Is_L
Is_Class2	⇐	petal_length_Is_L
Is_Class3	⇐	petal_length_Is_H
		petal_width_Is_H

Discrete Data Set: Animals

In this example, we will analyze a data set providing attributes of different animals where each attribute is expressed by a Boolean value.

Set up data

```
animalData = Import["animals.data", "Table"];

TableForm[animalData, TableSpacing → {1, 1}]
```

animal	small	medium	big	2legs	4legs	hair	hooves	mane	feathers	hunt	run	fly	swim	cla
hen	1	0	0	1	0	0	0	0	1	0	0	0	0	1
goose	1	0	0	1	0	0	0	0	1	0	0	1	1	1
hawk	1	0	0	1	0	0	0	0	1	1	0	1	0	1
fox	0	1	0	0	1	1	0	0	0	1	0	0	0	2
dog	0	1	0	0	1	1	0	0	0	0	1	0	0	2
tiger	0	0	1	0	1	1	0	0	0	1	1	0	0	2
lion	0	0	1	0	1	1	0	1	0	1	1	0	0	2
dove	1	0	0	1	0	0	0	0	1	0	0	1	0	1
horse	0	0	1	0	1	1	1	1	0	0	1	0	0	3
owl	1	0	0	1	0	0	0	0	1	1	0	1	0	1
zebra	0	0	1	0	1	1	1	1	0	0	1	0	0	3
wolf	0	1	0	0	1	1	0	1	0	1	1	0	0	2
cat	1	0	0	0	1	1	0	0	0	1	0	0	0	2
duck	1	0	0	1	0	0	0	0	1	0	0	1	1	1
cow	0	0	1	0	1	1	1	0	0	0	0	0	0	3
eagle	0	1	0	1	0	0	0	0	1	1	0	1	0	1

From this table, we want to extract the animal labels, the feature labels and the data (without the class).

```
{m, n} = Dimensions[animalData];
animalLabels = animalData[[Range[2, m], 1]];
featureLabels = animalData[[1, Range[2, n - 1]]];
trainData = DataSet[animalData[[Range[2, m], Range[2, n - 1]]], featureLabels];
testData = trainData;
```

Clustering

In order to identify coherent groups of animals, we first compute a clustering and then create a set of descriptions for these clusters.

```
clustersData = CreateWARD[
   trainData,
   Size → 4
  ];
```

```
testPreds = CreatePredicatesBin[trainData, Range[Length[featureLabels]]];
testVars = DefPredicateVars[testPreds];
```

To create a set of descriptions for the clusters, we employ the method `CreateClusterDescriptions`.

```
descr = CreateClusterDescriptions[trainData, clustersData, testVars];
PrintDescriptions[descr, Info → False]
```

Class	{}	Condition
Is_Class1	⇐	Is_4legs && Is_hunt
Is_Class2	⇐	{}
Is_Class3	⇐	Is_2legs && Is_big
Is_Class4	⇐	Is_small && Is_medium

To compare these descriptions with the animals in each cluster, we can recall the labels (animal names) belonging to each cluster using `GetClusterLabels`.

```
GetClusterLabels[clustersData, animalLabels] // TableForm
```

```
Class1 → {hen, goose, dove, duck}
Class2 → {hawk, owl, eagle}
Class3 → {fox, dog, wolf, cat}
Class4 → {tiger, lion, horse, zebra, cow}
```

■ Creating a SOM

We can also compute a SOM to find out similarities between the considered animals and afterwards assign the animal names to the corresponding parts of the SOM using the commands `GetSOMLabels` and `PrintLabeledSOM`.

```
somSize = {6, 6};
```

```
somAnimals = CreateSOM[trainData, Size → somSize, Type → "BATCH"];
```

```
somLabels = GetSOMLabels[somAnimals, testData, animalLabels];
```

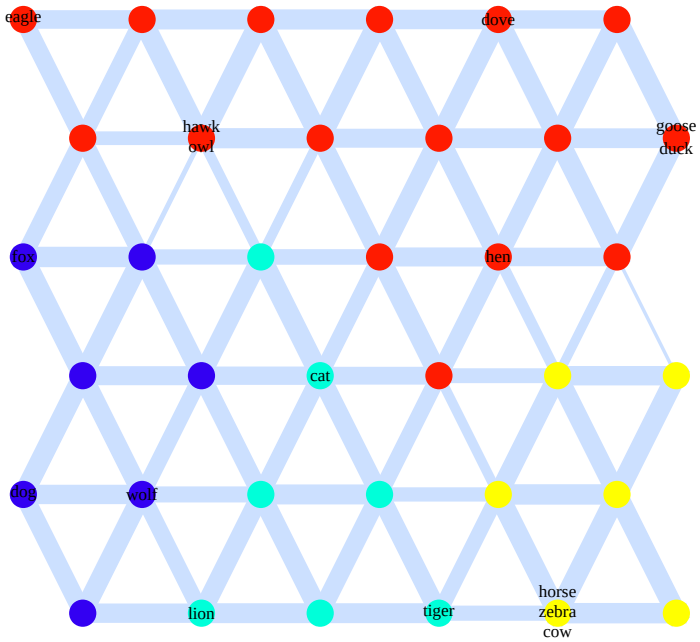
```
PrintLabeledSOM[somLabels]
```

{}	lion	{}	tiger	horse zebra cow	{}
dog	wolf	{}	{}	{}	{}
{}	{}	cat	{}	{}	{}
fox	{}	{}	{}	hen	{}
{}	hawk owl	{}	{}	{}	goose duck
eagle	{}	{}	{}	dove	{}

Now that we have projected the animals onto the two-dimensional grid, we can compute a clustering. We decide to compute 4 clusters.

```
centersWARD = CreateWARD[
    somAnimals,
    Size → 4
  ];
```

```
$TextStyle = {FontWeight → "Bold", FontSize → 16};
Show@Graphics[{
    Hue[0.6, 0.2, 1],
    PlotSOMUGrid[somAnimals, Goal → centersWARD,
     Thickness → 0.03, Labels → somLabels, PointSize → 0.04]
  }, PlotRange → All]
Null
Clear[$TextStyle];
```




```
GetClusterLabels[centersWARD, somLabels] // TableForm
```

```
Class1 → {dog, wolf, fox}
Class2 → {lion, tiger, cat}
Class3 → {horse, zebra, cow}
Class4 → {hen, hawk, owl, goose, duck, eagle, dove}
```

■ Creating Descriptions for SOM Clusters

Our main concern is to identify common properties of animals belonging to the same cluster. We obtain the interesting properties by creating a set of descriptions.

```
testPreds = CreatePredicatesBin[somAnimals, Range[Length[featureLabels]]];
testVars = DefPredicateVars[testPreds];
```

```
descr = CreateClusterDescriptions[somAnimals, centersWARD, testVars];
PrintDescriptions[descr, Info → False]
```

Class	{}	Condition
Is_Class1	⇐	Is_small && Is_big && Is_2legs
Is_Class2	⇐	Is_2legs && Is_medium && Is_hooves
Is_Class3	⇐	{}
Is_Class4	⇐	Is_4legs

We can see that above descriptions are more detailed than the previous ones. This is due to the fact that the SOM also contains nodes to which no animals are assigned at all. These nodes, however, are also part of the descriptions.

Application Example: Project Portfolio

We now want to give an example where we will analyze the project portfolio of a software company. For that purpose, the project managers have been asked to describe their projects with respect to a list of 240 characteristic topics. In total, we collected the descriptions of 31 projects.

Set up Data

```
projDataRaw = Import["projCharacteristic.txt", "Table"];
tmpData = Transpose[projDataRaw];
projData = tmpData[[Range[2, Length@tmpData], Range[2, Length[First[tmpData]]]]];
projLabels = tmpData[[Range[Length@projData], 1]];
featureLabels = tmpData[[1, Range[2, Length@projDataRaw]]];
Clear[projDataRaw];
Clear[tmpData];
```

Below, the list of project names is shown.

```
projLabels
```

```
{Item, AdvLearn, AgenCom, Ariadne, Avitech, CustWeb, DBAgents, EvalVoiceXML,
 FeatLearn, FeDWare, IABADU, Ideal, InFormMe!SCS, Inspire, MathKnow, MathSoft,
 Matrioshka, MESComponents, Moses, MoveD!, NetBanking.invisibleWire,
 NetBanking.webServices, Oxygen, QModel, RemoteDiagConsulting,
 ScriptAPI, Tome_Warder, VirtMould, MatrixApply, SELF, EDE-Solver}
```

The topics which have been used to describe these projects are actual keywords of the IT branch.

```
featureLabels
```

```
{adaptable, agency, agent, ai, algebra, analysis, apache, api, applets, architecture,
 authentication, authorization, automation, banking, browser, business-to-business,
 calculus, cascading, classes, classification, client-server, com, com+, e-commerce,
 communication, comparison, component, concept, configurable, connectivity,
 content, controller, corba, criteria, customizable, database, datamodel, dcom,
 decentralized, decision, design, development, devices, differential, distributed,
 distribution, documentation, datawarehouse, dynamic, education, efficiency, ejb,
 email, embedded, engineering, enterprise, equations, evaluation, exceptions,
 exploitation, expression, extensible, extension, fat, federation, flexible, formal,
 forms, frames, framework, functions, fuzzy, game, generation, generic, genetic,
 grasshopper, gröbner, gui, hardware, heterogeneous, hierarchically, high-speed,
 html, http, huber, image, implementation, incomplete, inconsistent, infrastructure,
 installation, insurance, integral, integration, intelligent, interaction,
 interface, internet, interpretable, intranet, java, javabeans, javascript,
 jdbc, jsp, dbt, kbt, knowledge, language, large, layer, legacy, level, library,
 linear, linguistic, literature, logical, logicographic, maintenance, management,
 manufacturing, marketplace, mathematica, mathematics, matrix, mechatronic, mes,
 messages, meta, method, machine_learning, mobile, model, model-view-controller,
 module, multiplatform, multi-tier, navigation, .net, network, neural, notation,
 num, numeric, objects, odbc, olap, opc, operating_system, optimization, oracle, pda,
 pdf, performance, pervasive, pilot, pin, planning, platform, platform-independent,
 plc, portal, prediction, pre-processing, presentation, process_automation,
 product, protocol, prototype, prototyping, provider, proving, proxy, queries,
 reactive, real-time, regularization, relational, remote, replication, report,
 requirements, research, retrieval, reusability, reuse, roles, rules, swt,
 sym, scenario, security, self-organizing, semantics, semi-structured, server,
 service, servlet, signal, simplification, simulation, soap, solution, standard,
 state-of-the-art, strategic, strategy, symbolic, test, theorem, theorema,
 theoretical, thin, tier, tomcat, tool, toolbox, touchscreen, tracking, trading,
 traffic, training, transaction, transport, trust, ubiquitous, uml, umts, uncertainty,
 unification, visualization, wap, web, web-based, wireless, wizard, workflow, xml}
```

A typical project description consists of a vector of zeros and ones, where one indicates that the project is concerned with the corresponding topic.

```
projData[[1]]
```

```
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,
 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,
 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0,
 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0,
 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0}
```

■ Creating a SOM

Next, we train a SOM with the given data. As we have only 31 records, we increase the cool-down rate with option Cooldown of `CreateSOM`. (The DataSet::mindiff warnings can be ignored in this case.)

```
trainData = projData;
trainDataSet = Def["softwareProjects", DataSet[trainData, featureLabels]];
testDataSet = DataSet[projData, featureLabels];
somSize = {10, 10};

somProjects = CreateSOM[trainDataSet, Size → somSize];
```

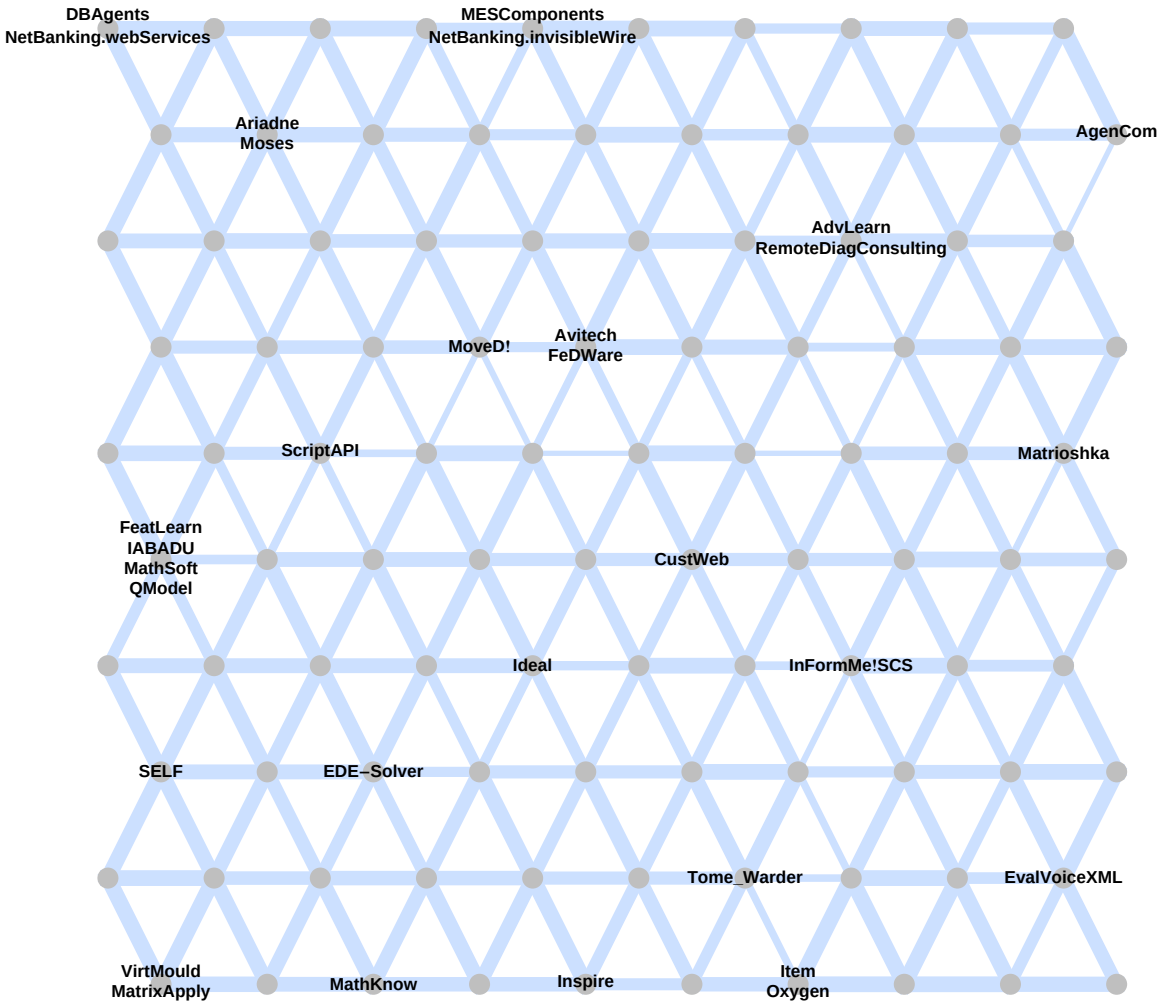
Based on the computed SOM, we can now recall the labels according to our original data set and labels using `GetSOMLabels`. The results are printed using `PrintLabeledSOM`.

```
somLabels = GetSOMLabels[somProjects, testDataSet, projLabels];

PrintLabeledSOM[somLabels]
```

VirtMould MatrixApply	{}	MathKnow	{}	Inspire	{}	
{}	{}	{}	{}	{}	{}	To
SELF	{}	EDE-Solver	{}	{}	{}	
{}	{}	{}	{}	Ideal	{}	
FeatLearn IABADU MathSoft QModel	{}	{}	{}	{}	CustWeb	
{}	{}	ScriptAPI	{}	{}	{}	
{}	{}	{}	MoveD!	Avitech FeDWare	{}	
{}	{}	{}	{}	{}	{}	
{}	Ariadne Moses	{}	{}	{}	{}	
DBAgents NetBanking.webServices	{}	{}	{}	MESComponents NetBanking.invisibleWire	{}	

We make a graphical plot of the SOM, adding labels with the Labels option.

```
Show[Graphics[
  {Hue[0.6, 0.2, 1],
   PlotSOMUGrid[somProjects, Labels → somLabels, PointSize → 0.020]
  },
  PlotRange → All, ImageSize → 600],
 TextStyle → {FontFamily → "Helvetica", FontWeight → "Bold", FontSize → 9}
]
```

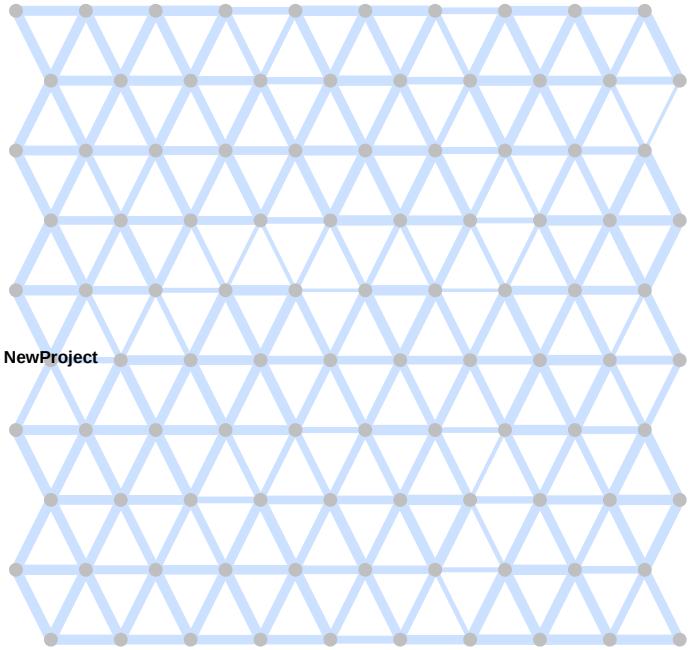


■ Identifying the Position of a New Project

If we want to know how a new project is related to the existing projects (e.g., in order to contact an adequate project manager), we can describe the new project using the previously defined topic list and map it directly onto the SOM. Then we can investigate the neighborhood of the node to find out relations to other projects.

```
newProject = {1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
   0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
   0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0,
   0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0,
   0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1};

newLabels = GetSOMLabels[somProjects, DataSet[{newProject}], {"NewProject"}];
```

```
Show[Graphics[
  {Hue[0.6, 0.2, 1],
   PlotSOMUGrid[somProjects, Labels → newLabels, PointSize → 0.020]},
  BaseStyle → {FontFamily → "Helvetica", FontWeight → "Bold", FontSize → 9},
  PlotRange → All
 ]]
```



■ Projects Concerning a Specific Topic

We can also figure out which projects are related to a specific topic (e.g. knowledge).

```
actPos = Position[featureLabels, "knowledge"][[1, 1]];
```

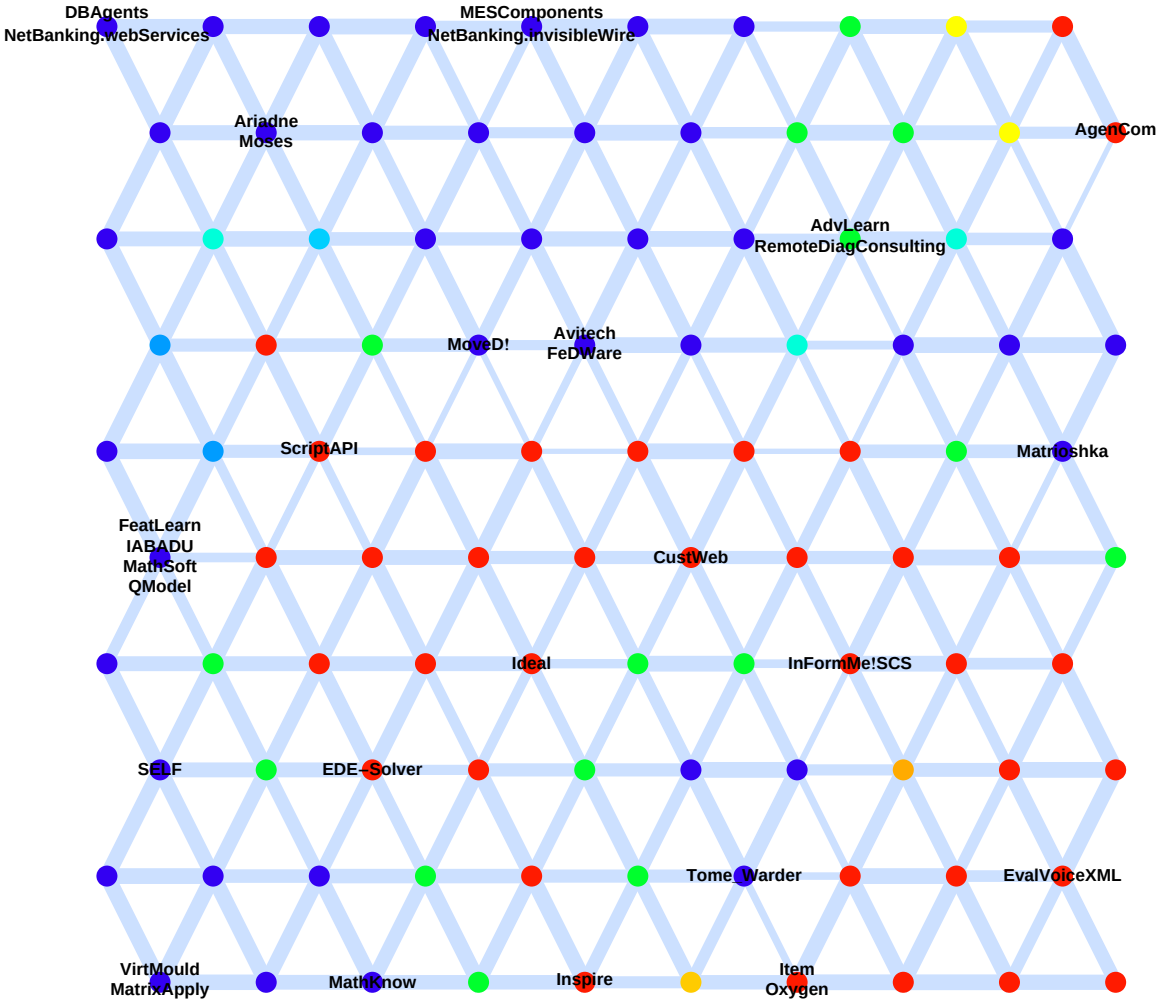
First, we will check which projects are directly related to the specified topic.

```
actProj = Extract[projLabels, Position[projData[[All, actPos]], 1]]
```

```
{Item, AdvLearn, AgenCom, CustWeb, EvalVoiceXML,
 Ideal, InFormMe!SCS, Inspire, Oxygen, ScriptAPI, EDE-Solver}
```

Next, we will plot the SOM and mark projects related to the topic with red and projects which are not related with blue.

```
Show[Graphics[
  {Hue[0.6, 0.2, 1],
   PlotSOMUGrid[somProjects, Goal → actPos, Labels → somLabels, PointSize → 0.020]
  },
  PlotRange → All, ImageSize → 600,
  BaseStyle → {FontFamily → "Helvetica", FontWeight → "Bold", FontSize → 9}
 ]]
```



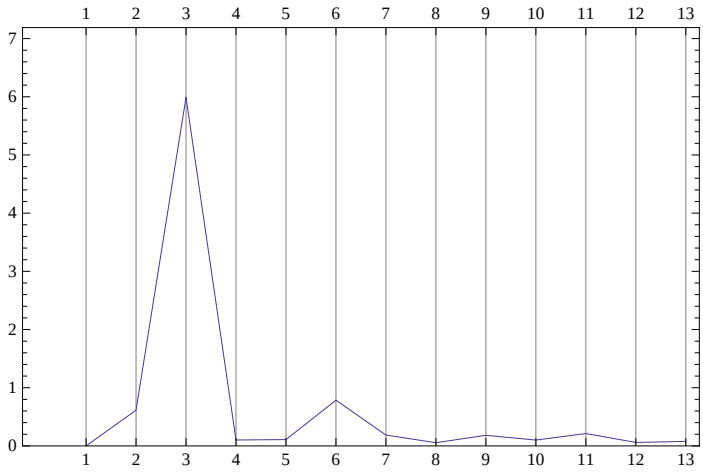
Clusters

To find out how the projects can be grouped together, we employ a WARD clustering.

First, we identify the most useful number of clusters.

```
centersWARD = CreateWARD[
    somProjects,
    Size → 1
  ];
```

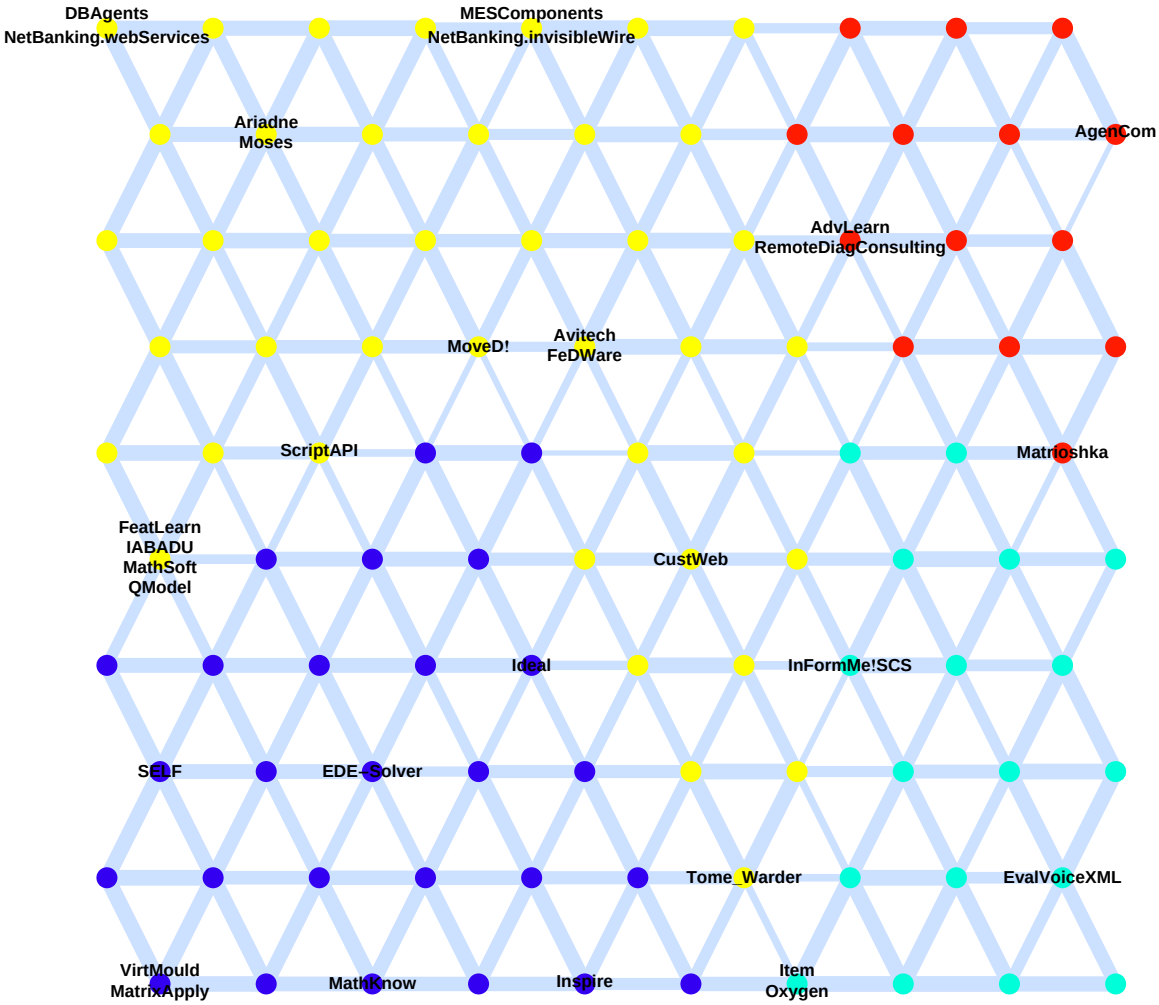
```
wardIndex = ("WIndex" /. centersWARD)[[1]];
PlotLineGraph[wardIndex]
```



We decide to calculate 4 clusters.

```
centersWARD = CreateWARD[
    somProjects,
    Size → 4
  ];
```

```
Show[Graphics[
  {Hue[0.6, 0.2, 1],
   PlotSOMUGrid[somProjects,
    Goal → centersWARD, Labels → somLabels, PointSize → 0.020]
  },
  PlotRange → All, ImageSize → 600,
  BaseStyle → {FontFamily → "Helvetica", FontWeight → "Bold", FontSize → 9}
 ]]
```



Creating Descriptions

To better understand what the obtained clusters are about, we will now compute a set of descriptions for each cluster.

```
testPreds =
  CreatePredicatesBin[somProjects, Range[Length[featureLabels]], PredType → "IS"];
testVars = DefPredicateVars[testPreds];
```

```
Options[CreateClusterDescriptions]
```

```
{Clusters → All, ExpLevel → 10, Logic → Automatic,
 MaxIter → 500, MaxLevel → 10, MaxNegIter → 5, MaxRules → 100,
 MaxTolerance → 0.01, MinConf → 0.8, MinConfFallBack → None,
 MinDetail → 0.1, MinDiff → 0.1, MinSup → 0.1, RulesAlgorithm → CreateMINER}
```

```
descr = CreateClusterDescriptions[somProjects, centersWARD, testVars, MinSup → 0.1];
```

```
PrintDescriptions[descr, Info → False]
```

Class	{}	Condition
Is_Class1	⇐	Is_architecture && Is_semantics
Is_Class2	⇐	Is_communication && Is_algebra && Is_client-server
Is_Class3	⇐	Is_mathematica && Is_decision Is_method && Is_decision Is_method && Is_exploitation
Is_Class4	⇐	Is_performance && Is_evaluation && Is_expression

```
GetClusterLabels[centersWARD, somLabels] // TableForm
```

```
Class1 → {VirtMould, MatrixApply, MathKnow, Inspire, SELF, EDE-Solver, Ideal}
Class2 → {Item, Oxygen, EvalVoiceXML, InFormMe!SCS}
Class3 → {Tome_Warder, FeatLearn, IABADU, MathSoft, QModel, CustWeb, ScriptAPI, MoveD!, Avi
Class4 → {Matrioshka, AdvLearn, RemoteDiagConsulting, AgenCom}
```

Image Analysis

This section very briefly shows how machine learning techniques can be applied to image analysis.

Set-up

```
Needs["mlf`"];
```

First, the image data is read in from a file using RGB color information.

```
img = Import["marille.jpg"];
Show[img]
```

As we are not only interested in RGB colors, we add HLS information (hue, lightness and saturation) to our image data before we transfer the data to *mlf*.

```
imgMatrix = Reverse@ImageData[img, "Byte"];
{m, n, c} = imgSize = Dimensions[imgMatrix]
```

```
{322, 184, 3}
```

```
imgMatrixFull = Flatten[MapIndexed[
    Join[#2, #1, Apply[Apply[EnhanceHLS, RGBtoHLS], #1]] &, imgMatrix, {2}], 1];
```

```
mlfImage = Def[
    "imageData",
    DataSet[imgMatrixFull,
     {"X", "Y", "Red", "Green", "Blue", "Hue", "Lightness", "Saturation"}]
  ];
```

Clustering

First we want to cluster the individual pixels into similar regions, i.e., image segmentation, using the k-means algorithm:

```
noClusters = 4;
```

```
imgClusters = CreateKMeans[
    mlfImage,
    Size → noClusters,
    Weights → {m / n, 1, 1, 1, 1, 1, 1, 1}
  ];
```

Visualizing results

In the following 4 plots, the color red marks pixels which belong to the corresponding cluster (given in the plot label).

```
imgDOMs = GetDOMs[imgClusters];
```

```
TableForm[Table[ListContourPlot[
    Partition[imgDOMs[[All, i]], imgSize[[2]]],
    Frame → False,
    AspectRatio → Automatic,
    ContourStyle → None,
    PlotLabel → "Cluster " <> ToString[i],
    ColorFunction → (CFRed@# &),
    DisplayFunction → Identity,
    ClippingStyle → Automatic
  ], {i, noClusters}], TableDirections → Row]
```

Cluster 1 Cluster 2 Cluster 3 Cluster 4

Now we make a composite image of all clusters with a different color for each cluster.

```
imgDOMs = GetDOMs[imgClusters];
imgRaster = Map[List @@ (GetClusterColor@#) &, imgDOMs];
imgRes = Raster[Partition[imgRaster, n]];
Show[Graphics[imgRes], AspectRatio → m / n]
```

■ Self-organizing Maps (SOMs)

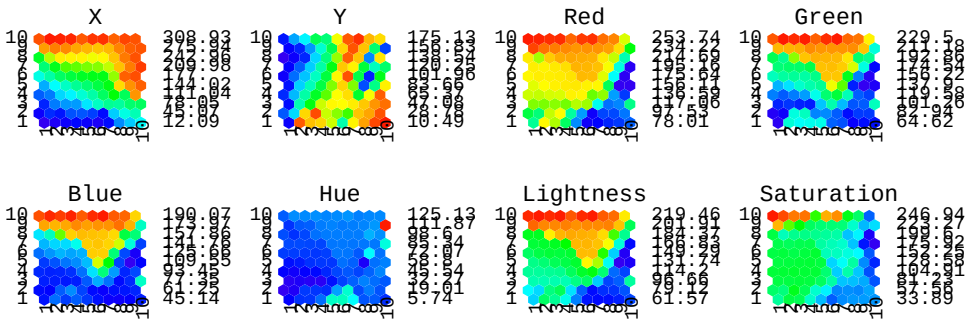
We can also use a SOM to cluster the image. Below, we only use the pixel coordinates and the HLS information to train the SOM (see option Weights).

```
imgSOM = CreateSOM[
    mlfImage,
    Type → BATCH,
    Size → {10, 10},
    Weights → {m / n, 1, 0, 0, 0, 1, 1, 1},
    MaxIter → 100
   ];
```

■ Different visualizations of the SOM

Standard visualization of the trained SOM:

```
PlotSOMHexRaster[imgSOM, Range[8], ImageSize → 500]
```



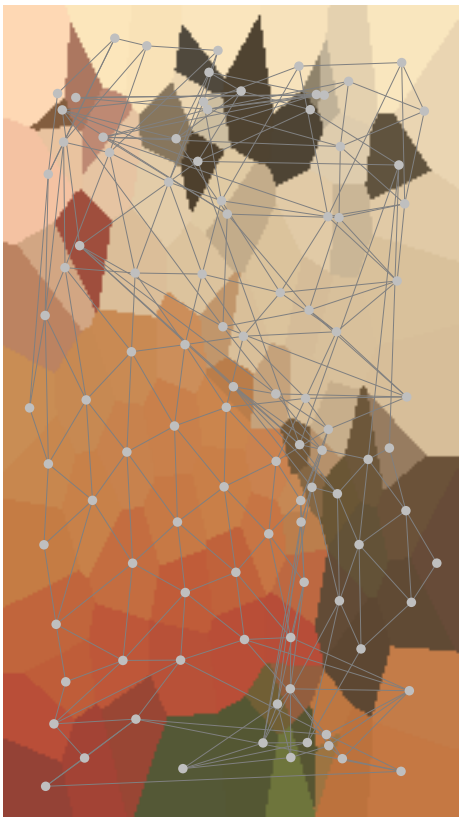
The following plot shows the 10×10 nodes of the SOM where each node is colored with the actual color represented by this node, i.e. {r,g,b} where r, g, and b are the values of the attributes Red, Green, and Blue of the node.

```
imgSOMColor = Raster[
    Partition[
     MLFGetData[imgSOM, All, Range[3, 5]] / 255,
     GetParam[imgSOM, Size][[1, 2]]
    ]
  ];

Show[Graphics[imgSOMColor], AspectRatio → 1]
```

Visualization of the SOM in the (x, y) space (pixel coordinate space): Here we recall the (r, g, b) triples of each pixel from the SOM using the pixel coordinates only (and only the best-matching unit is considered). We plot this color at the pixel coordinate. Additionally, we visualize the grid of the SOM in the pixel coordinate space using `PlotSOMGrid`.

```
imgSOMRecall =
  MLFGetData[RecallBM[mlfImage, imgSOM, Weights → {1, 1, 0, 0, 0, 0, 0, 0}, NSize → 0]];
rgbRecall = imgSOMRecall[[All, Range[3, 5]]];

imgRaster = rgbRecall / 255;

imgRes = Raster[Partition[imgRaster, n]];

gridPlot = Graphics[PlotSOMGrid[
    imgSOM,
    Dims → {2, 1},
    PointSize → 0.02,
    Thickness → 0.001]
  ];
```

```
Show[{Graphics[imgRes], gridPlot}, AspectRatio → m / n]
```



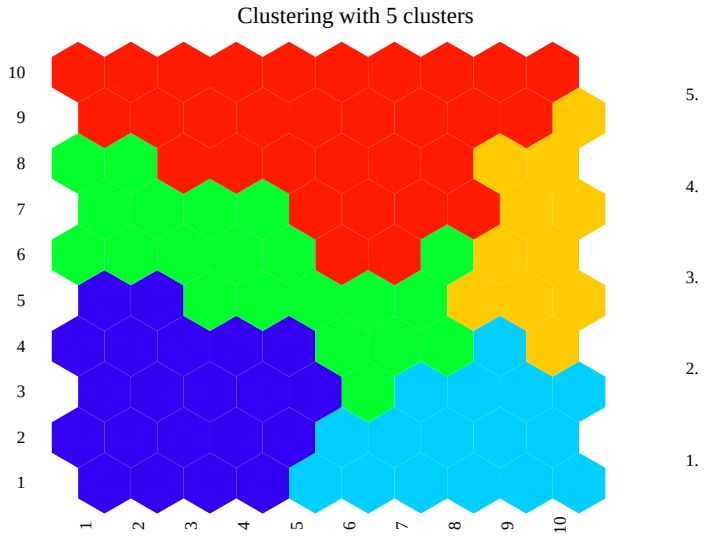
■ Creating a Clustering upon a SOM

Based on the SOM created above, we calculate a WARD clustering:

```
imgClusters = CreateWARD[
    imgSOM,
    Size → 5
  ];
```

We plot the clustering in the SOM grid space:

```
PlotSOMHexRaster[imgSOM, imgClusters]
```



and in the (x, y) space, using a distinct color for each cluster:

```
curDOMs = GetDOMs[imgClusters];
imgSOMRecall = RecallBM[mlfImage, imgSOM, NSize → 1, Index → True];
imgSOMRecClusters = Extract[curDOMs, imgSOMRecall];
imgRaster = Map[List @@ (GetClusterColor@#) &, imgSOMRecClusters];
imgRes = Raster[Partition[imgRaster, n]];
Show[Graphics[imgRes], AspectRatio → m / n]
```

More About

Template for Unsupervised Data Analysis

Related Tutorials

Supervised Data Analysis

Approximation of Numerical Functions

Unsupervised Data Analysis; including image analysis (this tutorial)

Object and Data Handling, Fuzzy Logic Operations, Visualization

Related Wolfram Education Group Courses

XXXX

Tutorial - Basic Functionality of *mlf*

In this chapter, the basic concepts of the ***machine learning framework*** (***mlf***) will be described.

Loading *mlf*

In order to use the *machine learning framework* package in the *Mathematica* front end, the user has to load it in an open *Mathematica* notebook.

```
Needs["mlf`"];
```

This loads the *machine learning framework* package, which contains the algorithmic kernel (written in C++). A link connection to the *mlf* kernel is established. The current link connections can be inspected using the command `Links`:

```
Links[]
```

```
{LinkObject[6eh_shm, 1, 1], LinkObject['/Applications/Mathematica
    8.0.app/SystemFiles/Links/JLink/JLink.app/Contents/MacOS/JavaApplicationStub'
    -init "/tmp/m000001100891", 2, 2],
 LinkObject[hpb_shm, 3, 3], LinkObject[9fc_shm, 4, 4],
 LinkObject[633_shm, 5, 5],
 LinkObject[x3z_shm, 6, 6],
 LinkObject[, 7, 7],
 LinkObject[/Library/Mathematica/Applications/mlf/mlf, 14, 8],
 LinkObject['/Applications/Mathematica
    8.0.app/Contents/MacOS/MathKernel' -subkernel -noinit -mathlink, 867, 9],
 LinkObject['/Applications/Mathematica 8.0.app/Contents/MacOS/MathKernel'
    -subkernel -noinit -mathlink, 868, 10],
 LinkObject['/Applications/Mathematica 8.0.app/Contents/MacOS/MathKernel'
    -subkernel -noinit -mathlink, 869, 11],
 LinkObject['/Applications/Mathematica 8.0.app/Contents/MacOS/MathKernel'
    -subkernel -noinit -mathlink, 870, 12]}
```

The license management of the *machine learning framework* is performed within the *mlf* kernel. As soon as the package is loaded, one license is locked (in case of a floating license) by the current user. The license can only be freed again by terminating the *Mathematica* kernel.

To determine the version number of the *machine learning framework*, a special command is used.

```
MLFVersion[]
```

```
Machine learning framework for Mathematica 8 and 9
2.1.0 for MacOSX (x86_64) (Jun  3 2013)
```

Basic Concepts

Introduction

With *mlf*, *Mathematica*'s functionality is extended in several ways. As *mlf* is designed to work with large sets of data, a dedicated memory concept has been established. Additionally, a new mechanism for handling objects has been introduced. This is done due to the need of increased flexibility, speed, and type safety.

Basically, there are the following three types of objects in *mlf*:
* basic data types,
* operations, and
* algorithms.

The *basic data types* are used to store information. *Operations* can be used to manipulate or combine existing objects. These two object types are handled in a symbolic manner, unless specified otherwise. Finally, *algorithms* can be used to perform complex computations.

Handling objects

Typical objects within *mlf* are numeric and logical values, operations, or data sets. For most objects, commands are available to define them.

```
FAnd[0.5, 0.4]
```

```
Obj`LogicAnd[Obj`LogicConst[0.5], Obj`LogicConst[0.4]]
```

In this example, a new *fuzzy-and* object with two constant (logical) operands has been defined. The next example defines a numeric vector and assigns it to the *Mathematica* variable *obj*.

```
obj = NVector[Range[10]]
```

```
Obj`RealVector[{1., 2., 3., 4., 5., 6., 7., 8., 9., 10.}]
```

obj

```
Obj`RealVector[{1., 2., 3., 4., 5., 6., 7., 8., 9., 10.}]
```

As already mentioned, *mlf* uses its own memory management independent from *Mathematica*. With the command `Def`, it is possible to create a named object in *mlf* memory. After the definition, we can work with the object via its name. The name of an object is also called the symbolic reference to that object.

```
myFirstNamedObjVar = Def["MyFirstNamedObj", NVal[0.5]]
```

```
Name`MyFirstNamedObj
```

To work with the object we just created in *mlf* memory, the command `GetObject` is used. This command takes as argument the name of the object we want to refer to. When we refer to an *mlf* object via its name, we can also say that we resolve the symbolic reference to that object. The name of the object can be specified either as a string or as a symbol in the dedicated namespace *Name*`. Therefore, the following three statements are equivalent and each refers to the same previously defined object residing in *mlf* memory.

```
GetObject["MyFirstNamedObj"]
```

```
Obj`RealConst[0.5]
```

```
GetObject[myFirstNamedObjVar]
```

```
Obj`RealConst[0.5]
```

```
GetObject[Name`MyFirstNamedObj]
```

```
Obj`RealConst[0.5]
```

`DelObject` is applied to delete an object from *mlf* memory. As with `GetObject`, the name can be specified either as a string or as a symbol in the dedicated namespace Name`. The command `DelObject` command returns 1 if the object existed and could be deleted; 0 is returned if no object with the specified name could be found in *mlf* memory.

```
DelObject["MyFirstNamedObj"]
```

```
1
```

The following two variants are equivalent to the preceding expression, but since the object has just been deleted, it cannot be done again and 0 is returned.

```
DelObject[myFirstNamedObjVar]
DelObject[Name`MyFirstNamedObj]
```

```
0
```

```
0
```

If one tries to resolve a symbolic reference for which no object resides in *mlf* memory, an error message is generated.

```
GetObject["NotThere"]
```

```
mlf::error : The object 'NotThere' does not exist in the kernel..
```

```
$Failed
```

```
GetObject[notThereVar]
```

```
mlf::error : The object 'ereVar' does not exist in the kernel..
```

```
$Failed
```

```
GetObject[Name`NotThere]
```

```
mlf::error : The object 'NotThere' does not exist in the kernel..
```

```
$Failed
```

Data Sets

Creating a data set

In order to store large amounts of data together with some additional information (e.g., column headers), a special container called *data set* is introduced.

As a data set is basically only a two-dimensional matrix of numeric (or categorical) values, it can be directly derived from a matrix applying the command `DataSet` with the data matrix as the first and the column headers as the optional second argument.

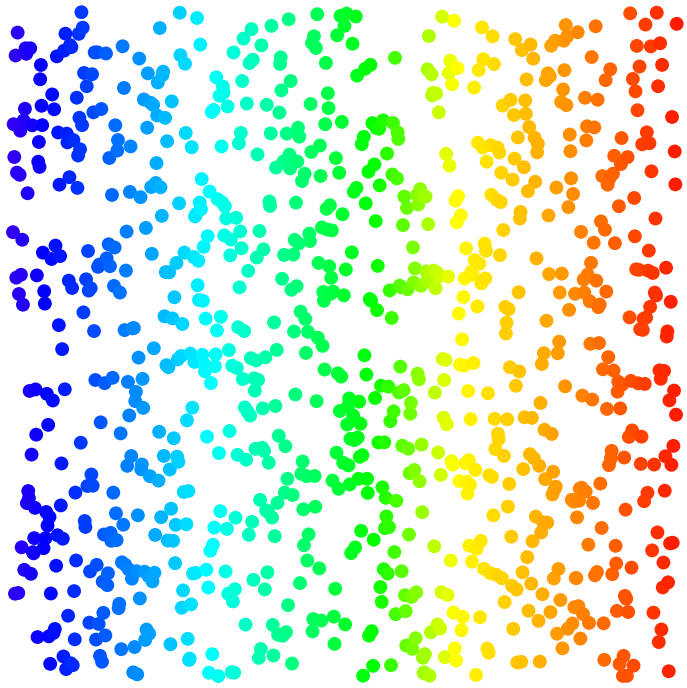
As we create a symbolic reference to the data set, we do not need to take care of size and memory consumption of the data structure.

```
data = Table[{r = N[i / 1000], r + 0.5, Random[]}, {i, 1, 1000}];
myDataSet = CreateDataSet[data, {"x1", "x2", "x3"}]
```

```
Name`DataSet12
```

To plot a data set, a special function called `PlotAttributes` is available.

```
Show[Graphics[PlotAttributes[
    myDataSet,
    Dims → {1, 3},
    Goal → 2,
    PointSize → 0.02
  ]]]
```



Load data from a file

Usually data will be available from a file and not be created within *Mathematica*. If the data are already in a suitable format, a data set can be created from a file with one single command:

```
irisData = LoadData["iris.txt"];
```

To split the data into training and test data, the fraction of data to be used for the training set is given by the option `TrainPart`:

```
{trainData, testData} = LoadData["iris.txt", TrainPart → 0.7];
```

Alternatively, to load a given data file into *Mathematica* and edit the data before creating a data set, the *Mathematica* function `Import` can be used.

```
dataRaw = Import["iris.txt", "Table"];
```

```
dataRaw[[Range[10]]] // TableForm
```

sepal_length	sepal_width	petal_length	petal_width	class
5.1	3.5	1.4	0.2	Iris-versicol
4.9	3.	1.4	0.2	Iris-versicol
4.7	3.2	1.3	0.2	Iris-versicol
4.6	3.1	1.5	0.2	Iris-versicol
5.	3.6	1.4	0.2	Iris-versicol
5.4	3.9	1.7	0.4	Iris-versicol
4.6	3.4	1.4	0.3	Iris-versicol
5.	3.4	1.5	0.2	Iris-versicol
4.4	2.9	1.4	0.2	Iris-versicol

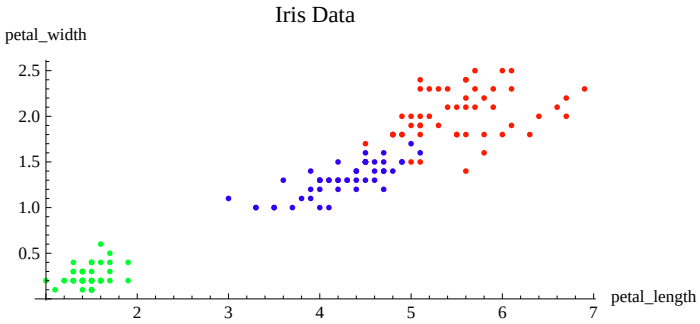
We have now loaded the data file, which contains the column headers in the first row. To define a new data set with this data, we have to extract the first row to get hold of the column names.

```
headers = First[dataRaw];
data = Rest[dataRaw];
```

```
myDataSet = DataSet[data, headers];
```

With `PlotAttributes` , it is very straight forward to visualize the data. As we know that the fifth row contains the classification of the iris flowers, we are able to color the output accordingly.

```
Show[Graphics[PlotAttributes[myDataSet, Dims → {3, 4}, Goal → 5, PointSize → 0.01],
  Axes → True, AxesLabel → headers[[{3, 4}]], PlotLabel -> "Iris Data"]]
```



Now we can start to analyze this data set using *supervised* (see chapter 2 and 3) or *unsupervised* (see chapter 4) methods.

Starting with *mlf* 1.2, also direct ODBC access to databases and Microsoft Excel files is supported, which, however, requires *Mathematica* 5.0 or higher and .NET installed on your machine - see help to `ImportODBC`.

Fuzzy Logic related

■ Defining partitions for the data set

To define a set of predicates (a fuzzy partition) for each dimension of the data set, we do not need to specify all sets and predicates manually, but we can use the predefined commands `CreatePartition` and `CreatePredicates`. The first command returns a list of the form {fuzzySets, fuzzyPredicates}, while the latter only returns a list of predicates.
Predicates are generated by applying the operations *is*, *is-at-least*, and *is-at-most* to the fuzzy sets found. Afterwards, those predicates are removed whose support (number of samples fulfilling the predicate) is too small.

There are various parameters which influence how the sets and predicates are defined. You can display them with Options.

```
Options[CreatePartition]
```

```
{NoOfSets → 5, Overlap → 0.2, Width → 1., PredType → ALL,
 SetType → PWL, MinSup → 0.025, Border → False, Linguistic → True,
 SeparateSets → False, SetNames → Automatic, CutPoints → Automatic,
 Minimum → Automatic, Maximum → Automatic, Confidence → 0.02, Labels → {},
 MaxLengthOfCombinations → 1, CascadingCombinations → False, OutputColumn → 1}
```

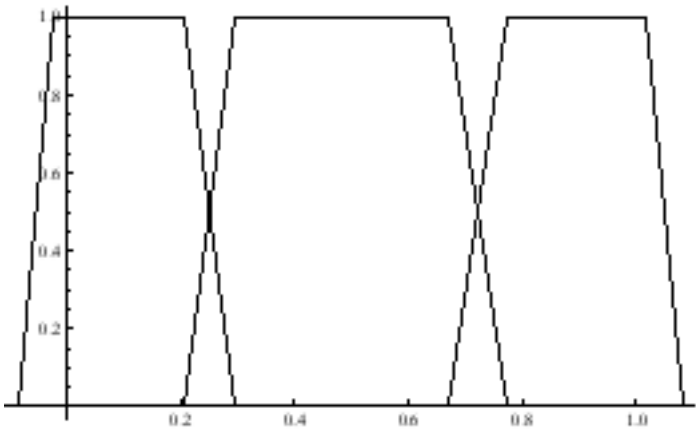
Now we create a three-set partition for the third attribute of our data set. Since we want to apply piecewise-linear fuzzy sets, we specify PWL as set type (for exponential fuzzy sets use EXP as set type). With the option

Border→True, we can specify that we want the fuzzy partition to include the lower and upper bound of our domain, too.

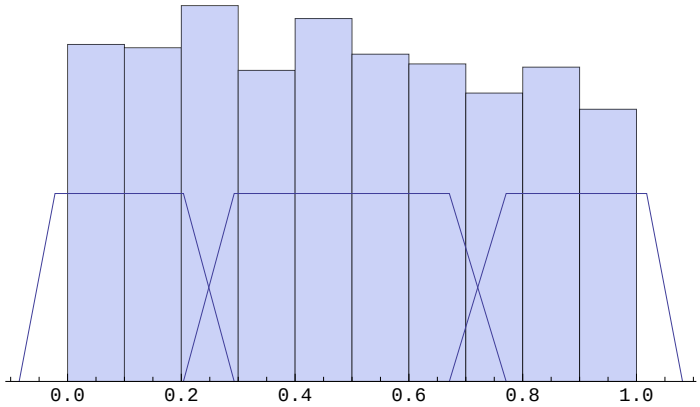
```
data = Table[{r = N[i / 1000], r + 0.5, Random[]}, {i, 1, 1000}];
myDataSet = CreateDataSet[data, {"Col1", "Col2", "Col3" }];
res = CreatePartition[myDataSet, 3, 3, SetType -> "PWL", Border → True];
```

The following fuzzy sets have been created:

```
plotData = PlotFuzzySet[res]
```




```
plotData=PlotFuzzySetHistogram[res,MLFGetData[myDataSet,All,3]]
```



And the following predicates have been defined:

```
res[[2, All, 1]] // TableForm
```

```
Col3_IsAtLeast_M
Col3_IsAtMost_M
Col3_Is_H
Col3_Is_L
Col3_Is_M
```

■ Create predicates with separate fuzzy sets

Fuzzy sets and fuzzy predicates are closely related. When a fuzzy predicate is automatically created, it usually contains a copy of the underlying fuzzy set. However, if one wants to be able to modify the fuzzy set afterwards, the fuzzy set has to be defined outside the predicate. This can be done by applying `CreatePartition` with option SeparateSets set to True. This approach is particularly useful when a fuzzy controller is constructed first and its fuzzy sets are tuned afterwards.

```
testPartitions = CreatePartition[
   myDataSet, 3, 3,
   SeparateSets → True
  ];
```

As we can see below, the original predicate contains the complete fuzzy set.

```
res[[2, 1]]
```

```
Col3_IsAtLeast_M → Obj`LogicDataPred[2, AL,
  Obj`FuzzySetPWL[{Obj`Pair[0.203674, 0.], Obj`Pair[0.292897, 1.],
    Obj`Pair[0.671051, 1.], Obj`Pair[0.770904, 0.]}]]
```

In contrast, the newly created predicate only contains a reference to the corresponding fuzzy partition and the fuzzy set.

```
testPartitions[[2, 1]]
```

```
Col3_IsAtLeast_M → Obj`LogicDataPred[2, AL, Obj`FuzzySetVar[Col3_Partition, M]]
```

A list of all fuzzy sets created for a given dimension is stored in a variable named like the column appended with postfix _Partition. You can get a list of all partition names using the command `GetPartitionNames` with the list of column headers as argument.

```
GetObject["Col3_Partition"]
```

```
Obj`FuzzySetMap[{H → Obj`FuzzySetPWL[{Obj`Pair[0.631538, 0.],
      Obj`Pair[0.694997, 1.], Obj`Pair[0.97061, 1.], Obj`Pair[1.03364, 0.]}],
  L → Obj`FuzzySetPWL[{Obj`Pair[-0.0377992, 0.], Obj`Pair[0.0252264, 1.],
      Obj`Pair[0.311925, 1.], Obj`Pair[0.376157, 0.]}],
  M → Obj`FuzzySetPWL[{Obj`Pair[0.311925, 0.], Obj`Pair[0.376157, 1.],
      Obj`Pair[0.631538, 1.], Obj`Pair[0.694997, 0.]}]}]
```

```
GetMembers[]
```

```
{AGE_IsAtLeast_H, AGE_IsAtLeast_L, AGE_IsAtLeast_M, AGE_IsAtMost_H,
 AGE_IsAtMost_L, AGE_IsAtMost_M, AGE_Is_H, AGE_Is_L, AGE_Is_M, AGE_Is_VH,
 AGE_Is_VL, AlcalinityOfAsh_IsAtLeast_H, AlcalinityOfAsh_IsAtLeast_L,
 AlcalinityOfAsh_IsAtLeast_M, AlcalinityOfAsh_IsAtMost_H, AlcalinityOfAsh_IsAtMost_L,
 AlcalinityOfAsh_IsAtMost_M, AlcalinityOfAsh_Is_H, AlcalinityOfAsh_Is_L,
 AlcalinityOfAsh_Is_M, AlcalinityOfAsh_Is_VH, AlcalinityOfAsh_Is_VL,
 Alcohol_IsAtLeast_H, Alcohol_IsAtLeast_L, Alcohol_IsAtLeast_M, Alcohol_IsAtMost_H,
 Alcohol_IsAtMost_L, Alcohol_IsAtMost_M, Alcohol_Is_H, Alcohol_Is_L, Alcohol_Is_M,
 Alcohol_Is_VH, Alcohol_Is_VL, Ash_IsAtLeast_H, Ash_IsAtLeast_L, Ash_IsAtLeast_M,
 Ash_IsAtMost_H, Ash_IsAtMost_L, Ash_IsAtMost_M, Ash_Is_H, Ash_Is_L, Ash_Is_M,
 Ash_Is_VH, Ash_Is_VL, B_IsAtLeast_H, B_IsAtLeast_L, B_IsAtLeast_M, B_IsAtMost_H,
 B_IsAtMost_L, B_IsAtMost_M, B_Is_H, B_Is_L, B_Is_M, B_Is_VH, B_Is_VL, CHAS_Is_0,
 CHAS_Is_1, CRIM_IsAtLeast_H, CRIM_IsAtLeast_L, CRIM_IsAtLeast_M, CRIM_IsAtMost_H,
 CRIM_IsAtMost_L, CRIM_IsAtMost_M, CRIM_Is_H, CRIM_Is_L, CRIM_Is_M, CRIM_Is_VH,
 CRIM_Is_VL, Childs_IsAtLeast_H, Childs_IsAtLeast_L, Childs_IsAtMost_H,
 Childs_IsAtMost_L, Childs_IsAtMost_M, Childs_Is_H, Childs_Is_L, Childs_Is_VH,
 Childs_Is_VL, Class_Is_'democrat', Class_Is_'republican', Class_Is_C1, Class_Is_C2,
 Class_Is_C3, Col3_Partition, ColorIntensity_IsAtLeast_H, ColorIntensity_IsAtLeast_L,
 ColorIntensity_IsAtLeast_M, ColorIntensity_IsAtMost_H, ColorIntensity_IsAtMost_L,
 ColorIntensity_IsAtMost_M, ColorIntensity_Is_H, ColorIntensity_Is_L,
 ColorIntensity_Is_M, ColorIntensity_Is_VH, ColorIntensity_Is_VL, CreditRequest_Is_no,
 CreditRequest_Is_yes, DIS_IsAtLeast_H, DIS_IsAtLeast_L, DIS_IsAtLeast_M,
 DIS_IsAtMost_H, DIS_IsAtMost_L, DIS_IsAtMost_M, DIS_Is_H, DIS_Is_L, DIS_Is_M,
 DIS_Is_VH, DIS_Is_VL, DataSet1, DataSet10, DataSet10Equalized1, DataSet11,
 DataSet12, DataSet13, DataSet2, DataSet21, DataSet22, DataSet23, DataSet3, DataSet4,
 DataSet5, DataSet6, DataSet7, DataSet8, DataSet9, EqualizeHistogramDataSet1,
 FamiliyStatus_Is_female, FamiliyStatus_Is_married, FamiliyStatus_Is_single,
 FamiliyStatus_Is_widower, Flavanoids_IsAtLeast_H, Flavanoids_IsAtLeast_L,
 Flavanoids_IsAtLeast_M, Flavanoids_IsAtMost_H, Flavanoids_IsAtMost_L,
 Flavanoids_IsAtMost_M, Flavanoids_Is_H, Flavanoids_Is_L, Flavanoids_Is_M,
 Flavanoids_Is_VH, Flavanoids_Is_VL, Hue_IsAtLeast_H, Hue_IsAtLeast_L, Hue_IsAtLeast_M,
 Hue_IsAtMost_H, Hue_IsAtMost_L, Hue_IsAtMost_M, Hue_Is_H, Hue_Is_L, Hue_Is_M,
 Hue_Is_VH, Hue_Is_VL, INDUS_IsAtLeast_H, INDUS_IsAtLeast_L, INDUS_IsAtLeast_M,
 INDUS_IsAtMost_H, INDUS_IsAtMost_L, INDUS_IsAtMost_M, INDUS_Is_H, INDUS_Is_L,
 INDUS_Is_M, INDUS_Is_VH, INDUS_Is_VL, Income_IsAtLeast_H, Income_IsAtLeast_L,
 Income_IsAtLeast_M, Income_IsAtMost_H, Income_IsAtMost_L, Income_IsAtMost_M,
 Income_Is_H, Income_Is_L, Income_Is_M, Income_Is_VH, Income_Is_VL, LSTAT_IsAtLeast_H,
 LSTAT_IsAtLeast_L, LSTAT_IsAtLeast_M, LSTAT_IsAtMost_H, LSTAT_IsAtMost_L,
 LSTAT_IsAtMost_M, LSTAT_Is_H, LSTAT_Is_L, LSTAT_Is_M, LSTAT_Is_VH, LSTAT_Is_VL,
 Magnesium_IsAtLeast_H, Magnesium_IsAtLeast_L, Magnesium_IsAtLeast_M,
 Magnesium_IsAtMost_H, Magnesium_IsAtMost_L, Magnesium_IsAtMost_M,
 Magnesium_Is_H, Magnesium_Is_L, Magnesium_Is_M, Magnesium_Is_VH, Magnesium_Is_VL,
 MalicAcid_IsAtLeast_H, MalicAcid_IsAtLeast_L, MalicAcid_IsAtLeast_M,
 MalicAcid_IsAtMost_H, MalicAcid_IsAtMost_L, MalicAcid_IsAtMost_M, MalicAcid_Is_H,
```

```
MalicAcid_Is_L, MalicAcid_Is_M, MalicAcid_Is_VH, MalicAcid_Is_VL, NOX_IsAtLeast_H,
NOX_IsAtLeast_L, NOX_IsAtLeast_M, NOX_IsAtMost_H, NOX_IsAtMost_L, NOX_IsAtMost_M,
NOX_Is_H, NOX_Is_L, NOX_Is_M, NOX_Is_VH, NOX_Is_VL, Name_Is_Anita_Conte,
Name_Is_Anita_Farner, Name_Is_Anita_Fontana, Name_Is_Anita_Gehrig,
Name_Is_Anita_Gruner, Name_Is_Anita_Hauser, Name_Is_Anita_Indermaur,
Name_Is_Anita_Maurer, Name_Is_Anita_Mieschke, Name_Is_Anita_Sager,
Name_Is_Anita_Schnyder, Name_Is_Anita_Silbermann, Name_Is_Anita_Steinmann,
Name_Is_Anita_Thüring, Name_Is_Anita_Waser, Name_Is_Anita_de_Vargas,
Name_Is_Anna-Rösli_Bachmann, Name_Is_Anna-Rösli_Böckli, Name_Is_Anna-Rösli_Conte,
Name_Is_Anna-Rösli_Gehrig, Name_Is_Anna-Rösli_Gschwind, Name_Is_Anna-Rösli_Hauser,
Name_Is_Anna-Rösli_Maurer, Name_Is_Anna-Rösli_Mieschke, Name_Is_Anna-Rösli_Perrin,
Name_Is_Anna-Rösli_Reber, Name_Is_Anna-Rösli_Steinmann, Name_Is_Anna-Rösli_Thüring,
Name_Is_Anna-Rösli_Weber, Name_Is_Anna-Rösli_de_Vargas, Name_Is_Astrid_Bachmann,
Name_Is_Astrid_Conte, Name_Is_Astrid_Farner, Name_Is_Astrid_Gehrig,
Name_Is_Astrid_Gruner, Name_Is_Astrid_Gschwind, Name_Is_Astrid_Hauser,
Name_Is_Astrid_Indermaur, Name_Is_Astrid_Keller, Name_Is_Astrid_Maurer,
Name_Is_Astrid_Perrin, Name_Is_Astrid_Sager, Name_Is_Astrid_Schneider,
Name_Is_Astrid_Schnyder, Name_Is_Astrid_Silbermann, Name_Is_Astrid_Steinmann,
Name_Is_Astrid_Waser, Name_Is_Cornelia_Conte, Name_Is_Cornelia_Dubach,
Name_Is_Cornelia_Farner, Name_Is_Cornelia_Gehrig, Name_Is_Cornelia_Gschwind,
Name_Is_Cornelia_Hauser, Name_Is_Cornelia_Keller, Name_Is_Cornelia_Maurer,
Name_Is_Cornelia_Mieschke, Name_Is_Cornelia_Reber, Name_Is_Cornelia_Silbermann,
Name_Is_Cornelia_Steinmann, Name_Is_Cornelia_Thüring, Name_Is_Cornelia_Waser,
Name_Is_Cornelia_Weber, Name_Is_Cornelia_de_Vargas, Name_Is_Elisabeth_Böckli,
Name_Is_Elisabeth_Conte, Name_Is_Elisabeth_Dubach, Name_Is_Elisabeth_Gschwind,
Name_Is_Elisabeth_Indermaur, Name_Is_Elisabeth_Keller, Name_Is_Elisabeth_Maurer,
Name_Is_Elisabeth_Mieschke, Name_Is_Elisabeth_Reber, Name_Is_Elisabeth_Schneider,
Name_Is_Elisabeth_Silbermann, Name_Is_Elisabeth_Thüring, Name_Is_Elisabeth_Weber,
Name_Is_Elisabeth_de_Vargas, Name_Is_Ernst_Conte, Name_Is_Ernst_Farner,
Name_Is_Ernst_Fontana, Name_Is_Ernst_Gehrig, Name_Is_Ernst_Keller,
Name_Is_Ernst_Maurer, Name_Is_Ernst_Mieschke, Name_Is_Ernst_Reber,
Name_Is_Ernst_Sager, Name_Is_Ernst_Schnyder, Name_Is_Ernst_Silbermann,
Name_Is_Ernst_Thüring, Name_Is_Ernst_Weber, Name_Is_Ernst_de_Vargas,
Name_Is_Franz_Bachmann, Name_Is_Franz_Conte, Name_Is_Franz_Dubach,
Name_Is_Franz_Farner, Name_Is_Franz_Gehrig, Name_Is_Franz_Gschwind,
Name_Is_Franz_Indermaur, Name_Is_Franz_Keller, Name_Is_Franz_Perrin,
Name_Is_Franz_Reber, Name_Is_Franz_Schnyder, Name_Is_Franz_Thüring,
Name_Is_Franz_Waser, Name_Is_Franz_Weber, Name_Is_Gerhard_Bachmann,
Name_Is_Gerhard_Conte, Name_Is_Gerhard_Farner, Name_Is_Gerhard_Fontana,
Name_Is_Gerhard_Gehrig, Name_Is_Gerhard_Gruner, Name_Is_Gerhard_Gschwind,
Name_Is_Gerhard_Hauser, Name_Is_Gerhard_Indermaur, Name_Is_Gerhard_Keller,
Name_Is_Gerhard_Mieschke, Name_Is_Gerhard_Perrin, Name_Is_Gerhard_Reber,
Name_Is_Gerhard_Schneider, Name_Is_Gerhard_Steinmann, Name_Is_Gerhard_Thüring,
Name_Is_Gerhard_Weber, Name_Is_Gerhard_de_Vargas, Name_Is_Henri_Bachmann,
Name_Is_Henri_Böckli, Name_Is_Henri_Conte, Name_Is_Henri_Dubach,
Name_Is_Henri_Hauser, Name_Is_Henri_Indermaur, Name_Is_Henri_Keller,
Name_Is_Henri_Mieschke, Name_Is_Henri_Perrin, Name_Is_Henri_Sager,
Name_Is_Henri_Schnyder, Name_Is_Henri_Thüring, Name_Is_Henri_Weber,
Name_Is_Louise_Bachmann, Name_Is_Louise_Böckli, Name_Is_Louise_Conte,
Name_Is_Louise_Dubach, Name_Is_Louise_Farner, Name_Is_Louise_Fontana,
Name_Is_Louise_Gehrig, Name_Is_Louise_Hauser, Name_Is_Louise_Keller,
Name_Is_Louise_Mieschke, Name_Is_Louise_Perrin, Name_Is_Louise_Sager,
Name_Is_Louise_Schneider, Name_Is_Louise_Schnyder, Name_Is_Louise_Silbermann,
Name_Is_Louise_Steinmann, Name_Is_Louise_Thüring, Name_Is_Louise_Waser,
Name_Is_Louise_Weber, Name_Is_Louise_de_Vargas, Name_Is_Martin_Bachmann,
Name_Is_Martin_Böckli, Name_Is_Martin_Conte, Name_Is_Martin_Dubach,
Name_Is_Martin_Farner, Name_Is_Martin_Fontana, Name_Is_Martin_Gruner,
Name_Is_Martin_Gschwind, Name_Is_Martin_Hauser, Name_Is_Martin_Indermaur,
Name_Is_Martin_Maurer, Name_Is_Martin_Sager, Name_Is_Martin_Schneider,
Name_Is_Martin_Schnyder, Name_Is_Martin_Silbermann, Name_Is_Martin_Steinmann,
Name_Is_Martin_Thüring, Name_Is_Martin_Waser, Name_Is_Martin_de_Vargas,
Name_Is_Michael_Bachmann, Name_Is_Michael_Böckli, Name_Is_Michael_Conte,
Name_Is_Michael_Dubach, Name_Is_Michael_Fontana, Name_Is_Michael_Gruner,
Name_Is_Michael_Hauser, Name_Is_Michael_Keller, Name_Is_Michael_Maurer,
Name_Is_Michael_Perrin, Name_Is_Michael_Reber, Name_Is_Michael_Sager,
Name_Is_Michael_Silbermann, Name_Is_Michael_Steinmann, Name_Is_Michael_Thüring,
Name_Is_Michael_Waser, Name_Is_Michael_Weber, Name_Is_Michael_de_Vargas,
Name_Is_Natascha_Conte, Name_Is_Natascha_Farner, Name_Is_Natascha_Gehrig,
Name_Is_Natascha_Indermaur, Name_Is_Natascha_Maurer, Name_Is_Natascha_Mieschke,
```

```
Name_Is_Natascha_Sager, Name_Is_Natascha_Silbermann, Name_Is_Natascha_Thüring,
Name_Is_Natascha_Waser, Name_Is_Natascha_Weber, Name_Is_Natascha_de_Vargas,
Name_Is_Otto_Bachmann, Name_Is_Otto_Conte, Name_Is_Otto_Dubach, Name_Is_Otto_Farner,
Name_Is_Otto_Fontana, Name_Is_Otto_Gruner, Name_Is_Otto_Gschwind,
Name_Is_Otto_Hauser, Name_Is_Otto_Keller, Name_Is_Otto_Perrin, Name_Is_Otto_Sager,
Name_Is_Otto_Schnyder, Name_Is_Otto_Silbermann, Name_Is_Otto_Steinmann,
Name_Is_Otto_Thüring, Name_Is_Otto_Waser, Name_Is_Paul_Bachmann, Name_Is_Paul_Böckli,
Name_Is_Paul_Farner, Name_Is_Paul_Gehrig, Name_Is_Paul_Gruner, Name_Is_Paul_Gschwind,
Name_Is_Paul_Keller, Name_Is_Paul_Mieschke, Name_Is_Paul_Sager,
Name_Is_Paul_Schneider, Name_Is_Paul_Schnyder, Name_Is_Paul_Silbermann,
Name_Is_Paul_Steinmann, Name_Is_Paul_Waser, Name_Is_Paul_Weber,
Name_Is_Paul_de_Vargas, Name_Is_Peter_Bachmann, Name_Is_Peter_Dubach,
Name_Is_Peter_Farner, Name_Is_Peter_Gehrig, Name_Is_Peter_Gschwind,
Name_Is_Peter_Hauser, Name_Is_Peter_Keller, Name_Is_Peter_Mieschke,
Name_Is_Peter_Perrin, Name_Is_Peter_Reber, Name_Is_Peter_Sager,
Name_Is_Peter_Schneider, Name_Is_Peter_Schnyder, Name_Is_Peter_Steinmann,
Name_Is_Peter_Thüring, Name_Is_Peter_Weber, Name_Is_Peter_de_Vargas,
Name_Is_Priska_Dubach, Name_Is_Priska_Farner, Name_Is_Priska_Fontana,
Name_Is_Priska_Gehrig, Name_Is_Priska_Gruner, Name_Is_Priska_Gschwind,
Name_Is_Priska_Hauser, Name_Is_Priska_Indermaur, Name_Is_Priska_Keller,
Name_Is_Priska_Maurer, Name_Is_Priska_Perrin, Name_Is_Priska_Reber,
Name_Is_Priska_Schneider, Name_Is_Priska_Schnyder, Name_Is_Priska_Silbermann,
Name_Is_Priska_Steinmann, Name_Is_Priska_Thüring, Name_Is_Priska_de_Vargas,
Name_Is_Rahel_Bachmann, Name_Is_Rahel_Conte, Name_Is_Rahel_Dubach,
Name_Is_Rahel_Farner, Name_Is_Rahel_Fontana, Name_Is_Rahel_Gehrig,
Name_Is_Rahel_Gruner, Name_Is_Rahel_Gschwind, Name_Is_Rahel_Hauser,
Name_Is_Rahel_Maurer, Name_Is_Rahel_Mieschke, Name_Is_Rahel_Perrin,
Name_Is_Rahel_Reber, Name_Is_Rahel_Sager, Name_Is_Rahel_Schneider,
Name_Is_Rahel_Schnyder, Name_Is_Rahel_Silbermann, Name_Is_Rahel_de_Vargas,
Name_Is_Remo_Böckli, Name_Is_Remo_Farner, Name_Is_Remo_Fontana,
Name_Is_Remo_Gruner, Name_Is_Remo_Hauser, Name_Is_Remo_Indermaur,
Name_Is_Remo_Mieschke, Name_Is_Remo_Perrin, Name_Is_Remo_Schnyder,
Name_Is_Remo_Silbermann, Name_Is_Remo_Waser, Name_Is_Remo_Weber,
Name_Is_Remo_de_Vargas, Name_Is_Robert_Bachmann, Name_Is_Robert_Böckli,
Name_Is_Robert_Conte, Name_Is_Robert_Dubach, Name_Is_Robert_Gruner,
Name_Is_Robert_Gschwind, Name_Is_Robert_Hauser, Name_Is_Robert_Indermaur,
Name_Is_Robert_Keller, Name_Is_Robert_Maurer, Name_Is_Robert_Mieschke,
Name_Is_Robert_Perrin, Name_Is_Robert_Reber, Name_Is_Robert_Sager,
Name_Is_Robert_Schnyder, Name_Is_Robert_Steinmann, Name_Is_Robert_Thüring,
Name_Is_Robert_Weber, Name_Is_Robert_de_Vargas, Name_Is_Samuel_Bachmann,
Name_Is_Samuel_Böckli, Name_Is_Samuel_Conte, Name_Is_Samuel_Farner,
Name_Is_Samuel_Fontana, Name_Is_Samuel_Gruner, Name_Is_Samuel_Gschwind,
Name_Is_Samuel_Hauser, Name_Is_Samuel_Maurer, Name_Is_Samuel_Perrin,
Name_Is_Samuel_Reber, Name_Is_Samuel_Sager, Name_Is_Samuel_Schneider,
Name_Is_Samuel_Schnyder, Name_Is_Samuel_Thüring, Name_Is_Samuel_Waser,
Name_Is_Samuel_Weber, Name_Is_Samuel_de_Vargas, Name_Is_Sarah_Böckli,
Name_Is_Sarah_Conte, Name_Is_Sarah_Farner, Name_Is_Sarah_Gehrig,
Name_Is_Sarah_Gruner, Name_Is_Sarah_Gschwind, Name_Is_Sarah_Hauser,
Name_Is_Sarah_Indermaur, Name_Is_Sarah_Keller, Name_Is_Sarah_Maurer,
Name_Is_Sarah_Perrin, Name_Is_Sarah_Schneider, Name_Is_Sarah_Schnyder,
Name_Is_Sarah_Steinmann, Name_Is_Sarah_Thüring, Name_Is_Sarah_Waser,
Name_Is_Sarah_Weber, Name_Is_Verena_Bachmann, Name_Is_Verena_Böckli,
Name_Is_Verena_Conte, Name_Is_Verena_Fontana, Name_Is_Verena_Gehrig,
Name_Is_Verena_Gruner, Name_Is_Verena_Gschwind, Name_Is_Verena_Mieschke,
Name_Is_Verena_Perrin, Name_Is_Verena_Reber, Name_Is_Verena_Sager,
Name_Is_Verena_Steinmann, Name_Is_Verena_Thüring, Name_Is_Verena_Waser,
Name_Is_Willi_Böckli, Name_Is_Willi_Conte, Name_Is_Willi_Gruner,
Name_Is_Willi_Gschwind, Name_Is_Willi_Hauser, Name_Is_Willi_Indermaur,
Name_Is_Willi_Keller, Name_Is_Willi_Maurer, Name_Is_Willi_Mieschke,
Name_Is_Willi_Reber, Name_Is_Willi_Schnyder, Name_Is_Willi_Silbermann,
Name_Is_Willi_Steinmann, Name_Is_Willi_Waser, Name_Is_Willi_Weber,
Name_Is_Willi_de_Vargas, Name_Is_Yulia_Bachmann, Name_Is_Yulia_Böckli,
Name_Is_Yulia_Dubach, Name_Is_Yulia_Fontana, Name_Is_Yulia_Gruner,
Name_Is_Yulia_Gschwind, Name_Is_Yulia_Hauser, Name_Is_Yulia_Maurer,
Name_Is_Yulia_Mieschke, Name_Is_Yulia_Perrin, Name_Is_Yulia_Reber,
Name_Is_Yulia_Sager, Name_Is_Yulia_Schneider, Name_Is_Yulia_Silbermann,
Name_Is_Yulia_Steinmann, Name_Is_Yulia_Waser, Name_Is_Yulia_de_Vargas,
NoisyDataSet1, NoisyDataSet2, NoisyDataSet3, NonflavanoidPhenols_IsAtLeast_H,
NonflavanoidPhenols_IsAtLeast_L, NonflavanoidPhenols_IsAtLeast_M,
```

```
NonflavanoidPhenols_IsAtMost_H, NonflavanoidPhenols_IsAtMost_L,
NonflavanoidPhenols_IsAtMost_M, NonflavanoidPhenols_Is_H, NonflavanoidPhenols_Is_L,
NonflavanoidPhenols_Is_M, NonflavanoidPhenols_Is_VH, NonflavanoidPhenols_Is_VL,
OD280OD315OfDilutedWines_IsAtLeast_H, OD280OD315OfDilutedWines_IsAtLeast_L,
OD280OD315OfDilutedWines_IsAtLeast_M, OD280OD315OfDilutedWines_IsAtMost_H,
OD280OD315OfDilutedWines_IsAtMost_L, OD280OD315OfDilutedWines_IsAtMost_M,
OD280OD315OfDilutedWines_Is_H, OD280OD315OfDilutedWines_Is_L,
OD280OD315OfDilutedWines_Is_M, OD280OD315OfDilutedWines_Is_VH,
OD280OD315OfDilutedWines_Is_VL, PTRATIO_IsAtLeast_H, PTRATIO_IsAtLeast_L,
PTRATIO_IsAtLeast_M, PTRATIO_IsAtMost_H, PTRATIO_IsAtMost_L,
PTRATIO_IsAtMost_M, PTRATIO_Is_H, PTRATIO_Is_L, PTRATIO_Is_M, PTRATIO_Is_VH,
PTRATIO_Is_VL, Proanthocyanins_IsAtLeast_H, Proanthocyanins_IsAtLeast_L,
Proanthocyanins_IsAtLeast_M, Proanthocyanins_IsAtMost_H, Proanthocyanins_IsAtMost_L,
Proanthocyanins_IsAtMost_M, Proanthocyanins_Is_H, Proanthocyanins_Is_L,
Proanthocyanins_Is_M, Proanthocyanins_Is_VH, Proanthocyanins_Is_VL,
Profession_Is_administration, Profession_Is_executive, Profession_Is_semiSkilled,
Profession_Is_skilled, Profession_Is_teacher, Profession_Is_unknown,
Profession_Is_unskilled, Proline_IsAtLeast_H, Proline_IsAtLeast_L,
Proline_IsAtLeast_M, Proline_IsAtMost_H, Proline_IsAtMost_L, Proline_IsAtMost_M,
Proline_Is_H, Proline_Is_L, Proline_Is_M, Proline_Is_VH, Proline_Is_VL,
Properties_Is_N, Properties_Is_Y, RAD_IsAtLeast_H, RAD_IsAtLeast_L,
RAD_IsAtLeast_M, RAD_IsAtMost_H, RAD_IsAtMost_L, RAD_IsAtMost_M, RAD_Is_H,
RAD_Is_L, RAD_Is_M, RAD_Is_VH, RAD_Is_VL, RM_IsAtLeast_H, RM_IsAtLeast_L,
RM_IsAtLeast_M, RM_IsAtMost_H, RM_IsAtMost_L, RM_IsAtMost_M, RM_Is_H, RM_Is_L,
RM_Is_M, RM_Is_VH, RM_Is_VL, Savings_Is_N, Savings_Is_Y, Sex_Is_female,
Sex_Is_male, TAX_IsAtLeast_H, TAX_IsAtLeast_L, TAX_IsAtLeast_M, TAX_IsAtMost_H,
TAX_IsAtMost_L, TAX_IsAtMost_M, TAX_Is_H, TAX_Is_L, TAX_Is_M, TAX_Is_VH,
TAX_Is_VL, TempDataSet1, TempDataSet2, TestData1, TestData2, TestData3,
TestData4, TestData5, TestData6, TestData7, TmpView, TotalPhenols_IsAtLeast_H,
TotalPhenols_IsAtLeast_L, TotalPhenols_IsAtLeast_M, TotalPhenols_IsAtMost_H,
TotalPhenols_IsAtMost_L, TotalPhenols_IsAtMost_M, TotalPhenols_Is_H,
TotalPhenols_Is_L, TotalPhenols_Is_M, TotalPhenols_Is_VH, TotalPhenols_Is_VL,
TrainData1, TrainData10, TrainData11, TrainData12, TrainData13, TrainData14,
TrainData15, TrainData16, TrainData17, TrainData18, TrainData19, TrainData2,
TrainData20, TrainData21, TrainData22, TrainData23, TrainData24, TrainData25,
TrainData26, TrainData27, TrainData3, TrainData4, TrainData5, TrainData6,
TrainData7, TrainData8, TrainData9, ZN_IsAtLeast_H, ZN_IsAtLeast_L, ZN_IsAtLeast_M,
ZN_IsAtMost_H, ZN_IsAtMost_L, ZN_IsAtMost_M, ZN_Is_H, ZN_Is_L, ZN_Is_M, ZN_Is_VH,
ZN_Is_VL, _EvalMissingVals, _Means, adoption-of-the-budget-resolution_Is_'n',
adoption-of-the-budget-resolution_Is_'y', aid-to-nicaraguan-contras_Is_'n',
aid-to-nicaraguan-contras_Is_'y', anti-satellite-test-ban_Is_'n',
anti-satellite-test-ban_Is_'y', class_Is_0_VL, class_Is_1_L, class_Is_2_M,
class_Is_3_H, class_Is_4_VH, class_Is_Iris-setosa, class_Is_Iris-versicol,
class_Is_Iris-virginica, crime_Is_'n', crime_Is_'y', deltaErrorB,
duty-free-exports_Is_'n', duty-free-exports_Is_'y', education-spending_Is_'n',
education-spending_Is_'y', eigenVects, el-salvador-aid_Is_'n',
el-salvador-aid_Is_'y', export-administration-act-south-africa_Is_'n',
export-administration-act-south-africa_Is_'y', handicapped-infants_Is_'n',
handicapped-infants_Is_'y', immigration_Is_'n', immigration_Is_'y',
mx-missile_Is_'n', mx-missile_Is_'y', petal_length <= 2.875 (L),
petal_length <= 4.516 (M), petal_length <= 5.432 (H),
petal_length == [1.09, 1.543] (VL), petal_length == [1.543, 2.875] (L),
petal_length == [2.875, 4.516] (M), petal_length == [4.516, 5.432] (H),
petal_length == [5.432, 6.81] (VH), petal_length >= 1.543 (L),
petal_length >= 2.875 (M), petal_length >= 4.516 (H), petal_length_IsAtLeast_H,
petal_length_IsAtLeast_L, petal_length_IsAtLeast_M, petal_length_IsAtMost_H,
petal_length_IsAtMost_L, petal_length_IsAtMost_M, petal_length_Is_H,
petal_length_Is_L, petal_length_Is_M, petal_length_Is_VH, petal_length_Is_VL,
petal_width <= 0.8375 (L), petal_width <= 1.562 (M), petal_width <= 2.049 (H),
petal_width == [0.052, 0.2787] (VL), petal_width == [0.2787, 0.8375] (L),
petal_width == [0.8375, 1.562] (M), petal_width == [1.562, 2.049] (H),
petal_width == [2.049, 2.548] (VH), petal_width >= 0.2787 (L),
petal_width >= 0.8375 (M), petal_width >= 1.562 (H), petal_width_IsAtLeast_H,
petal_width_IsAtLeast_L, petal_width_IsAtLeast_M, petal_width_IsAtMost_H,
petal_width_IsAtMost_L, petal_width_IsAtMost_M, petal_width_Is_H,
petal_width_Is_L, petal_width_Is_M, petal_width_Is_VH, petal_width_Is_VL,
physician-fee-freeze_Is_'n', physician-fee-freeze_Is_'y', quantErrorB, radHistoryB,
religious-groups-in-schools_Is_'n', religious-groups-in-schools_Is_'y',
sepal_length <= 5.538 (L), sepal_length <= 6.052 (M), sepal_length <= 6.687 (H),
```

```
sepal_length == [4.334, 5.048] (VL), sepal_length == [5.048, 5.538] (L),
sepal_length == [5.538, 6.052] (M), sepal_length == [6.052, 6.687] (H),
sepal_length == [6.687, 7.766] (VH), sepal_length >= 5.048 (L),
sepal_length >= 5.538 (M), sepal_length >= 6.052 (H), sepal_length_IsAtLeast_H,
sepal_length_IsAtLeast_L, sepal_length_IsAtLeast_M, sepal_length_IsAtMost_H,
sepal_length_IsAtMost_L, sepal_length_IsAtMost_M, sepal_length_Is_H,
sepal_length_Is_L, sepal_length_Is_M, sepal_length_Is_VH, sepal_length_Is_VL,
sepal_width <= 2.868 (L), sepal_width <= 3.167 (M), sepal_width <= 3.559 (H),
sepal_width == [2.162, 2.561] (VL), sepal_width == [2.561, 2.868] (L),
sepal_width == [2.868, 3.167] (M), sepal_width == [3.167, 3.559] (H),
sepal_width == [3.559, 4.138] (VH), sepal_width >= 2.561 (L),
sepal_width >= 2.868 (M), sepal_width >= 3.167 (H), sepal_width_IsAtLeast_H,
sepal_width_IsAtLeast_L, sepal_width_IsAtLeast_M, sepal_width_IsAtMost_H,
sepal_width_IsAtMost_L, sepal_width_IsAtMost_M, sepal_width_Is_H,
sepal_width_Is_L, sepal_width_Is_M, sepal_width_Is_VH, sepal_width_Is_VL,
superfund-right-to-sue_Is_'n', superfund-right-to-sue_Is_'y',
synfuels-corporation-cutback_Is_'n', synfuels-corporation-cutback_Is_'y', trainGPR1,
trainSVM1, water-project-cost-sharing_Is_'n', water-project-cost-sharing_Is_'y',
x1_IsAtLeast_H, x1_IsAtLeast_L, x1_IsAtLeast_M, x1_IsAtMost_H, x1_IsAtMost_L,
x1_IsAtMost_M, x1_Is_H, x1_Is_L, x1_Is_M, x1_Is_VH, x1_Is_VL, x2_IsAtLeast_H,
x2_IsAtLeast_L, x2_IsAtLeast_M, x2_IsAtMost_H, x2_IsAtMost_L, x2_IsAtMost_M,
x2_Is_H, x2_Is_L, x2_Is_M, x2_Is_VH, x2_Is_VL, x3_IsAtLeast_H, x3_IsAtLeast_L,
x3_IsAtLeast_M, x3_IsAtMost_H, x3_IsAtMost_L, x3_IsAtMost_M, x3_Is_H, x3_Is_L,
x3_Is_M, x3_Is_VH, x3_Is_VL, yc_Is_0_VL, yc_Is_1_L, yc_Is_2_M, yc_Is_3_H, yc_Is_4_VH}
```

```
GetParam[GetObject["Col3_Partition"][[1]], "H"]
```

```
Obj`FuzzySetPWL[{Obj`Pair[0.631538, 0.],
  Obj`Pair[0.694997, 1.], Obj`Pair[0.97061, 1.], Obj`Pair[1.03364, 0.]}]
```

■ Defining a view

To select a (fuzzy) subset of samples from a given data set, it is possible to define views on this data set.

A view is defined with the command `DataSetViewBuf`, where the data set has to be given as the first argument and a filter operation as the second argument. The filter operation is usually a single fuzzy predicate or a logical combination of predicates.

We use one of the previously generated predicates to select all samples from our data set where the third attribute is not high.

```
res[[2, 3, 1]]
```

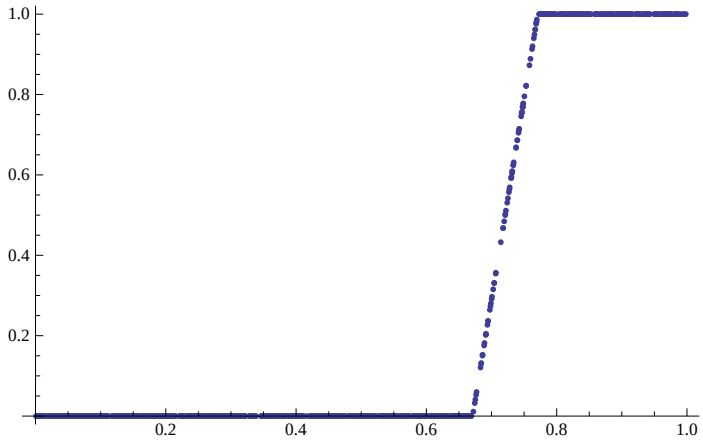
```
Col3_Is_H
```

```
viewBuf = Def["MyView", DataSetViewBuf[myDataSet, res[[2, 3, 2]]]];
viewNew = GetObject[viewBuf];
```

After we have processed the view, we can inspect the degrees of membership for all samples of our data set.

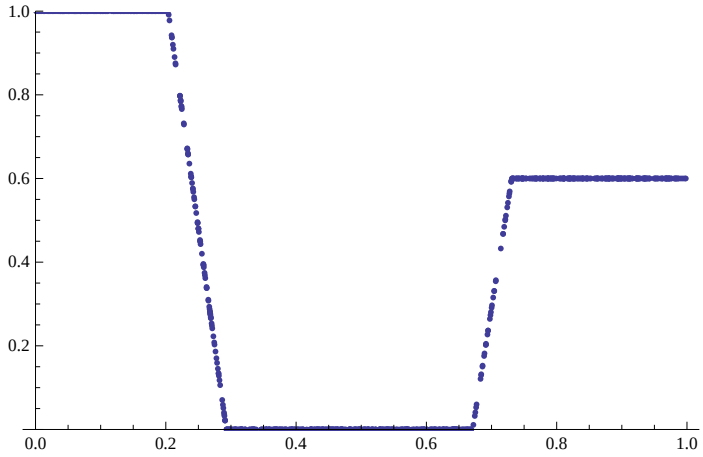
```
doms = GetParam[viewNew, "Doms"][[1]];
dataRow = MLFGetData[viewNew, All, 3];
```

```
ListPlot[Transpose@{dataRow, doms}]
```



By combining several predicates with a logical operation, we can create more complex types of views, too.

```
viewBuf2 = Def["MyView2",
    DataSetViewBuf[myDataSet, FOr[FAnd[0.6, res[[2, 3, 2]]], res[[2, 4, 2]]], LogicM]];
viewNew2 = GetObject[viewBuf2];
ListPlot[Transpose@{dataRow, GetParam[viewNew2, "Doms"][[1]]}, PlotRange -> {0, 1}]
```



■ Combined Predicates

Categorical predicates can be combined with "OR". Suppose you have an attribute "colour" with possible values {"red", "green", "blue"}; then in addition to simple predicates such as "colour_Is_red", you can also create combined predicates such as "colour_Is_red_OR_green".
Let us create a respective data set:

```
data = Table[
  Module[{x},
   x = Random[];
   {
    If[x < 0.2, "red",
     If[x < 0.4, "yellow",
      If[x < 0.6, "green",
       If[x < 0.8, "blue", "violet"]
      ]
     ]
    ],
    Module[{y},
     y = x + 0.1 * Random[] - 0.1 * Random[];
     If[y < 0.25, "hot",
      If[y < 0.5, "warm",
       If[y < 0.75, "cool", "freezing"]
      ]
     ]
    ]
   }
  ],
  {i, 1, 100}
 ]
```

```
{{yellow, warm}, {blue, cool}, {red, hot}, {yellow, warm}, {red, hot}, {red, hot},
 {violet, freezing}, {green, cool}, {yellow, warm}, {red, hot}, {red, hot},
 {red, hot}, {yellow, warm}, {green, cool}, {yellow, warm}, {blue, cool},
 {green, cool}, {violet, freezing}, {violet, freezing}, {green, warm}, {red, hot},
 {yellow, warm}, {red, hot}, {blue, cool}, {red, hot}, {violet, freezing},
 {green, cool}, {violet, freezing}, {green, warm}, {red, warm}, {green, warm},
 {yellow, hot}, {yellow, hot}, {violet, freezing}, {green, warm}, {violet, freezing},
 {red, hot}, {blue, cool}, {green, warm}, {red, hot}, {green, cool}, {green, warm},
 {blue, cool}, {red, hot}, {blue, freezing}, {green, cool}, {violet, freezing},
 {violet, freezing}, {yellow, warm}, {green, warm}, {violet, freezing},
 {violet, freezing}, {green, cool}, {blue, freezing}, {yellow, hot},
 {yellow, hot}, {blue, cool}, {violet, freezing}, {violet, freezing}, {red, hot},
 {blue, freezing}, {blue, cool}, {violet, freezing}, {yellow, hot}, {yellow, hot},
 {violet, freezing}, {blue, freezing}, {red, hot}, {blue, cool}, {green, warm},
 {yellow, hot}, {blue, cool}, {green, cool}, {yellow, warm}, {yellow, warm},
 {green, cool}, {blue, cool}, {yellow, warm}, {yellow, hot}, {blue, cool},
 {green, warm}, {blue, cool}, {red, hot}, {violet, freezing}, {green, cool},
 {green, cool}, {yellow, warm}, {blue, freezing}, {yellow, warm}, {blue, cool},
 {yellow, warm}, {yellow, hot}, {violet, freezing}, {yellow, warm}, {red, hot},
 {yellow, warm}, {yellow, warm}, {violet, freezing}, {green, cool}, {yellow, hot}}
```

```
headers = {"colour", "temperature"};
```

```
myData = Def["MyData", DataSet[data, headers]];
```

For comparison, let us first create the simple predicates. For categorical predicates, we use the function `CreatePredicatesIs` (which creates predicates with names like "A_Is_x", therefore the "Is" in the name of the function):

```
testPreds = CreatePredicatesIs[myData, 1]
```

```
{colour_Is_blue → Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[1, 1]}]],
 colour_Is_green → Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[2, 1]}]],
 colour_Is_red → Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[3, 1]}]],
 colour_Is_violet → Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[4, 1]}]],
 colour_Is_yellow → Obj`LogicDataPred[0, EQ, Obj`FuzzySetPWL[{Obj`Pair[5, 1]}]]}
```

For simpler access, we define variables which serve as pointers to the predicates:

```
testVars = DefPredicateVars[testPreds]
```

```
{Obj`LogicVar[colour_Is_blue],
 Obj`LogicVar[colour_Is_green], Obj`LogicVar[colour_Is_red],
 Obj`LogicVar[colour_Is_violet], Obj`LogicVar[colour_Is_yellow]}
```

To add combinations of these predicates, we define a maximum length of such combinations greater than 1:

```
testPreds = CreatePredicatesIs[myData, 1, MaxLengthOfCombinations → 2];
```

```
testVars = DefPredicateVars[testPreds]
```

```
{Obj`LogicVar[colour_Is_blue], Obj`LogicVar[colour_Is_green],
 Obj`LogicVar[colour_Is_red], Obj`LogicVar[colour_Is_violet],
 Obj`LogicVar[colour_Is_yellow], Obj`LogicVar[colour_Is_green_OR_blue],
 Obj`LogicVar[colour_Is_red_OR_blue], Obj`LogicVar[colour_Is_violet_OR_blue],
 Obj`LogicVar[colour_Is_yellow_OR_blue], Obj`LogicVar[colour_Is_red_OR_green],
 Obj`LogicVar[colour_Is_violet_OR_green], Obj`LogicVar[colour_Is_yellow_OR_green],
 Obj`LogicVar[colour_Is_violet_OR_red], Obj`LogicVar[colour_Is_yellow_OR_red],
 Obj`LogicVar[colour_Is_yellow_OR_violet]}
```

NB: the number of predicates can easily grow very high:

```
testPreds = CreatePredicatesIs[myData, 1, MaxLengthOfCombinations → 3];
```

```
testVars = DefPredicateVars[testPreds]
```

```
{Obj`LogicVar[colour_Is_blue], Obj`LogicVar[colour_Is_green],
 Obj`LogicVar[colour_Is_red], Obj`LogicVar[colour_Is_violet],
 Obj`LogicVar[colour_Is_yellow], Obj`LogicVar[colour_Is_green_OR_blue],
 Obj`LogicVar[colour_Is_red_OR_blue], Obj`LogicVar[colour_Is_violet_OR_blue],
 Obj`LogicVar[colour_Is_yellow_OR_blue], Obj`LogicVar[colour_Is_red_OR_green],
 Obj`LogicVar[colour_Is_violet_OR_green], Obj`LogicVar[colour_Is_yellow_OR_green],
 Obj`LogicVar[colour_Is_violet_OR_red], Obj`LogicVar[colour_Is_yellow_OR_red],
 Obj`LogicVar[colour_Is_yellow_OR_violet],
 Obj`LogicVar[colour_Is_red_OR_green_OR_blue],
 Obj`LogicVar[colour_Is_violet_OR_green_OR_blue],
 Obj`LogicVar[colour_Is_yellow_OR_green_OR_blue],
 Obj`LogicVar[colour_Is_violet_OR_red_OR_blue],
 Obj`LogicVar[colour_Is_yellow_OR_red_OR_blue],
 Obj`LogicVar[colour_Is_yellow_OR_violet_OR_blue],
 Obj`LogicVar[colour_Is_violet_OR_red_OR_green],
 Obj`LogicVar[colour_Is_yellow_OR_red_OR_green],
 Obj`LogicVar[colour_Is_yellow_OR_violet_OR_green],
 Obj`LogicVar[colour_Is_yellow_OR_violet_OR_red]}
```

Alternatively, we can define "cascading" predicates, where the values are ordered with respect to their significance for a particular output attribute. We choose "temperature" as the output attribute:

```
testPreds =
  CreatePredicatesIs[myData, 1, CascadingCombinations → True, OutputColumn → 2];
```

```
testVars = DefPredicateVars[testPreds]
```

```
{Obj`LogicVar[colour_Is_blue], Obj`LogicVar[colour_Is_green_OR_blue],
 Obj`LogicVar[colour_Is_red_OR_green_OR_blue],
 Obj`LogicVar[colour_Is_violet_OR_red_OR_green_OR_blue]}
```

■ Relational Predicates

For numeric attributes, one can define predicates describing relations between different attributes, such as attr1 is greater or equal (GE) attr2.

```
irisData = LoadData["iris.txt", "Table"];
```

```
relPreds1 = CreateRelationalPredicates[irisData, {1, 2, 3}];
```

```
relVars1 = DefPredicateVars[relPreds1]
```

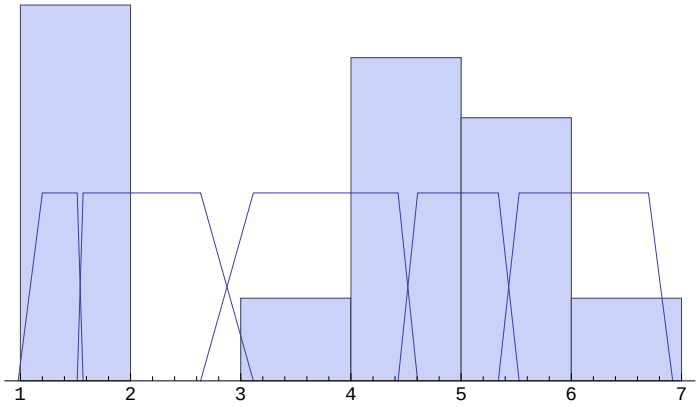
```
{Obj`LogicVar[sepal_length_EQ_sepal_width],
 Obj`LogicVar[sepal_length_GE_sepal_width], Obj`LogicVar[sepal_length_GT_sepal_width],
 Obj`LogicVar[sepal_length_LE_sepal_width], Obj`LogicVar[sepal_length_LT_sepal_width],
 Obj`LogicVar[sepal_length_EQ_petal_length],
 Obj`LogicVar[sepal_length_GE_petal_length],
 Obj`LogicVar[sepal_length_GT_petal_length],
 Obj`LogicVar[sepal_length_LE_petal_length],
 Obj`LogicVar[sepal_length_LT_petal_length],
 Obj`LogicVar[sepal_width_EQ_petal_length],
 Obj`LogicVar[sepal_width_GE_petal_length], Obj`LogicVar[sepal_width_GT_petal_length],
 Obj`LogicVar[sepal_width_LE_petal_length], Obj`LogicVar[sepal_width_LT_petal_length]}
```

Handling outliers

```
data = GetOriginalData[ irisData ];
AppendTo[data, {1.5, 1, 9, 1, data[[-1, -1]]}];
myDataSetOL = Def["MyDataSetOL", DataSet[data]];
```

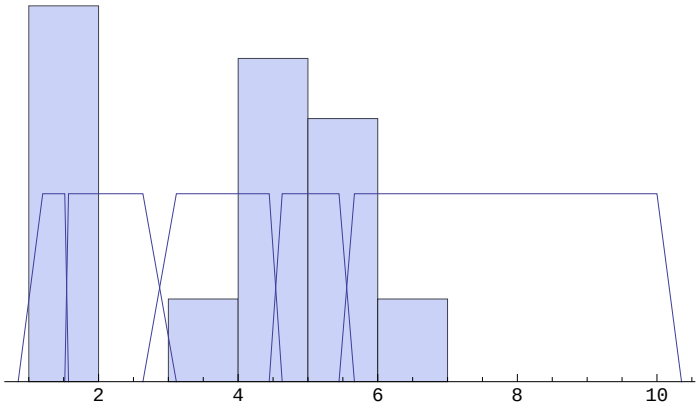
`CreatePartition` automatically takes care of outliers:

```
testPartitions = CreatePartition[myDataSetOL, 3, 5];
PlotFuzzySetHistogram[testPartitions, MLFGetData[data, All, 3]]
```



To prevent this behavior, we can set the maximum of the data manually:

```
testPartitions = CreatePartition[myDataSetOL, 3, 5, Maximum → 10];
PlotFuzzySetHistogram[testPartitions, MLFGetData[data, All, 3]]
```



Numerics

In this section, numerical extensions to the language of *Mathematica* are presented.

A numeric constant is specified using `NVal`.

```
obj = NVal[0.4]
```

```
Obj`RealConst[0.4]
```

Numeric addition is specified by `NAdd`, subtraction by `NSub`. If `NAdd` or `NSub` is used as a binary operator, the two objects to add/subtract must be given as arguments. If either of these operators is used as an n-ary operator, the objects to be added/subtracted have to be wrapped in a list and passed as a single argument.

```
obj = NAdd[0.5, 0.7]
```

```
Obj`RealAdd[Obj`RealConst[0.5], Obj`RealConst[0.7]]
```

```
obj = NAdd[{0.5, 0.7, 12}]
```

```
Obj`RealNAdd[
 Obj`RealList[{Obj`RealConst[0.5], Obj`RealConst[0.7], Obj`RealConst[12]}]]
```

With `EvalObject`, the object is evaluated.

```
EvalObject[obj]
```

```
13.2
```

Of course, an expression can be directly evaluated, too.

```
EvalObject[NAdd[0.3, 0.8]]
```

```
1.1
```

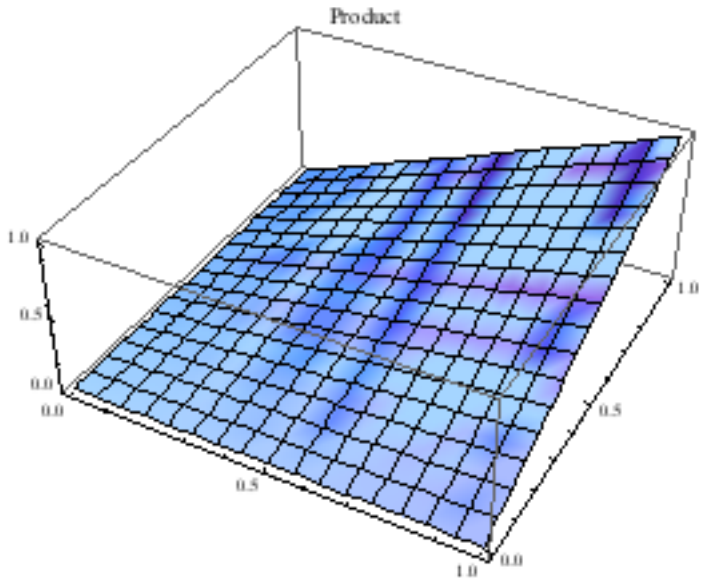
It is convenient to define regular *Mathematica* functions using numerical expressions.

```
f[x_, y_] := EvalObject[NSub[x, y]];
f[1, 2]
```

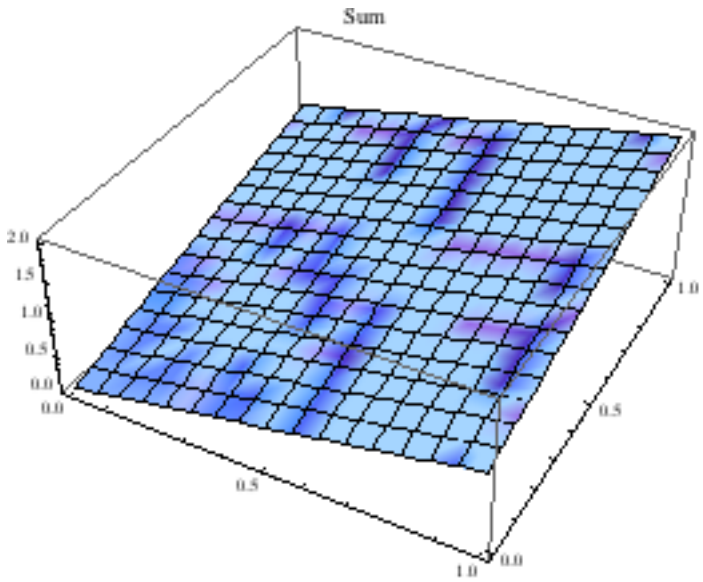
SetDelayed::write : Tag List in {0.3, 0, 0.2, 0.9, 0.25, 0.25}[x_, y_] is Protected. ≫

```
{0.3, 0, 0.2, 0.9, 0.25, 0.25}[1, 2]
```

```
Plot3D[EvalObject[NMul[x, y]], {x, 0, 1}, {y, 0, 1}, PlotLabel -> "Product"]
```



```
Plot3D[EvalObject[NAdd[x, y]], {x, 0, 1}, {y, 0, 1}, PlotLabel -> "Sum"]
```



Logics

mlf employs the formalism of fuzzy logic to express vague logical coherences. Fuzzy logic is an extension of classical, two-valued, propositional logic by allowing not only the two truth values True and False (1 and 0), but any value in the interval $[0, 1] \in \mathbb{R}$.

Simple Expressions

Basics

Logical operations can be defined in a way similar to classical logic. With the two values 0 (`FFalse`) and 1 (`FTrue`), classical propositional logic is embedded into fuzzy logic, i.e. when allowing only the truth values 0 and 1, the operations in fuzzy logic agree completely with the operations in classical, two-valued logic. The basic operations defined for fuzzy logic are `FVal`, `FAnd`, `FOr`, and `FNot`.

First we define a simple, parameterized logical expression which computes the disjunction of two values.

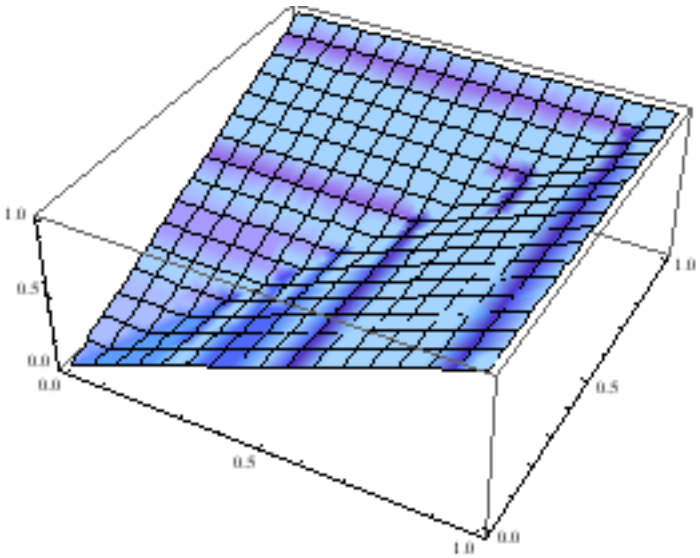
```
logicExpr[x_, y_] := FOr[x, y]
```

To evaluate this expression, the command `EvalObject` is used.

```
EvalObject[logicExpr[0.3, 0.8]]
```

```
0.8
```

```
Plot3D[EvalObject[logicExpr[x, y]], {x, 0, 1}, {y, 0, 1}, PlotRange → All]
```



Like `NAdd` and `NSub`, the commands `FAnd` and `FOr` can be used as either binary or n-ary operators. For binary application, two objects have to be given as arguments, while for n-ary use, the objects must be wrapped in a list and passed as a single `List` argument.

```
EvalObject[FAnd[0.7, 0.2]]
valueList = {0.5, 0.3, 0.6, 0.5};
EvalObject[FOr[valueList]]
EvalObject[FAnd[valueList]]
```

```
0.2

0.6

0.3
```

Fuzzy logic operations are derived from so called t-norms and t-conorms (the t stands for triangular). Although the number of possible t-norms is infinite, *mlf* currently supports only the three most common ones. The available logics are the so-called Min/Max logic (`LogicM`), product logic (`LogicP`), and Lukasiewicz logic (`LogicL`).

Per default, Min/Max logic is used. However, It is possible to specify a different logic when evaluating an expression.

```
EvalObject[FAnd[0.8, FNot[0.5]]]
```

```
0.5
```

```
EvalObject[FAnd[0.8, FNot[0.5]], Logic → LogicM]
EvalObject[FAnd[0.8, FNot[0.5]], Logic → LogicP]
EvalObject[FAnd[0.8, FNot[0.5]], Logic → LogicL]
```

```
0.5

0.4

0.3
```

■ Different t-norms

The logical *and* operation of two terms x and y is defined as the t-norm of these two terms.
For Min/Max logic, it is defined as

FAnd[x, y]:= Min[x, y];

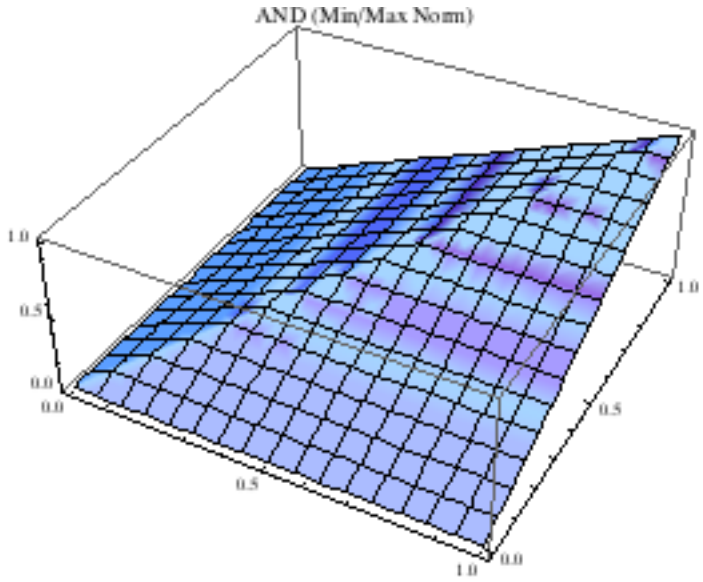
for product logic, it is defined as

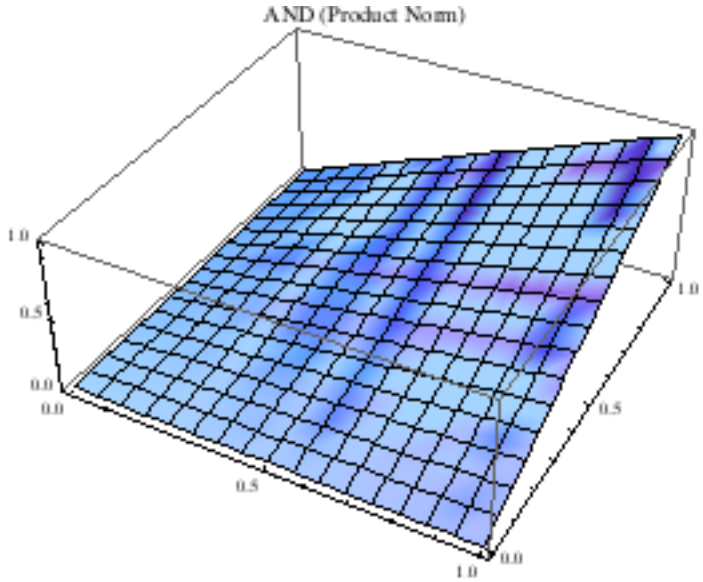
FAnd[x, y]:= x*y ;

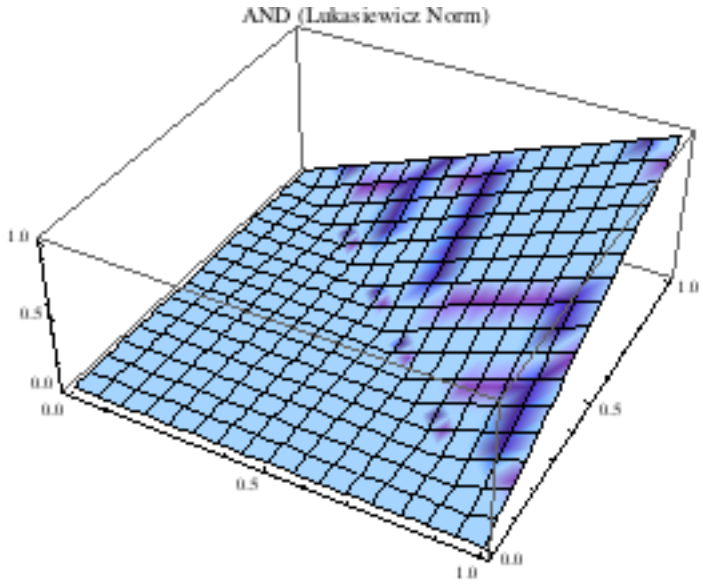
and finally, for Lukasiewicz logic, it is defined as

FAnd[x, y] := Max[(x + y - 1), 0] .

```
logicExpr[x_, y_] := FAnd[x, y]
Plot3D[EvalObject[logicExpr[x, y], Logic → LogicM],
 {x, 0, 1}, {y, 0, 1}, PlotLabel → "AND (Min/Max Norm)"]
Plot3D[EvalObject[logicExpr[x, y], Logic → LogicP],
 {x, 0, 1}, {y, 0, 1}, PlotLabel → "AND (Product Norm)"]
Plot3D[EvalObject[logicExpr[x, y], Logic → LogicL], {x, 0, 1},
 {y, 0, 1}, PlotLabel → "AND (Lukasiewicz Norm)"]
```







■ **Different t-conorms**

The logical *or* operation of two terms x and y is defined as the t-conorm of these two terms.
For Min/Max logic, it is defined as

FOr[x, y] := Max[x, y];

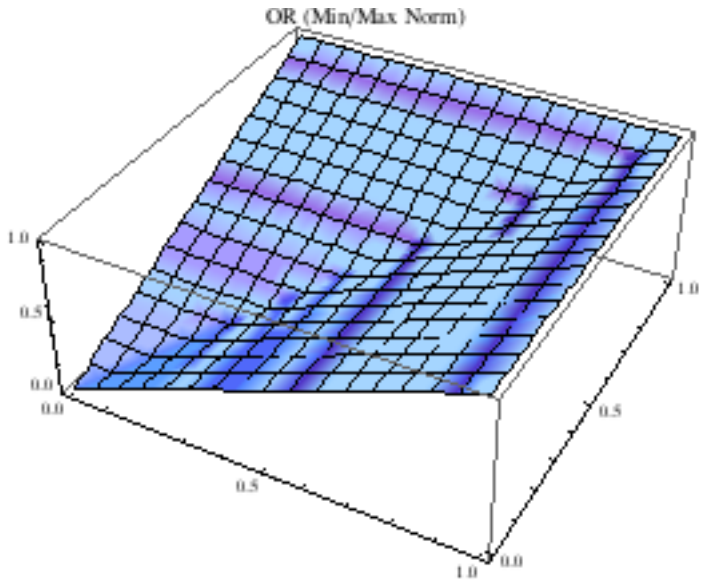
for product logic,

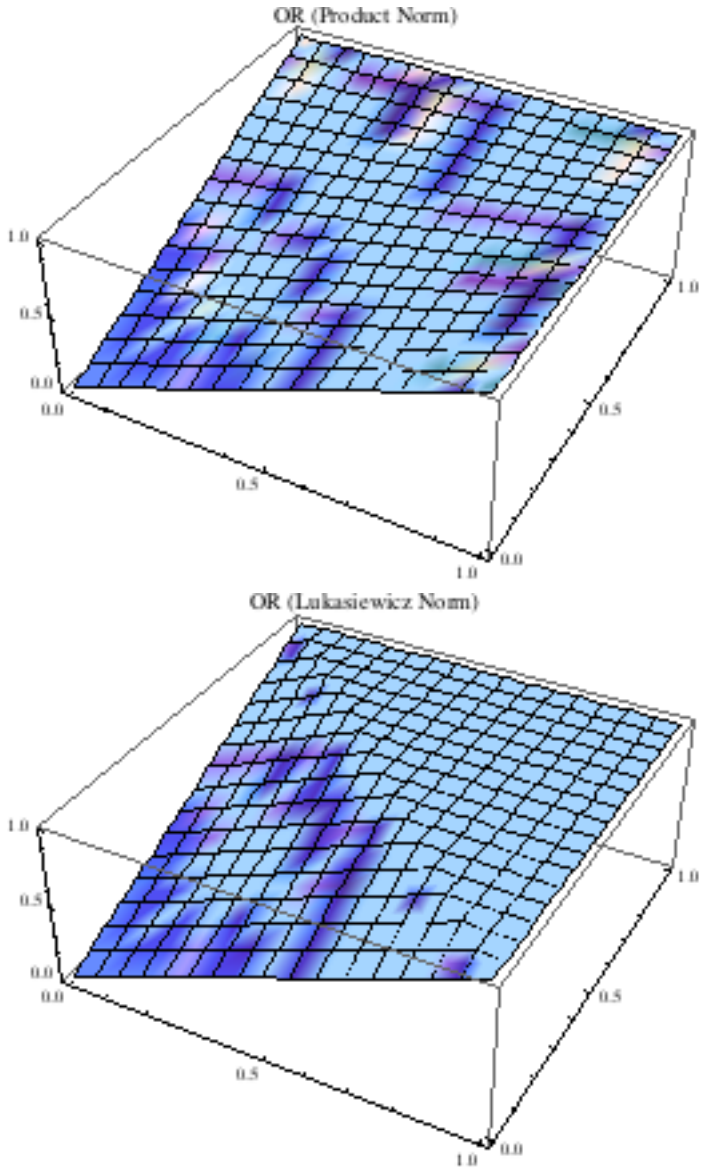
FOr[x, y] := x + y - x*y ;

and for Lukasiewicz logic,

FOr[x, y] := Min[x + y, 1] .

```
logicExpr[x_, y_] := FOr[x, y]
Plot3D[EvalObject[logicExpr[x, y], Logic → LogicM],
 {x, 0, 1}, {y, 0, 1}, PlotLabel → "OR (Min/Max Norm)"]
Plot3D[EvalObject[logicExpr[x, y], Logic → LogicP],
 {x, 0, 1}, {y, 0, 1}, PlotLabel → "OR (Product Norm)"]
Plot3D[EvalObject[logicExpr[x, y], Logic → LogicL], {x, 0, 1},
 {y, 0, 1}, PlotLabel → "OR (Lukasiewicz Norm)"]
```





■ Simple fuzzy sets and fuzzy predicates

To start with this topic, let us first give a motivation for the notion of a predicate. Generally speaking, a predicate is used for checking whether a property referring to some underlying system is valid for a given, concrete property value (or values). For example, if we want to establish some formalism to check if a person is taller than 180 cm, we can define the predicate taller-than-180-cm() which takes the height of a person as an argument. If we now apply the predicate to some distinct person of 175 cm height, the predicate will state that this property is not fulfilled. Expressed a bit more formally, applying taller-than-180-cm(175) returns False. In this example, we implicitly regard the internationally standardized metric system of measuring as the underlying system we refer to. If we apply our predicate to another person of height 183 cm - taller-than-180-cm(183) - then, of course, True is returned.
In fuzzy logic, from the user's perspective, the notion of a predicate is the same as described above except that the possible truth values returned are not only True and False but can take any value in the interval $[0,1] \in \mathbb{R}$.

If we switch to fuzzy logic terminology, we can say that fuzzy predicates usually describe some kind of membership function of fuzzy sets. While in classical set theory, the characteristic function $\mu_S(x)$ is defined as a mapping in $S \to \{0,1\}$, fuzzy sets have a characteristic function given as $\mu_S(x) : S \to [0, 1]$, where $\mu_S(x)$ characterizes the *degree of membership* of x in S, i.e. the truth value of the statement $x \in S$.

In *mlf*, the following distinct types of fuzzy sets are employed:
* *piecewise-linear* fuzzy sets,
* *exponential* fuzzy sets,
* *tri-cubic* fuzzy sets,

* *radial-basis-function* fuzzy sets, and
* *B-spline* fuzzy sets.

To express the membership degree of some value in a given fuzzy set, *mlf* employs the following three basic types of fuzzy predicates:
* is
* is-at-least, and
* is-at-most.

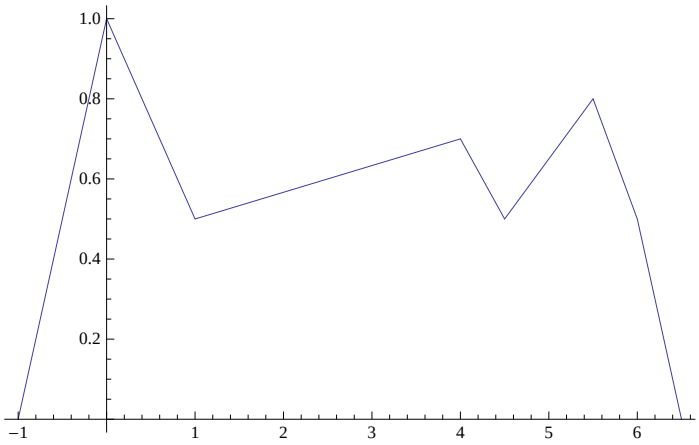
■ **Fuzzy sets/predicates with piecewise linear membership functions**

We will first take a look at piecewise-linear fuzzy sets. These fuzzy sets are defined by specifying a list of {numericValue, fuzzyValue} elements, called *support points*. With the command `FuzzySetPWL` and the support points wrapped in a list as argument, it is possible to generate arbitrary linear fuzzy sets.

```
fs := FuzzySetPWL[
  {{-1, 0}, {0, 1}, {1, 0.5}, {4, 0.7}, {4.5, 0.5}, {5.5, 0.8}, {6, 0.5}, {6.5, 0}}]
```

A fuzzy set can be plotted directly with the command `PlotFuzzySet`.

```
PlotFuzzySet[fs]
```

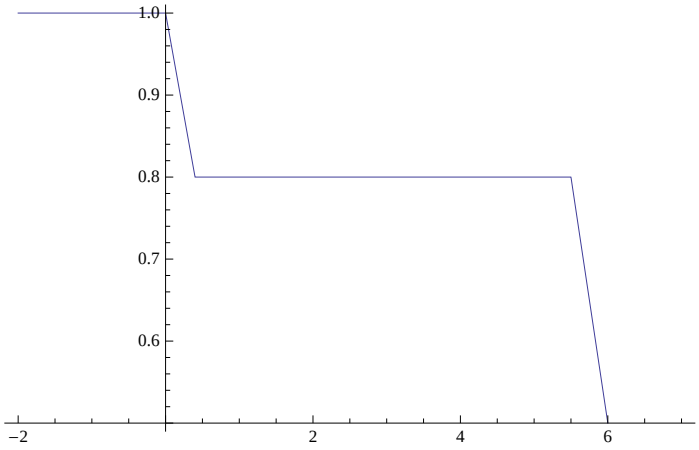


The very basic fuzzy predicate *is* can be defined with the command `FPredIs`, with the value as the first and the fuzzy set as the second argument. To define *is-at-most* and *is-at-least* predicates, the commands `FPredIsAM` and `FPredIsAL`, respectively, are applied.

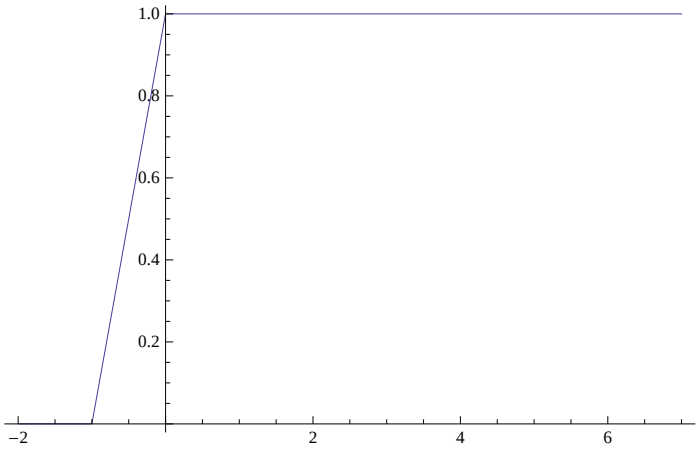
```
fpredIs[x_] := FPredIs[x, fs]
fpredIsAtMost[x_] := FPredIsAM[x, fs]
fpredIsAtLeast[x_] := FPredIsAL[x, fs]
```

From an operational point of view, a predicate is a function on some underlying fuzzy set. Therefore, to plot the membership function of a predicate, we have to evaluate the predicate before plotting it, i.e. we have to plot the evaluation graph of the predicate.

```
Plot[EvalObject[fpredIsAtMost[x]], {x, -2, 7}]
```

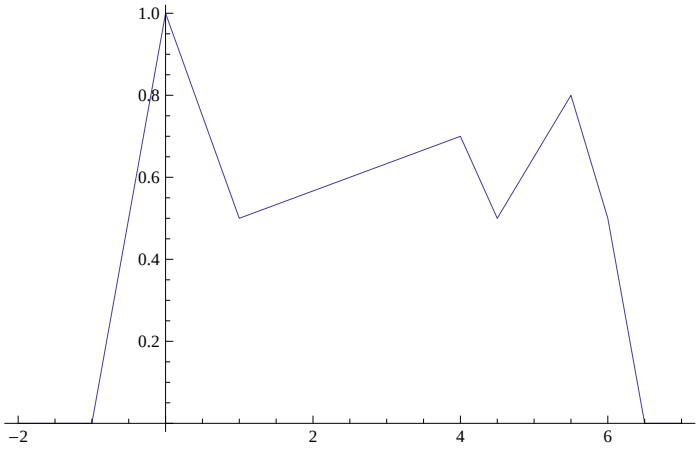


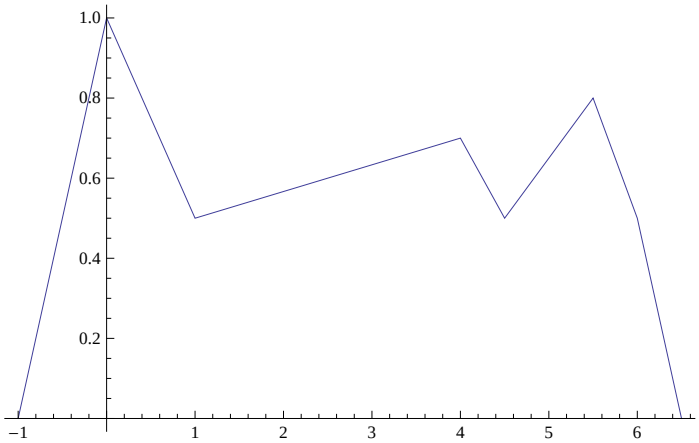
```
Plot[EvalObject[fpredIsAtLeast[x]], {x, -2, 7}]
```



In case of the *is*-predicate, the plot of the evaluation graph of the predicate is equivalent to the plot of the underlying fuzzy set itself (apart from a slightly different layout applied).

```
Plot[EvalObject[fpredIs[x]], {x, -2, 7}]
PlotFuzzySet[fs]
```





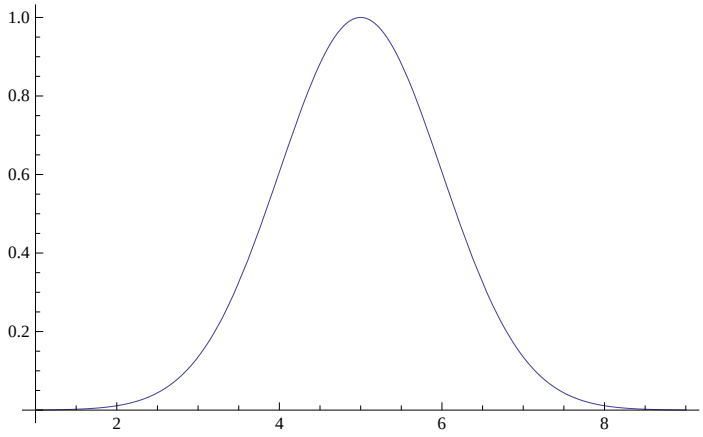
- **Fuzzy sets/predicates with exponential membership functions**

When a continuous membership function is needed, exponential (or bell-shaped) fuzzy sets are very convenient. In order to get an exponential fuzzy set, the command `FuzzySetExp` can be used. The command takes the center and the radius of the curve as first and second argument, respectively.
Predicates for *is*, *is-at-least*, and *is-at-most* over an exponential fuzzy set can be given in the same way as shown above for piecewise-linear fuzzy sets.

```
fsExp := FuzzySetExp[5, 1]
fpredIsAtMost[x_] := FPredIsAM[x, fsExp]
fpredIsAtLeast[x_] := FPredIsAL[x, fsExp]
```

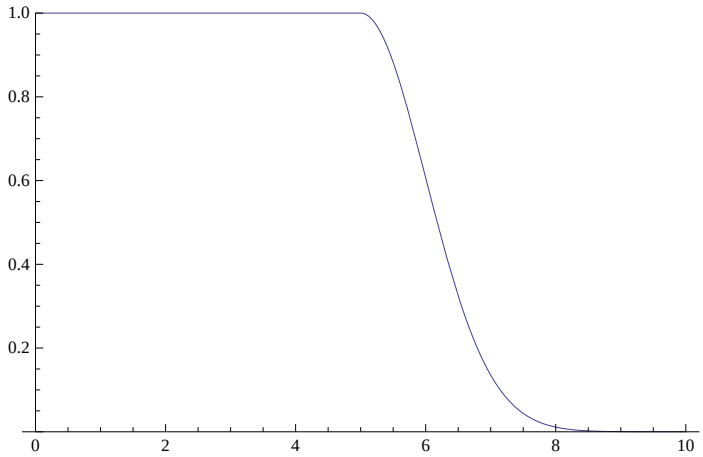
The exponential fuzzy set has got the characteristic bell shape.

```
PlotFuzzySet[fsExp]
```

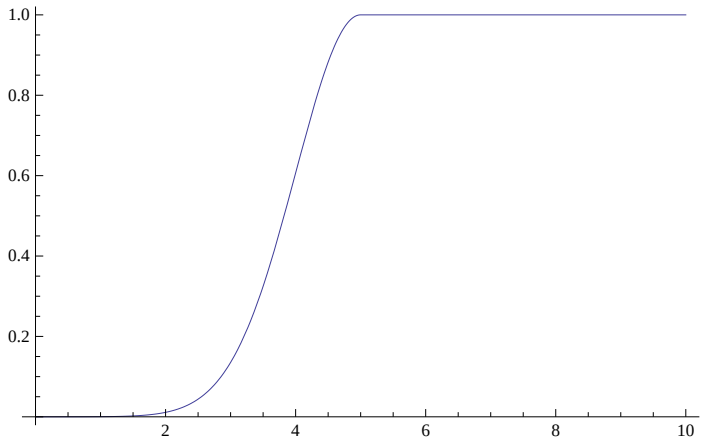


Naturally, we can also plot the membership functions of the defined predicates by plotting the evaluation graphs of the predicates.

```
Plot[EvalObject[fpredIsAtMost[x]], {x, 0, 10}, PlotRange → {0, 1}]
```



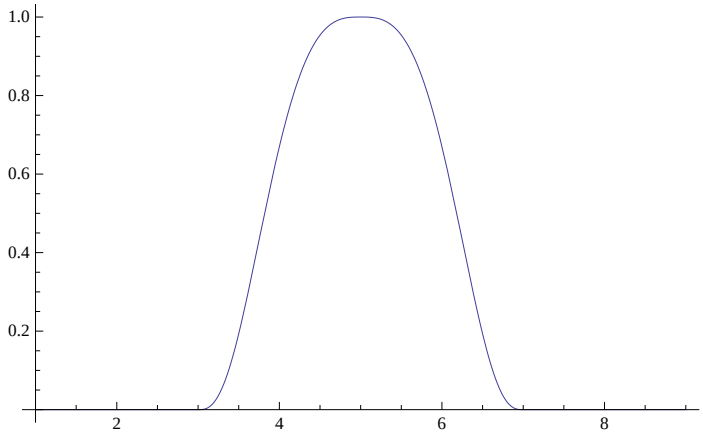

```
Plot[EvalObject[fpredIsAtLeast[x]], {x, 0, 10}]
```



■ **Fuzzy sets/predicates with tri-cubic membership functions**

Tri-cubic fuzzy sets combine the numeric benefits of exponential fuzzy sets with the local support of PWL fuzzy sets.

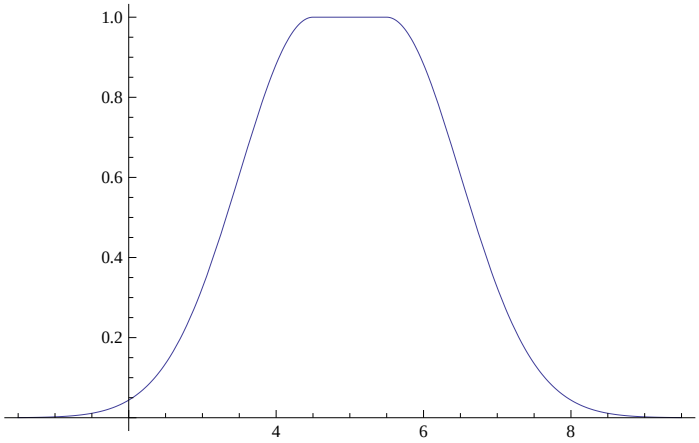
```
fsTriCube := FuzzySetTriCube[5, 1];
PlotFuzzySet[fsTriCube, PlotRange → All]
```



■ **Fuzzy sets/predicates with radial-basis functions**

Another very common choice for continuous membership functions are radial basis functions (RBF).
To create an RBF fuzzy set, apply `FuzzySetRBF` with the center, width, and center support as arguments.

```
fsRBF := FuzzySetRBF[5, 1, 1];
PlotFuzzySet[fsRBF]
```

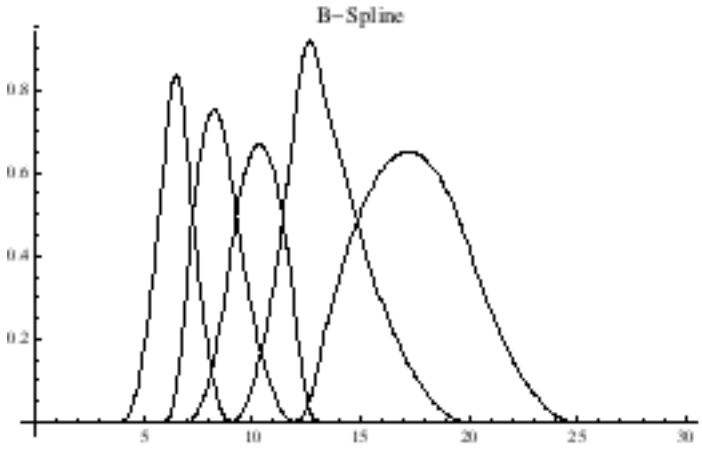


■ **Fuzzy sets/predicates with b-splines**

To define a partition of the domain under consideration, *B-splines* are very useful.
B-splines are available with the command `BSpline`. The command takes three arguments, i.e. the list of knots (knotsList), the number of splines (noOfSplines), and the order (order), where the number of knots in knotsList must equal to the sum of noOfSplines and order. Expressed more formally, the condition Length[knots] === noOfSplines + order must hold.

```
noOfSplines = 5;
knots = {4, 6, 7, 9, 12, 13, 20, 25};
bspline := BSpline[knots, noOfSplines, Length[knots] - noOfSplines]
fsBS[i_] := FuzzySetBSpline[bspline, i]
```

```
PlotFuzzySet[Table[fsBS[i], {i, noOfSplines}],
 PlotRange → {{0, 30}, All}, PlotLabel → "B-Spline"]
```



Since the first parameter of `BSpline` is a list - of which the length can be obtained easily - and the condition described above must hold, the third parameter can always be derived automatically and is therefore optional.

```
BSpline[knots, noOfSplines]
```

```
Obj`BSpline[{4, 6, 7, 9, 12, 13, 20, 25}, 5, 3]
```

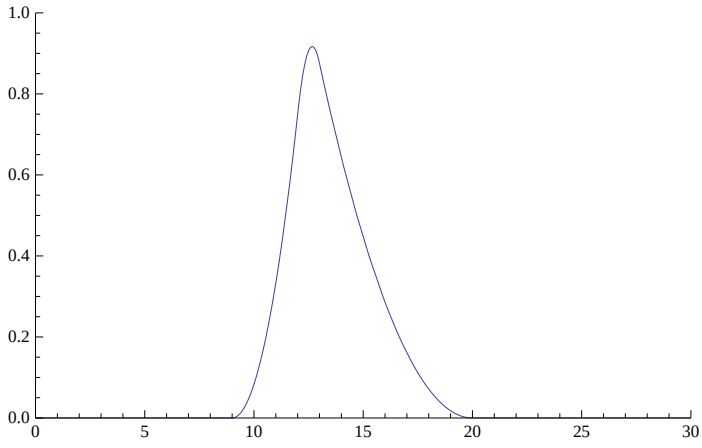
■ Scaling, intersection, and union of fuzzy sets

Transformations on fuzzy sets are defined on t-norms and t-conorms. To scale a fuzzy set, for each point of the set, the conjunction with the scale factor is computed. Intersection is given as the conjunction of the evaluated sets, whereas disjunction is used for the union. For piecewise-linear fuzzy sets, these operations are performed on the underlying set of support points, while for all other fuzzy sets, these operations are computed symbolically.

The default logic for all operations is Min/Max logic.

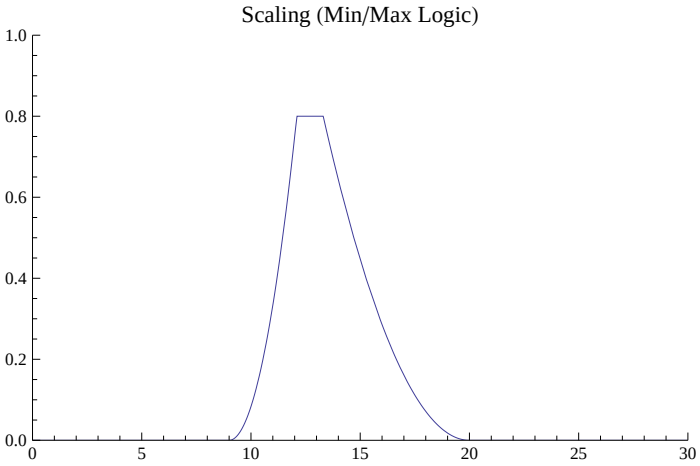
```
noOfSplines = 5;
knots = {4, 6, 7, 9, 12, 13, 20, 25};
bspline := BSpline[knots, noOfSplines]
fsBS[i_] := FuzzySetBSpline[bspline, i]
```

```
PlotFuzzySet[fsBS[4], PlotRange → {{0, 30}, {0, 1}}]
```



```
fsScaled = FuzzySetScaled[fsBS[4], 0.8];
PlotFuzzySet[fsScaled, PlotRange → {{0, 30}, {0, 1}},
 PlotLabel → "Scaling (Min/Max Logic)"]
```

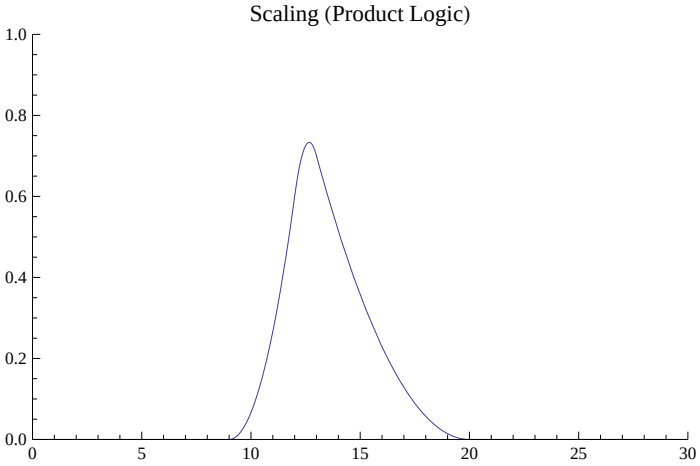
mlf::error : Fuzzy set not completely defined.



Note: the error message,"Fuzzy set not completely defined", does not refer to a critical error, but to the fact that `PlotFuzzySet` encountered a definition that requires some interpolation rather than a clearly defined set.

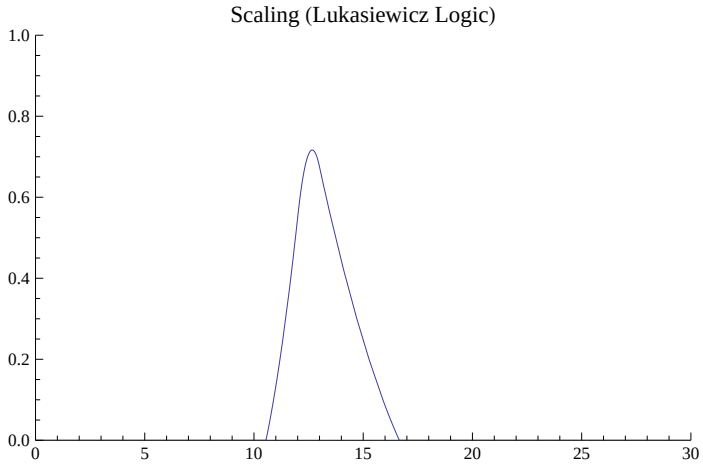
```
fsScaled = FuzzySetScaled[fsBS[4], 0.8, LogicP];
PlotFuzzySet[fsScaled, PlotRange → {{0, 30}, {0, 1}},
 PlotLabel → "Scaling (Product Logic)"]
```

mlf::error : Fuzzy set not completely defined.

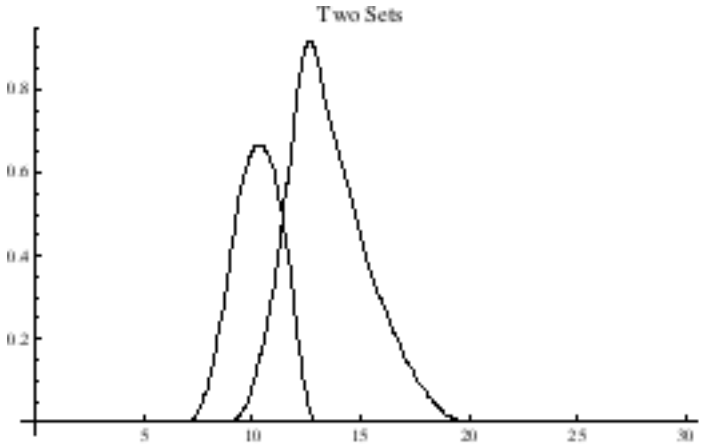


```
fsScaled = FuzzySetScaled[fsBS[4], 0.8, LogicL];
PlotFuzzySet[fsScaled, PlotRange → {{0, 30}, {0, 1}},
 PlotLabel → "Scaling (Lukasiewicz Logic)"]
```

mlf::error : Fuzzy set not completely defined.

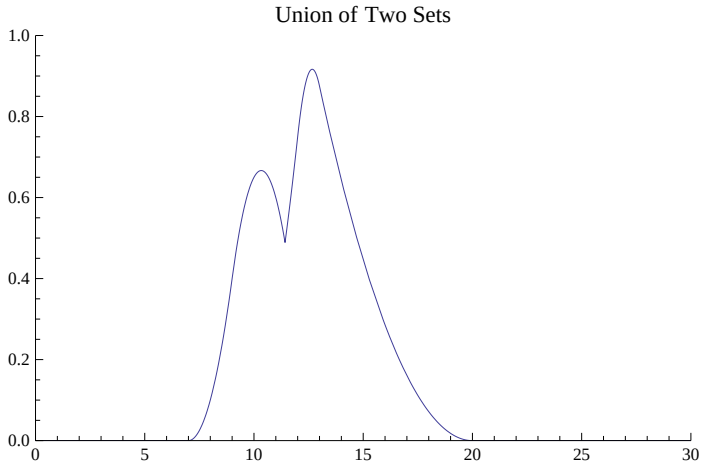



```
PlotFuzzySet[{fsBS[3], fsBS[4]}, PlotRange → {{0, 30}, {0, 1}}, PlotLabel → "Two Sets"]
```



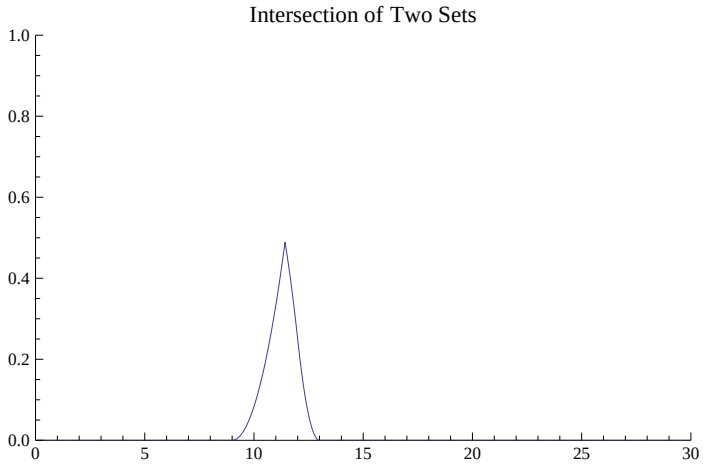

```
fsJoined := FuzzySetJoined[{fsBS[3], fsBS[4]}]
PlotFuzzySet[fsJoined, PlotRange → {{0, 30}, {0, 1}}, PlotLabel → "Union of Two Sets"]
```

mlf::error : Fuzzy set not completely defined.



```
fsIntersect := FuzzySetIntersect[{fsBS[3], fsBS[4]}]
PlotFuzzySet[fsIntersect, PlotRange → {{0, 30}, {0, 1}},
 PlotLabel -> "Intersection of Two Sets"]
```

mlf::error : Fuzzy set not completely defined.

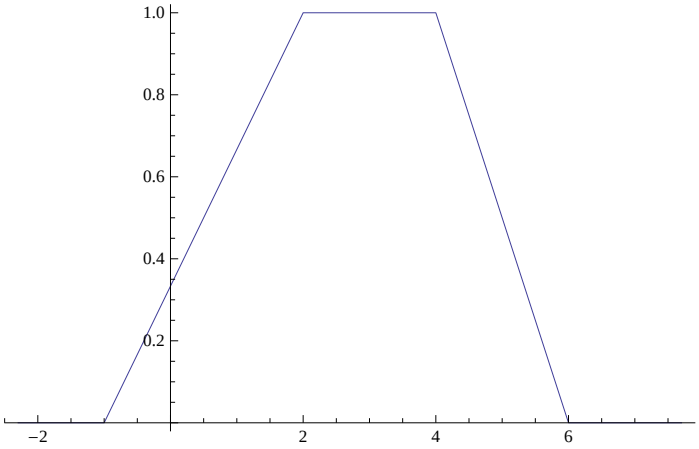


■ Fuzzy predicates for vectors

We have now seen how to define a predicate for a single value. Usually, however, it is more comfortable to apply a predicate directly to an element of a vector.
Column predicates are declared in a similar way as simple predicates for *is*, *is-at-least*, and *is-at-most*. The commands `FColPredIs`, `FColPredIsAL`, and `FColPredIsAM` are applied, respectively. These commands take the column index of the vector as the first and the fuzzy set to operate on as the second parameter.
Additionaly, the command `FColPredIsEx` allows you to create an equality predicate working on a crisp value instead of a fuzzy set.
For the next example, let us say we have two values alpha and beta given in some vector. To work with the two elements of this vector in a more model-oriented way, we define three column predicates. Using these predicates, we can work with the given vector elements symbolically, which is more intuitive than specifying bare numeric column indices.

```
isSmall = Def["IsSmall", FuzzySetPWL[{{-1, 0}, {2, 1}, {4, 1}, {6, 0}}]];
predAlphaIsSmall = Def["AlphaIsSmall", FColPredIs[1, isSmall]];
predBetaIsSmall = Def["BetaIsSmall", FColPredIs[2, isSmall]];
predAlphaIsOne = Def["AlphaIsOne", FColPredIsEx[1, 1]];
```

```
PlotFuzzySet[isSmall]
```




```
EvalObject[predAlphaIsSmall, {1, 5}]
```

```
0.666667
```

```
EvalObject[predBetaIsSmall, {1, 5}]
```

```
0.5
```

```
EvalObject[predAlphaIsOne, {1, 5}]
```

```
1.
```

```
EvalObject[predAlphaIsOne, {2, 5}]
```

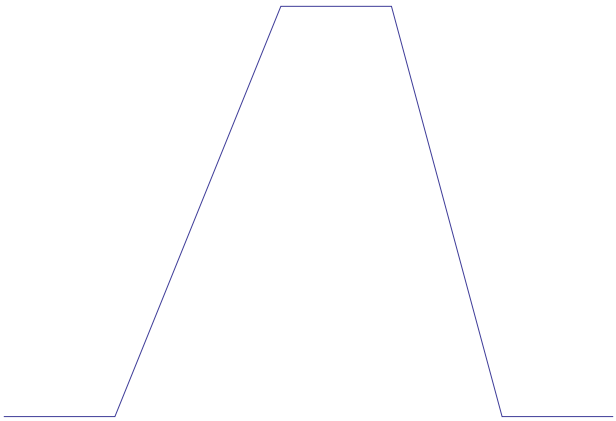
```
0.
```

To evaluate a predicate on a list of vectors, the command `EvalDataSet` is used.

```
myData = DataSet[Table[{i, Random[]}, {i, -3, 8, 1}], {"i", "Random"}];
```

```
evals = EvalDataSet[predAlphaIsSmall, myData];
```

```
ListPlot[evals, Joined → True, Axes → False]
```



■ Predicates in expressions

Predicates can be easily used in regular logical expressions.
In the following example, we compute the conjunction of two predicates given over a two-dimensional space. First we define the fuzzy sets.

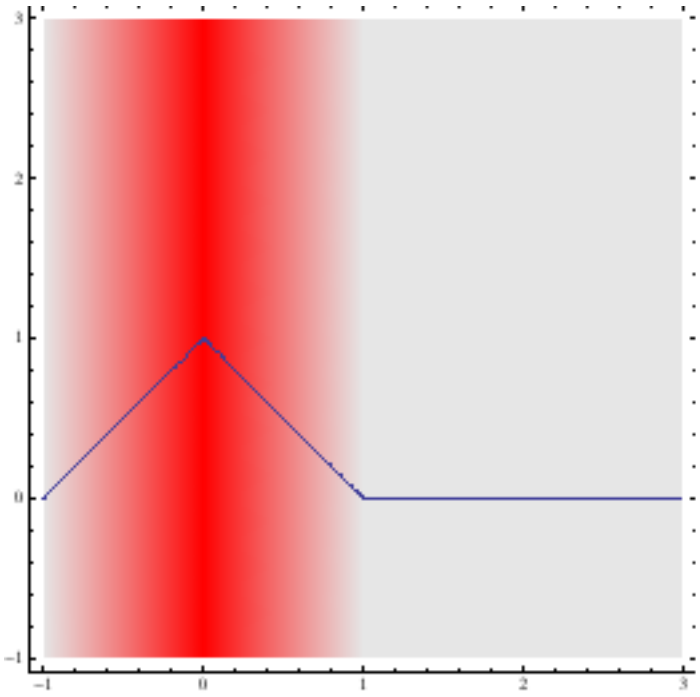
```
fsSmall := FuzzySetPWL[{{-1, 0}, {0, 1}, {1, 0}}]
fsMedium := FuzzySetPWL[{{0, 0}, {1, 1}, {2, 0}}]
fsBig := FuzzySetPWL[{{1, 0}, {2, 1}, {3, 0}}]
```

Based on these fuzzy sets, we now create the column predicates.

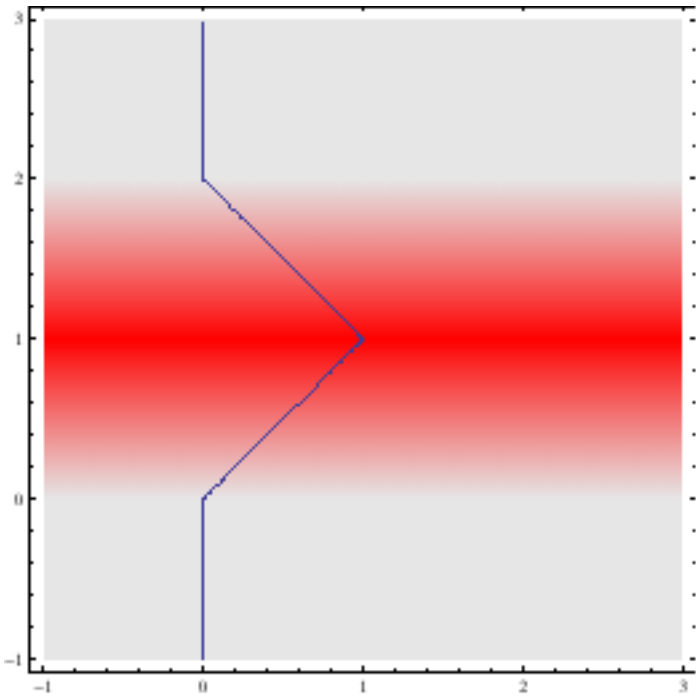
```
fpAisSmall := FColPredIs[1, fsSmall]
fpAisMedium := FColPredIs[1, fsMedium]
fpAisBig := FColPredIs[1, fsBig]
fpBisSmall := FColPredIs[2, fsSmall]
fpBisMedium := FColPredIs[2, fsMedium]
fpBisBig := FColPredIs[2, fsBig]
```

Since we will construct a logical expression using the fuzzy predicates fpAisSmall and fpBisMedium further on, let us first have a look at the contour plot of these predicates.

```
Show[{DensityPlot[EvalObject[fpAisSmall, {x, y}],
    {x, -1, 3}, {y, -1, 3}, ColorFunction → (CFRed@# &)],
  Plot[EvalObject[fpAisSmall, {x, 0}], {x, -1, 3}, PlotRange → {0, 1}, PlotPoints → 30]}]
```



```
Show[{DensityPlot[EvalObject[fpBisMedium, {x, y}], {x, -1, 3}, {y, -1, 3},
    ColorFunction → (CFRed@# &)], Plot[EvalObject[fpBisMedium, {0, y}], {y, -1, 3},
     PlotRange → {0, 1}, PlotPoints → 30] /. x_Line :> Map[Reverse, x, {2}]}]
```

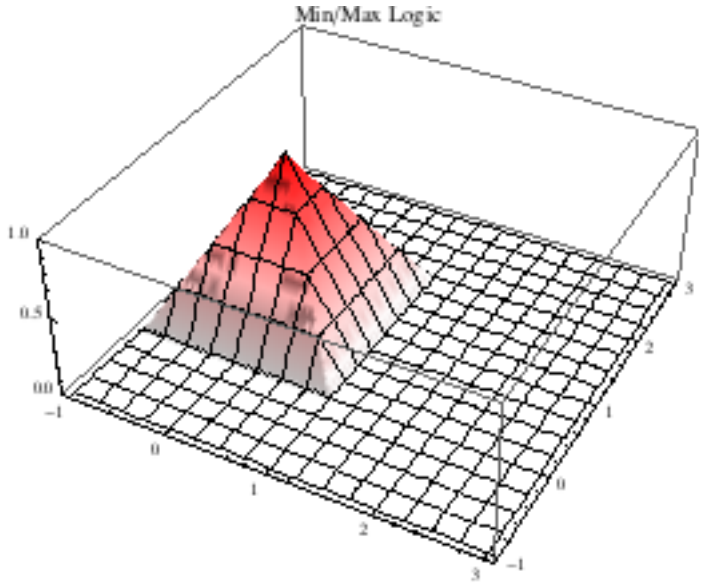


Now we formulate the new expression based on these fuzzy predicates.

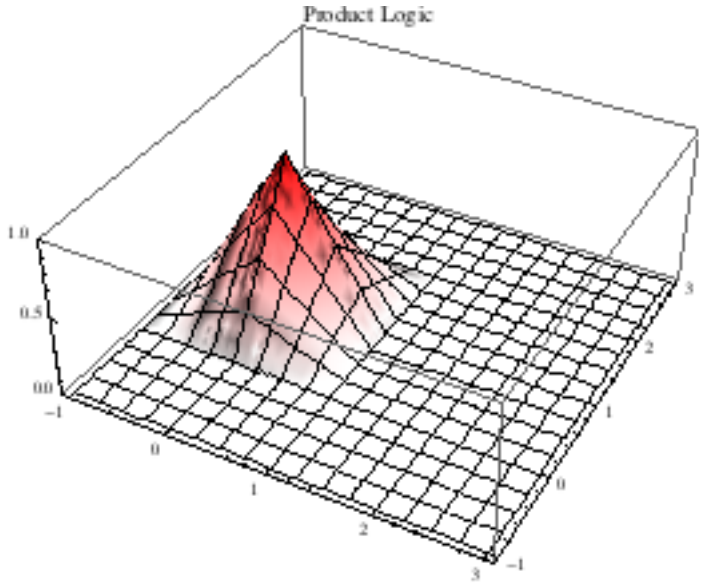
```
logicExpr = Def["MyExpr", FAnd[fpAisSmall, fpBisMedium]];
```

By evaluating the expression defined previously, we can find those regions where A is small and B is medium.

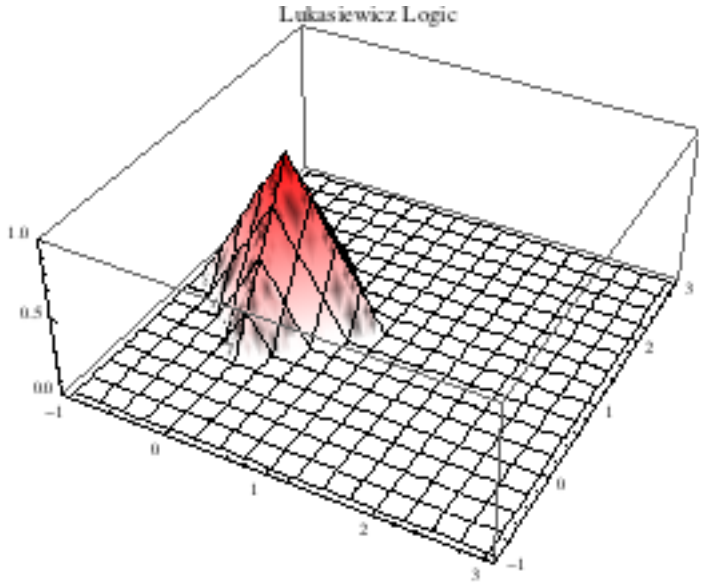
```
Plot3D[
 EvalObject[logicExpr, {x, y}], {x, -1, 3}, {y, -1, 3},
 PlotRange → {0, 1}, PlotPoints → 30,
 PlotLabel -> "Min/Max Logic", ColorFunction → (CFRed[#3] &)]
```




```
Plot3D[
 EvalObject[logicExpr, {x, y}, Logic → LogicP], {x, -1, 3}, {y, -1, 3},
 PlotRange → {0, 1}, PlotPoints → 30,
 PlotLabel -> "Product Logic", ColorFunction → (CFRed[#3] &)]
```



```
Plot3D[
 EvalObject[logicExpr, {x, y}, Logic → LogicL], {x, -1, 3}, {y, -1, 3},
 PlotRange → {0, 1}, PlotPoints → 30,
 PlotLabel -> "Lukasiewicz Logic", ColorFunction → (CFRed[#3] &)]
```



■ Multi-dimensional fuzzy sets

Multi-dimensional fuzzy sets are not objects themselves like the fuzzy sets introduced so far. In fact, they are implemented by means of fuzzy predicates which make use of other fuzzy sets and work on vectors. Since they work on vectors like column predicates, their names also start with FCol, as we will see below. Whenever we talk about multi-dimensional fuzzy sets, this notion applies.

There are two kinds of multi-dimensional fuzzy sets in *mlf*:
* *spheric* fuzzy sets and
* *non-spheric* fuzzy sets.

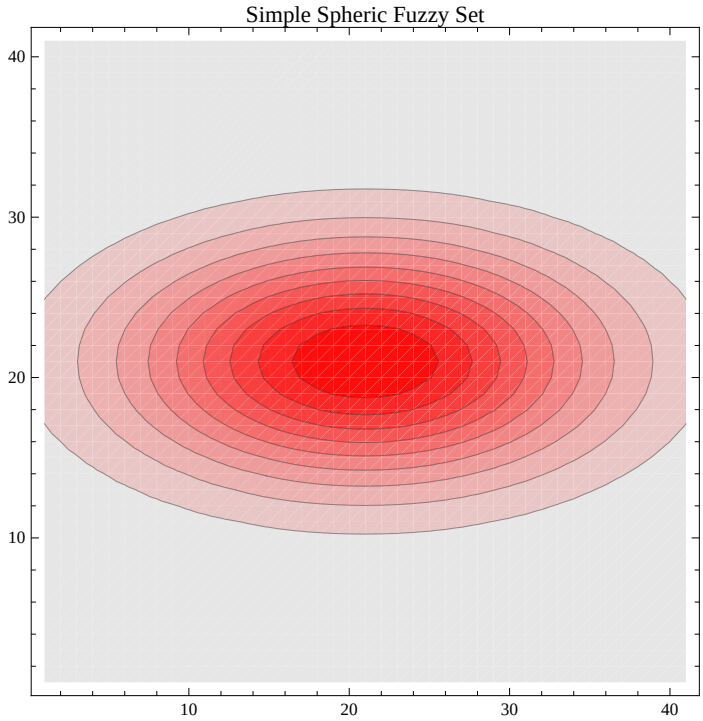
■ Spheric fuzzy sets

A spheric fuzzy set is constructed by rotating an arbitrary fuzzy set around a given center. Actually, only the positive part of the set is rotated. Additionally, it is possible to define weights for each dimension.
To construct a spheric fuzzy set, the command `FColPredSpheric` is used which consequently takes three arguments, i.e. the center, the fuzzy set, and the weights.

```
simpleSphericPred = Def["simpleSphericPred",
    FColPredSpheric[{20, 20}, FuzzySetExp[0, 10], {1, 0.5}]
  ];
```

Like a column predicate, the spheric fuzzy set can now be applied to a vector using command `EvalObject`.

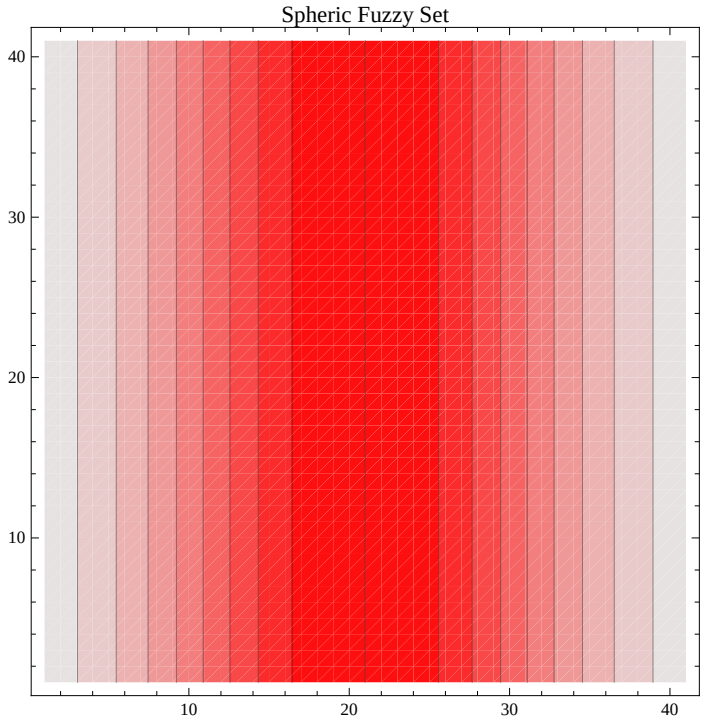
```
evalMatrix =
  Table[
   Table[
    EvalObject[simpleSphericPred, {x, y}],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel -> "Simple Spheric Fuzzy Set", ColorFunction → (CFRed@# &)]
```



When we modify the second weight parameter from 0.5 to 0, a very different result is obtained.

```
sphericPred = Def["sphericPred",
    FColPredSpheric[{20, 20}, FuzzySetExp[0, 10], {1, 0}]
  ];
```

```
evalMatrix =
  Table[
   Table[
    EvalObject[sphericPred, {x, y}],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel -> "Spheric Fuzzy Set", ColorFunction → (CFRed@# &)]
```



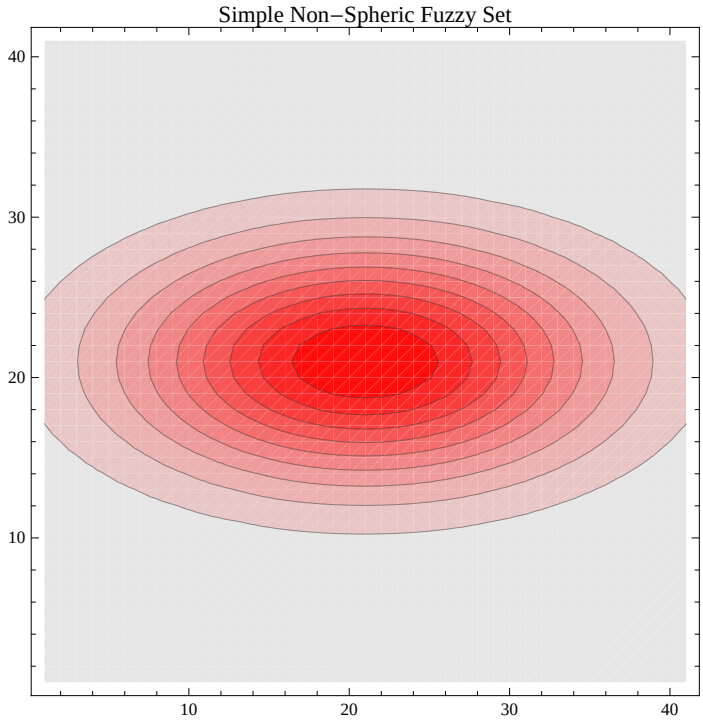
- **Non-spheric fuzzy sets**

  Non-spheric fuzzy sets make use of a distinct fuzzy set for each dimension. These fuzzy sets are then combined using a t-norm.
  We define a non-spheric fuzzy set using the command `FColPredND`, where ND stands for n-dimensional. The argument to this command is a list which wraps the distinct fuzzy sets that apply to each dimension.

```
simpleNonSphericPred = Def["simpleNonSphericPred",
   FColPredND[{
     FuzzySetExp[20, 10],
     FuzzySetExp[20, 5]
    }]];
```

  Like a spheric fuzzy set, the n-dimensional fuzzy set can now be applied to a vector using the command `EvalObject`.

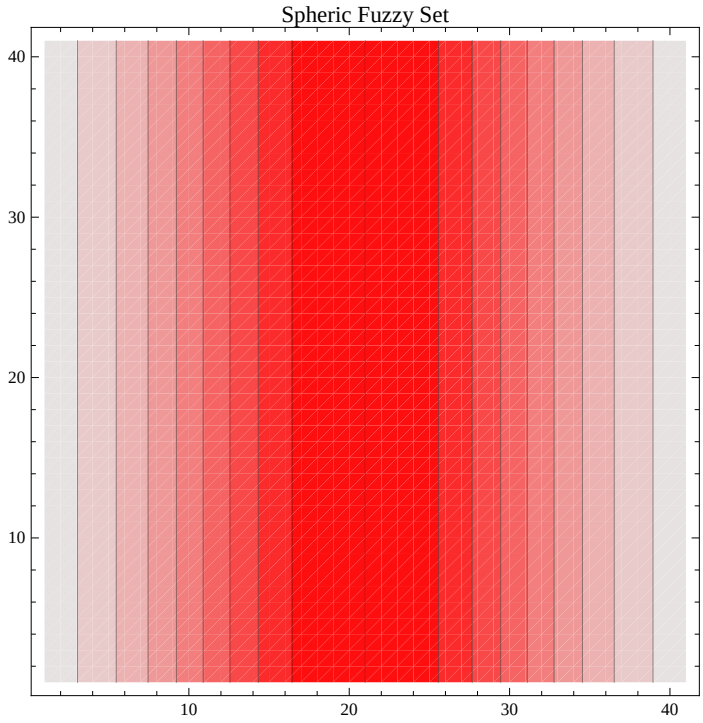
```
evalMatrix =
  Table[
   Table[
    EvalObject[simpleNonSphericPred, {x, y}, Logic → LogicP],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel -> "Simple Non-Spheric Fuzzy Set", ColorFunction → (CFRed@# &)]
```



As we can see below, it is also possible to eliminate one or more dimensions. To omit a dimension from the definition, one can pass a list of fuzzy sets which is shorter then the actual number of dimensions or specify an empty fuzzy set using the dummy object Obj`Empty.

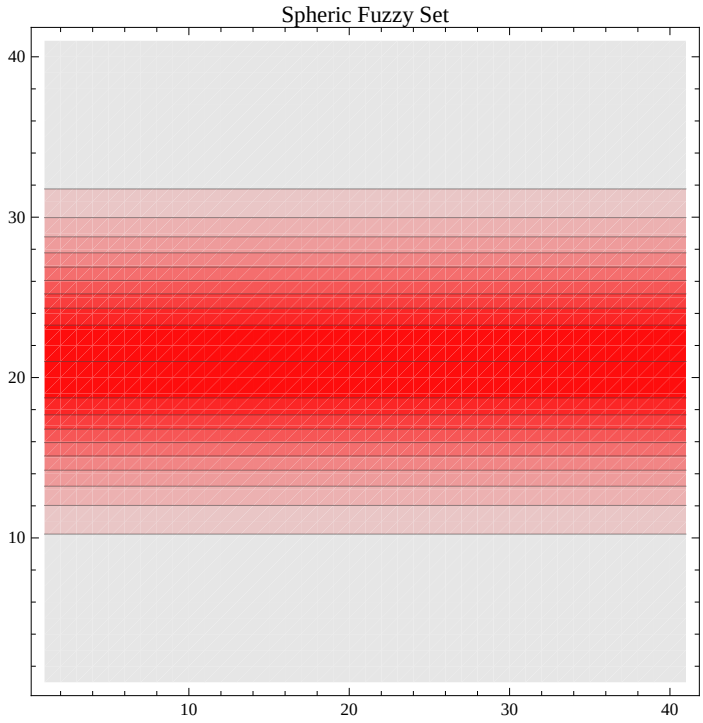
```
nonSphericPredA = Def["nonSphericPredA",
   FColPredND[
    {FuzzySetExp[20, 10]}
   ]];
```

```
evalMatrix =
  Table[
   Table[
    EvalObject[nonSphericPredA, {x, y}, Logic → LogicP],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel → "Spheric Fuzzy Set", ColorFunction → (CFRed@# &)]
```



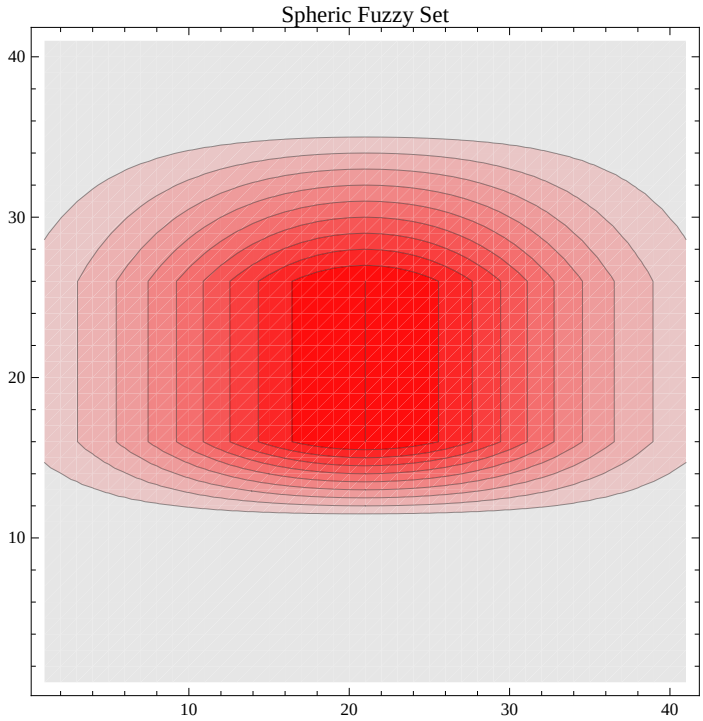

```
nonSphericPredB = Def["nonSphericPredB",
   FColPredND[
    {Obj`Empty,
     FuzzySetExp[20, 5]}
   ]];
```

```
evalMatrix =
  Table[
   Table[
    EvalObject[nonSphericPredB, {x, y}, Logic → LogicP],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel -> "Spheric Fuzzy Set", ColorFunction → (CFRed@# &)]
```




```
complexNonSphericPred = Def["complexNonSphericPred",
   FColPredND[{FuzzySetExp[20, 10], FuzzySetPWL[{{10, 0}, {15, 1}, {25, 1}, {35, 0}}]}]];
```

```
evalMatrix =
  Table[
   Table[
    EvalObject[complexNonSphericPred, {x, y}, Logic → LogicP],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel → "Spheric Fuzzy Set", ColorFunction → (CFRed@# &)]
```



■ Multi-dimensional fuzzy predicates

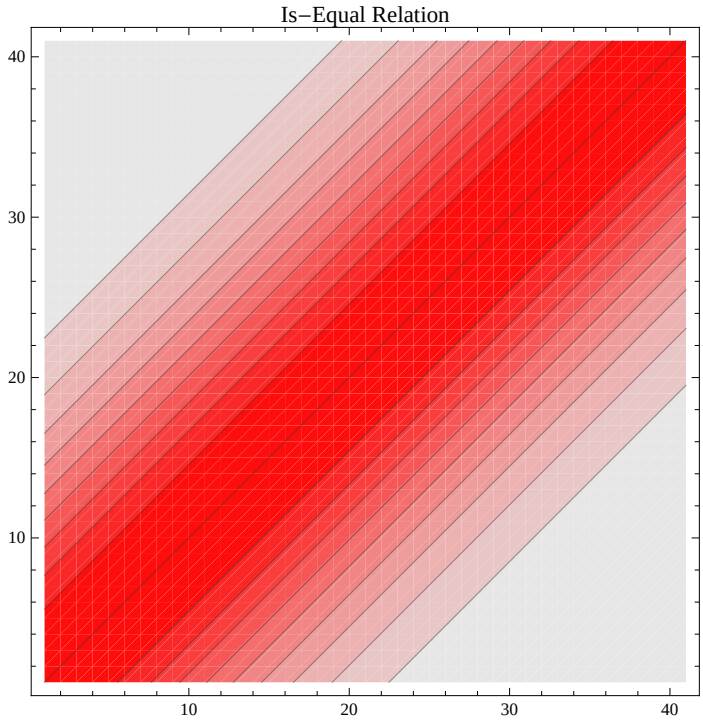
We obtain the simplest form of multi-dimensional fuzzy predicates if we base them on a combination of fuzzy sets, defined on a single dimension, by means of a t-norm or t-conorm. Alternatively, it is possible to define multi-dimensional fuzzy sets and use them for the predicates. A third possibility is to define predicates as some kind of fuzzy relations between two or more attributes. This is done by defining a fuzzy predicate on a computed attribute, e.g. (x+y) >= 10.

■ Fuzzy relations

Using standard predicates, it is easily possible to define comparison operations which provide placeholders for the original attributes. For implementing these placeholders, the command `NColVal` is applied.

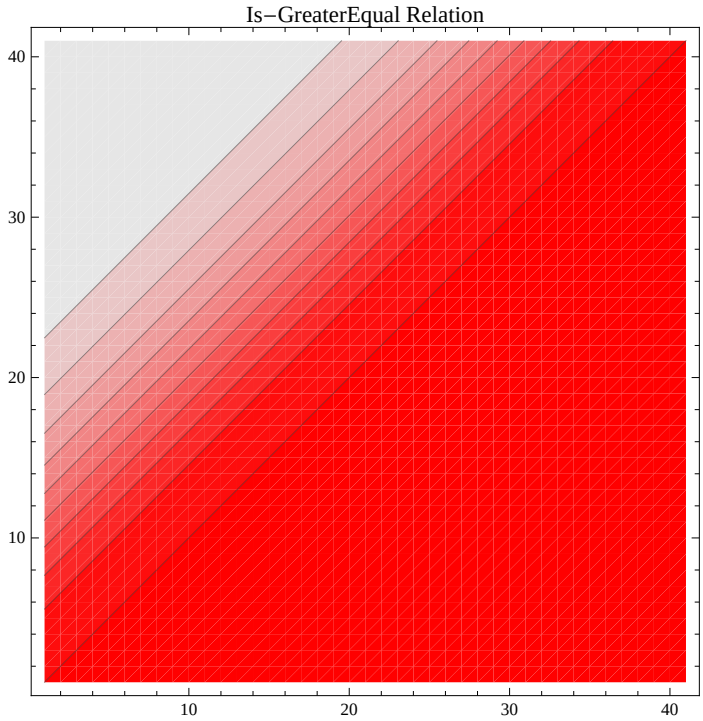
```
Def["myDist", NSub[NColVal[1], NColVal[2]]];

predIsEQ = Def["myDistPred", FPredIs[Ref["myDist"], FuzzySetExp[0, 10]]];
```

```
evalMatrix =
  Table[
   Table[
    EvalObject[predIsEQ, {x, y}],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel → "Is-Equal Relation", ColorFunction → (CFRed@# &)]
```




```
predIsGE = Def["myDistPred", FPredIsAL[Ref["myDist"], FuzzySetExp[0, 10]]];
```

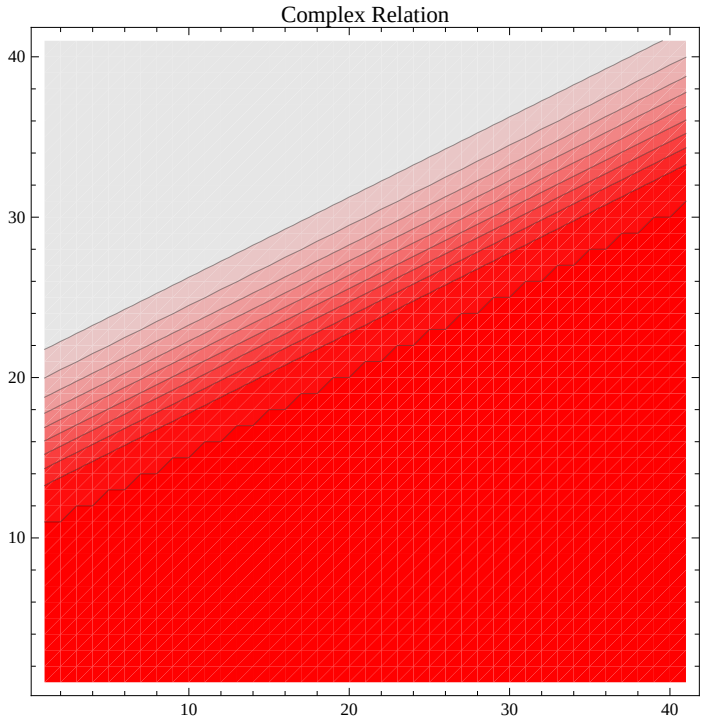
```
evalMatrix =
  Table[
   Table[
    EvalObject[predIsGE, {x, y}],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel → "Is-GreaterEqual Relation", ColorFunction → (CFRed@# &)]
```



It is also possible to define more complex distance functions using *mlf*'s numerical operations.

```
Def["myNewDist", NSub[NAdd[20, NColVal[1]], NMul[NColVal[2], 2]]];

predIsGT = Def["myDistPred", FPredIsAL[Ref["myNewDist"], FuzzySetExp[0, 10]]];
```

```
evalMatrix =
  Table[
   Table[
    EvalObject[predIsGT, {x, y}],
    {x, 0, 40}],
   {y, 0, 40}];
ListContourPlot[evalMatrix,
 PlotLabel → "Complex Relation", ColorFunction → (CFRed@# &)]
```



Logic Inference Models

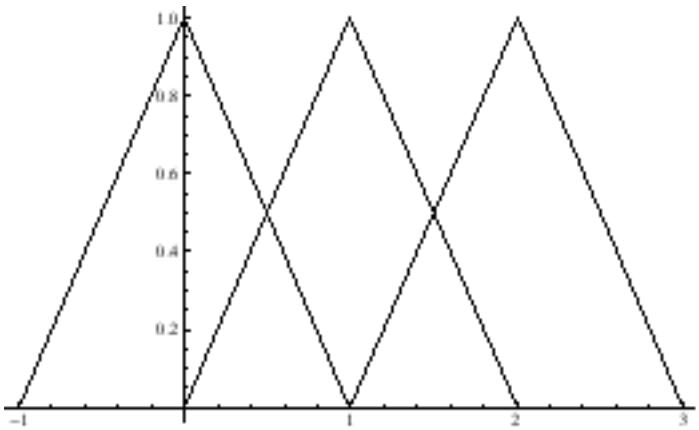
Basically, there are two common types of fuzzy inference models - *Sugeno* and *Mamdani* inference (see `Fuzzy Inference`).

- **Sugeno inference**
- **A simple example**

First we define three fuzzy sets and the corresponding predicates.

```
fsSmall := FuzzySetPWL[{{-1, 0}, {0, 1}, {1, 0}}]
fsMedium := FuzzySetPWL[{{0, 0}, {1, 1}, {2, 0}}]
fsBig := FuzzySetPWL[{{1, 0}, {2, 1}, {3, 0}}]
fpSmall := FColPredIs[1, fsSmall]
fpMedium := FColPredIs[1, fsMedium]
fpBig := FColPredIs[1, fsBig]
```

```
PlotFuzzySet[{fsSmall, fsMedium, fsBig}]
```



Using these fuzzy sets, we state three rules for our goal function using the command `FRule` which takes the condition as the first and the consequence as second argument. For a Sugeno controller, the consequence can be an arbitrary numerical expression.

```
logicRule1 := FRule[fpSmall, NVal[2]]
logicRule2 := FRule[fpMedium, NVal[5]]
logicRule3 := FRule[fpBig, NVal[4]]
SugenoFI = FRuleSetSugeno[{logicRule1, logicRule2, logicRule3}];
```

To compute the inference result, the commands `FuzzyInference` and `FuzzyInferenceVal` are used. Both take the controller as the first argument and the data as the second argument. While the first one computes a symbolic solution of the controller, the latter directly evaluates the result and returns only a single value.

```
FuzzyInference[SugenoFI, {1}]
FuzzyInferenceVal[SugenoFI, {1}]
```

```
Obj`RealDiv[
 Obj`RealNAdd[Obj`RealList[{Obj`RealMul[Obj`RealConst[0.], Obj`RealConst[2.]],
    Obj`RealMul[Obj`RealConst[1.], Obj`RealConst[5.]],
    Obj`RealMul[Obj`RealConst[0.], Obj`RealConst[4.]]}]], Obj`RealConst[1.]]
```

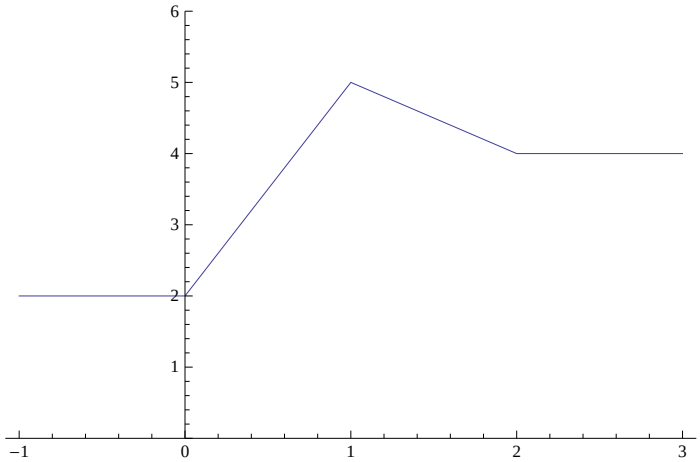
```
5.
```

We can now write a new function using the inference method. Remember that the second argument of the evaluation has to be a list of values, not just a single value.

```
newFunc[x_] :=
 FuzzyInferenceVal[SugenoFI, {x}]
```

In order to show how our controller behaves, we can plot the result of the inference process.

```
Plot[newFunc[x], {x, -1, 3}, PlotRange → {0, 6}]
```



As expected, we get a 2 for low values, a 5 for medium ones, and a 4 for big values.
This function, however, is not very smooth. To overcome this deficiency, we might use exponential fuzzy sets and a different logic.

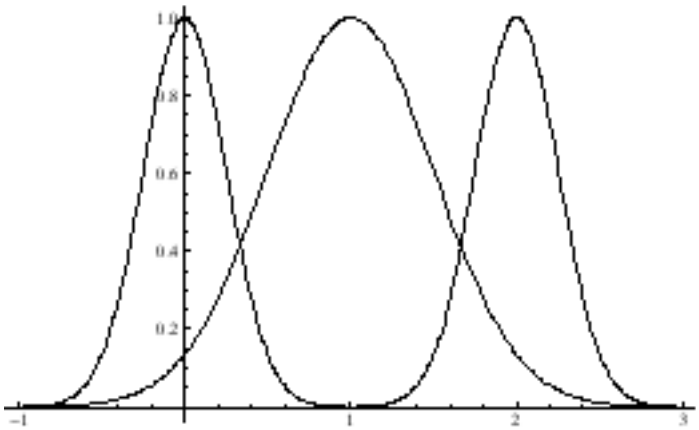
■ **A more complex example**

This is basically the same example as above, but now we want to create a two-dimensional controller.

First we consider three fuzzy sets and the corresponding predicates.

```
fsSmall := FuzzySetExp[0, 0.25]
fsMedium := FuzzySetExp[1, 0.5]
fsBig := FuzzySetExp[2, 0.25]
```

```
PlotFuzzySet[{fsSmall, fsMedium, fsBig}]
```




```
fpAisSmall := FColPredIsAM[1, fsSmall]
fpAisMedium := FColPredIs[1, fsMedium]
fpAisBig := FColPredIsAL[1, fsBig]
fpAisALMedium := FColPredIsAL[1, fsMedium]
fpAisAMMedium := FColPredIsAM[1, fsMedium]
fpBisSmall := FColPredIs[2, fsSmall]
fpBisMedium := FColPredIs[2, fsMedium]
fpBisBig := FColPredIs[2, fsBig]
```

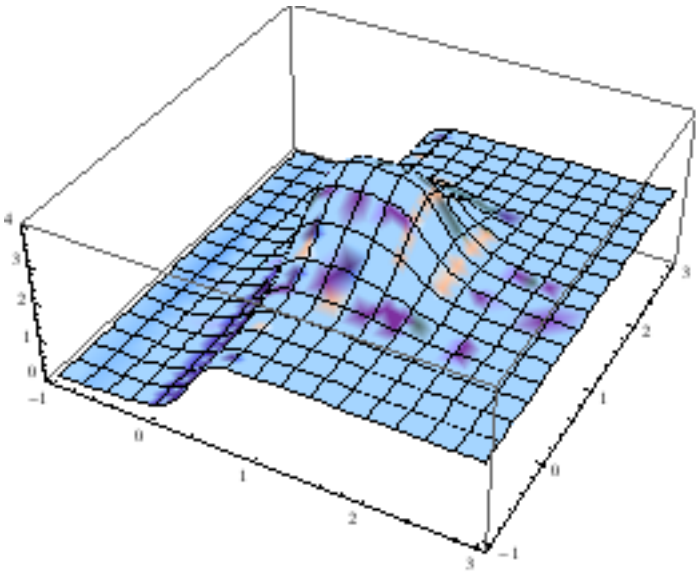
Using these fuzzy sets, we define three rules for our goal function.
To speed up later computations, we can keep the controller in *mlf* memory using the command `Def` .

```
logicRule1 := FRule[FAnd[fpAisALMedium, FOr[fpAisBig, FNot[fpBisMedium]]], NVal[2]]
logicRule2 := FRule[FAnd[fpAisMedium, fpBisMedium], NVal[4]]
logicRule3 := FRule[fpAisSmall, NVal[0]]
controller = FRuleSetSugeno[{logicRule1, logicRule2, logicRule3}, LogicP];
mySugenoCont = Def["MyController", controller];
```

To see how our controller behaves, we plot the result of the inference process.

```
newFunc2D[x_List] :=
 FuzzyInferenceVal[mySugenoCont, x]
```

```
Plot3D[newFunc2D[{x, y}], {x, -1, 3}, {y, -1, 3}, PlotRange → All]
```



Mamdani inference

Sugeno inference is applicable for most kinds of practical problems. However, an implementation of Mamdani inference is also available. While the result of Sugeno inference is always a single numerical value, the result of Mamdani inference is a symbolic fuzzy set. Currently symbolic inference is restricted to piecewise-linear fuzzy sets. It is suggested to use Mamdani inference for testing or educational purposes only. For practical problems, however, one might prefer Sugeno inference.

Parameter style (one-dimensional)

First we define three fuzzy sets.

```
fsSmall := FuzzySetPWL[{{-1, 0}, {0, 1}, {1, 0}}]
fpSmall[x_] := FPredIs[NVal[x], fsSmall]
fsMedium := FuzzySetPWL[{{0, 0}, {1, 1}, {2, 0}}]
fpMedium[x_] := FPredIs[NVal[x], fsMedium]
fsBig := FuzzySetPWL[{{1, 0}, {2, 1}, {3, 0}}]
fpBig[x_] := FPredIs[NVal[x], fsBig]
```

We could also have given the three b-spline fuzzy sets instead of the above piecewise-linear fuzzy sets.

```
fsSmallBS := fsBS[0]
fsMediumBS := fsBS[1]
fsBigBS := fsBS[2]
```

Using these fuzzy sets, we define three rules for the two parameters x and y.

```
logicRule1[x_, y_] := FRule[FOr[fpSmall[x], fpMedium[y]], fsBig]
logicRule2[x_, y_] := FRule[FAnd[fpMedium[x], fpMedium[y]], fsMedium]
logicRule3[x_, y_] := FRule[FOr[fpBig[x], fpSmall[y]], fsSmall]
logicRuleSet[x_, y_] :=
 FRuleSetMamdani[{logicRule1[x, y], logicRule2[x, y], logicRule3[x, y]}, LogicM]
```

We can now evaluate this rule set for a specific mapping of x and y.

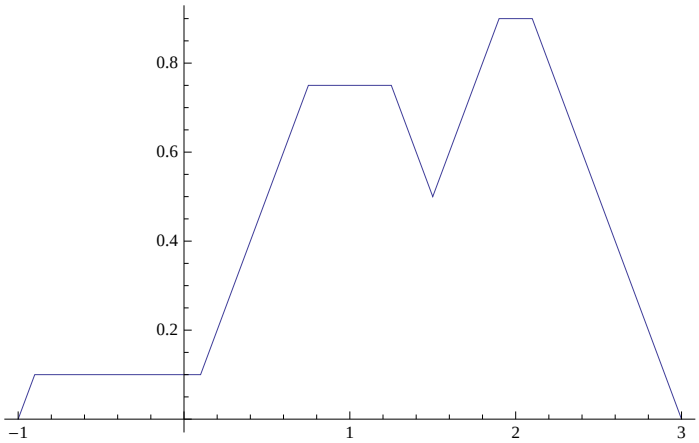
```
FuzzyInferenceVal[logicRuleSet[0.75, 0.9]]
```

```
1.40506
```

This evaluation created only a single value. We will now compute the respective fuzzy set, assign it to a variable, and then plot its contours.

```
fsResult = FuzzyInference[logicRuleSet[0.75, 0.9]];
```

```
PlotFuzzySet[fsResult]
```



To obtain a single value for our goal parameter, we can defuzzify the final fuzzy set (currently the center of gravitiy is computed for defuzzyfication).

```
EvalObject[fsResult]
```

```
1.40506
```

- **Cached style (two-dimensional)**

For the upcoming example, we use the following three fuzzy sets.

```
fsSmall := FuzzySetPWL[{{1, 0}, {2, 1}, {3, 1}, {4, 0}}]
fsMedium := FuzzySetPWL[{{2, 0}, {3, 1}, {4, 1}, {5, 0}}]
fsBig := FuzzySetPWL[{{3, 0}, {4, 1}, {5, 1}, {6, 0}}]
fpAisSmall := FColPredIs[1, fsSmall]
fpAisMedium := FColPredIs[1, fsMedium]
fpAisBig := FColPredIs[1, fsBig]
fpBisSmall := FColPredIs[2, fsSmall]
fpBisMedium := FColPredIs[2, fsMedium]
fpBisBig := FColPredIs[2, fsBig]
```

Using these fuzzy sets, we define four rules for the two parameters x and y.

```
logicRuleSet :=
 FRuleSetMamdani[{
    FRule[FAnd[fpAisSmall, fpBisMedium], fsBig],
    FRule[FAnd[fpAisMedium, fpBisBig], fsMedium],
    FRule[FAnd[fpAisMedium, fpBisSmall], fsBig],
    FRule[FAnd[fpAisBig, fpBisMedium], fsSmall]
  }]
myContr = Def["RuleSet", logicRuleSet];
```

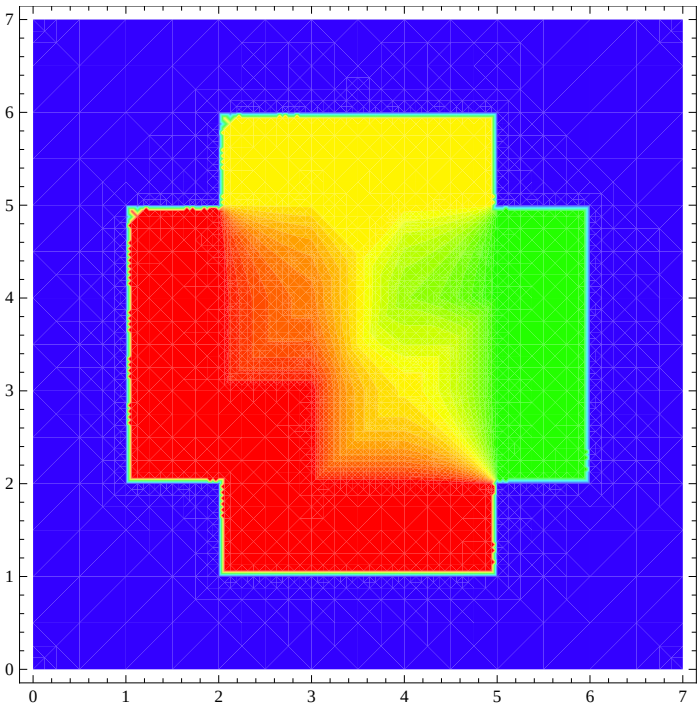
Taking the rule set as a named object increases computational speed, as the binary representation of the rule set is kept in *mlf* memory and does not need to be restored each time the rule set is evaluated.
We can now evaluate this rule set for a specific mapping of x and y.

```
inference[x_, y_] :=
 FuzzyInferenceVal[myContr, {x, y}]
```

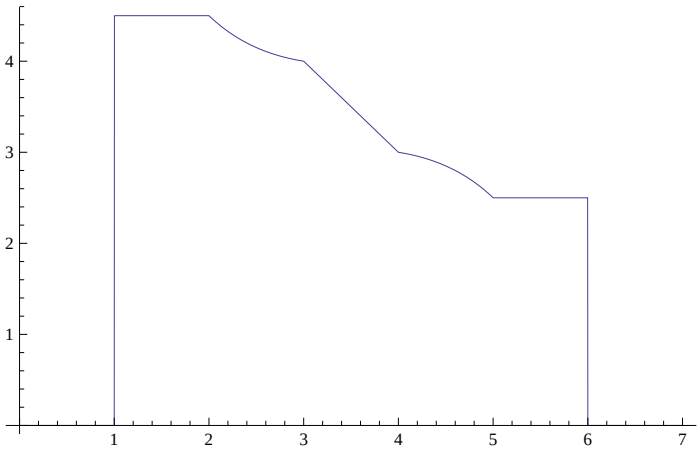
```
inference[3, 4]
```

```
4.
```

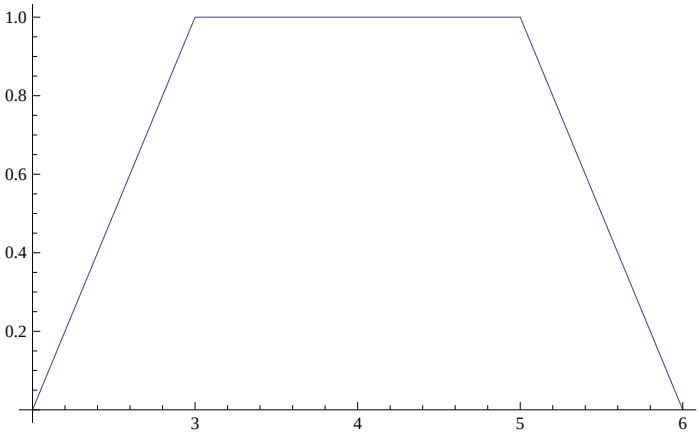
```
ContourPlot[inference[x, y], {x, 0, 7}, {y, 0, 7},
 ColorFunction → (CFTemp2@# &), ContourLines → False, Contours → 50]
```




```
Plot[inference[x, 4], {x, 0, 7}]
```




```
PlotFuzzySet[FuzzyInference[myContr, {3, 4}]]
```

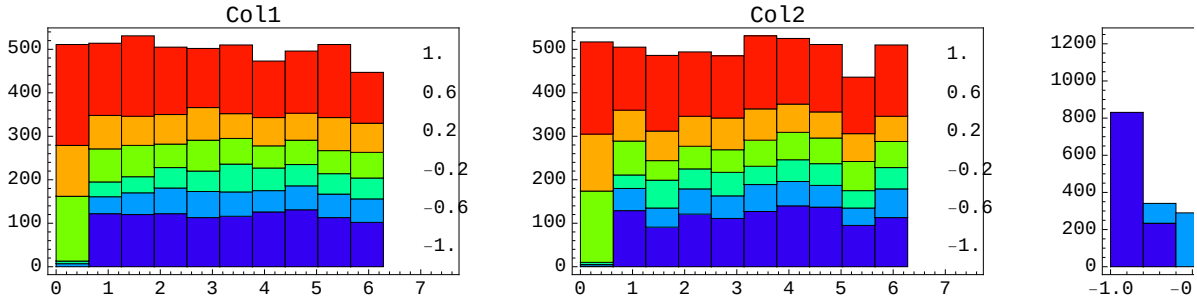


Visualization

```
myData = DataSet[Table[
     {x = Random[] * 2 * Pi,
      y = Random[] * 2 * Pi,
      z = Sin[x * y]},
     {i, 5000}]];
```

To get a first impression of how a data set is distributed, we can plot histograms of the three dimensions where the data are colored w.r.t. the goal column.

```
PlotHistogram[myData, Goal → 3, ColorBar → True, ImageSize → 800]
```

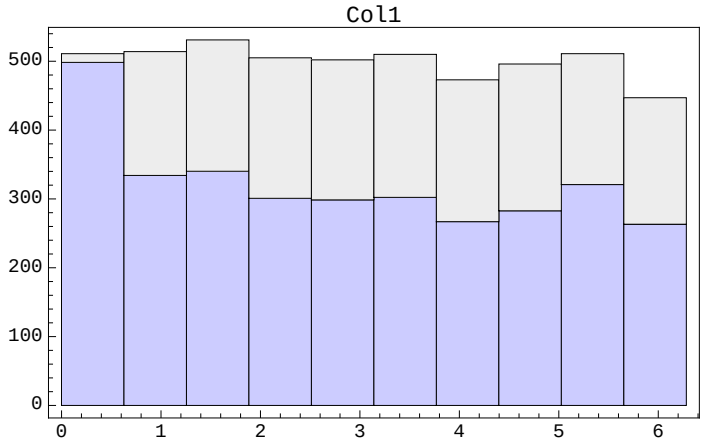


Blue represents samples where z (Col3) is very low, while red represents samples where z is very high.

We can also see what the distribution of the data with respect to a particular condition is. For instance, we can check the distribution of x with respect to the condition that z is at least 0.2, or more specifically, that z is at least within a (tri-cubic) fuzzy set with center 0.2:

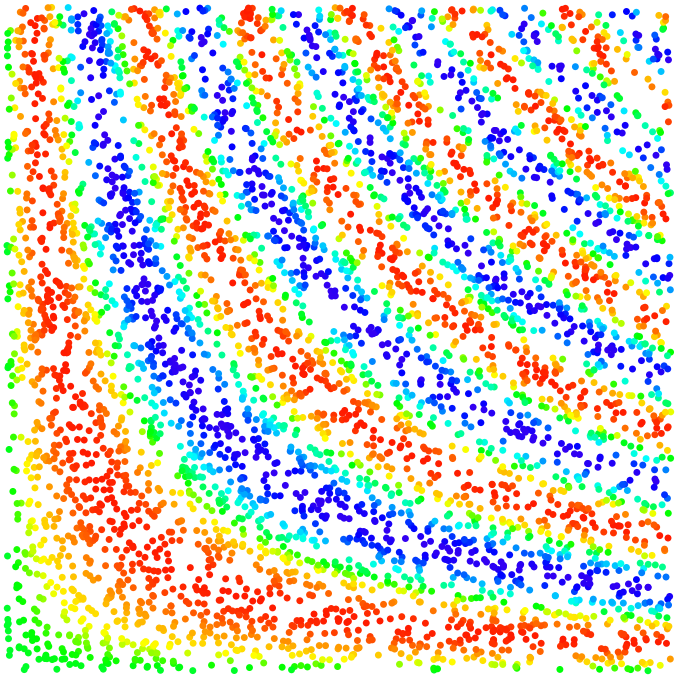
```
selection1 = FColPredIsAL[3, FuzzySetTriCube[0.2, 0.3]];
```

```
PlotHistogram[myData, 1, Goal → selection1]
```



The distribution of the data can also be visualized with a scatter plot for two dimensions and coloring the output according to another dimension. We do this using the command `PlotAttributes`, with the data as the first argument and the dimensions to plot and the goal column as options.

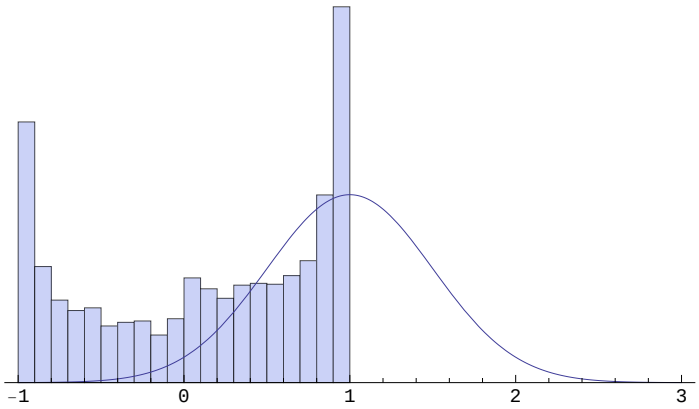
```
Show[Graphics[PlotAttributes[myData, Dims → {1, 2}, Goal → 3, PointSize → 0.01]]]
```



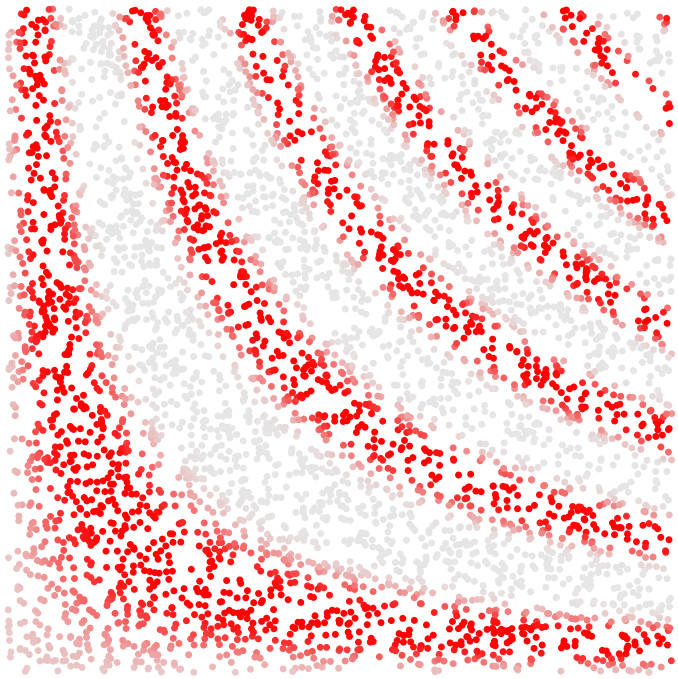
To visualize a specific part of the data, we can use the command `PlotFuzzySetHistogram` with predicates as goal parameter. The resulting graph shows all samples which satisfy the predicate in red, while all other samples are shown in gray.

```
curPred = FColPredIsAL[3, FuzzySetExp[1, 0.5]];
```

```
PlotFuzzySetHistogram[curPred[[3]], MLFGetData[myData, All, 3]]
```



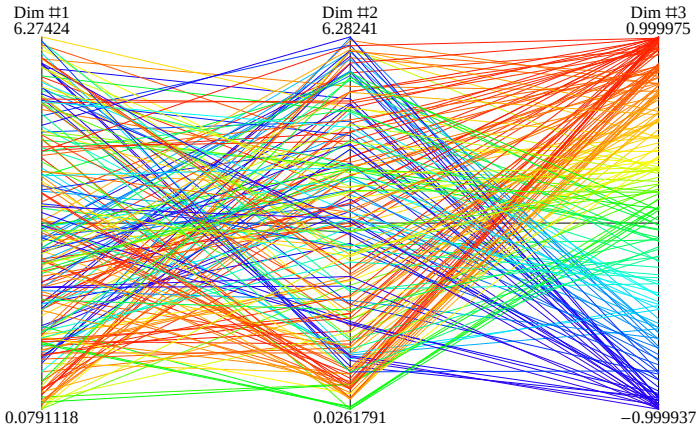
```
Show[Graphics[PlotAttributes[myData,
    Dims → {1, 2},
    Goal → curPred,
    PointSize → 0.01
   ]]]
```



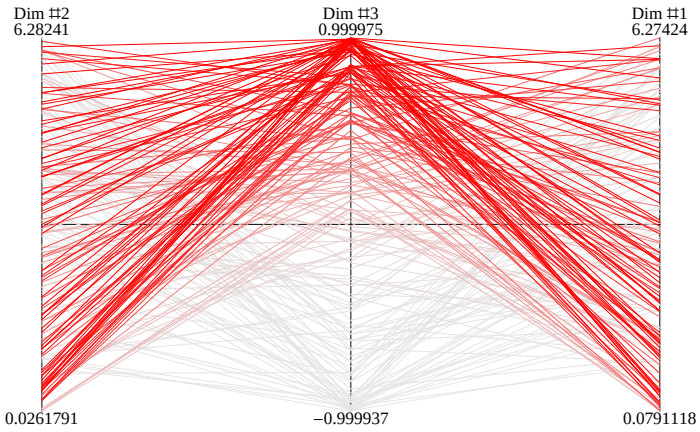
Another important visualization technique makes use of parallel coordinates to visualize high-dimensional data. In *mlf*, the command `PlotParallelCoordinates` can be used to create such a plot. By specifying the goal parameter, it is possible to set the color coding of the output. If no goal parameter is specified, the lines are colored according to their record number.

```
myData = Table[
    {x = Random[] * 2 * Pi,
     y = Random[] * 2 * Pi,
     z = Sin[x * y]},
    {i, 200}];
```

```
PlotParallelCoordinates[myData, Goal → 3]
```



```
PlotParallelCoordinates[myData, Dims → {2, 3, 1},
 Goal → FColPredIsAL[3, FuzzySetExp[1, 0.5]]]
```



Saving/Loading Computations and Expressions to/from an XML File

In order to save the current state of the system, the command `SaveState` is available. This command writes an XML file which contains all objects that are currently defined in the *mlf* memory. You can see which objects are currently defined by applying the function `GetMembers`.

To clear all objects in *mlf* memory at once, the command `ClearState` is used; this can actually be seen as re-initializing *mlf*.

```
ClearState[]
```

```
State of mlf completely cleared
```

```
GetMembers[]
```

```
{_EvalMissingVals}
```

```
Def["ObjectA", NVal[5.1]];
Def["ObjectB", IVal[3]];
Def["ObjectC", NVal[12.4]];
```

```
GetMembers[]
```

```
{ObjectA, ObjectB, ObjectC, _EvalMissingVals}
```

```
Off[mlf::info]
```

```
SaveState["mathData.xml"]
```

```
1
```

To load all the objects previously stored, the command `ReadState` is used. This resets the system to the state which was stored in the xml file.

```
ReadState["mathData.xml"]
```

```
1
```

```
GetObject["ObjectA"]
```

```
Obj`RealConst[5.1]
```

It is also possible to save not all objects but only a specific set of objects to file, which is done using the command `SaveObjects`.

```
Def["Object1", NVal[5.1]];
Def["Object2", IVal[3]];
```

```
SaveObjects["mathData2.xml", {"Object1", "Object2"}];
```

To add all the objects stored in a distinct file to the current state, `ReadObjects` can be used. This command works similar to `ReadState` but preserves all previously defined objects and prints out a warning if an existing

object is overridden.

```
DelObject["Object1"];
```

```
ReadObjects["mathData2.xml"]
```

```
1
```

```
GetMembers[]
```

```
{Object1, Object2, ObjectA, ObjectB, ObjectC, _EvalMissingVals}
```

```
GetObject["Object1"]
```

```
Obj`RealConst[5.1]
```