DTREG

Predictive Modeling Software

Phillip H. Sherrod

www.dtreg.com

DTREG (pronounced D-T-Reg) builds classification and regression decision trees, neural networks, support vector machine (SVM), GMDH polynomial networks, gene expression programs, K-Means clustering, discriminant analysis and logistic regression models that describe data relationships and can be used to predict values for future observations. DTREG also has full support for time series analysis.

DTREG accepts a dataset containing of number of rows with a column for each variable. One of the variables is the “target variable” whose value is to be modeled and predicted as a function of the “predictor variables”. DTREG analyzes the data and generates a model showing how best to predict the values of the target variable based on values of the predictor variables.

DTREG can create classical, single-tree models and also TreeBoost and Decision Tree Forest models consisting of ensembles of many trees. DTREG also can generate Neural Networks, Support Vector Machine (SVM), Gene Expression Programming/Symbolic Regression, K-Means clustering, GMDH polynomial networks, Discriminate Analysis, Linear Regression, and Logistic Regression models.

DTREG includes a full Data Transformation Language (DTL) for transforming variables, creating new variables and selecting which rows to analyze.

Contents

Data Mining and Modeling

"*Predicting the future is hard, especially if it hasn't happened yet.*"
– Yogi Berra

Data Mining

The process of extracting useful information from a set of data values is called "**data mining**". Many techniques have been developed for data mining, and there is an art to selecting and applying the best method for a particular situation.

DTREG (pronounced D-T-Reg) builds classification and regression decision trees, neural networks, support vector machine (SVM), gene expression programming (GEP), K-Means clustering, discriminant analysis and logistic regression models that describe data relationships and predict values for future observations.

DTREG also has full support for time series analysis. Most of the model types such as neural networks, gene expression programming and SVM can be used to model time series using lag variables generated by DTREG.

Data mining has great commercial and scientific value. Consider these cases:

1. A company has collected data showing how much of their product consumers buy. For each consumer, the company has demographic and economic information such as age, gender, education, hobbies, income and occupation. Since the company has a limited advertising budget, they want to determine how to use the demographic data to predict which people are the most likely buyers of their product so they can focus their advertising on that group. A decision tree is an excellent tool for this type of analysis because it shows which combination of attributes best predict the purchase of the product. And, a decision tree can be used to "score" a set of individuals and rank them by the probability that they will respond positively to a marketing effort. For information about how Lift and Gain tables and charts are used for customer targeting, please see page 201.

2. A political campaign wants to maximize the turnout of their supporters on Election Day. Exit polling has been done during previous elections giving a breakdown of voting patterns by precinct, race, gender, age and other factors. DTREG can analyze this data and generate a decision tree identifying which sets of voters should be targeted for get-out-the-vote efforts for upcoming elections.

3. A bank wants to reduce the default rate on personal loans. Using historical data collected for previous borrowers, the bank can use DTREG to generate a decision tree that can then be used to "score" candidate borrowers to predict the likelihood

that they will default on their loans.

4. An emergency room treats patients with chest pain. Based on factors such as blood pressure, age, gender, severity of pain, location of pain, and other measurements, the caregiver must decide whether the pain indicates a heart attack or some less critical problem. A decision tree can be generated to decide which patients require immediate attention.

Data Modeling

One of the most useful applications of statistical analysis is the development of a model to represent and explain the relationship between data items (variables). Many types of models have been developed, including linear and nonlinear regression (function fitting), discriminant analysis, logistic regression, support vector machines, neural networks and decision trees. Each method has its advantages: there is no single modeling method that is best for all applications. DTREG provides the best, state-of-the-art modeling methods including neural networks, decision trees, TreeBoost, decision tree forests, support vector machines (SVM), gene expression programming, K-Means clustering, discriminant analysis and logistic regression. By applying the right method to the problem, the analyst using DTREG should be able to match or exceed the predictive ability of any other modeling program.

Supervised and Unsupervised Machine Learning

Methods for analyzing and modeling data can be divided into two groups: "*supervised learning*" and "*unsupervised learning.*" Supervised learning requires input data that has both predictor (independent) variables and a target (dependent) variable whose value is to be estimated. By various means, the process "learns" how to model (predict) the value of the target variable based on the predictor variables. Decision trees, regression analysis and neural networks are examples of supervised learning. If the goal of an analysis is to predict the value of some variable, then supervised learning is recommended approach.

Unsupervised learning does not identify a target (dependent) variable, but rather treats all of the variables equally. In this case, the goal is not to predict the value of a variable but rather to look for patterns, groupings or other ways to characterize the data that may lead to understanding of the way the data interrelates. Cluster analysis, correlation, factor analysis (principle components analysis) and statistical measures are examples of unsupervised learning.

Time Series Analysis

A time series is a sequence of values occurring over a period of time. Often time series describe economic conditions such as price fluctuations, hotel occupancy, airline passenger load, etc. DTREG can build models using methods such as neural networks, gene expression programming and SVM to model time series and make future forecasts.

Classes of Variables

You can specify three classes of variables when performing analyses:

Target variable -- The "target variable" is the variable whose values are to be modeled and predicted by other variables. It is analogous to the dependent variable (i.e., the variable on the left of the equal sign) in linear regression. There must be one and only one target variable.

Predictor variable -- A "predictor variable" is a variable whose values will be used to predict the value of the target variable. It is analogous to the independent variables (i.e., the variables on the right side of the equal sign) in linear regression. There must be at least one predictor variable specified, and there may be many predictor variables. If more than one predictor variable is specified, DTREG will determine how the predictor variables can be combined to best predict the values of the target variable. For time series analysis, DTREG can automatically generate lag variables that can be used as predictor variables.

Weight variable -- Optionally, you can specify a "weight variable". If a weight variable is specified, it must a numeric (continuous) variable whose values are greater than or equal to 0 (zero). The value of the weight variable specifies the weight given to a row in the dataset. For example, a weight value of 2 would cause DTREG to give twice as much weight to a row as it would to rows with a weight of 1; the effect on model training is the same as two occurrences of the row in the dataset. Weight values may be real (non-integer) values such as 2.5. A weight value of 0 (zero) causes the row to be ignored. If you do not specify a weight variable, all rows are given equal weight.

An integer weight value has the same effect on model training as duplicating rows the equivalent number of times in the training data. Since the goal of model training is to tune parameters to minimize the overall error (or variance) of the training data, weighted (or duplicated) rows that are misclassified add a greater amount to the total error than un-weighted rows, so they have an increased influence on the model.

While duplicating rows is equivalent to integer weighting during the training process, there is a difference during the testing and validation process. If you manually duplicate rows in the training data and specify that you want DTREG to use cross validation for testing or you want it to hold out a subset of the data for testing, some of the duplicated copies of the rows may be used for training, and some duplicate copies of the same rows may be used in the test/validation data. This results in the testing data using some of the same data used for training, and it renders the test results – which are supposed to be based on independent data – invalid. For this reason, if you manually duplicate rows rather than using weighting, you must do the validation using the Score function (see page 166) rather than using cross validation or hold-out validation.

Types of Variables

Variables may be of two types: ***continuous*** and ***categorical***.

Continuous variables with ordered values -- A continuous variable has numeric values such as 1, 2, 3.14, -5, etc. The relative magnitude of the values is significant (e.g., a value of 2 indicates twice the magnitude of 1). Examples of continuous variables are blood pressure, height, weight, income, age, and probability of illness. Some programs call continuous variables "ordered", "ordinal", "interval" or "monotonic" variables. If a variable is numeric and the values indicate relative magnitude or order, then the variable should be declared as continuous even if the numbers are discrete and do not form a continuous scale.

Categorical variables with unordered values -- A categorical variable has values that function as labels rather than as numbers. Some programs call categorical variables "nominal" variables. For example, a categorical variable for gender might use the value 1 for male and 2 for female. The actual magnitude of the value is not significant; coding male as 7 and female as 3 would work just as well. As another example, marital status might be coded as 1 for single, 2 for married, 3 for divorced and 4 for widowed. DTREG allows you to use non-numeric (character string) values for categorical variables. So your dataset could have the strings "Male" and "Female" or "M" and "F" for a categorical gender variable. Because categorical values are stored and compared as string values, a categorical value of 001 is different than a value of 1. In contrast, values of 001 and 1 would be equal for continuous variables.

Using DTREG

Once you understand the concept of predictive models, it is very easy to use DTREG to analyze data and build many types of models.

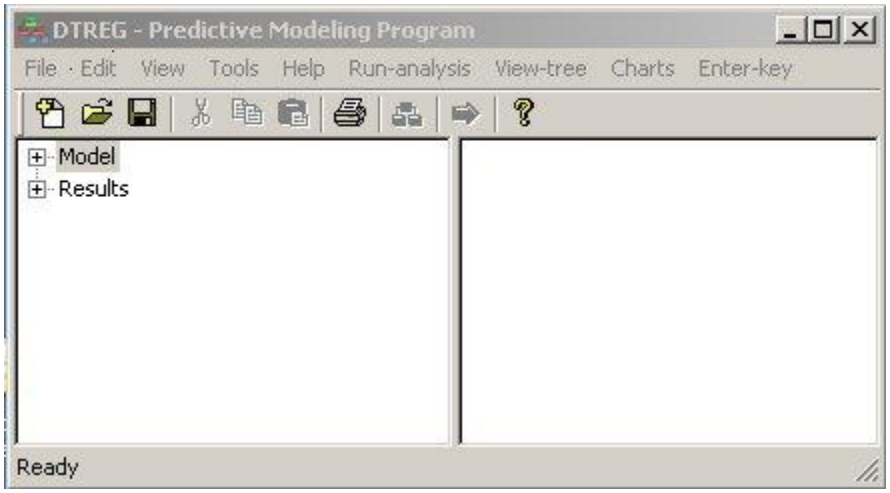
Installing DTREG

To install DTREG, run the installation program named **DTREGsetup.exe**. A "wizard" screen will guide you through the installation process. You can accept the default installation location (C:\Program files\DTREG) or select a different folder location. When the installation finishes, you should see this icon for DTREG on your desktop:

To launch DTREG, double-click the Shortcut to DTREG icon on your desktop.

DTREG's Main Screen

When you launch DTREG, its main screen displays:

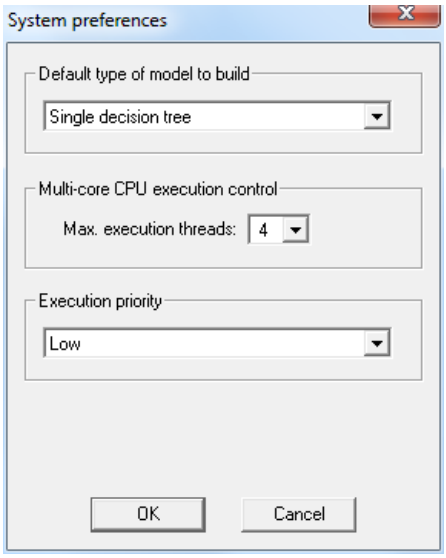


From this screen, you can

- Create a new project to build a model by clicking 
- Open an existing project by clicking 
- Set options and enter your registration key.

Setting DTREG Preferences

To set DTREG preferences, click “Tools” on the menu bar and select “Preferences” from the dropdown menu.



Default type of model to build: Select which type of model you would like DTREG to create for new projects (single tree, SVM neural network, etc.). You can always change the type of a model later by modifying its properties.

Max. execution threads: Specify how many execution threads you want DTREG to use during its computations. If you have a multi-CPU system, you can increase the speed of calculation by allowing DTREG to use more than one CPU, but this will place a heavier load on your system.

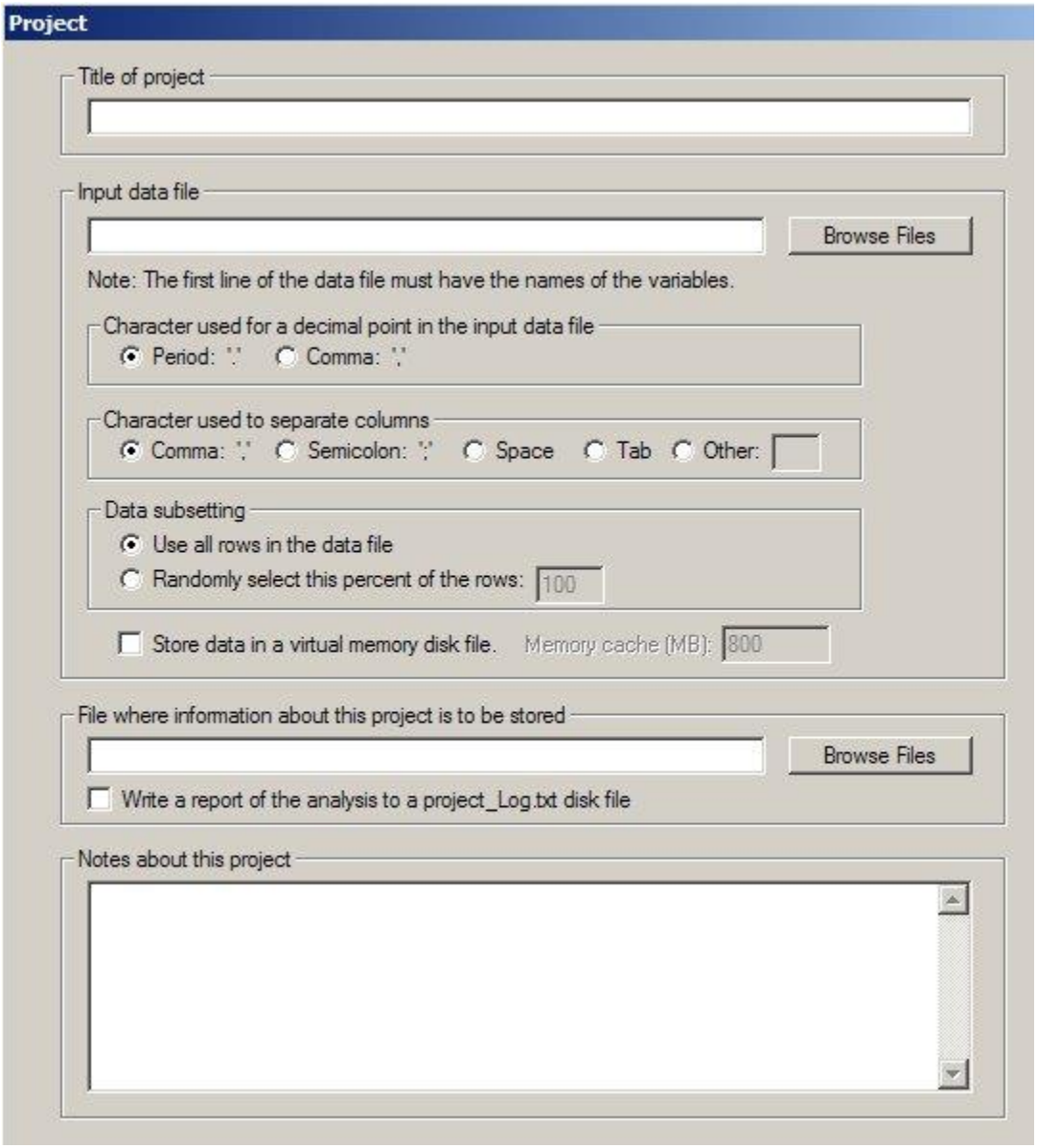
Execution priority: Specify the preferred execution for DTREG to use during an analysis. Currently the execution priority is only applied to neural network training processes.

Entering DTREG Registration Key

When you register DTREG, you will receive a registration key. To enter your key, click "Enter-key" on the main menu and enter your key in the screen that appears:

Creating a New Project

To create a new project, click the leftmost icon on the toolbar that looks like this:  Project "wizard" screens will guide you through setting up the project. The first screen looks like this:



There are several fields on this page.

- **Title of project** – This is an optional field. If you wish, you can specify a title to be displayed for this project.
- **Input data file** – This is a required field. Specify the device, folder and name of the file containing the input (learning) dataset to be used to build the tree. The data must be in a comma separated value (CSV) file with the names of the

variables on the first line. Please see page 36 for detailed information about the format of input data files. You can click the "Browse files" button to browse for the file rather than typing it in.

- **Character used for a decimal point in the input data file** – Select whether a period or a comma will be used to indicate the decimal point in numeric values in the input data file. The American standard decimal point marker is a period while the European standard is a comma. This setting affects only data read from the input file; a period always is used as the decimal point marker in the generated report.
- **Character used to separate columns** – Select the character that will be used to separate columns in the input file. The default separator is a comma, but you may select any character you wish to use.
- **Data subsetting** – If you wish, you can tell DTREG to use only a subset of the records in the data file for the analysis. This speeds up the analysis and is useful when experimenting with different model settings. If you tell DTREG to use a subset of the data, specify the percentage of the rows that you want it to use. Since random selection is used to select the rows, the actual number of rows used may be slightly different than the percent you specify.
- **File where information about this project is to be stored** – This is a required field. Specify the name of the project file where DTREG will store parameters and computed values for the project. DTREG project files are stored with the type "**.dtr**" (for example, "Iris.dtr"). You can click the "Browse file" button to browse for the directory where you want to store the file.
- **Notes about this project** – This is an optional field. You can enter any notes that you want to store with the project data.

After you finish filling in these fields, click the "Next" button at the bottom of the screen to advance to the next screen. The following property pages will be displayed:

- Time series/Regular predictive model (see page 41)
- Variables (see page 41)

New Project Example

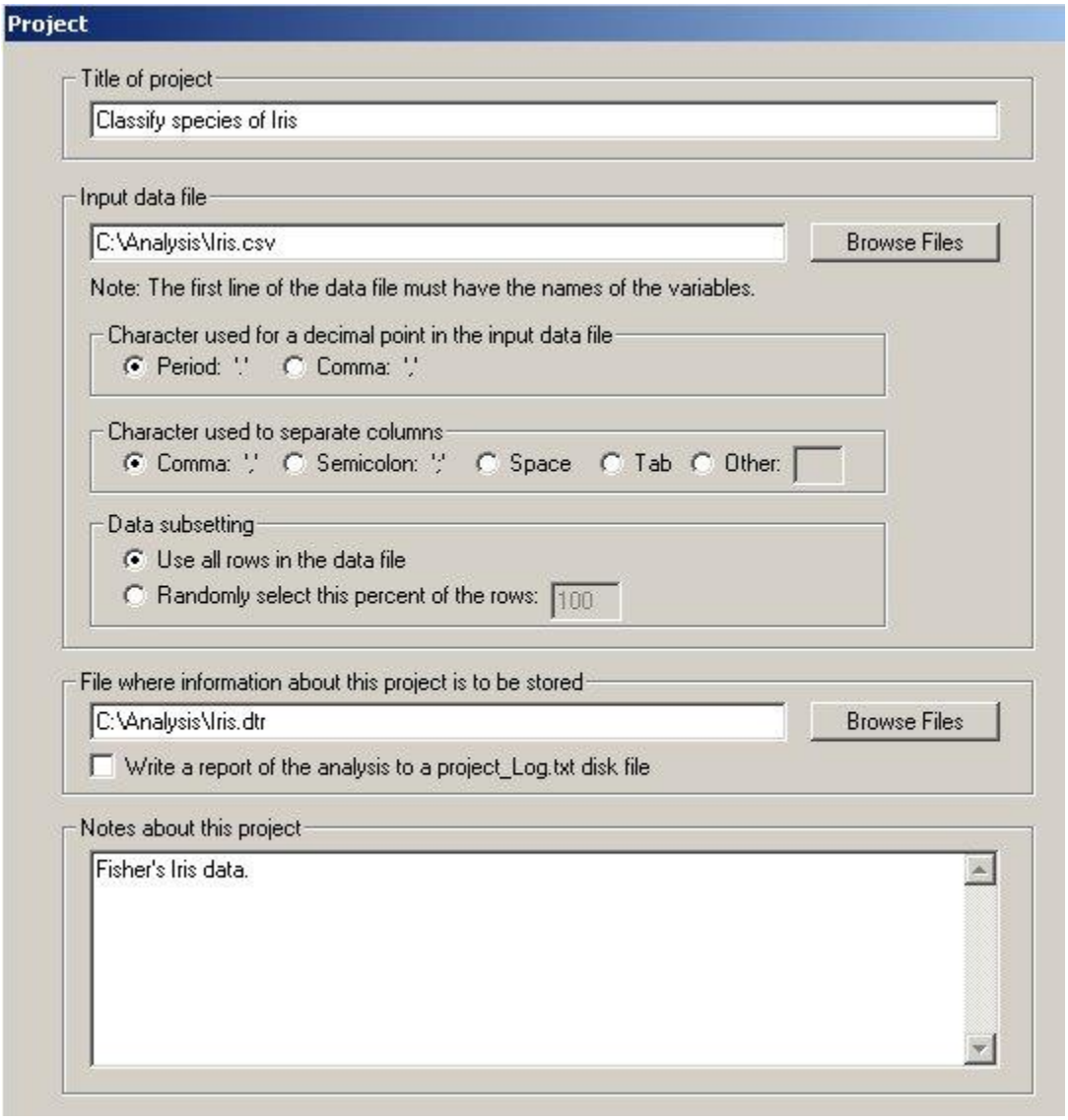
To illustrate the process of creating a new project, let's consider a concrete case. We will look at the steps involved in setting up a DTREG project to classify species of irises based on measurements of the plants. The data we will use is from the classic study devised by R. A. Fisher in 1936 (Fischer, 1936). First, we need to prepare a data file to be read by DTREG. Such an example data file is provided with the DTREG distribution and installed in the Examples directory under the DTREG installation directory. The name of the file is Iris.csv. Here are a few lines from that file:

```
Species,"Sepal length","Sepal width","Petal length","Petal width"
Setosa,5.1,3.5,1.4,0.2
Setosa,4.9,3,1.4,0.2
Setosa,4.7,3.2,1.3,0.2
Versicolor,7,3.2,4.7,1.4
Versicolor,6.4,3.2,4.5,1.5
Versicolor,6.9,3.1,4.9,1.5
Virginica,6.3,3.3,6,2.5
Virginica,5.8,2.7,5.1,1.9
Virginica,7.1,3,5.9,2.1
```

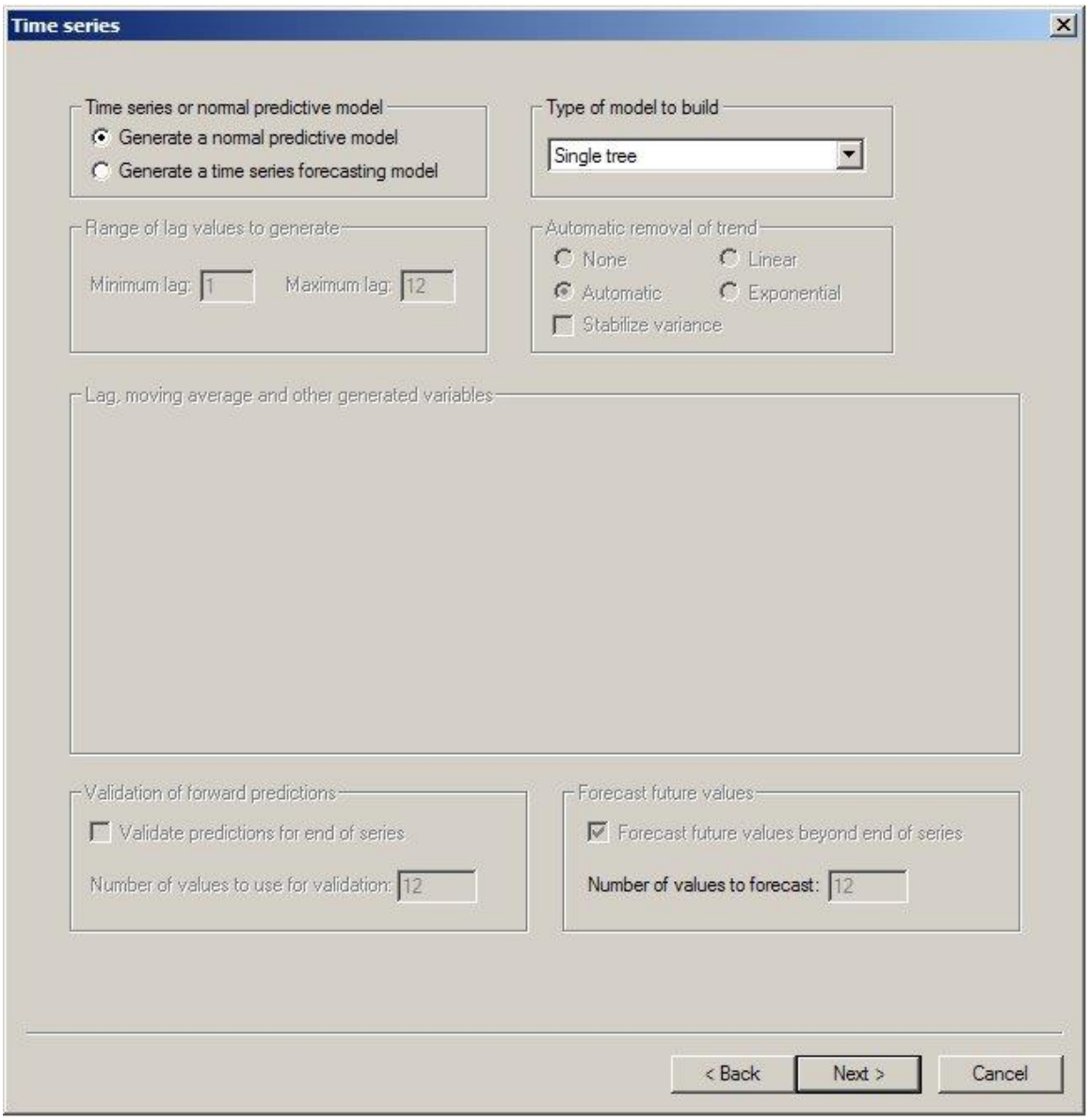
The first line of the file has the names of the variables separated by whatever character you selected as the column delimiter (by default it is a comma). In this case, there are 5 variables: Species, Sepal length, Sepal width, Petal length and Petal width. Variable names and values that contain spaces or the column separator character should be enclosed in quote marks. The records following this are the actual data observations (one per plant). There is one value for each of the five variables. See page 36 for additional information about the format of data files.

In this example, we are trying to predict the species of iris, so "Species" is a categorical target variable. The other four variables are continuous predictor variables.

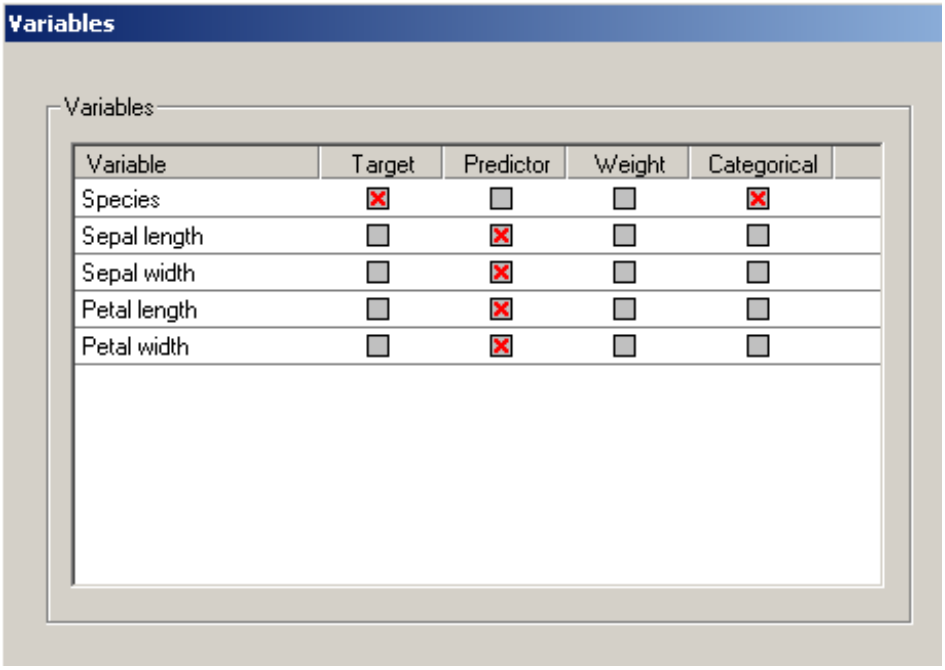
Here is the first screen we set up for this project:



On the second screen specify whether this is a time series analysis or a regular predictive model. Also, select the type of model to build (single tree, TreeBoost, neural network, etc.)

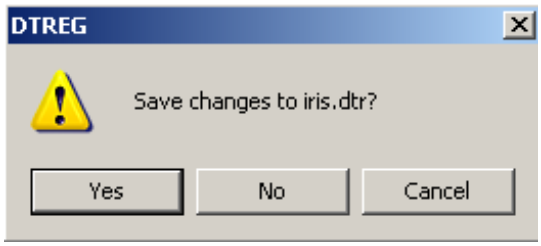


On the third screen, specify information about the variables:



Species is the target variable, and it is categorical. The other four variables are continuous predictor variables.

After you finish the last setup screen for the project, DTREG asks if you want to save the settings for the project:



We will click "Yes" and save the project settings in a file named Iris.dtr.

Opening an Existing Project

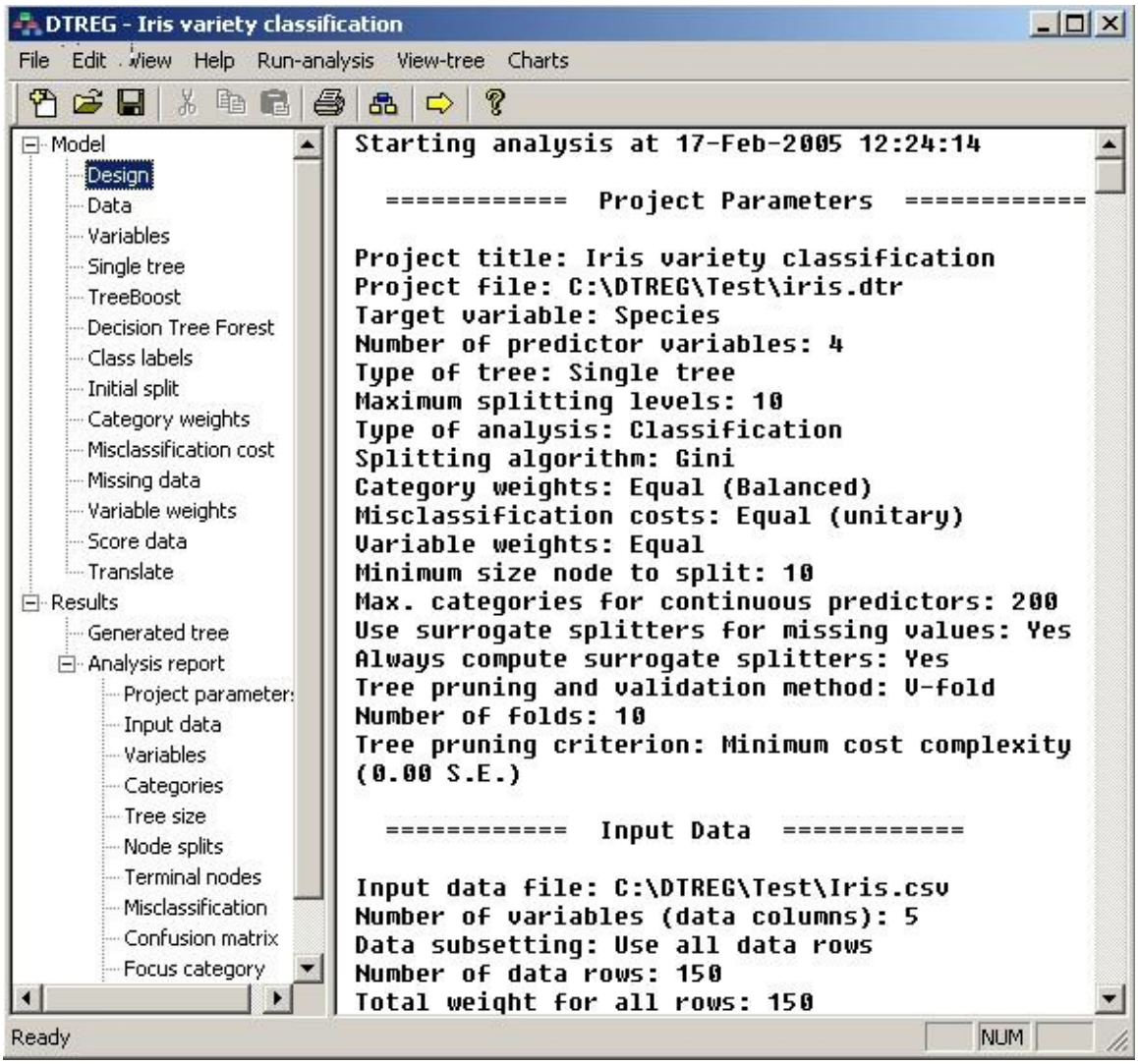
All of the information about a DTREG project is stored in a project database. This includes parameters that control the analysis, information about variables, the name of the data input file, the generated report, and information required to construct and display the generated predictive model. These project files have the file type ".dtr". You can open project files, examine the report, modify parameters and rerun the analysis.

The actual input data is not stored in the project file but remains in the original comma separated value (CSV) file. The project file stores only the name of the input data file.

To open an existing project file, click the  icon on the toolbar.

If you are reopening a project that was opened recently, you can click the "File" entry on the main menu line, and select the project from the list of recent projects.

Once you open a project, the last report generated for it will be displayed in the right panel, and the left panel will show a list of property pages you can select to review and change option settings.



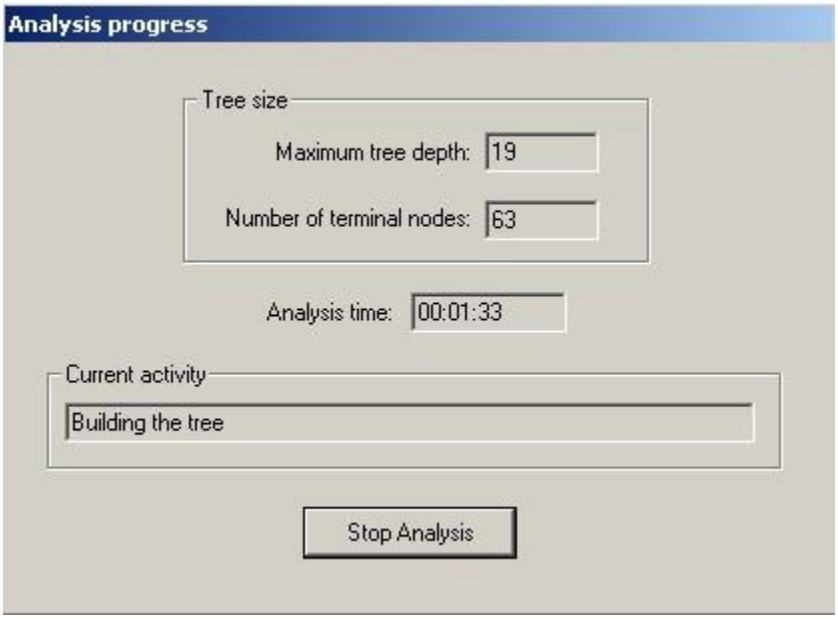
Example Projects Installed With DTREG

The DTREGsetup installation program installs a set of example projects in a folder named "Examples" under the DTREG installation directory. This is C:\Program files\DTREG\Examples, unless you selected a different folder during installation. A good way to get started using DTREG is to browse the examples in that directory and run some of them. See page 373 for additional information about example analyses.

Running an Analysis

Once you have created a new project or opened an existing project, you can tell DTREG to perform an analysis. To do this, click the  icon on the toolbar. You can also click "Run-analysis" on the main menu.

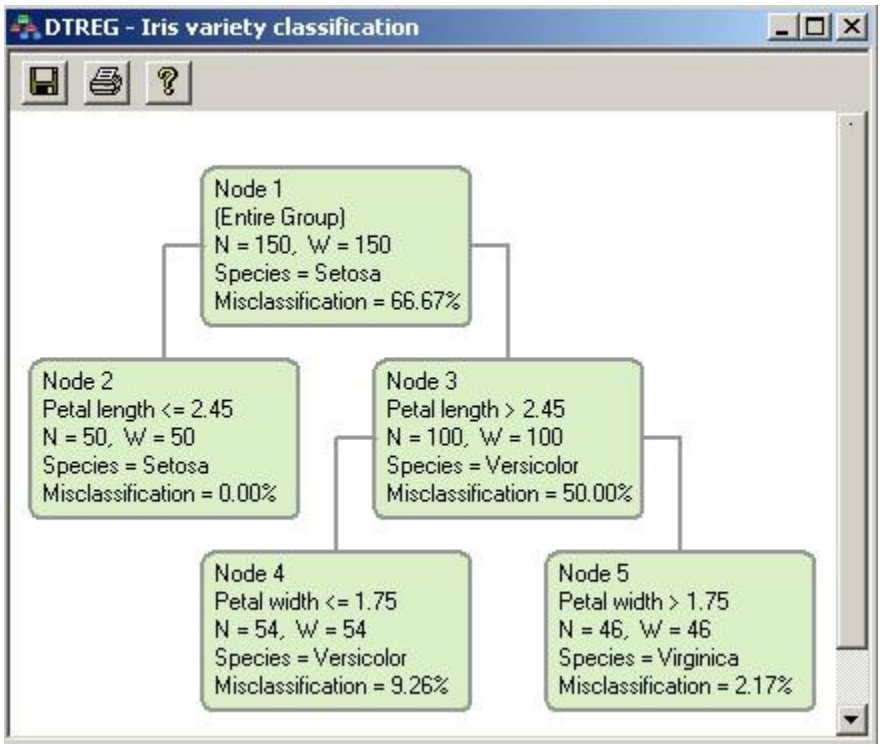
While an analysis is running, a progress screen similar to this will be displayed:



When the analysis finishes, the new report will be displayed in the main right panel.

Viewing the Generated Decision Tree

Once an analysis has been completed, you can view the generated decision tree by clicking the toolbar icon or by clicking "View-tree" on the main menu. To save the decision tree in a jpg, png or bmp disk file, click the disk icon. To print the decision tree, click the printer icon.



Command Line Operation

In production environments it may be useful to operate DTREG in command-line mode to build models. Model building parameters are stored in a command file and in a "template project". DTREG can then be run from the command line or using a batch (.bat) file to build new models. Command line operation is available only in the Enterprise Version of DTREG.

In order to use DTREG in command line mode, three things are required: (1) a template project describing the analysis; (2) a command file providing information about files and operations; (3) a command line to invoke DTREG.

The Template Project

Because DTREG has many types of models and many options and parameters for each type of model, it is impractical to have a command language to describe all of these features. Instead, a *Template Project* is used to describe the type of model to be created and to specify options and parameters for the operation. To create a template project, run

DTREG in interactive mode, specify a data file with the variables to be analyzed, select the type of model to build, and select options and parameters. Then save the project in a standard DTREG project file (.dtr file).

The Command File

The command file is a text file that contains commands that control the model building process. You can use Notepad, Wordpad or any other text editor to create the command file. The suggested file type is .cmd, but you can use any extension you wish.

The following commands may be placed in a command file. Some commands are required, and some are optional.

FOLDER (optional) – Specifies a default folder where all of the files are specified. If the FOLDER command is not specified, you must specify the folder as part of the file specification on each command.

Syntax: FOLDER *device_and_folder*

Example: FOLDER C:\Work\Campaign1

PROJECT (required) – Specifies the name of the file with the template project.

Syntax: PROJECT *file_name*

Example: PROJECT C:\Campaign1\AdModel.dtr

OUTPUT (optional) – Specifies the name of the file where the generated model is to be written. If no OUTPUT command is specified, then the model is not saved. If you simply want to use the template project to score data, it is not necessary to specify an OUTPUT command.

Syntax: OUTPUT *file_name*

Example: OUTPUT C:\Campaign1\AdTreeBoost.dtr

DATA (optional) – Specifies the name of the data file that will be used to train the model. If you are just performing scoring and not building a model, then you can omit the DATA command.

Syntax: DATA *file_name*

Example: DATA C:\Campaign1\Tennessee.csv

REPORT (optional) – Specifies the name of the file where the analysis report is to be written. If you do not use a REPORT command, then the analysis report is not saved.

Syntax: REPORT *file_name*

Example: REPORT C:\Campaign1\TennesseeLog.txt

SCOREINPUT (optional) – Specifies the name of a data file is to be used as input for scoring by the generated model. If you want scoring done, you must specify both a SCOREINPUT and a SCOREOUTPUT command. Omit the SCOREINPUT and SCOREOUTPUT commands if you are building a model and do not want scoring done.

Syntax: SCOREINPUT *file_name*

Example: SCOREINPUT C:\Campaign1\Nashville.csv

SCOREOUTPUT (optional) – Specifies the name of the data file where output from the scoring function (with predicted target values) is to be written.

Syntax: SCOREOUTPUT *file_name*

Example: SCOREOUTPUT C:\Campaign1\NashvillePredict.csv

BUILDMODEL (optional) – Specify this command if you want DTREG to build a new predictive. If you only want to use the template project to do scoring, you should omit the BUILDMODEL command.

Syntax: BUILDMODEL

TRANSLATE (optional) – Specifies that DTREG is to convert the model to C source code and write it to the specified file. This option is available only in the Enterprise Version of DTREG.

Syntax TRANSLATE *file_name*

Example: TRANSLATE C:\Campaign1\NeuralCode.c

REM (optional) – Comment line.

Example: REM Analysis of Tennessee data

Example Command Files

This is an example command file to build a model:

```
REM Build a new model whose name is NewProject.dtr
FOLDER  C:\Campaign1
PROJECT  OriginalProject.dtr
OUTPUT  NewProject.dtr
DATA  TrainingData.csv
BUILDMODEL
```

This is an example file that does scoring using an existing project but which does not build a new project:

```
REM  Score the Tennessee.csv file
FOLDER  C:\Campaign1
PROJECT  ResponseModel.dtr
SCOREINPUT  Tennessee.csv
SCOREOUTPUT  TennesseeScore.csv
```

The Command Line

The command file to start DTREG in command line mode is:

DTREG /cmd="*command_file*" [/*options*]

The /cmd="*command_file*" switch specifies the name of the command file that is to be executed. You should provide a full file specification including device and folder. For example:

DTREG /cmd="C:\Campaign1\BuildModel.cmd"

Options:

The following optional switches may be specified:

/MINIMIZE – Start DTREG in minimized mode so that it does not display its screen.

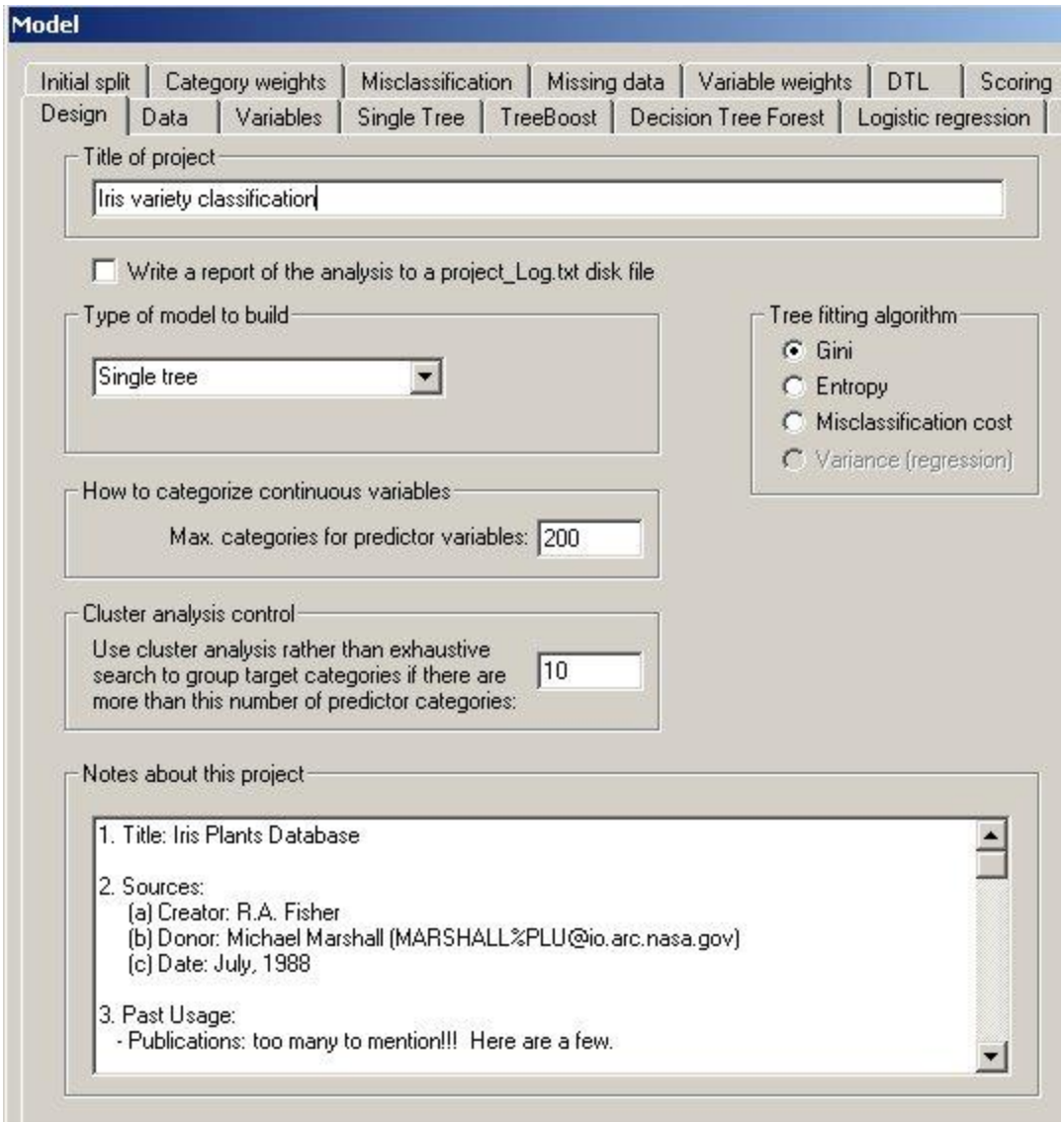
/HIDE – Do not display the execution screen at all.

Specifying Properties for a Model

You can specify properties for a model when you create it initially or you can change the properties for a project you have already created. The properties for a model display in the left panel and correspond to the project property screens.

To specify properties for a model, click one of the items shown under "Model" in the left panel:

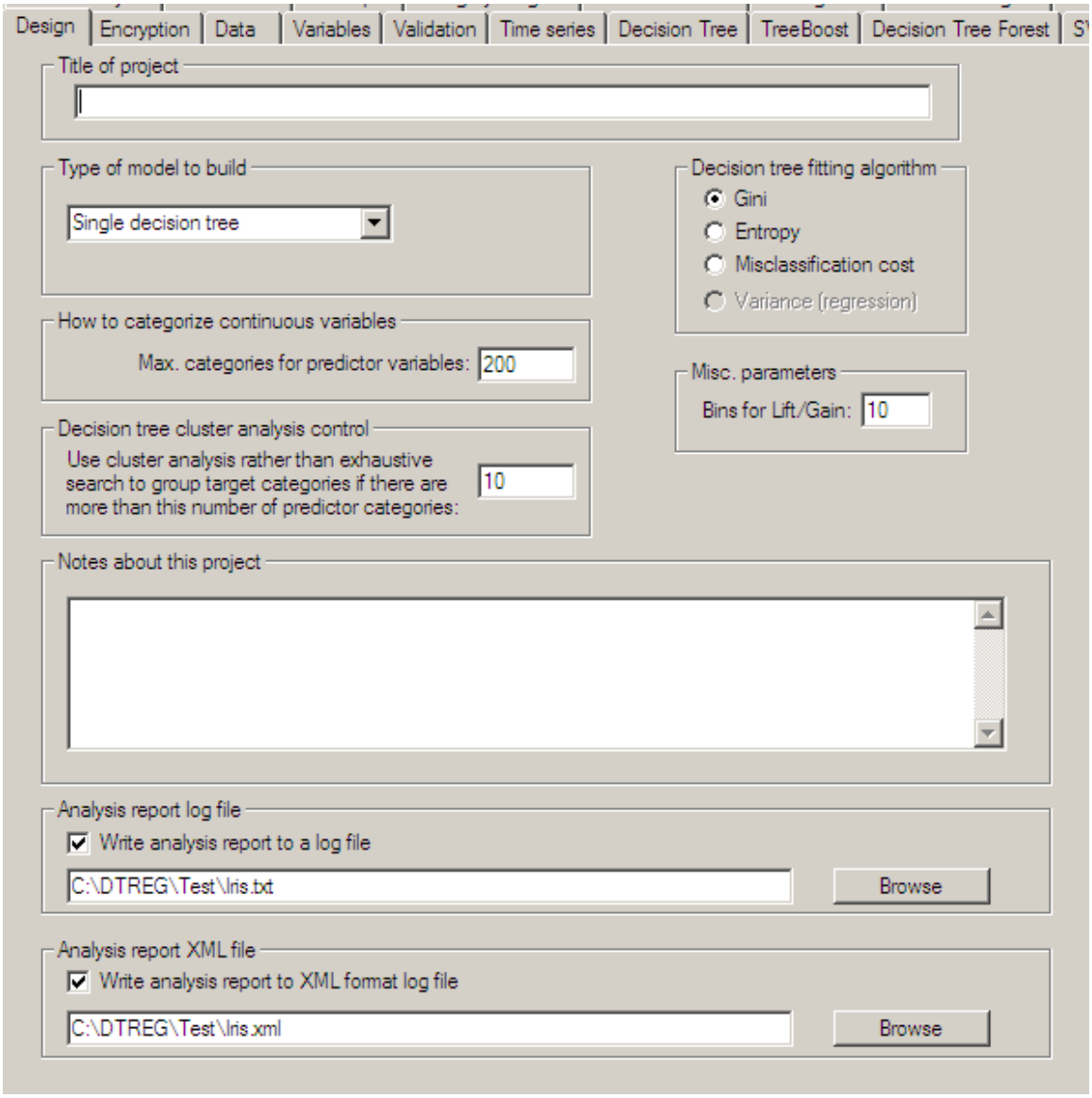
The Model screen displays with tabs for each property, similar to the one shown below:



Each property page is described below.

Design Property Page

The Design property page specifies general information about the model.



Title of project – Specify a descriptive title for the project. This is simply commentary information and may be omitted if you wish.

Write a report of the analysis to a *project*_Log.txt disk file – If this box is checked, DTREG will generate an analysis log file named *project*_Log.txt where *project* is the name of the DTREG project file. The log file contains the same information that is displayed in the analysis output panel.

Type of model to build – Select the type of model that DTREG should build.

How to categorize continuous variables – The values of continuous predictor variables are grouped into categories before they are used to build a decision tree. Specify in this field the maximum number of categories that are to be used to group continuous predictor variable values. The more categories you allow, the smaller and more precise the category ranges will be. However, as you increase the number of categories, the computation time also increases. If you allow up to 100 categories, then each category will be 1% of the range of the values.

Decision tree cluster analysis control – This value tells DTREG when to switch from an exhaustive search of predictor categories to a faster but slightly less accurate clustering method. This control is enabled only when building a classification tree. When the target variable is categorical and a predictor variable is also categorical, an exhaustive search would require DTREG to evaluate a potential split for every possible combination of categories of the predictor variable. The number of splits is equal to $2^{(k-1)}-1$ where k is the number of categories of the predictor variable. For example, if there are 5 predictor categories, 15 splits are tried; if there are 10 categories, 511 splits are tried; if there are 16 categories, 32767 splits are tried. Because of this exponential growth, the computational time makes it impractical to do an exhaustive search for more than about 12 predictor categories. To handle this situation, DTREG will switch to a faster but slightly less accurate method when the number of categories of a predictor variable exceeds the value you specify for this parameter. This allows DTREG to build classification trees even when a categorical predictor has hundreds or even thousands of categories.

Tree fitting algorithm – This parameter applies to single decision tree models. Select which algorithm you want DTREG to use to split nodes in the tree. TreeBoost models are always built using an algorithm that minimizes misclassification costs, so the algorithm selection boxes will be disabled for TreeBoost models. Here are the choices:

- **Gini** -- The Gini splitting method is the default and recommended method for classification trees. Each split is chosen to maximize the heterogeneity of the categories of the target variable in the child nodes.

- **Entropy** – The Entropy splitting method is an alternate method that can be selected for classification trees. Experiments have shown that entropy and Gini generally yield similar trees after pruning.

- **Misclassification cost** -- This method causes DTREG to use the split that minimizes the misclassification cost among the child nodes.

- **Variance** -- The variance splitting method is always used for regression trees. It causes DTREG to use the split that minimizes the sum of variance (i.e. sum of squared errors) in the child nodes.

Bins for Lift/Gain – Specify how many "bins" to use when computing and displaying the table of lift and gain values (see page 201).

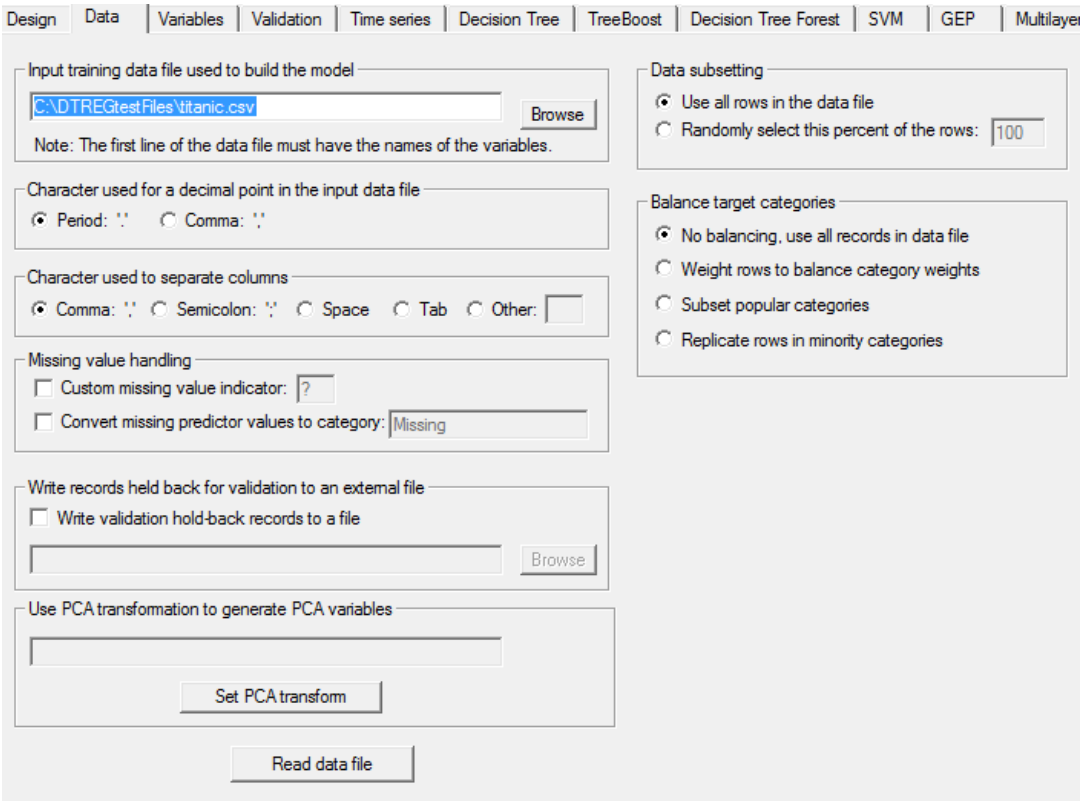
Notes about this project – This is a free-form text field where you can enter any notes you want to save regarding the project.

Analysis report log file – If you wish, you can direct DTREG to write a copy of the analysis report (show in the right portion of the screen) to a log file.

Analysis report XML file – If you enable this option, DTREG will create an analysis report in XML format so that it can be read by other programs. This feature is available only in the Enterprise Version of DTREG.

Data Property Page

The Data Property Page allows you to select the data file you want to use for the project.



Data File Format

The data file must be a text (ASCII) file with the values for one row (case) per line. Most database and spreadsheet programs such as Access and Excel can generate Comma Separated Value (CSV) formatted files that you can use as input to DTREG.

Data Subsetting – If you wish, you can tell DTREG to use only a subset of the records in the data file for the analysis. This speeds up the analysis and is useful when experimenting with different model settings. If you tell DTREG to use a subset of the data, specify the percentage of the rows that you want it to use. Since random selection is used to select the rows, the actual number of rows used may be slightly different than the percent you specify.

Balance Target Categories – When there is a large imbalance between the number of data rows with different target categories, model training tends to give priority to minimizing the error on the categories with more rows. The result is that the popular categories have low misclassification rates, but the categories with fewer training rows have a high level of misclassification error. DTREG provides three ways to mitigate the problems with target category imbalance:

1. **Weight rows of minority categories** – If you check this box, DTREG will increase the weight (importance) of data rows that have minority categories of the target variable. The weights are adjusted so that the sum of the weights for the rows with each target category are the same. This option may be used with cross-validation and leave-one-out validation.
2. **Subset popular categories** – If you check this box, DTREG will attempt to balance the number of data rows having each category of the target variable by selecting only a subset of the records that have target categories with excessive rows. This option causes DTREG to select only a subset of the rows with the most popular categories so that the data used for training will have approximately the same number of rows for all target categories as the least popular category has. Note that using this option means that rows with popular categories will not be included in the analysis.
3. **Replicate rows in minority categories** – If you check this box then DTREG will replicate (duplicate) data rows in the input file that are members of minority target categories (i.e., categories with fewer rows than the most popular category). The result will be that each target category will have approximately the same number of rows as the category with the maximum number of rows in the input file. If you use this option, then you cannot rely on cross-validation to validate the model, because the hold-out rows in validation folds may be replicated copies of rows that are used to train the model. The only way to do legitimate validation with this option is to split the data file outside of DTREG, then build the model with some of the data and run the held-out portion through the Score function (see page 163) to measure the misclassification error.

Write validation hold-back records to a file – If you check this box, you can specify a file where DTREG will write the records held back for validation. This is useful when you want to use the records selected for validation for your own, external tests. Note that this option is effective only when you specify that validation is to be done by holding back a percentage of the input dataset.

Set PCA transform – If you click this button, DTREG will prompt you for an auxiliary project file containing a PCA transformation function created by a previous analysis. The PCA transformation will be bound to this project, and new variables with names PC*n* will be created for the PCA transformed values. See page 352 for detailed information about using this feature. This feature is available only in the Enterprise Version of DTREG.

There are four selections related to the format of the input data file:

1. **Character used for a decimal point in the input data file** – Select whether a period or a comma will be used to indicate the decimal point in numeric values in the input data file. The American standard decimal point marker is a period while the European standard is a comma. This setting affects only data read from the input file; a period always is used as the decimal point marker in the generated report.
2. **Character used to separate columns** – Select the character that will be used to separate columns in the input file. The default separator is a comma, but you may select any character you wish to use.
3. **Custom missing value indicator** – Specify the character that will be used to indicate missing values in the data file. If a data field is entirely blank or consists only of the question mark character ("?") DTREG treats it as a missing value. If wish to specify a character to denote missing values in addition to question mark, check this box and specify the character in the associated edit box.
4. **Convert missing predictor values to category** – Normally, when the value of a predictor variable is missing (not specified or specified as '.' or '?'), it is treated internally with a special missing value code that means no information is available. DTREG has a number of techniques for imputing the estimated values of missing values including surrogate splitters and surrogate variables. However, some analyses may need to treat a missing value like another category of a variable rather than as an unknown value. If you check this option and a missing value is encountered for a categorical predictor variable, then the missing value is converted to the specified category label, and it is handled just as if the category label string had been specified in the data file. The replacement of missing values with a category label is done only for predictor variables (not target variables) that are declared to be categorical. Missing values for continuous predictor variables are always treated as missing (unknown) values.

The first row in the file must contain the names of the variables. If a variable name contains commas, you must enclose it in quote marks. You may enclose variable names in quotes even if they do not contain commas. If a variable name or a data value contains a quote character (") you must enclose the value in quote marks and specify a double quote mark to represent each single quote mark in the value. For example a value Toys "R" Us would be specified "Toys ""R"" Us".

Here is an example of a data file. Note that the third variable, "Gross income" is enclosed in quotes.

```
Age, Sex, "Gross income"
20,Male,25000
30,Female,42000
55,Male,76000
43,Male,44000
50,Female,82000
```

Specifying Variable Attributes in the Data File

You can optionally follow the name of a variable by a set of attributes enclosed in curly braces. Here is an example of such a data line:

```
Age{Continuous,Predictor}, Sex{Categorical,Predictor}, "Gross income"{Continuous,Target}
20,Male,25000
30,Female,42000
55,Male,76000
43,Male,44000
50,Female,82000
```

Note that the attributes are in curly braces, they go immediately after the name of the variable and before the character that separates variables. If multiple attributes are specified, they are separated by commas in the list. If the name of a variable is in quote marks, the attributes follow the closing quote mark. Here are the available attributes:

Target	This is the target variable
Predictor	This is a predictor variable
Weight	This is the weight variable
Unused	This variable is not used (default)
Categorical	Variable has categorical values
Continuous	Variable has continuous values (default)
Character	Variable is to be treated as a character string

The more common way to set variable attributes is using the Variable Property Page described on page 41.

Continuing Data Lines

Long data lines can be continued to the following line by placing a backslash ('\') character as the last character on the line being continued. For example, the following continued line:

```
Age, Sex,\
"Gross income"
```

is equivalent to:

```
Age, Sex, "Gross income"
```

Specifying Missing Values in Data Files

To indicate a missing value in a dataset, use the following:

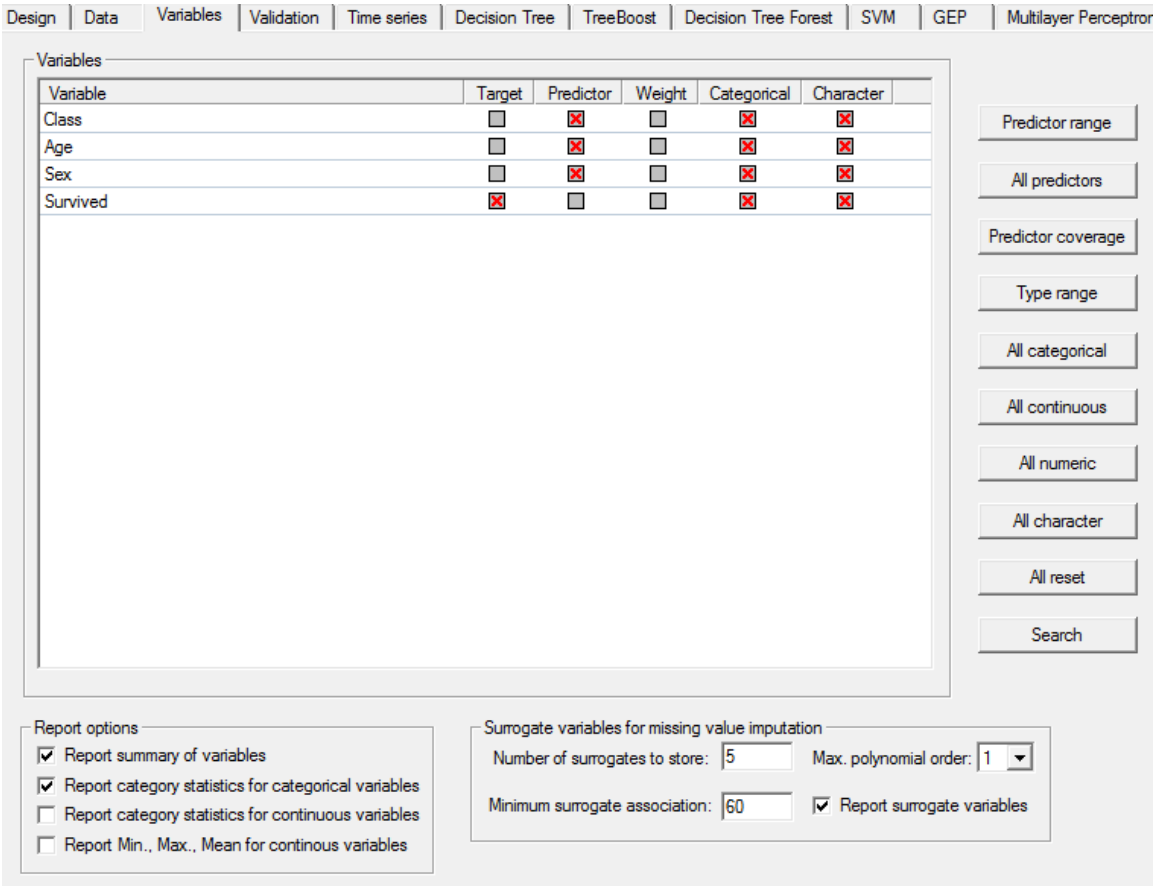
- A field that is entirely empty (nothing between the commas).
- The question mark character ('?')
- A single period ('.').
- A character you specify using the "Custom missing value indicator" specification.

For example, in the following data set the value of Age is missing in the first row, the value of Sex is missing in the second row, and the value of Gross income is missing in the third row.

```
Age, Sex, "Gross income"
.,Male,25000
30,,42000
55,Male,?
43,Male,44000
50,Female,82000
```

Variables Property Page

The Variables property page is used to specify the class and category of each variable.



The list will show the name of each variable as was found on the first row of the data file for the project (see description of the Data property screen on page 36).

The following columns are shown next to the variable names. Click on a box in a column to turn a property on or off for a variable.

- **Target** – If this box is checked, the selected variable is the target variable for the model. One and only one variable may be designated as the target variable.

- **Predictor** – If this box is checked, the selected variable will be used as a predictor variable when creating the model. You must select at least one predictor variable, and you may select many predictor variables.

- **Weight** – If this box is checked, the selected variable will be used as the weight variable. If a weight variable is selected, its values will be used to weight the rows of the data. If no weight variable is selected, all rows receive the same weight.

- **Categorical** – Check this box if the variable is categorical (nominal). Leave the box unchecked if the variable is continuous or ordinal. Categorical variables may have either numeric or text (e.g. "Male" or "Female") values in the data file. Continuous variables must have numeric values.

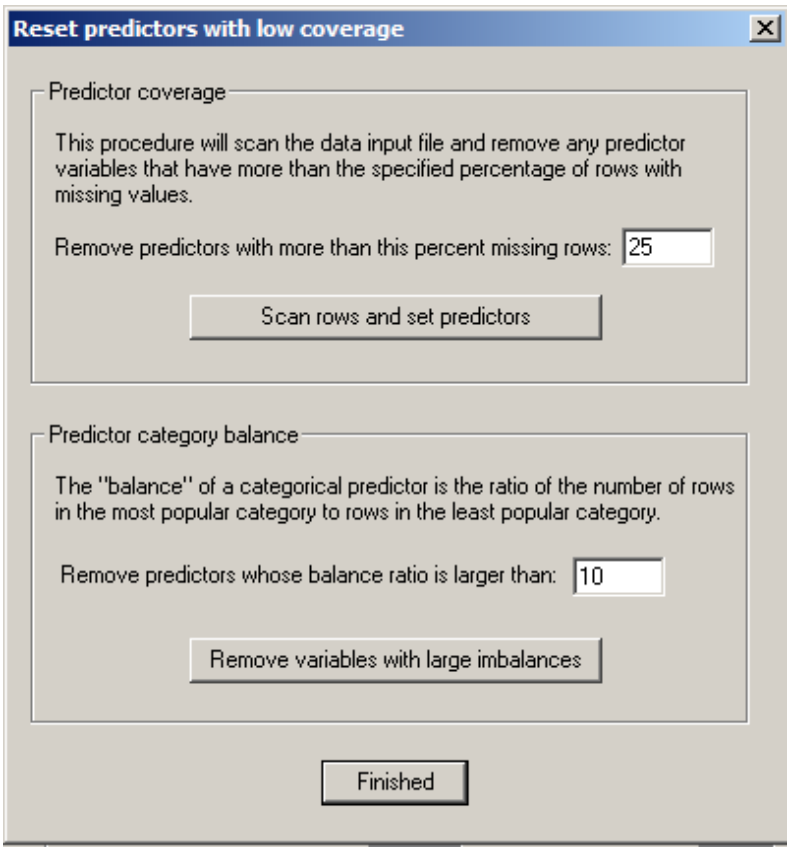
- **Character** – Check this box if the values of the variable can have general character values such as "Male", "Female", "Yes", "No", etc. Leave the box unchecked if the values of the variable are strictly numeric. Only categorical variables can store character values; continuous variables store only numeric values. The default setting for categorical variables is character type. Note: the setting of this attribute only affects the code generated by the Translate function (see page 169). It does not affect the building of the model or the operation of the Score function. C and C++ code generated for variables declared to have character values are defined with char[*nnn*] declarations. Numeric variables are defined as type long. SAS® code initializes character or numeric variables depending on this setting. It is legal to declare a categorical variable to be of type character even if it has only numeric values.

Several buttons are shown at the right side of the list:

- **Predictor range** – If you have a lot of variables and want to set all of the variables in a range to be predictor variables or not to be predictor variables, then click this button. It will display a screen where you can select the first and last variables in the range that you want to designate as being (or not being) predictors. This is often a lot easier and faster than clicking hundreds of boxes.

- **All predictors** – Click this button to check the predictor boxes for all variables. Note, you must then select one of the variables as the target variable.

- **Predictor coverage** – If you click this button, the following screen will be displayed:



These procedures can be used to reduce the number of predictor variables when there are many missing values in the data or when the distribution of rows for a categorical variable are highly unbalanced. There are two procedures:

Predictor coverage – If you use this procedure, DTREG will scan the data rows and remove any predictor variables (set them to unused) if the percentage of data rows with missing values for a variable exceeds the specified value.

Predictor category balance – If you use this procedure, DTREG will check each categorical predictor variable and compute the "balance" which is the ratio of the number of rows with the most popular category and the number of rows with the least popular category. If the ratio exceeds the specified value, then the variable is removed as a predictor. .

- **Type range** – Click this button if you want to set the type of a range of variables to be categorical or continuous. It will display a screen where you can select the first and last variables in the range whose type is to be set.

- **All continuous** – Click this button to uncheck the categorical checkboxes for all variables (i.e. to set the class of all variables to be continuous).

- **All numeric** – Click this button to uncheck all of the character checkboxes for all variables (i.e., set all variables to hold only numeric values). The boxes for variables that are known to have non-numeric values will remain checked.

- **All character** – Click this button to check the character attribute boxes for all variables. Continuous variables can never store character values, so only categorical variables are affected.

- **All reset** – Click this button to reset (uncheck) all boxes.

- **Search** – If there are a large number of variables, you can click Search to locate a variable whose name contains a specified string. The list of variables will be scrolled so the matching variable is visible, but it may not be displayed at the top of the screen. Often, the variable located will be positioned near the bottom of the screen.

There are three category distribution report options available at the bottom of the page:

- **Report category statistics for categorical variables** – If selected, a Summary of Categories report will be generated with information about the categories for all categorical predictor and target variables. For additional information, please see page 181.
- **Report category statistics for continuous variables** – If selected, a report will be generated with information about the categories for all continuous predictor and target variables.
- **Report min., max., mean for continuous** variables – If selected, DTREG will report the minimum, maximum, mean and standard deviation for each continuous predictor variable.

Surrogate Variables
This section of the page is used to set parameters for surrogate variables. Please see page 358 for detailed information about surrogate variables.

- **Number of surrogates to store** – This is the maximum number of surrogate variables that DTREG will store for each predictor variable. Fewer surrogates may be stored if no significant associations are found.

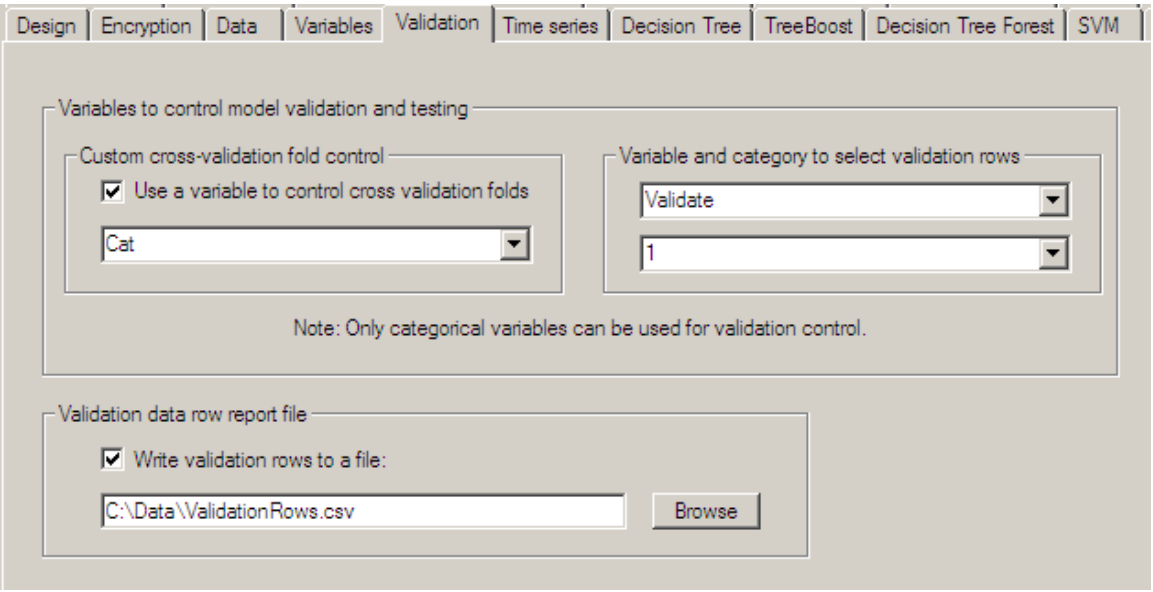
- **Minimum surrogate association** – The association computed for each potential surrogate is compared to this value. If the association is smaller than this, then the surrogate is excluded.

- **Maximum polynomial order** – This controls whether linear, quadratic, or cubic functions are used for surrogate associations. If a polynomial order greater than 1 is specified, DTREG computes the association for all polynomials up to that order, and it only uses the higher order polynomials if they provide superior fit (association) over lower-order polynomials.

- **Report surrogate variables** – If this option is checked, then DTREG adds a table to the analysis report showing which surrogate variables were stored for each predictor along with the polynomial coefficients and the association. See page 182 for an example of the surrogate variable report.

Validation Property Page

The Validation Property Page is used to select variables to control hold-out validation and cross-validation. .



Cross-validation Control Variable
Normally when cross validation is used to evaluate the quality of a model, DTREG assigns a random set of rows to each validation fold after stratifying on the target variable. If you wish, you can select a variable whose values will determine which cross validation fold each row will be placed in rather than using random selection. If a variable is used, it must be a categorical variable; there will be one fold for each category of the variable.

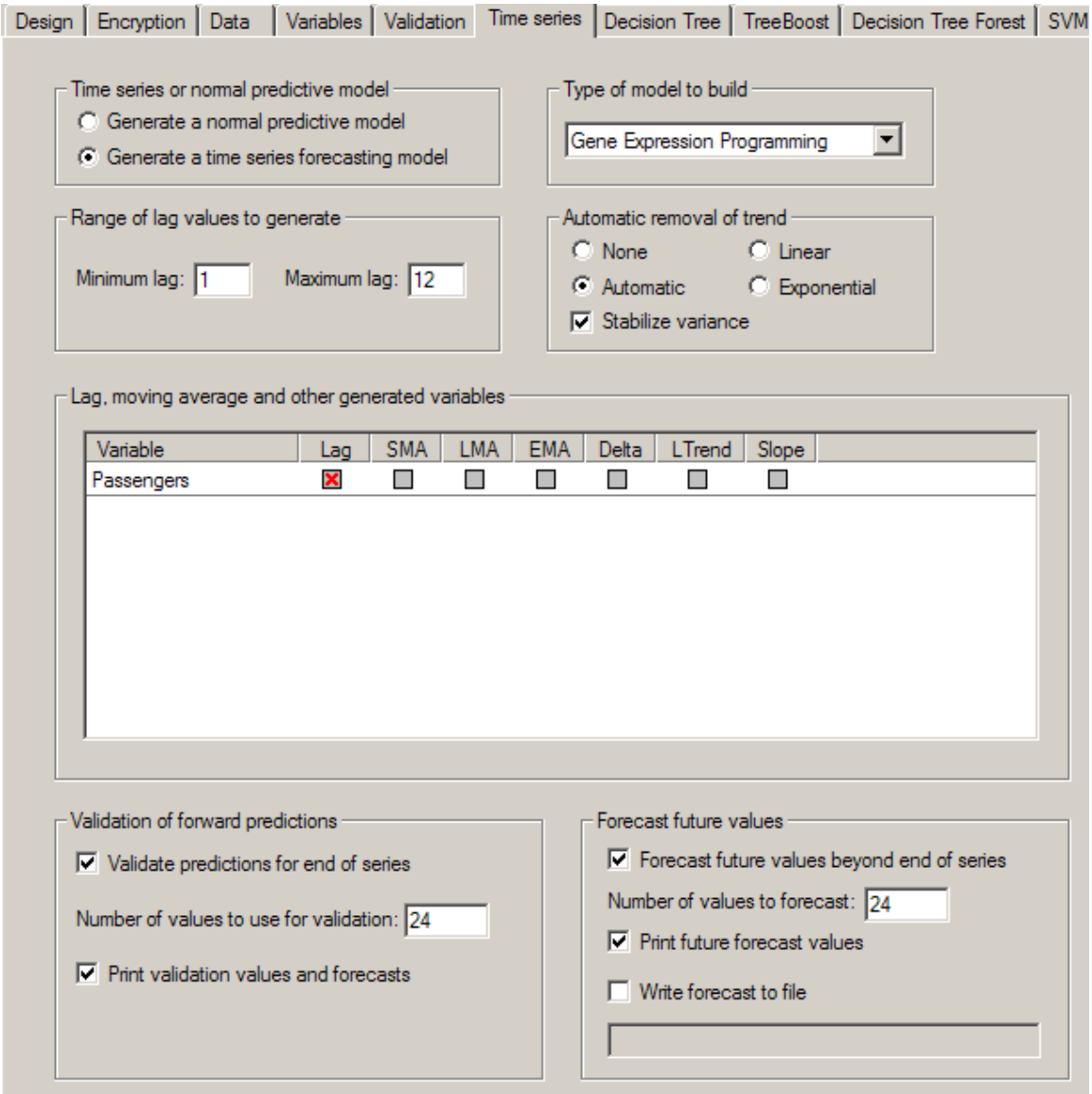
A cross validation control variable is useful for a situation where you have a number of similar observations that are clustered in a small number of groups. If the observations within a cluster are very similar (i.e., cohorts), then performing cross validation where observations from the same cluster are both used to build a validation model and evaluate it will result in overly optimistic results. In this case, it would be proper to use the cluster number to control the cross validation folds so nearly similar cases are grouped in a fold.

Validation row selection variable This variable and the associated category selects the rows that are to be held-out and used for model validation if you select the validation method "Use variable to select validation rows" on the property page for the model. Select the variable to control hold-out rows in the upper field, and select the category of that variable that is to be held out in the lower field. Any rows that have categories other than the specified category will be used in the training data for the model. Only categorical variables may be used as hold-out control variables. You can use the DTL facility (see page 153) to create a selection variable.

Validation data row report file – If you enable this option and specify a file name in the edit field, then DTREG will write a record to the file showing which rows were used for validation, and it will show the predicted value for each validation row. Each row in the file has 4 columns: (1) the data row number (1 based), (2) the actual value of the target variable, (3) the predicted value of the target variable, and (4) an indication of correct/incorrect classification or the residual value if doing regression.

Time Series Property Page

The Time Series property page is used to specify whether a time series or a regular predictive model is to be created. You also specify parameters for time series models. See the chapter beginning on page 139 for additional information about time series analysis.



Time series or normal predictive model – Select whether you wish to build a time series model or a normal predictive model to predict a target variable from predictor variables.

Type of model to build – Select which type of model you want DTREG to build for the time series. The following types of models may be used: (1) Single tree, (2) TreeBoost, (3) SVM, (4) Gene expression programming, (5) Multilayer perceptron neural

network, (6) GRNN neural network, (7) GMDH polynomial network, (8) RBF neural network, (9) cascade correlation. While single trees may be used, they are not recommended because they do a poor job of predicting continuous values. Gene expression programming models work well, because they can generate very general functions. GRNN neural networks also often work well.

Range of lag values to generate – Time series models normally use lagged values of the target variable as predictor variables. A lag variable has the value of the target variable that occurred a specified number of periods ago. Specify the minimum and maximum periods for which you want DTREG to generate lag variables.

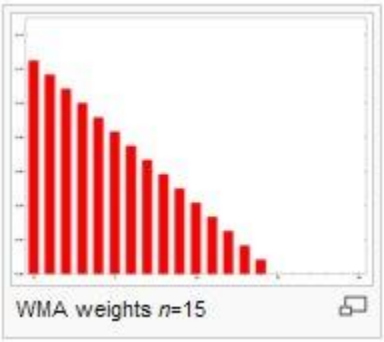
Automatic removal of trend – Usually it is easier to build accurate time series models if the series is stable over time. If a series has a linear or exponential growth trend, DTREG can remove the trend by fitting a linear or exponential function to the series and then subtracting that function from the time series values. There are several choices:

- **None** – Do not attempt to remove a trend from the series.
- **Linear** – Fit a linear function to the series and use it to remove the trend.
- **Exponential** – Fit an exponential function to the series.
- **Automatic** – Try both linear and exponential functions and use the best one.
- **Stabilize variance** – If the variance (amplitude) of the series is increasing or decreasing regularly over time, this option causes DTREG to attempt to stabilize it so that the variance is constant. Note: about 20% of the time variance stabilization improves models and 80% it hurts them, so try it both ways and compare the results. See page 143 for additional information.

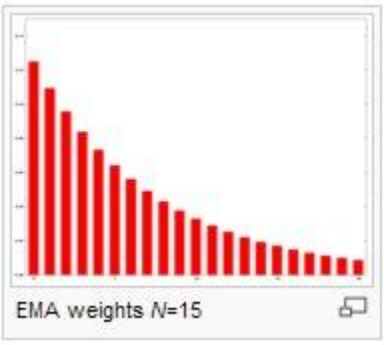
Generated variables – In this section you can select if you want DTREG to generate lag or other types of variables. For a time series, lag variables are almost always generated for the target variable. If there are additional predictor variables you can optionally instruct DTREG to generate lag variables for them too. Several types of variables can be generated:

- **Lag** – A lag variable is the value of a variable that occurred the specified number of observations in the past. For example, if Y[100] is the current value of the Y variable, then a lag variable with a lag setting of 10 would have the value of Y[90].
- **SMA** – Simple moving average. This is the average value of the variable over the number of preceding observations equal to the specified lag. The values receive equal weight when averaging.

- **LMA** – Linearly weighted moving average. This is a moving average computed using a linear weighting with the observations closest to the current observation receiving more weight than older observations.

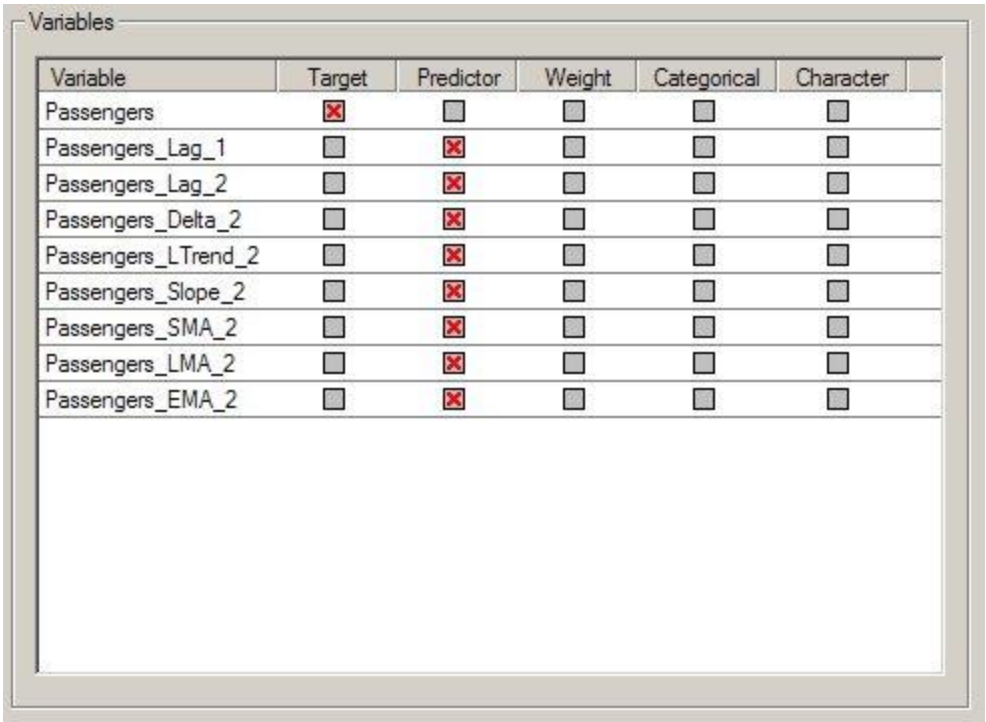


- **EMA** – Exponentially weighted moving average. This is a moving average computed using an exponential weighting function with the observations closest to the current observation receiving more weight than older observations.



- **Delta** – This is the difference between the value of the variable one step before the current observation and the value (lag+1) steps behind. For example, if the lag is set to 10, then delta for Y[100] = Y[99] – Y[89].
- **LTrend** – This is the value of an observation predicted by a linear equation fitted through the number of points preceding the point. The lag value controls how many points before the current point are used for the fitted line.
- **Slope** – This is the slope of a linear equation fitted through the number of points preceding the point. The lag value controls how many points before the current point are used for the fitted line.

Variables generated by DTREG are available for selection as predictor variables on the Variables property page. Here is an example of variables generated for the Passengers variable with a lag of 2:



Note that you can select which of the lag variables you wish to use as predictors.

Validation of forward predictions – If you enable this option, DTREG will use the specified number of observations at the end of the series to validate (check) the predictions of the model. The model will be built using only the observations before those being held out for validation, and then the model will be used to generate predicted values for the held-out validation observations, and they will be compared. Statistics on the quality of the fit will be written to the analysis report, and the Time Series chart can be used to view the predicted and actual values. See page 145 for additional information.

Print validation values and forecasts – If this option is selected, DTREG will include the values of the validation rows and the predicted forecast values in the analysis report.

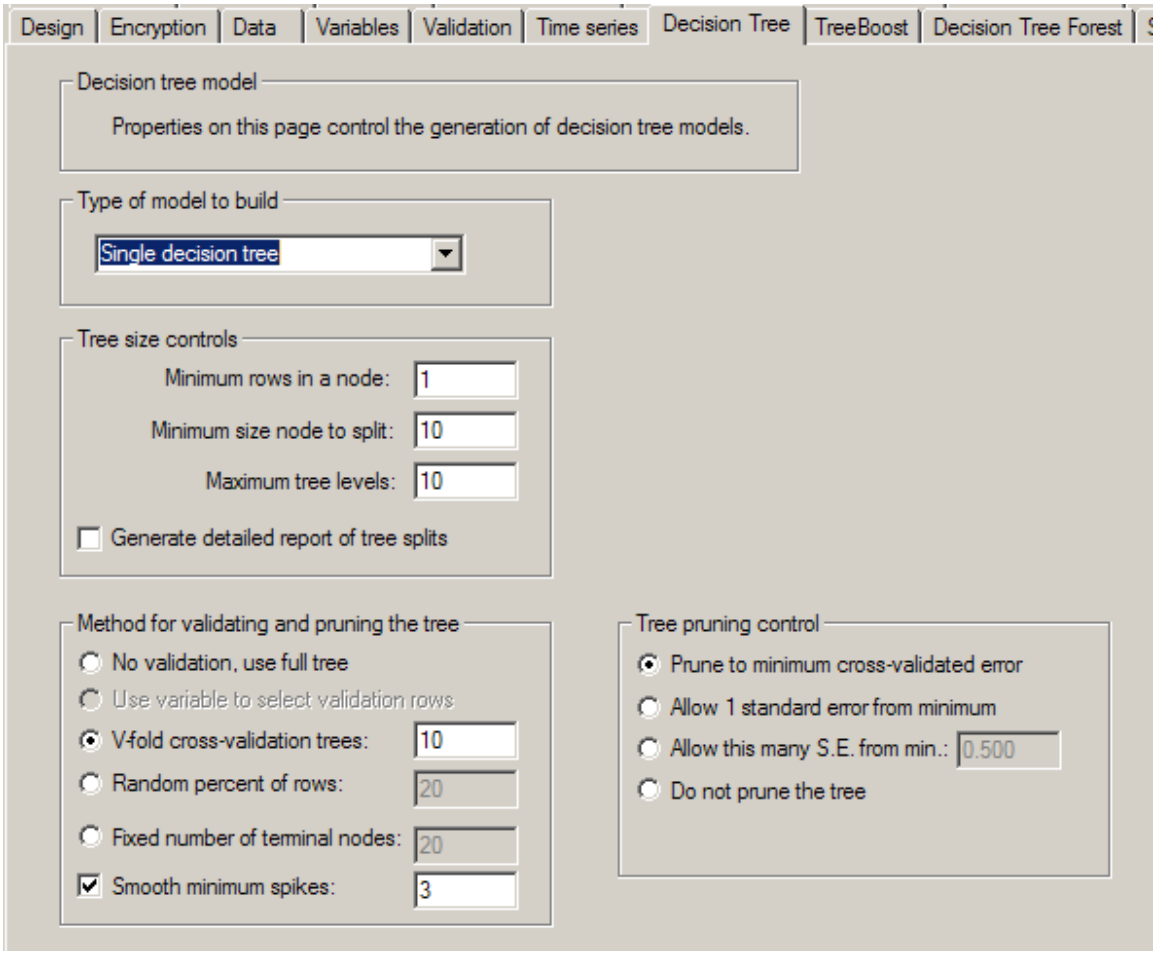
Forecast future values – If this option is selected, then DTREG will use the time series model to generate forecasts for future observations beyond the end of the series. The forecast values are written to the analysis report, and the Time Series chart can be used to view them.

Print future forecast values – If this option is selected, DTREG will display in the analysis report forecast values for the specified number of periods.

Write forecast to file – If you check this box, you can specify where the forecast values are to be written.

Single Tree Model Property Page

The Single Tree property page is used to specify parameters for single tree models.



Type of model to build – Select the type of model that DTREG should build. The controls on this screen are disabled for any type of model other than single tree.

Minimum rows in a node – This specifies the minimum number of rows that may fall in a node after splitting. A split will not be allowed if either of the resulting child nodes had fewer than this number of rows.

Minimum size node to split – This specifies that a node (group) should never be split if it contains fewer rows than the specified value.

Maximum tree levels – Specify the maximum number of levels in the tree that you want DTREG to construct when it is building the tree. It is best to let DTREG initially build a large tree with many levels and then allow the pruning phase of the analysis to remove levels. See page 366 for information about how pruning is done.

Generate report of tree splits – If you check this box DTREG will write a report of each node split to the analysis log. If the tree is large, this report will be large.

Method for validating and pruning the tree – select the method to be used by DTREG to test the tree that it builds.

No validation, use full tree – If you check this button, DTREG will build the full decision tree for the model and will do no testing or pruning. A full, unpruned tree is sometimes called an "exploratory tree".

V-Fold cross-validation – If you check this button, DTREG will use V-fold cross-validation to determine the statistically optimal tree size. You may specify how many "folds" (cross-validation trees) are to be used for the validation; a value of 10 is recommended. Specifying a larger value increases the computation time and rarely results in a more optimal tree. For a detailed description of V-fold cross validation, please see page 369.

Random percent of rows – If you check this button, DTREG will hold back from the model building process the specified percent of the data rows. The rows are selected randomly from the full dataset, but they are chosen so as to stratify the values of the target variable. Once the model is built, the rows that were held back are run through the tree and the misclassification rate is reported. If you enable tree pruning, the tree will be pruned to the size that optimizes the fit to the random test rows. The advantage of this method over V-fold cross-validation is speed – only one tree has to be created rather than (V+1) trees that are required for V-fold cross-validation. The disadvantage is that the random rows that are held back do not contribute to the model as it is constructed, so the model may be an inferior representation of the training data. Generally, V-fold cross-validation is the recommended method for small to medium size data sets where computation time is not significant, and random-holdback validation can be used for large datasets where the time required to build (V+1) trees would be excessive.

Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

Fixed number of terminal nodes – If you check this button, DTREG will prune the tree to the specified number of terminal nodes. The cost-complexity values computed for the tree are used to guide the pruning so that the least significant nodes are pruned to reduce the tree to the specified size. When this option is selected, cross-validation trees are not generated, so it is much faster than doing full cross-validation on large trees; however, there is no assurance that the generated tree has the optimal number of nodes. This option is useful when you are generating exploratory trees.

Smooth minimum spikes – If you check this button, DTREG will smooth out fluctuations in the error rate for various size models by averaging the misclassification rates for neighboring tree sizes. During the pruning process, DTREG must identify the tree size that produces the minimum misclassification error (residual) for the validation data; this is the optimal size to which the tree will be pruned. Sometimes the error rate fluctuates as the tree size increases, and an anomalous minimum "spike" may occur in a region where the surrounding error rates are much higher. This happens more often when using random-row-holdback validation than when using V-fold cross-validation which tends to average out error rate values. If you enable smoothing of minimum spikes, DTREG averages each error-rate/tree-size value with its neighboring values. The effect is to cause DTREG to seek regions where the minimum values are consistently low rather than isolated low values. The generated trees may be larger, but they usually are more stable when used for scoring. The value associated with this button specifies how many values are to be averaged for smoothing. For example, a smoothing value of 3 causes DTREG to compute the average of three points – the center point and the neighboring points on the left and right.

Tree Pruning Control – Select options in this group to control how DTREG prunes the tree to the optimal size. Note: You must select V-fold cross validation to enable tree pruning. For additional information about how tree pruning is performed, please see page 366.

Prune to minimal cross-validated error – If you select this option, DTREG will prune the tree to the number of nodes that produce the minimal error in the cross-validation trees. This is the theoretically optimal tree size, but it may be only marginally better than a smaller tree with a slightly larger error value. For additional information, please see page 371.

Allow one standard error from minimum – If you select this option, DTREG will be allowed to prune the tree to a smaller number of nodes such that the cross-validated error cost of the smaller tree is no more than one standard error from the minimal cross-validated error value. The advantage of selecting this option is that DTREG generates a smaller and simpler tree; however, the tree may not be quite as good at predicting future values as the larger, optimal tree. Research has shown that the misclassification cost values tend to decrease to a valley as the tree size is pruned and then increase gradually once the pruned tree size passes the optimal size. Typically, the decrease is not smooth and there is some roughness in the cost values around the optimal point; so, allowing pruning to a smaller, slightly less optimal tree is probably not statistically significant, and you end up with a smaller, simpler model.

Allow this many S.E. from min. – If you check this box, you can specify an exact number of standard error intervals to allow the pruning to select a smaller tree. If you specify 1 for the standard error interval, then this option is equivalent to selecting "Allow one standard error from minimum".

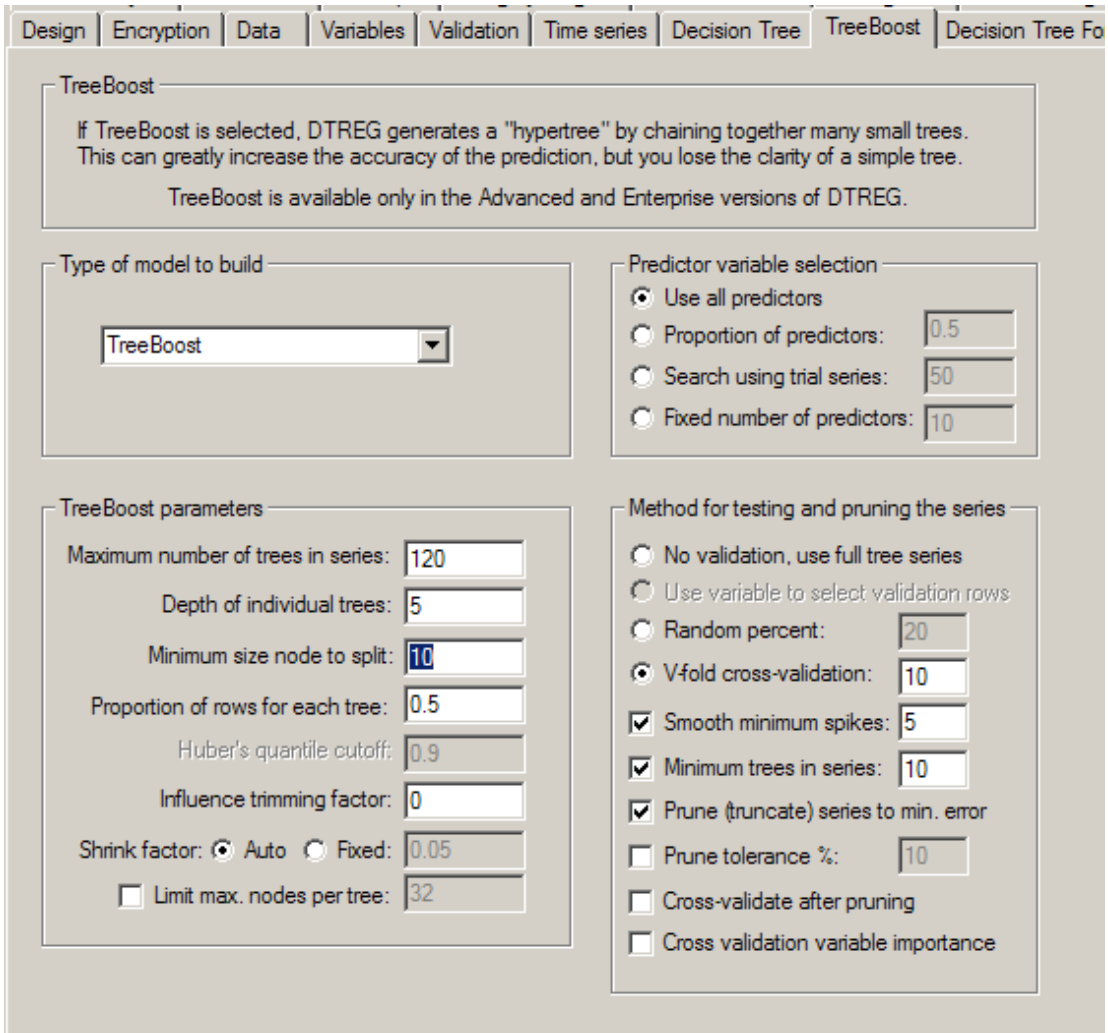
Do not prune the tree – Select this option if you want DTREG to perform cross-validation but not prune the tree. You will get the cross-validation statistics, but the full, unpruned tree will be generated.

TreeBoost Property Page

TreeBoost models often can provide greater predictive accuracy than single-tree models, but they have the disadvantage that you cannot visualize them the way you can a single tree; TreeBoost models are more of a "black box".

For more technical information about TreeBoost, please see the chapter starting on page 245.

When you select the TreeBoost property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a type of model other than TreeBoost, the other controls on this screen will be disabled.

Maximum number of trees in series: Specify how many trees you want DTREG to generate in the TreeBoost series. If you select the appropriate options in the right panel, DTREG will prune (truncate) a series to the optimal size after building it. You can click Charts on the main menu followed by Model Size to view a chart that shows how the error rates vary with the number of trees in the series.

Depth of individual trees: Specify how many levels of splits each tree in the TreeBoost series should have. The number of terminal nodes in a tree is equal to 2^k where k is the number of levels. So, for example, a tree with a depth of 1 has two terminal nodes, a tree with a depth of 2 has 4 terminal nodes, and a tree with a depth of 3 has 8 terminal nodes. Because many trees contribute to the model generated by TreeBoost, usually it is not necessary for individual trees to be very large. Experiments have shown that trees with 4 to 8 levels generally perform well, but if there are a large number of predictor variables or there are many categories for the predictors, you should try increasing the tree depth to 10 or 12. The depth should be at least as large as the number of variable interactions. If you have a categorical predictor variable with many classes (for example, postal zip code) it may be necessary to increase the tree depth to allow DTREG to partition the data into more groups. If the predictions from a TreeBoost model are not as accurate as those from the corresponding single-tree model, try increasing the depth of the TreeBoost trees.

Minimum size node to split – This specifies that a node should never be split if it contains fewer rows than the specified value.

Proportion of rows in each tree: Research has shown (Friedman, 1999b) that TreeBoost generates the most accurate models with minimum over fitting if only a portion of the data rows are used to build each tree in the series. Specify for this parameter the proportion of rows that are to be used to build each tree in the series; a value of 0.5 is recommended (i.e., half of the rows). The specified proportion of the rows are chosen randomly from the full set of rows. (This is the *stochastic* part of stochastic gradient boosting.)

Huber's quantile cutoff: The TreeBoost algorithm uses Huber's M-regression loss function to evaluate error measurements for regression models (Huber, 1964). This loss function is a hybrid of ordinary least-squares (OLS) and least absolute deviation (LAD). For residuals less than a cutoff point, the squared error values are used. For residuals greater than the cutoff point, absolute values are used. The virtue of this method is that small to medium residuals receive the traditional least-squares treatment, but large residuals (which may be anomalous cases, mismeasurements or incorrectly coded values) do not excessively perturb the function. After the residuals are calculated, they are sorted by absolute value and the ones below the specified quantile cutoff point are then squared while those in the quantile above the cutoff point are used as absolute values. The recommended value is 0.9 which causes the smaller 90% of the residuals to be squared

and the most extreme 10% to be used as absolute values. Huber's quantile cutoff parameter is used only for regression analyses and not for classification analyses.

Influence trimming factor: This parameter is strictly for speed optimization; in most cases it has little or no effect on the final TreeBoost model. When building a TreeBoost model, the residual values from the existing tree series are used as the input data for the next tree in the series. As the series grows, the existing model may do an excellent job of fitting many of the data rows, and the new trees being constructed are only dealing with the unusual cases. "Influence trimming" allows DTREG to exclude from the next tree build process rows whose residual values are very small. The default parameter setting of 0.1 excludes rows whose total residual represent only 10% of the total residual weight. In some case, a small minority of the rows represent most of the residual weight, so most of the rows can be excluded from the next tree build. Influence trimming is only used when building classification models.

Shrinkage factor: Research has shown (Friedman, 2001) that the predictive accuracy of a TreeBoost series can be improved by apply a weighting coefficient that is less than 1 ($0 < v < 1$) to each tree as the series is constructed. This coefficient is called the "shrinkage factor". The effect is to retard the learning rate of the series, so the series has to be longer to compensate for the shrinkage, but its accuracy is better. Tests have shown that small shrinkage factors in the range of 0.1 yield dramatic improvements over TreeBoost series built with no shrinkage ($v = 1$). The tradeoff in using a small shrinkage factor is that the TreeBoost series is longer and the computational time increases.

If "Auto" shrinkage factor is selected, the shrinkage factor is calculated by DTREG based on the number of data rows in the training data set.

Let *NumRows* = the number of data rows in the training data set.
Then, *ShrinkFactor* = max(0.01, 0.1 * min(1.0, *NumRows*/10000))

If you prefer, you can select the "Fixed" option and specify a shrinkage factor.

If you experience significant over fitting of the TreeBoost model (much better fit on training data than test data), try decreasing the shrinkage factor. Note that "Auto" mode will never use a shrinkage factor less than 0.1. If over fitting is a problem, try switching to the "fixed" setting and specify values in the range of 0.05.

Limit max. nodes per tree: If you enable this option, DTREG will build each tree in the TreeBoost series to the maximum depth and then prune it by removing the least significant nodes so that it has no more than the specified number of terminal (leaf) nodes. It is recommended that you leave this box unchecked and limit the size of trees by setting the maximum tree depth. The main reason for pruning trees in the series is to reduce the amount of memory space required by very large models.

Pruning Methods for TreeBoost Series

TreeBoost series are less prone to problems with over fitting than single-tree models, but they can benefit from validation and pruning to the optimal size to minimize the error on a test dataset. In the case of a TreeBoost series, "pruning" consists of truncating the series to the optimal number of trees.

No validation, use full tree series: All of the data rows are used to "train" the TreeBoost series. No validation or pruning is performed.

Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

Random percent of rows – If you check this button, DTREG will hold back from the model building process the specified percent of the data rows. The rows are selected randomly from the full dataset, but they are chosen so as to stratify the values of the target variable. Once the model is built, the rows that were held back are run through the tree and the misclassification rate is reported. If you enable tree pruning, the tree will be pruned to the size that optimizes the fit to the random test rows. The advantage of this method over V-fold cross-validation is speed – only one TreeBoost series has to be created rather than (V+1) tree series that are required for V-fold cross-validation. The disadvantage is that the random rows that are held back do not contribute to the model as it is constructed, so the model may be an inferior representation of the training data. Generally, V-fold cross-validation is the recommended method for small to medium size data sets where computation time is not significant, and random-holdback validation can be used for large datasets where the time required to build (V+1) tree series would be excessive.

V-Fold cross-validation – If you check this button, DTREG will use V-fold cross-validation to determine the statistically optimal size for the TreeBoost series. You may specify how many "folds" (cross-validation trees) are to be used for the validation; a value in the range 3 to 10 is recommended. Specifying a larger value increases the computation time and rarely results in a more optimal tree. For additional information about V-fold cross validation, please see page 369.

This is the process used for cross validation of a TreeBoost series:

First, a primary series is created using all of the data rows. This series is grown to the maximum allowable length.

The data rows are randomly divided into *V* sets, where *V* is the number of folds. Hence, each set has 1/*V* of the total rows.

A TreeBoost series is created for each of the *V* folds (i.e., *V* TreeBoost series are created). The *n*th series is built using all of the row data sets *except for* the *n*th data set. In other words, one set of data (1/*V* rows) is excluded (held back) from each series, and it is a different set of rows that is held back each time.

After the *n*th series is created using all data rows except for those in the *n*th set, the rows in the *n*th set that were held back are used to compute the misclassification rate for the series. The misclassification rate is computed for the series using only the first tree, then the first two trees in the series, then the first three, up to the total length of the series. The error rate is stored for each possible number of trees in the series.

One the *V* cross-validation series have been created and their error rates have been computed using the held-back rows, the error rates for each length of series is averaged across the *V* series and the length with the minimum error average is used.

If pruning was requested, the primary series that was created using all data rows is then pruned to the length with the minimum error rate as determined by cross validation.

Smooth minimum spikes – If you check this button, DTREG will smooth out fluctuations in the error rate for various size models by averaging the misclassification rates for neighboring tree series sizes. Sometimes, the error rate fluctuates as the tree size increases, and an anomalous minimum "spike" may occur in a region where the surrounding error rates are much higher. This happens more often when using random-row-holdback validation than when using V-fold cross-validation which tends to average out error rate values. If you enable smoothing of minimum spikes, DTREG averages each error-rate/tree-size value with its neighboring values. The effect is to cause DTREG to seek regions where the minimum values are consistently low rather than isolated low values. The generated TreeBoost series may be longer, but they usually are more stable when used for scoring. The value associated with this button specifies how many values are to be averaged for smoothing. For example, a smoothing value of 3 causes DTREG to compute the average of three points – the center point and the neighboring points on the left and right.

Minimum trees in series – If you check this box, you can specify the minimum number of trees in the series after pruning. DTREG will not prune the series to a length shorter than the specified value. Some TreeBoost series have erratic behavior with small numbers of trees. Sometimes the error rate is very low with series consisting of one or two trees, then the error rate jumps up and gradually declines. In cases like this, the short

series is unreliable, and it is undesirable to prune to that length even if the minimum error occurs with one or two trees. By specifying the minimum number of trees in the series, you can guarantee that pruning will not truncate the series below a specified length.

Prune (truncate) series to minimum error: If this box is checked, DTREG will truncate the TreeBoost series at the length that has the minimum validation error as determined by the validation method selected above. If this box is not checked, then DTREG will use the validation method to measure the error rate, but the full series will be retained.

Prune tolerance percent: Check this box to allow DTREG to prune the series to a smaller number of trees than the minimum validation point. In many cases, the improvement from adding trees to a series may be small, and the error rate will decline slowly with a long, nearly-horizontal "tail" on the model-size chart. In cases like this, it is possible to prune many trees from the series with only a small increase in the error rate. If you enable this option, then DTREG will prune the series to a smaller size than the absolute minimum as long as the error rate does not increase by more than the percentage factor that you specify. For example, if the minimum error point in the series has an error (misclassification) rate of 20% and you specify a pruning tolerance factor of 10%, then DTREG will be allowed to prune the series to a shorter length as long as the error rate does not exceed 22% (20 + 0.10*20).

Cross validate after pruning: If this box is checked and V-fold cross validation is selected, DTREG will recomputed the cross-validated error rate after pruning the series so that the validation error accurately reflects the truncated series. This doubles the time required for cross validation. If this box is not checked, the error rate for the full, un-truncated TreeBoost series is used for validation statistics.

Cross validation variable importance: If you check this option, then DTREG will compute the importance of variables for each cross validation fold, it will calculate the geometric mean of the importance across folds, and it will provide a separate report of the variables showing the mean cross-validation importance. This is an alternate way to compute variable importance that minimizes the importance of variables that are relatively unimportant on any cross validation fold. It may give a more accurate message or variable importance when some variables are sensitive to specific data values.

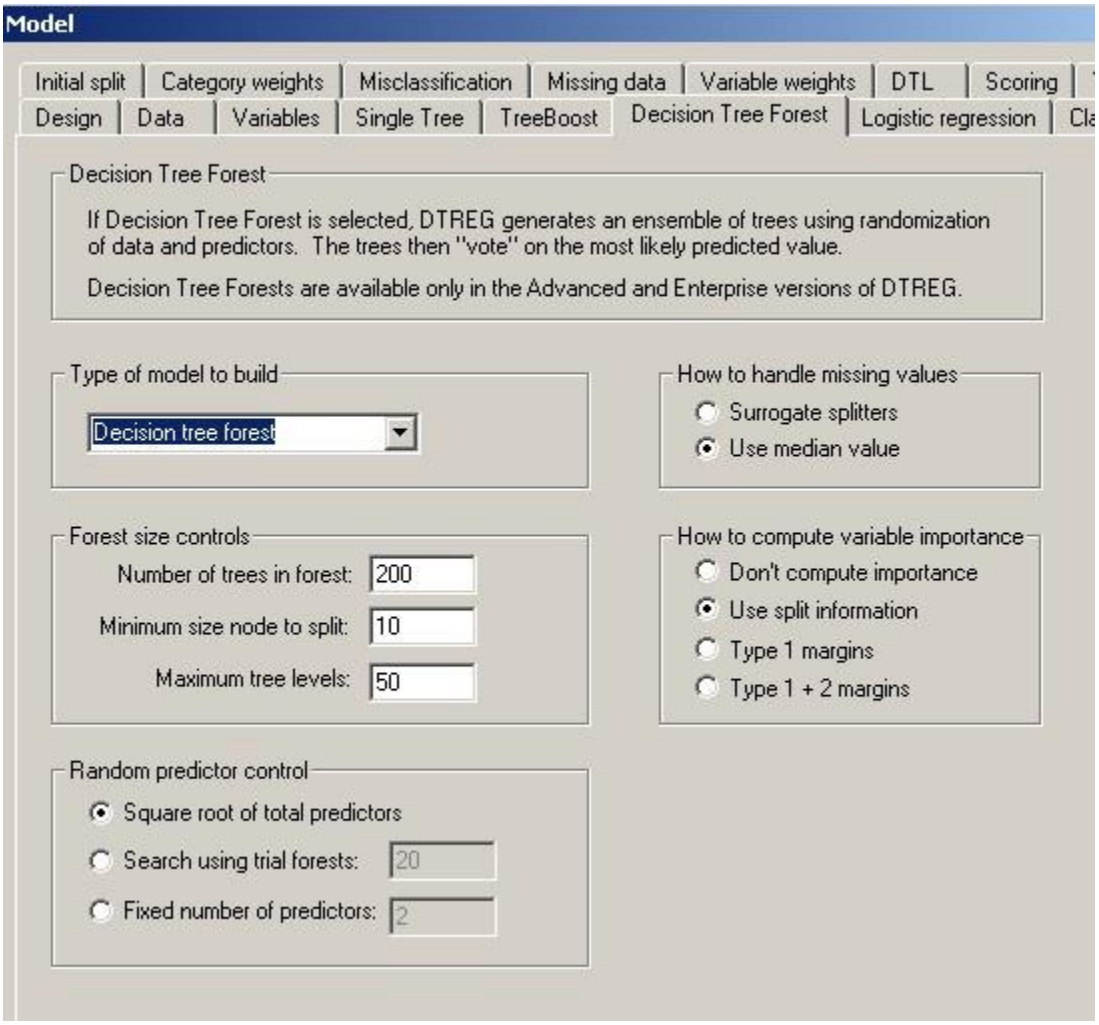
Predictor variable selection: In some cases, it may be possible to improve the quality of a TreeBoost model by considering only a random subset of the predictors for each split rather than all predictors. This is somewhat similar to the predictor selection method used by Decision Tree Forest Models. However, most of the time it is better to allow all predictors to be considered for each split, so you should always try building the model that way.

Decision Tree Forest Property Page

Decision tree forest models often can provide greater predictive accuracy than single-tree models, but they have the disadvantage that you cannot visualize them the way you can a single tree; decision tree forest models are more of a "black box".

For more technical information about decision tree forests, please see the chapter starting on page 249.

When you select the decision tree forest property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than decision tree forest, all of the other controls on this screen will be disabled.

Forest size controls

Generally, the larger a decision tree forest is, the more accurate the prediction. There are two types of size controls available (1) the number of trees in the forest and (2) the size of each individual tree.

Number of trees in forest -- This specifies how many trees are to be constructed in the decision tree forest. It is recommended that a minimum value of 100 be used.

Minimum size node to split – A node in a tree in the forest will not be split if it has fewer than this number of rows in it.

Maximum tree levels – Specify the maximum number of levels (depth) that each tree in the forest may be grown to. Some research indicates that it is best to grow very large trees, so the maximum levels should be set large and the minimum node size control would limit the size of the trees.

Random Predictor Control

When a tree is constructed in a decision tree forest, a random subset of the predictor variables are selected as candidate splitters for each node. The controls in this group set how many candidate predictors are considered as splitters for each node.

Square root of total predictors – If you select this option, DTREG will use the square root of the number of total predictor variables as the candidates for each node split. Leo Breiman recommends this as a default setting.

Search using trial forests – If you select this option, DTREG will built a set of trial decision tree forests using a different numbers of predictors and determine the optimal number of predictors to minimize the misclassification error. When doing the search, DTREG starts with 2 predictors and checks each possible number of predictors in steps of 2 up to but not including the total number of predictors. Once the optimal number of predictors is determined from the trial runs, that number is used to build the final decision tree forest. Clearly this method involves more computation than the other methods since multiple decision tree forests must be constructed. To save time, you can specify in the box on the option line a smaller number of trees in the trial forest than in the final forest.

Once the optimal number of predictors is determined, it is shown as the value with "Fixed number of predictors", so you can select that option for subsequent runs without having to repeat the search.

Fixed number of predictors – If you select this option, you can specify exactly how many predictors you want DTREG to use as candidates for each node split.

How to Handle Missing Values

Surrogate splitters – If this option is selected, DTREG will compute the association between the primary splitter selected for a node and all other predictors including predictors not considered as candidates for the split. If the value of the primary predictor variable is missing for a row, DTREG will use the best surrogate splitter whose value is known for the row. Use the Missing Data property page (see page 133) to control whether DTREG always computes surrogate splitters or only computes them when there are missing values in a node. See the chapter starting on page 357 for additional information about handling missing values.

Use median value – If this option is selected, DTREG replaces missing values for predictor variables with the median value of the variable over all data rows. While this option is less exact than using surrogate splitters, it is much faster than computing surrogates, and it often yields very good results if there aren't a lot of missing values; so it is the recommended option when building exploratory models.

How to Compute Variable Importance

DTREG offers three methods for computing the importance of predictor variables:

Use split information – DTREG calculates the importance of each variable by adding up the improvement in classification gained by each split that used the predictor. This is the same method used to compute the importance for single-tree and TreeBoost models. Generally, this method produces good results, and it can be calculated quickly.

Type 1 margins – DTREG first calculates the misclassification rate for the model using the actual data values for all predictors. Then for each predictor, it randomly permutes (rearranges) the values of the predictor and computes the misclassification rate for the model using the permuted values. The difference between the misclassification rate with the correctly ordered values and the misclassification rate for the permuted values is used as the measure of importance of the predictor. This method of calculating variable importance often is more accurate than calculating the importance from split information, but it takes much longer to compute because of the time required to permute the rows for each predictor.

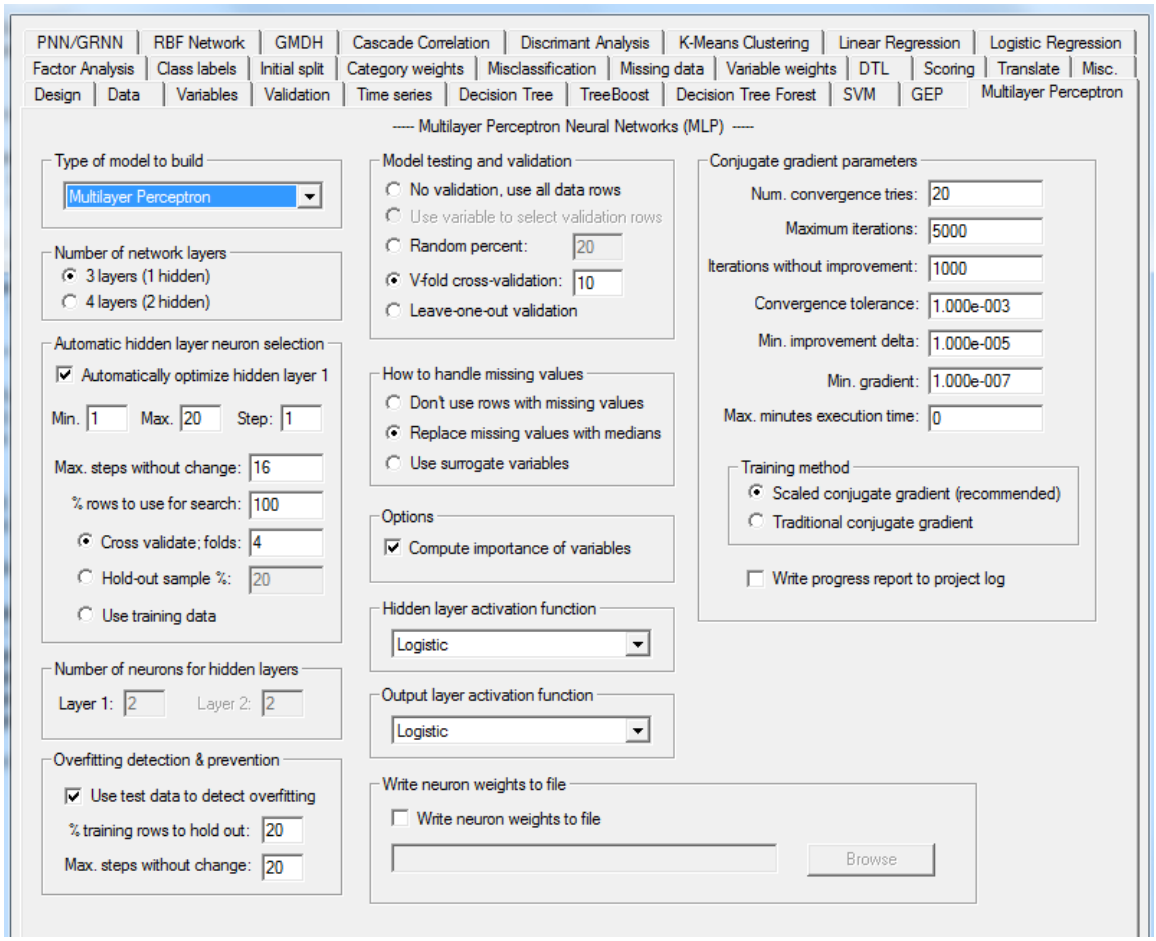
Type 1 + 2 margins – DTREG first calculates the importance using type 1 margins as described above. It then examines each data row and determines how many trees in the forest correctly voted for the row with the original data minus the number of trees that correctly voted for the row using the permuted data. The two measures of importance are then averaged. This is usually the most accurate measure of importance, but it is also the slowest to compute. In the case of a regression tree forest (i.e., continuous target variable), this method is the same as the "Type 1 margins" method.

Multilayer Perceptron Neural Networks (MLP) Property Page

A Multilayer Perceptron Neural Network (MLP) (also known as a Multilayer Feed-Forward neural network) is a model developed to simulate the function of neurons in a nervous system.

For additional information about multilayer perceptron networks, please see the chapter starting on page 253.

When you select the Multilayer Perceptron property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than multilayer perceptron neural network, all of the other controls on this screen will be disabled.

Number of network layers: Specify whether you want to create a neural network model with 3 total layers (one input, one hidden and one output) or with 4 layers (one input, two hidden and one output). It is rare to find a problem that requires more than 3 layers, so a 3 layer model is recommended.

Number of neurons

One of the basic parameters of a neural network is the number of neurons in the hidden layer(s). DTREG allows you to specify a fixed number of neurons, or you can allow allow it to search for the optimal number of neurons.

Automatically optimize hidden layer 1: One of the challenges in building multilayer perceptron neural networks is deciding how many neurons to use for the hidden layer(s). If you select too few neurons, the model may not be adequate to model complex data. If you select too many neurons, it may over-fit the data and result in poor generalization to new data. If you check this box, DTREG will try building multiple networks with different numbers of neurons in hidden layer 1 and evaluate how well they fit by using cross validation or a hold-out sample. This automatic selection only applies to hidden layer 1. If you elect to build a model with two hidden layers, you will have to manually select the number of neurons in hidden layer 2. If you enable automatic neuron optimization, you can view the Model Size chart to see how the error varies with different numbers of neurons (see page 209).

Minimum, Maximum and Step size for automatic search: If you enable the automatic search for the optimal number of neurons, specify in these fields the minimum and maximum number of neurons to try. Also specify how many neurons should be added for each trial.

Max. steps without change: When DTREG is performing the search for the optimal number of neurons, it builds models starting with the minimum specified number of neurons working up to the maximum. It evaluates the error of each model before increasing the number of neurons. If it builds the number of models specified by this value without seeing any improvement, it assumes it has passed the optimal size and stops the search.

% rows to use for search: You can tell DTREG to use only a random subset of the rows when performing the search for the optimal number of neurons. If you have a lot of training data, this can speed up the search.

Cross validate folds: If you want DTREG to evaluate the quality of each model during the search by using cross validation, check this box and specify the number of cross validation folds to use.

Hold-out sample %: If you want DTREG to hold out a portion of the data records from each trial model and then use the held-out rows to evaluate the model, specify the percent of rows to hold out in this field.

Use training data: If you select this option, then DTREG will evaluate the fit of the neural network using the same data that is uses to build the network. This is not a good

choice if you are trying to construct a network to be applied to new data that is not part of the training data. However, there are cases where the training data covers the entire set of possible values, and this option is appropriate. For example, if you are trying to build a neural network to model an exclusive OR (XOR) logic circuit, and the training data consists of all possible inputs and outputs, then it makes sense to look for the optimal network that models the training data.

Number of neurons for hidden layers: If you don't enable the automatic search for the optimal number of neurons, you can manually specify the number of neurons for each hidden layer in these fields. If you enable the automatic search, the optimal number of neurons found by the search will be shown in the Layer 1 field after the search is completed.

Over fitting Detection and Prevention

"Over fitting" occurs when the parameters of a model are tuned so tightly that the model fits the training data well but has poor accuracy on separate data not used for training. Multilayer perceptrons are subject to over fitting as are most other types of models.

DTREG has two methods for dealing with over fitting: (1) by selecting the optimal number of neurons as described below, and (2) by evaluating the model as the parameters are being tuned and stopping the tuning when over fitting is detected. This is known as "early stopping".

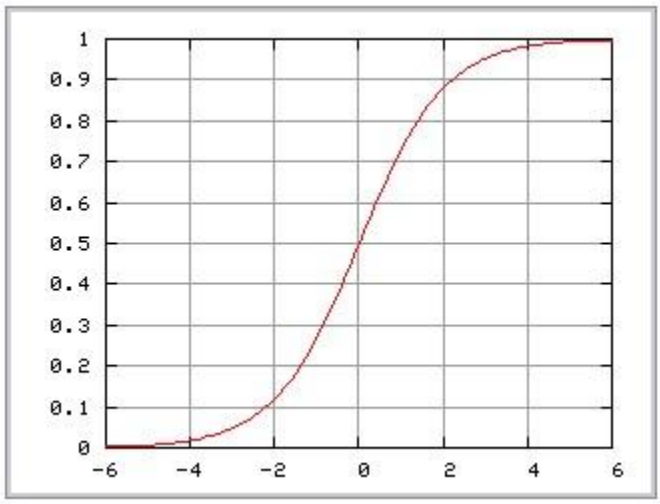
Use test data to detect over fitting: If you enable this option, DTREG holds out a specified percentage of the training rows and uses them to check for over fitting as model tuning is performed. The tuning process uses the training data to search for optimal parameter values. But as this process is running, the model is evaluated on the hold-out test rows, and the error from that test is compared with the error computed using previous parameter values. If the error on the test rows does not decrease after a specified number of iterations then DTREG stops the training and uses the parameters which produced the lowest error on the test data.

Percent training rows to hold out: Specify the percentage of the training rows that are held out and used to test for over fitting. Note, since these rows are held out, they do not contribute to the parameter optimization process.

Max. steps without change: If the error computed using the test error does not decrease (or if it increases) for this many iterations, then the training process is stopped, and the best parameters found are used for the model.

Activation Functions

Hidden layer activation function: You can select whether you want DTREG to use a linear or logistic (sigmoid) activation function for the hidden layers. A logistic function is recommended. Here is a plot of a logistic activation function:



Output layer activation function: You can select what type of activation function you want DTREG to use for the output layer. The choices are (1) a logistic (sigmoid) activation function, (2) a linear activation function or (3) a Softmax activation function. Softmax activation functions can be used only for classification analyses. Softmax produces more accurate probability estimates than the other types of activation functions, but it is slower to compute.

Model testing and validation

Select how you want DTREG to evaluate the neural network model once it has been created. You have five choices: (1) don't do any validation of the model (fast, but not recommended), (2) hold out a random percent of the rows during the model build and then run them through the model to evaluate its error, (3) use a control variable specified on the Validation Property Page to select which rows are held out for model validation, (4) perform cross validation with a specified number of folds, (5) perform cross validation with one row left out of each model build.

How to handle missing predictor variable values: If a row contains missing values for any of the predictor variables you can specify whether you want DTREG to (1) exclude the row from the model building process, (2) replace the missing values with the median value of the predictor variable, or (3) use surrogate variables. See page 358 for information about surrogate variables.

Compute importance of variables: If this box is checked, DTREG will compute and display the relative importance of each predictor variable. The calculation is performed using sensitivity analysis where the values of each variable are randomized and the effect on the quality of the model is measured.

Write neuron weights to a file: If this box is checked and a file name is entered in the field below it, DTREG will create a comma separated value file containing the values of the weights for each neuron. Here is an example of a weight file:

```
Layer,Neuron,Input,Offset,Weight,Function
1,N[1.1],"Time",5400.0000000,-0.0003555,Logistic
1,N[1.1],"1.0",0.0,-3.5025324,Logistic
2,N[2.1],"N[1.1]",0.0,4.6004631,Linear
2,N[2.1],"1.0",0.0,0.0970731,Linear
```

Conjugate gradient parameters

DTREG uses the conjugate gradient method to find the optimal network weights. For additional information about the conjugate gradient algorithm, see page 258.

Number of convergence tries – Specify how many sets of random starting values DTREG should use when trying to find the optimal set of network parameters. For each try, DTREG will create a set of random starting parameter values within the range specified by the Nguyen-Widrow algorithm and then use conjugate gradient to optimize them. Since there is no guarantee that conjugate gradient will converge to the global minimum, it is useful to try multiple, different random starting values. The network training time is directly proportional to the number of tries allowed.

Convergence tolerance: The conjugate gradient algorithm will iterate until the specified convergence tolerance is reached or it is stopped for another reason such as reaching the maximum allowed number of iterations. The convergence tolerance value specifies the proportion of residual unexplained variance that is left. That is, the convergence tolerance value specifies the remaining R^2 variance. For example, if a tolerance factor of 0.001 is specified, then the algorithm iterates until residual, unexplained R^2 reaches 0.001 which means the *explained* R^2 reaches 0.999 (99.9%).

Maximum iterations: Specify the maximum iterations you will allow DTREG to perform during the conjugate gradient optimization.

Iterations without improvement: After each iteration, DTREG measures the residual error of the model using the weight values calculated by the iteration. If the error does not improve after this many consecutive iterations, DTREG assumes the weights have converged to the optimal values, and it stops the conjugate gradient process.

Minimum improvement delta: This is the amount of improvement in the residual model error required for DTREG to count an iteration as having improved the model. If

the error is improved by less than this amount (or not at all), then no improvement is counted.

Min. gradient: If the largest weight gradient value is less than this parameter, DTREG assumes it has reached an optimal (flat) section of the error space and stops the conjugate gradient process. A gradient value measures the change in the model error relative to a change in a weight value, so a small gradient indicates that little improvement can be made by changing the weight value.

Max. minutes execution time: If this value is non-zero, DTREG will stop the conjugate gradient process after the specified number of minutes of run time and use the resulting weights as the final ones for the model.

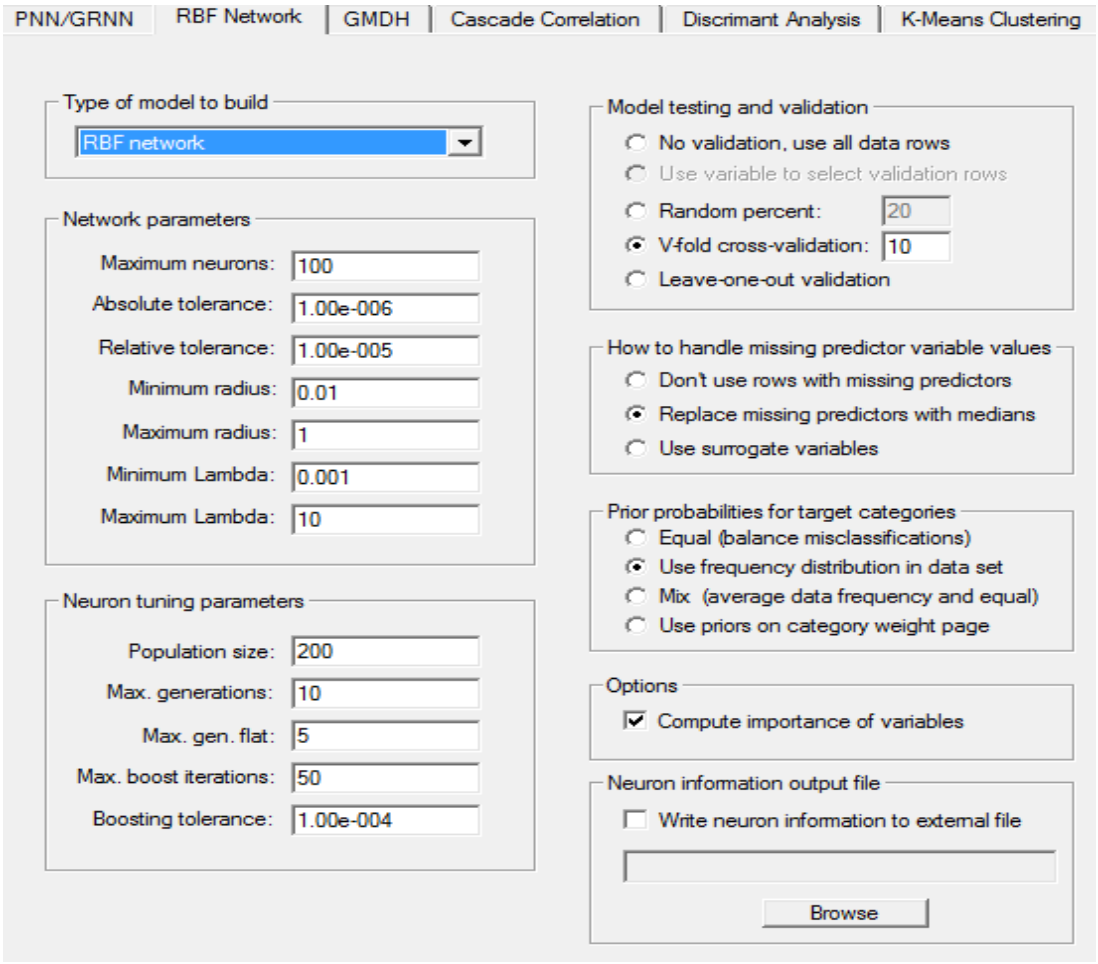
Training method: Specify whether you want DTREG to use the (1) scaled conjugate gradient or (2) traditional conjugate gradient algorithm. Usually, scaled conjugate gradient is faster than traditional conjugate gradient and produces results as least as good. For additional information about the conjugate gradient algorithms, see page 258258.

Write progress report to project log: If this box is checked, DTREG will write a report showing the improvement in the model after each conjugate gradient iteration.

RBF Neural Networks Property Page

A Radial Basis Function (RBF) neural network models data by fitting Gaussian functions to the training data. For more information about RBF networks, see the chapter beginning on page 261.

When you select the RBF Network property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than RBF Network, all of the other controls on this screen will be disabled.

Network Parameters

Maximum neurons: Specify the maximum number of neurons you allow to be used in the model. The RBF training algorithm stops adding neurons when it detects that over fitting may occur, so usually models will have fewer than the maximum allowed number of neurons.

Absolute tolerance: If the residual mean squared error (MSE) is reduced to this value, the training stops.

Relative tolerance: If the residual error is reduced by less than this amount by adding another neuron, the training stops.

Minimum radius: The minimum radius (spread) for neurons.

Maximum radius: The maximum radius (spread) for neurons. This parameter provides guidance for the maximum radius, but it is not an absolute limit. The training process may determine that a larger radius is required. If the validation error is significantly worse than the training error, try increasing the value of the maximum radius. If the training and validation errors are close but larger than you want, try decreasing the maximum radius.

Minimum lambda: This is the minimum value of the Lambda regularization parameter that will be used while computing weights as neurons are added to the network. If over fitting is indicated by the validation error being much larger than the training error, try increasing the minimum lambda.

Maximum lambda: This is the maximum value of the Lambda regularization parameter that will be used wile computing weights are neurons are added to the network.

Neuron Tuning Parameters

Population size: Part of the algorithm used by DTREG to build neural networks uses an evolutionary method called **Repeating Weighted Boosting Search** (**RWBS**). During the first part of this search, a population of candidate neurons is created with random centers and spreads (limited by the minimum and maximum specified radius). The population size parameter controls how many candidate neurons are created. It is advisable to increase the population if there are many predictor variables. A reasonable minimum population size is two times the number of predictor variables. Increasing the population size also may help the algorithm avoid local minima and find the optimal global solution.

Max. generations: This parameter controls the maximum number of generations of candidate neurons to be created by the RWBS evolutionary algorithm. Each generation uses a combination of the best neurons from the previous generation and new random neurons. The evolutionary process stops when the maximum number of generations is reached or no improvements are gained.

Max. gen. flat: If the RWBS evolutionary algorithm advances through this many consecutive generations without improvement, it stops.

Boosting tolerance: During each RWBS generation, candidate neurons are "mated" and the improvement in estimated leave-one-out error is computed. If the estimated error is

less than the boosting tolerance parameter, the boosting operation stops and the next generation begins.

Model Testing and Validation Parameters

No validation: The mode is trained but no validation is performed. This is fast, but it is not recommended because there is no way to measure how well the model is likely to generalize to new data.

Random percent: If this option is selected, a random percentage of the rows are held out during the validation training, then those held-out rows are run through the model and their error is reported as the validation error.

Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

V-fold cross validation: If this option is selected, *V* SVM models will be constructed with (*V*-1)/*V* proportion of the rows being used in each model. The remaining rows are then used to measure the accuracy of the model. The final model is built using all data rows. This method has the advantage of using all data rows in the final model, but the validation is performed in separately constructed models so there is some possibility that the misclassification rate for the final model may be different than the validation models.

Missing Value Parameters

How to handle missing predictor values: DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Prior Probability Parameters

Prior probabilities for target categories: Select the assumed prior probability distribution for the target variable categories. Traditionally (and in most benchmarks) the distribution in the training data set is used. If you wish to specify a custom set of prior probabilities, select the option "Use priors on category weight page", and set the values of the priors on the Category weight property page (see page 128).

Miscellaneous Options

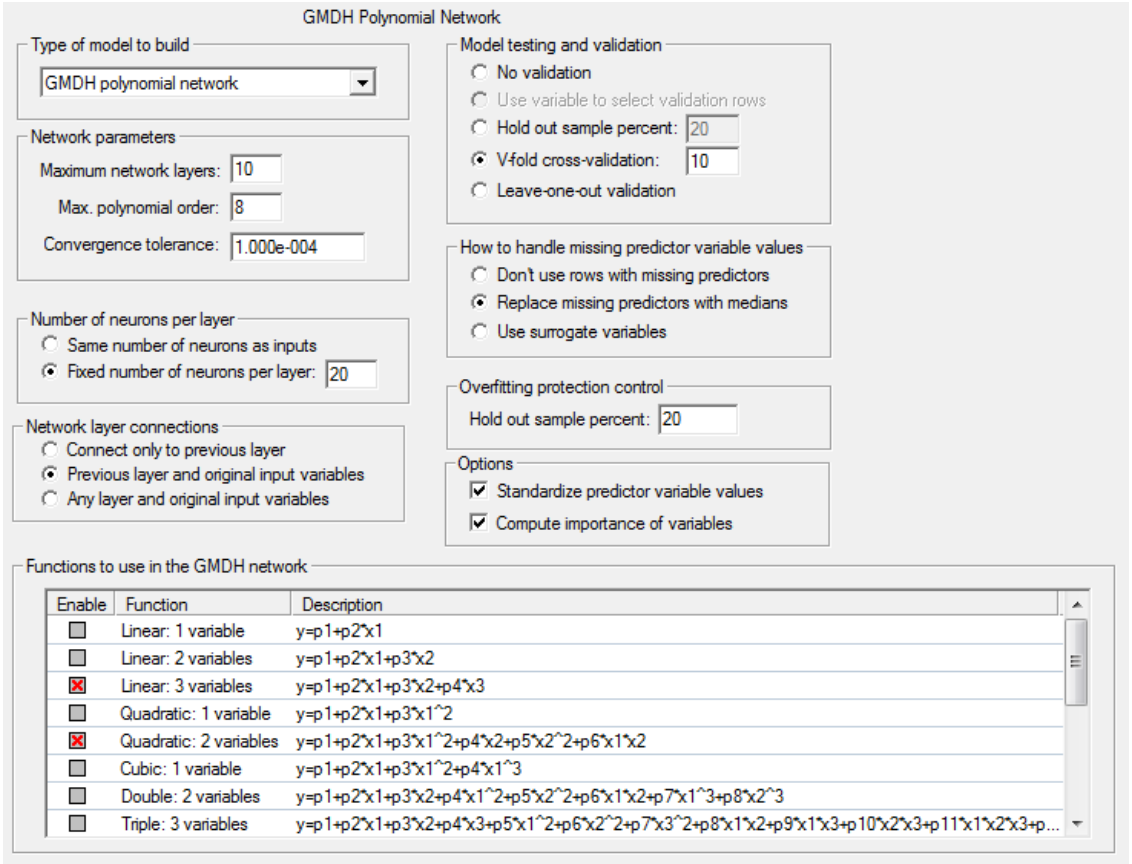
Compute importance of variables: If this box is checked, DTREG will compute and display the relative importance of each predictor variable. The calculation is performed using sensitivity analysis where the values of each variable are randomized and the effect on the quality of the model is measured.

Write neuron information to an external file: If this box is checked and a file name is entered below it, then DTREG will write information about each RBF neuron to the specified file. This information includes the center, width, and weight of each neuron. The generated file is a .csv Comma Separated Value file.

GMDH Polynomial Neural Networks Property Page

A GMDH polynomial neural network models data by fitting a network of polynomial functions to the training data. For more information about GMDH polynomial networks, see the chapter beginning on page 269.

When you select the GMDH Network property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than GMDH Polynomial Network, all of the other controls on this screen will be disabled.

Network Parameters

Maximum network layers: Specify the maximum number of layers in the neural network that the model may contain. The model may actually be created with fewer layers if the building process discovers that adding layers would harm or not improve the accuracy of the model.

Maximum polynomial order: Specify the highest power of a variable that a polynomial may contain. If the GMDH network is built using quadratic polynomials, then the order of the polynomials doubles on each layer.

Convergence tolerance: The training algorithm will add layers to the network until the specified convergence tolerance is reached or it is stopped for another reason such as reaching the maximum allowed number of layers or it detects that adding a layer will not improve the model. The convergence tolerance value specifies the proportion of residual unexplained variance that is left. That is, the convergence tolerance value specifies the remaining R^2 variance. For example, if a tolerance factor of 0.001 is specified, then the algorithm iterates until residual, unexplained R^2 reaches 0.001 which means the *explained* R^2 reaches 0.999 (99.9%).

Number of neurons per layer: This is the number of neurons that will be held in each layer of the network. You can specify an exact number, or you can select the option to use the same number of neurons as exist in the input layer.

Network layer connections: This parameter controls how neurons in the network are connected together. There are three choices:

1. **Connect only to previous layer** – This option tells DTREG that the inputs to one layer may come only from outputs generated by the next lower layer.
2. **Previous layer and the input variables** – This allows inputs to a layer to be connected to outputs from the previous layer and also to the original predictor variables.
3. **Any layer and original input variables** – This option allows DTREG to connect inputs to neurons in one layer to outputs from any lower level layer and also the input variables. Selecting this option usually results in slow training because the number of possible inputs increases as layers are added.

Model Testing and Validation Parameters

No validation: The mode is trained but no validation is performed. This is fast, but it is not recommended because there is no way to measure how well the model is likely to generalize to new data.

Random percent: If this option is selected, a random percentage of the rows are held out during the validation training, then those held-out rows are run through the model and their error is reported as the validation error.

Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

V-fold cross validation: If this option is selected, *V* SVM models will be constructed with (*V*-1)/*V* proportion of the rows being used in each model. The remaining rows are then used to measure the accuracy of the model. The final model is built using all data

rows. This method has the advantage of using all data rows in the final model, but the validation is performed in separately constructed models so there is some possibility that the misclassification rate for the final model may be different than the validation models.

Missing Value Parameters

How to handle missing predictor values: DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Over fitting Protection Control

Holdout sample percent: Specify the percent of the training rows that are to be used as the *control data* to detect model over fitting. See the description of the GMDH training algorithm on page 270 for information about how the control data is used.

Miscellaneous Options

Standardize Predictor Variable Values: If this box is checked, DTREG standardizes the values of continuous predictor variables by subtracting the mean and dividing by the standard deviation. The target variable values are not standardized.

Compute importance of variables: If this box is checked, DTREG will compute and display the relative importance of each predictor variable. The calculation is performed using sensitivity analysis where the values of each variable are randomized and the effect on the quality of the model is measured.

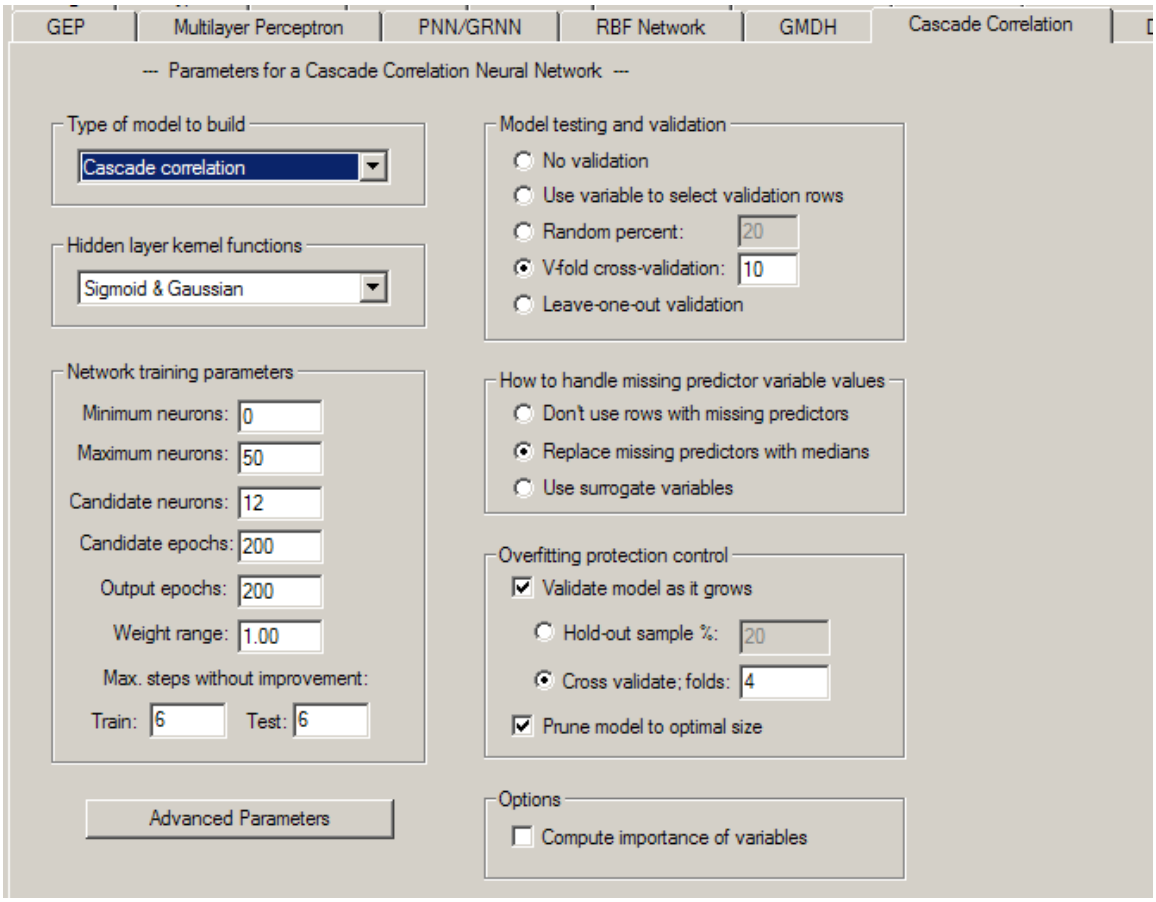
Functions to Use in the GMDH Network

Check which functions you wish to enable DTREG to use in the network. Traditional GMDH polynomial networks use only quadratic polynomials of two variables.

Cascade Correlation Neural Networks Property Page

A Cascade Correlation neural network is a type of self-organizing neural network. That is, it grows and adds neurons to the architecture as necessary to accurately model the data. For additional information about Cascade Correlation networks, please see the chapter beginning on page 273.

When you select the Cascade Correlation property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than Cascade Correlation, all of the other controls on this screen will be disabled.

Hidden layer kernel functions: Select the type of kernel functions that you want the model to be able to use in the hidden layer. The choices are (1) Sigmoid only, (2) Gaussian only, and (3) both Sigmoid and Gaussian functions. Generally it is best to allow the network to consider both sigmoid and Gaussian kernel functions and use whichever is best.

Minimum and Maximum Neurons: Specify the minimum and maximum number of neurons that may be used for the hidden layer. It is recommended that the minimum

number of neurons be set to 0 (zero). There are a surprising number of problems that can be solved best with just the output layer and no hidden neurons.

Candidate neurons: Specify how many candidate neurons are to be considered for each addition to the hidden layer. During the training process, DTREG generates a set of candidate neurons for each step. These candidate neurons have random weight values within the range specified by the Weight Range parameter. The candidate neurons will have either sigmoid, Gaussian or a mixture of kernel functions. Increasing the number of candidate neurons may reduce the number of neurons used in the hidden layer, but it will increase the training time.

Candidate epochs: This is the maximum number of iterative cycles that will be used to compute the weights for candidate neurons being considered for inclusion in the model.

Output epochs: This is the maximum number of iterative cycles that will be used to compute the weights for the output neurons.

Weight range: When candidate neurons are created, they are initially assigned weights whose values range from the negative of this value up to the positive of this value. Usually training is insensitive to the starting random values, so the range is no important.

Maximum steps without improvement: These two parameters specify how many neurons can be added without any reduction in the error. Each time a candidate neuron is added to the hidden layer, the model is re-trained and the error is computed. If the error is not reduced, a count is incremented. When validation is performed to find the optimal size of the network, a separate no-improvement count is kept for it. If the no-improvement counts reach the specified values, then training is terminated. Training also is terminated if the error on the training rows reaches zero.

Model testing and validation: Select how you want DTREG to evaluate the neural network model once it has been created. You have four choices: (1) don't do any validation of the model (fast, but not recommended), (2) hold out a random percent of the rows during the model build and then run them through the model to evaluate its error, (3) perform cross validation with a specified number of folds, (4) perform cross validation with one row left out of each model build.

How to handle missing predictor variable values: DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Over fitting Protection Control

Because cascade correlation networks add hidden-layer neurons during the training process, there is a serious risk of the model becoming so complex that it fits the training data very well but does not generalize well to new, unseen data; this is called *over fitting*. To prevent over fitting, DTREG tests the accuracy of the model using validation data after each neuron is added. It stops the building when over fitting is detected because the validation error reaches a minimum and starts to increase. You can view the Model Size chart (see page 209) to view how the error changes as neurons are added. The Model Size section of the analysis report also provides this information.

Validate model as it grows: Check this box to enable over fitting prevention. If you don't check this box, then the model will grow without restraint as it attempts to perfectly fit the training data.

Hold out sample percent or Cross-validation folds: Select whether you want the over fitting detection performed by using a hold-out set of rows or by using cross validation. Cross validation is recommended, but it slower than using a hold-out sample.

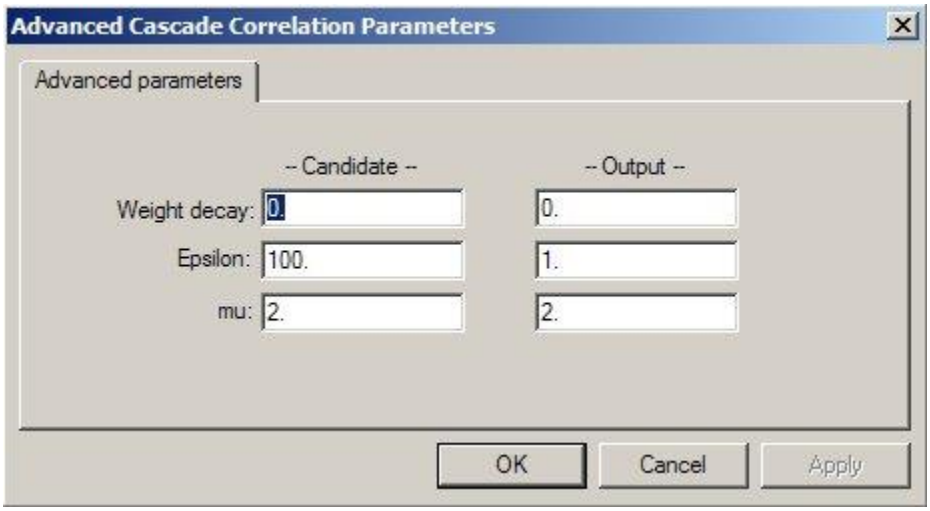
Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

Prune model to optimal size: If this box is checked, DTREG will prune the size of the model back to the number of neurons that generated the lowest error on the validation rows (i.e., the hold-out rows or the cross-validation error).

Compute importance of variables: If this box is checked, DTREG will compute and display the relative importance of each predictor variable. The calculation is performed using sensitivity analysis where the values of each variable are randomized and the effect on the quality of the model is measured.

Advanced Cascade Correlation Parameters

If you click the "Advanced Parameters" button, the following screen will be displayed:



Usually it is not necessary to change the parameters on this screen. The default values work well for most problems.

There are two sets of parameters, "**Candidate**" and "**Output**". The first set is used during the training of the weights of candidate neurons for the hidden layer. The second set is used when training the weights for the output layer neurons.

Weight decay: This is a regularization parameter that encourages weight values to remain close to zero. The larger the weight decay, the more tightly the parameters are forced toward zero. If you encounter a situation where weights seem to be going wild and the model error is getting worse rather than better, try using a weight decay value in the range of 0.001.

Epsilon and **mu**: These two parameters are used by the quickprop training algorithm. Here are suggestions for these parameters posted by Scott E. Fahlman, the co-inventor of Cascade Correlation:

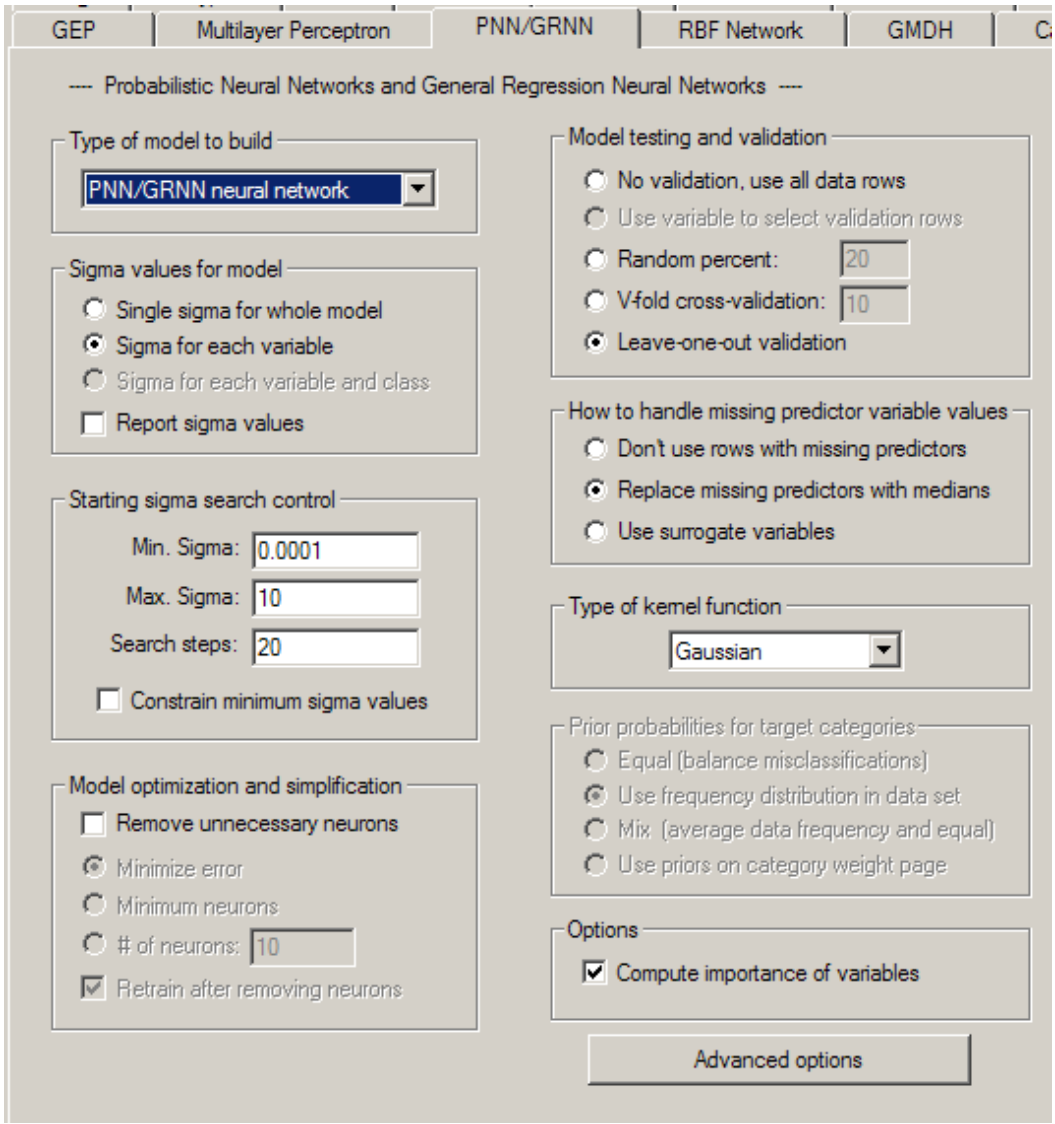
Epsilon: "This is the tricky one. It can vary over many orders of magnitude, depending on the problem (i.e. from 1000 to 0.01 or so). I've tried a number of kinds of normalization to keep this in a single range for all problems, but haven't found the magic bullet yet. Basically, you want to see steady improvement in the error measure or score. You'll occasionally see an epoch or two in which the score retreats from the best obtained so far, but if the lost ground isn't made up in the next few epochs, you're probably in the chaotic region and need to reduce the epsilon that is relevant to the current learning phase. If you see steady but weak convergence, especially near the end of the training phase, you want to turn it up."

Mu: "Set them at 2.0 and leave them there. If the training seems determined to oscillate, turn it down to 1.75 or 1.5. I think I've only gone higher than 2.0 for a few odd problems like XOR."

Probabilistic and General Regression Neural Networks Property Page

Probabilistic and General Regression Neural Networks are another type of neural network. For additional information about these networks, please see the chapter starting on page 279.

When you select the PNN/GRNN property page, you will see a screen like this:



The same parameter screen is used for probabilistic and general regression neural networks. If a classification analysis is being performed (with a categorical target variable), then a probabilistic neural network (PNN) is created. If a regression analysis is being performed (with a continuous target variable), then a general regression neural network (GRNN) is created.

Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than PNN/GRNN, all of the other controls on this screen will be disabled.

Sigma values for model: The sigma values control the radius of influence of each point in the model. DTREG provides three types of sigma values:

1. **Single sigma for whole model.** This is the simplest type of model; it uses a single sigma value for all points. This type of model is faster to build than the other types but usually is less accurate.
2. **Sigma for each variable.** This calculates a separate sigma value for each predictor variable in the model. This allows the influence of each variable on neighboring points to differ. This is the default and recommended choice because it is a good compromise between having a single sigma and allowing a separate sigma for each target category.
3. **Sigma for each variable and class.** This creates a separate sigma value for each predictor variable and for each target category. Usually there is little (if any) improvement over using a sigma for each variable, and in some cases the accuracy of the model suffers. This option is only available if the target variable is categorical

Report sigma values: If this box is checked, DTREG will show the computed sigma values in the project report.

Starting sigma search control: These parameters control the range of sigma values used during the initial search. Once the conjugate gradient method begins, the sigma values are allowed to move outside the range.

Constrain minimum sigma values: If you check this box then the sigma values will be constrained so that they cannot go smaller than the "Min. Sigma" parameter. Usually it is better to leave this box unchecked so that the optimization is free to select the best sigma values even if they are small. However, very small sigma values can sometimes result in a model that is "brittle": small changes in input predictor values cause large swings in the predicted target value.

Model testing and validation: Select how you want DTREG to evaluate the neural network model once it has been created. You have five choices: (1) don't do any validation of the model (fast, but not recommended), (2) hold out a random percent of the rows during the model build and then run them through the model to evaluate its error, (4) use a hold-out control variable specified on the Validation Property Page to select

which rows will be held out for testing, (4) perform cross validation with a specified number of folds, (5) perform cross validation with one row left out of each model build.

How to handle missing predictor variable values: DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Type of kernel function: The kernel function controls how the influence of a point declines as the radius from the point increases. DTREG supports two types of kernel functions:

1. **Gaussian:** A Gaussian function causes the influence of a point to decline according to the value (height) of a Gaussian distribution centered on the point. Gaussian functions are almost always the best kernel. The equation of the Gaussian function is:

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\left(\frac{-x^2}{2\sigma^2}\right)}$$

2. **Reciprocal:** The influence of the point decreases as a linear function of the distance from the point.

Prior probabilities for target categories: Select the assumed prior probability distribution for the target variable categories. Traditionally (and in most benchmarks) the distribution in the training data set is used. If you wish to specify a custom set of prior probabilities, select the option "Use priors on category weight page", and set the values of the priors on the Category weight property page (see page 128).

Compute importance of variables: If this box is checked, DTREG will compute and display the relative importance of each predictor variable. The calculation is performed using sensitivity analysis where the values of each variable are randomized and the effect on the quality of the model is measured.

Remove unnecessary neurons: If this box is checked, DTREG will optimize the PNN/GRNN model by removing neurons that are unnecessary. If this box is not checked, then all of the training rows will be retained in the model.

Removing unnecessary neurons has three benefits:

1. The size of the stored model is reduced.
2. The time required to apply the model during scoring is reduced.
3. Removing neurons often improves the accuracy of the model.

The process of removing unnecessary neurons is a slow, iterative process because the model must be evaluated with each remaining neuron to find the best one to remove. For models with more than 1000 training rows, the neuron removal process may become impractically slow. If you have a multi-CPU computer, you can speed up the process by

allowing DTREG to use multiple CPU's for the process. See page 16 for information about how to control CPU usage.

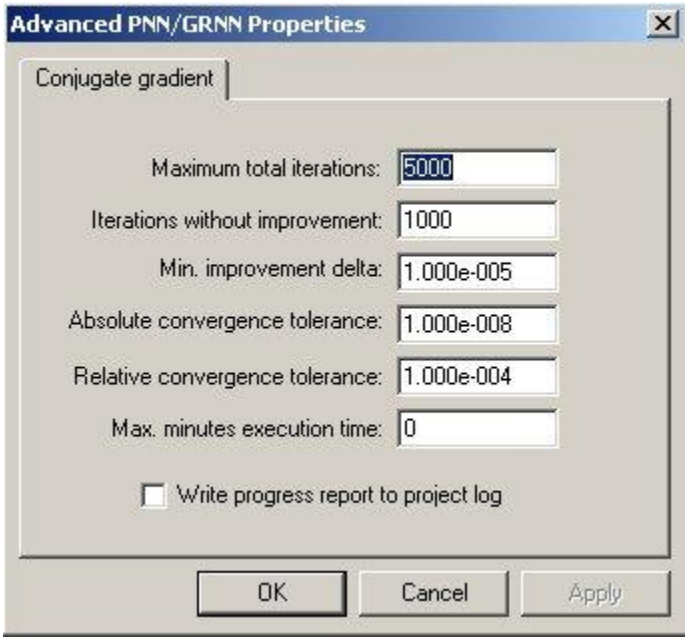
When unnecessary neurons are removed, the "Model Size" section of the analysis report shows how the error changes with different numbers of neurons. You can see a graphical chart of this by clicking Chart/Model size (see page 209).

There are three criteria that can be selected to guide the removal of neurons:

- **Minimize error** – If this option is selected, then DTREG removes neurons as long as the leave-one-out error remains constant or decreases. It stops when it finds a neuron whose removal would cause the error to increase above the minimum found.
- **Minimize neurons** – If this option is selected, DTREG removes neurons until the leave-one-out error would exceed the error for the model with all neurons.
- **# of neurons** – If this option is selected, DTREG reduces the least significant neurons until only the specified number of neurons remain.

Retrain after removing neurons: If this box is checked, DTREG will retrain the network (i.e., compute new Sigma values) using only the neurons left after the unnecessary neurons are removed. Sometimes this improves the quality of the model. If retraining does not improve the quality, the original Sigma values are used; so there is no harm in trying to retrain other than the retraining time. The Model Size report shows whether retraining improved the model.

Advanced options: Click this button to open the screen where you can set parameters for the conjugate gradient optimization process:



Maximum total iterations: Specify the maximum iterations you will allow DTREG to perform during the conjugate gradient optimization.

Iterations without improvement: After each iteration, DTREG measures the residual error of the model using the weight values calculated by the iteration. If the error does not improve after this many consecutive iterations, DTREG assumes the weights have converged to the optimal values, and it stops the conjugate gradient process.

Minimum improvement delta: This is the amount of improvement in the residual model error required for DTREG to count an iteration as having improved the model. If the error is improved by less than this amount (or not at all), then no improvement is counted.

Absolute convergence tolerance: If the residual error of the model is less than this parameter, DTREG assumes it has converged and stops the conjugate process.

Relative convergence tolerance: If the error declines by less than this amount during an iteration, DTREG will assume it has reached the optimal point and stop the process.

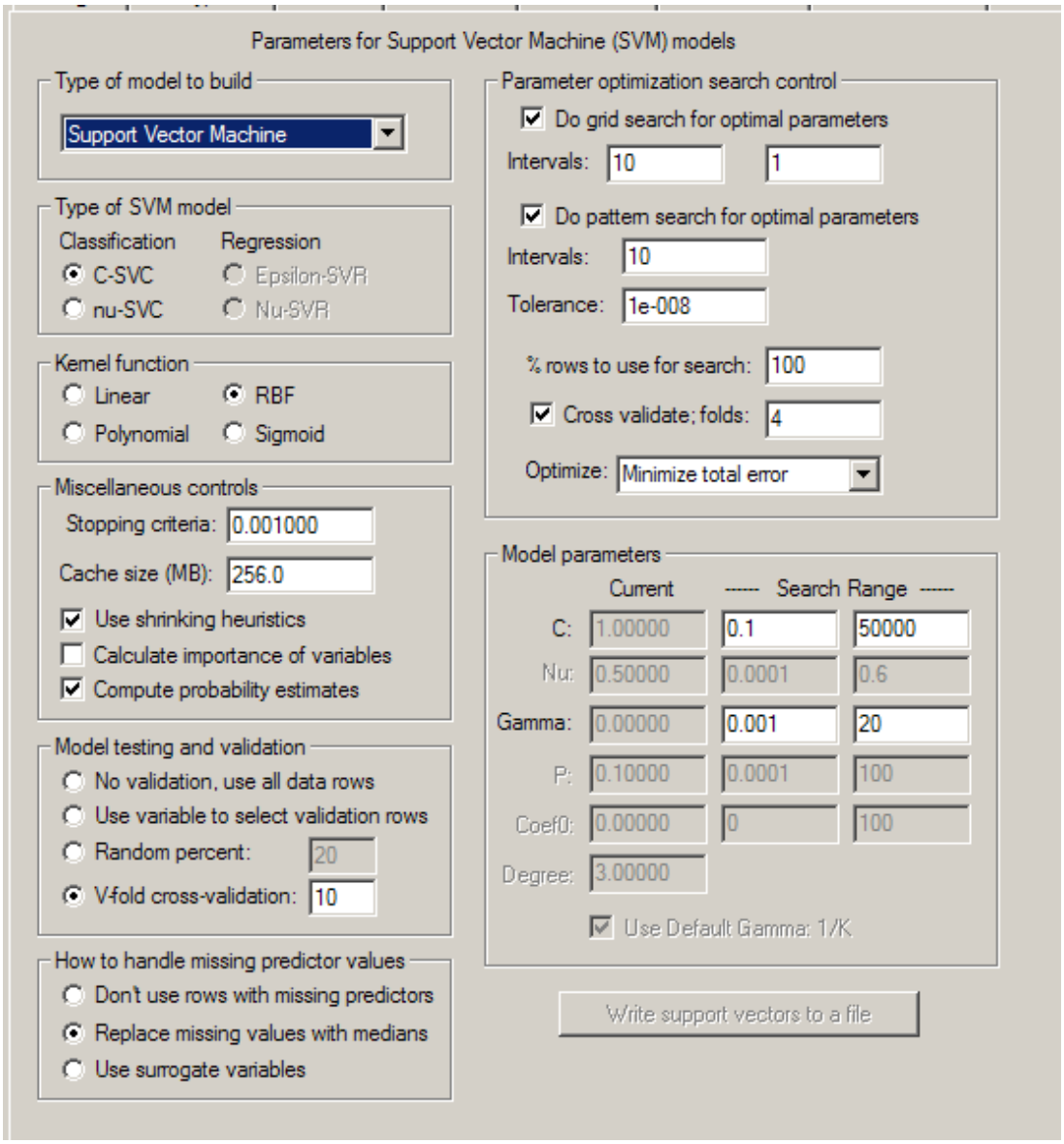
Max. minutes execution time: If this value is non-zero, DTREG will stop the conjugate gradient process after the specified number of minutes of run time and use the resulting weights as the final ones for the model.

Write progress report to project log: If this box is checked, DTREG will write a report showing the improvement in the model after each conjugate gradient iteration.

Support Vector Machine (SVM) Property Page

A Support Vector Machine (SVM) is a relatively new modeling method that has shown great promise at generating accurate models for a variety of problems. SVM seems to be particularly good at pattern recognition, but it also applicable to all other types of modeling applications. For more technical information about support vector machine models, please see the chapter starting on page 289.

When you select the SVM property page, you will see a screen like this:

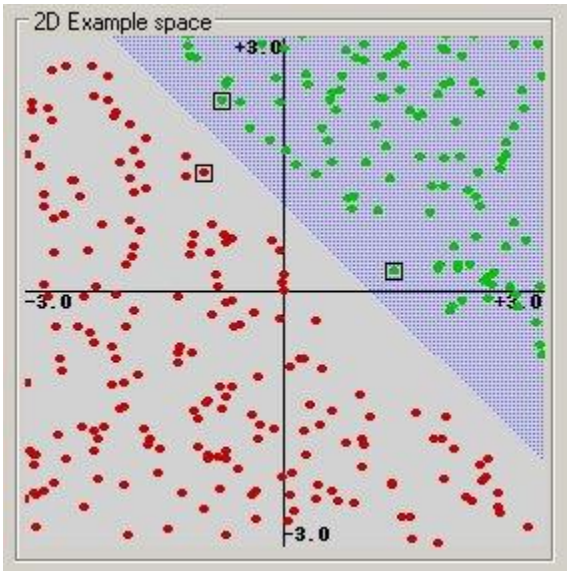


Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than support vector machine, all of the other controls on this screen will be disabled.

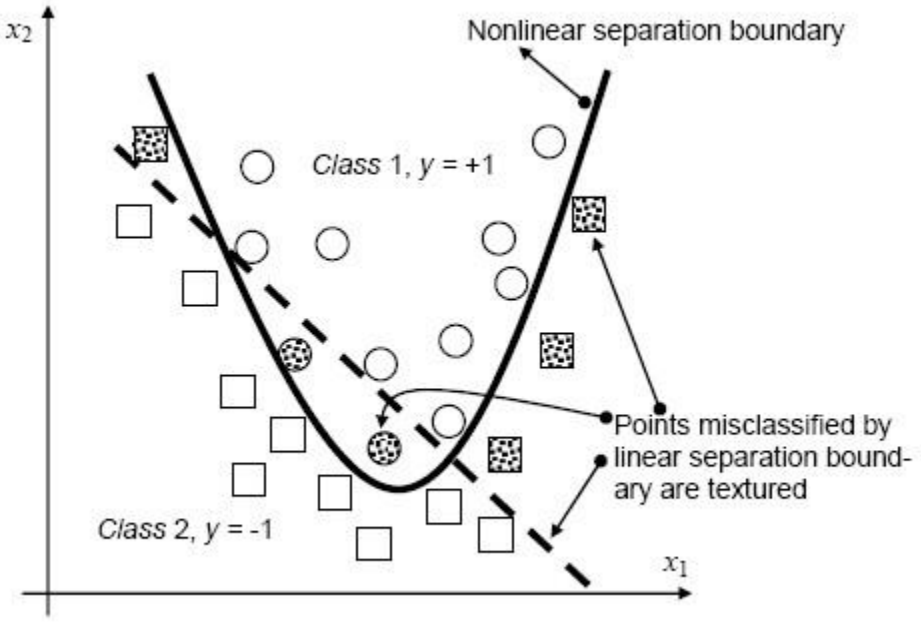
Type of SVM model – DTREG offers several types of SVM models. For classification models with a categorical target variable, you can select either C-SVC or ν-SVC models. For regression models with a continuous target variable, you can select either ε-SVR or ν-SVR models. For most applications, the results generated by the different models are quite similar. There is no way to predict in advance which method will perform better for a particular problem, so it is best to try each one.

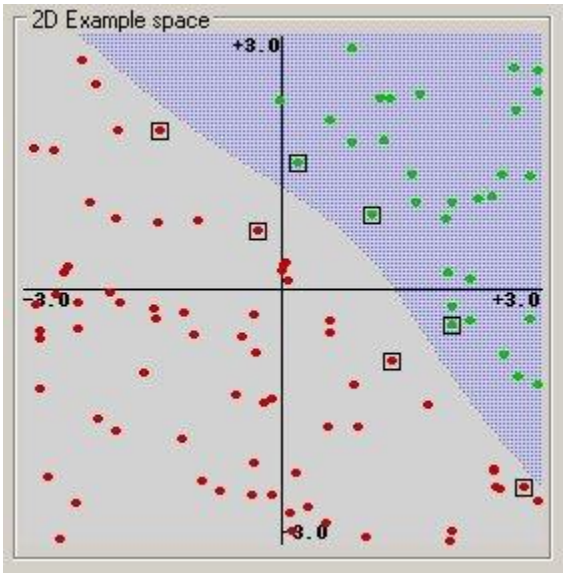
Kernel function – SVM models are built around a *kernel function* that transforms the input data into an *n*-dimensional space where a hyperplane can be constructed to partition the data. DTREG provides four kernel functions, Linear, Polynomial, Radial Basis Function (RBF) and Sigmoid (S-shaped). There is no way in advance to know which kernel function will be best for an application, but the RBF function has been found to do best job in the majority of cases.

Linear: u'*v

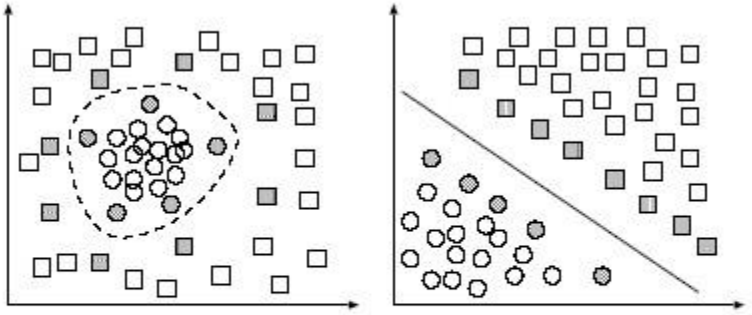


Polynomial: (gamma*u'*v + coef0)^degree
See the following figure from Kecman, 2004.



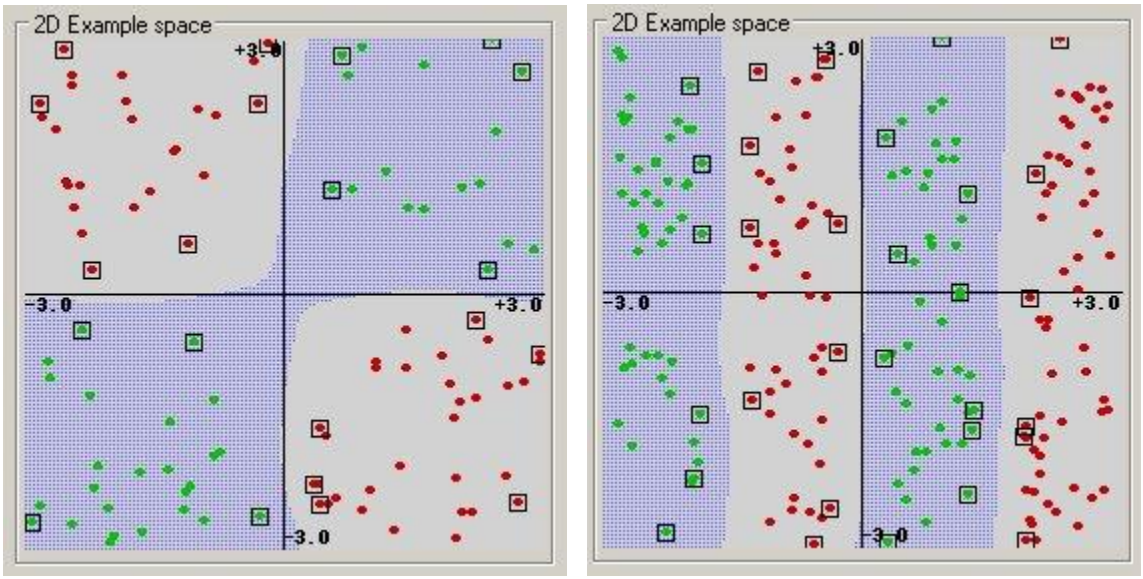


Radial basis function: exp(-gamma*|u-v|^2)
A Radial Basis Function (RBF) is the default and recommended kernel function. The RBF kernel non-linearly maps samples into a higher dimensional space, so it can handle nonlinear relationships between target categories and predictor attributes; a linear basis function cannot do this. Furthermore, the linear kernel is a special case of the RBF. A sigmoid kernel behaves the same as a RBF kernel for certain parameters. The RBF function has fewer parameters to tune than a polynomial kernel, and the RBF kernel has less numerical difficulties. The following chart fromYang, 2003 illustrates RBF mapping.

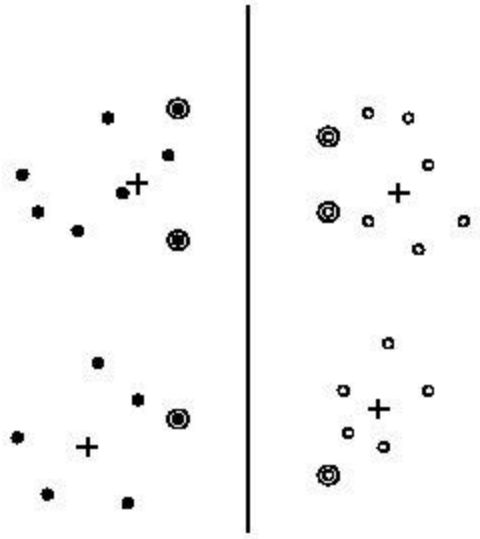


(a) Radial Basis Function (b) RBF mapping

Separable classification with Radial Basis kernel functions in different space. Left: original space. Right: feature space.

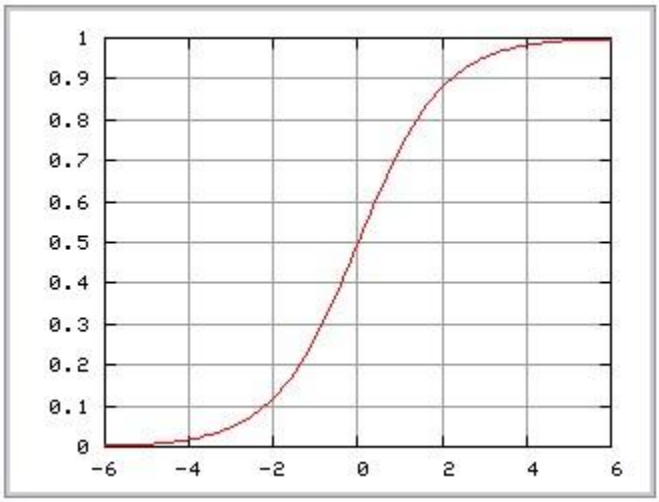


An SVM model using a radial basis function kernel has the architecture of an RBF network. However, the method for determining the number of nodes and their centers is different from standard RBF networks with the centers of the RBF notes on the support vectors (see the figure below from C. Campbell).



A classical RBF network finds the centers of RBF nodes by *k*-means clustering (marked by crosses). In contrast an SVM with RBF kernels uses RBF nodes centered on the support vectors (circled), i.e., the datapoints closest to the separating hyperplane (the vertical line illustrated).

Sigmoid: tanh(gamma*u'*v + coef0)



Stopping criteria (Epsilon) – This is a tolerance factor that controls when DTREG stops the iterative optimization process. The default value usually works well; you can reduce the tolerance to generate a more accurate model or increase the value to reduce the computation time. This parameter is called the Epsilon value in some other implementations of SVM.

Cache size – DTREG uses a cache to store truncated rows of the reordered kernel matrix. This cache avoids recomputing components of the kernel matrix and can speed up the

computation by a significant amount in some cases. The cache size value is specified in units of mega-bytes (MB). The default value is 256 (MB). Research has shown that on machines with lots of memory increasing the cache size up to 512 (MB) or even 1000 (1 GB) can improve performance.

Use shrinking heuristics – A SVM model is formed by selecting a hyperplane that partitions the data with maximum margin between the feature vectors that define points near overlap. Shrinking improves performance by allowing DTREG to ignore points that are far from overlapping and which are unlikely to influence the choice of the optimal separating hyperplane. Essentially, shrinking eliminates outlying vectors from consideration. Enabling shrinking heuristics can significantly speed up performance when the training data set is large; it is recommended that shrinking be enabled.

Calculate importance of variables – If this option is selected, DTREG will analyze the generated SVM model and generate a report on the relative significance of predictor variables.

Compute probability estimates – If this option is selected, DTREG generates an SVM model that is capable of estimating the probability for each target category rather than simply predicting the most likely category. This option is especially useful for problems with only two target categories because you can use the probability threshold features in DTREG to adjust the proportion of cases assigned each category. Note: when this option is selected, a different type of model is constructed, and the misclassification rate for the model may be different than for a model without probability calculations.

Model testing and validation – DTREG offers two methods for validating an SVM model:

Random percent holdback – If this option is selected, DTREG will select a random set of data rows and hold them out of the model building process. These rows will then be run through the generated model and the misclassification error rate will be reported.

Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

V-fold cross validation – If this option is selected, *V* SVM models will be constructed with (*V*-1)/*V* proportion of the rows being used in each model. The remaining rows are then used to measure the accuracy of the model. The final model is built using all data rows. This method has the advantage of using all data rows in the final model, but the validation is performed in separately constructed models so there is some possibility that the misclassification rate for the final model may be different than the validation models.

How to handle missing predictor values – DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Parameter optimization search control – The accuracy of an SVM model is largely dependent on the selection of the model parameters such as C, Gamma, P, etc. DTREG provides two methods for finding optimal parameter values, a **grid search** and a **pattern search**. A grid search tries values of each parameter across the specified search range using geometric steps. A pattern search (also known as a "compass search" or a "line search") starts at the center of the search range and makes trial steps in each direction for each parameter. If the fit of the model improves, the search center moves to the new point and the process is repeated. If no improvement is found, the step size is reduced and the search is tried again. The pattern search stops when the search step size is reduced to a specified tolerance.

Grid searches are computationally expensive because the model must be evaluated at many points within the grid for each parameter. For example, if a grid search is used with 10 search intervals and an RBF kernel function is used with two parameters (C and Gamma), then the model must be evaluated at 10*10 = 100 grid points. An Epsilon-SVR analysis has three parameters (C, Gamma and P) so a grid search with 10 intervals would require 10*10*10 = 1000 model evaluations. If cross-validation is used for each model evaluation, the number of actual SVM calculations would be further multiplied by the number of cross-validation folds (typically 4 to 10). For large models, this approach may be computationally infeasible.

A pattern search generally requires far fewer evaluations of the model than a grid search. Beginning at the geometric center of the search range, a pattern search makes trial steps with positive and negative step values for each parameter. If a step is found that improves the model, the center of the search is moved to that point. If no step improves the model, the step size is reduced and the process is repeated. The search terminates when the step size is reduced to a specified tolerance. The weakness of a pattern search is that it may find a local rather than global optimal point for the parameters. If the value of the model within the parameter space has ridges rather than being purely convex, the pattern search may get trapped in a local valley and miss the globally optimal point.

DTREG allows you to use both a grid search and a pattern search. When you check both boxes the grid search is performed first. Once the grid search finishes, a pattern search is performed over a narrow search range surrounding the best point found by the grid search. Hopefully, the grid search will find a region near the global optimum point and the pattern search will then find the global optimum by starting in the right region.

Do grid search for optimal parameters – If this option is selected, DTREG will perform a grid search to try to determine the optimal parameter values. For each relevant parameter, you can specify the lower and upper range to be searched. DTREG will try

values in the range using geometric steps and use cross validation to measure how well the model fits the data. SVM models are among the most accurate, but their performance is highly dependent on the parameters you specify, so a grid search is recommended. Generally, the search gets slower as the value of the *C* parameter gets larger, so it is best to restrict it to a reasonable range. For classification problems, the optimal value of *C* typically is in the range of 1 to 100. For regression problems, the optimal value of *C* may be much larger – a million or more.

The grid search **Intervals** value specifies how many values will be tried between the low and high values (including those values). The value specified in the field to the right of intervals is the **refinement** iteration value. Once DTREG has identified the best set of parameter values using the initial grid search, it will then perform smaller grid searches in the vicinity of the optimal point to further refine the optimal values. A refinement value of 1 (the default) does only the primary grid search. A value of 2 would do the grid search and then one finer-level search. You can specify large refinement values to increase the number of searches. Caution: the time required to do a grid search is proportional to the number of parameters times the number of intervals times the number of refinement steps; this can add up to a lot of time.

Do pattern search for optimal parameters – If this option is selected, DTREG will perform a pattern search to try to determine the optimal parameter values.

The pattern search **Intervals** value controls the starting step size. The first step will be set so that the number of steps required to cross the entire search range equals the specified number of intervals. The pattern search **Tolerance** value controls when the pattern search terminates. The search stops when the value of all parameters divided by the step size is less than the tolerance value.

Percent rows to use for search specifies what percent of the training rows are to be used for the search operation. Since a search operation is a very computationally expensive procedure, you can select a subset of the full training rows to use for the search.

Cross validate; folds Specifies if V-fold cross-validation is to be used by the search to calculate the optimal parameter values. If this option is selected, DTREG will perform cross validation when it is performing the search to determine the optimal parameters. If this option is not selected, DTREG searches for the optimal parameters using the error computed for the training data. For the most accurate parameter calculations it is best to use cross validation, but this will increase the time required to do the search.

Search optimization criterion – When performing a search for optimal parameters, you can select which criterion is to be used to determine the optimum function value:

- **Minimize total error** – The total misclassification error (or mean square error for regression) is minimized. This is the only available option for regression analyses.

- **Minimize weighted error** – The misclassification errors are weighted by multiplying errors by a factor to compensate for differences in the frequencies of the target categories. Misclassifications of categories with low frequencies receive more weight to help balance them compared to categories with higher frequencies.
- **Maximize AUC** – The parameter search finds the point that maximizes the area under the ROC curve (AUC). This option is only available for classification analyses where the target variable has two categories. Maximizing the AUC tends to balance the misclassifications between the classes and improves the discrimination. Note, AUC is also known as the "C-statistic".
- **Maximize sensitivity & specificity** – The parameters are optimized to produce the maximum geometric mean of sensitivity and specificity. This option is available only for classification analyses where the target variable has two categories.

Model parameters – There are a number of parameters such as *C, Nu, Gamma* that apply to the SVM model and the selected kernel function. Selecting the optimal values can significantly impact the accuracy of the model. DTREG will enable the appropriate parameter value boxes depending on the type of SVM model and kernel function that is selected. If a grid or pattern search is enabled, then additional boxes will be enabled where you can specify the lower and upper range of the search interval.

Write Support Vectors to a File – Click this button to open a dialog box where you can specify a file where the support vectors for the generated model should be written. This button is enabled only if a model has been built and support vectors have been found.

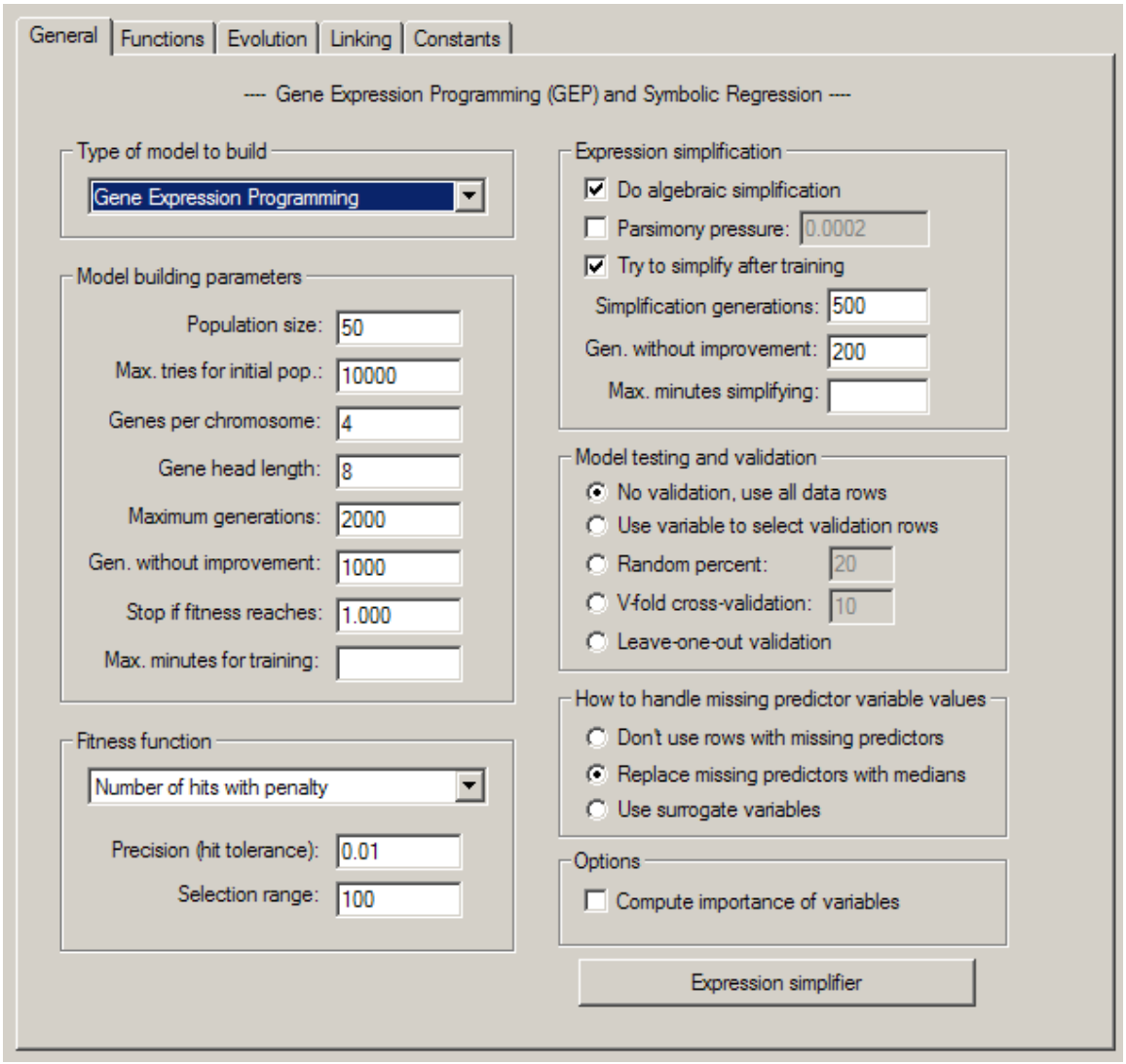
Gene Expression Programming (GEP) Property Pages

Gene Expression Programming is an algorithm for performing Symbolic Regression to try to a mathematical function that fits a set of data. Unlike traditional linear and non-linear regression, symbolic regression does not require the form of the function to be specified in advance. Using a genetic, evolution algorithm, symbolic regression finds a function to fit the data. For more detailed information about gene expression programming models, please see the chapter starting on page305.

Because of the number of parameters associated with gene expression programming, there are five property pages for GEP: General, Functions, Evolution, Linking and Constants.

GEP General Property Page

When you select the GEP General property page, you will see a screen like this:



Model Building Parameters

Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than gene expression programming, all of the other controls on this screen will be disabled.

Population size: This is the number of chromosomes in the population being evolved. Usually a population size in the range of 30 to 80 chromosomes works well.

Maximum tries for initial population: The first step in the GEP model building process is to create an initial population with a random set of functions and terminals. If the initial population contains no viable members, then a new population is tried with a different set of functions and terminals. This process is repeated until a population is found that has at least one viable member or the number of attempts specified by this parameter is reached.

Genes per chromosome: A chromosome is composed of one or more genes joined by a linking function. Usually one to ten genes per chromosome works well. More complex functions require more genes.

Gene head length: This specifies the number of symbols (variables, constants and functions) in the head section of each gene. Typically a head length in the range of two to sixteen works well. More complex functions require longer heads to allow for more variables and functions.

Maximum generations: This is the maximum number of generations that will be produced during the evolution process. There is no way to know how many generations will be required other than experimentation.

Generations without improvement: During the evolution process, DTREG notes when the model is improved because a new chromosome is found that is better than any previous one. If the number of generations specified by this parameter elapse without finding any improvement, the evolution process stops.

Stop if fitness reaches: If the fitness score of the best chromosome equals or exceeds the value of this parameter, the evolution process is stopped. The maximum possible fitness is 1.0.

Maximum minutes for training: Specify the maximum number of minutes of execution time you will allow DTREG spend on the training process. If the time limit is reached, the evolution process stops. If this field is left blank then no time limit is imposed.

Fitness Function Parameters

Fitness function: Select which function you want to be used to compute the fitness score. All fitness functions compute fitness scores that range from 0.0 to 1.0. A fitness of 0.0 means the model fits very poorly – it is worthless or not viable. A fitness score of 1.0 means the model fits the data perfectly.

Mean squared error (MSE) [classification and regression] – This is the mean value of the squared difference between the actual target value and the predicted target value. The formula is:

$$variance = \sum_{i=1}^{N} (P_i - T_i)^2$$

$$fitness = \frac{1}{1 + \frac{variance}{N}}$$

Where P_i is the predicted value for row *i* and T_i is the actual target value; *N* is the number of rows in the training data set.

Explained variance R^2 [regression] – This is the proportion of the initial variance in the training data that is explained by the GEP model.

$$\frac{initialvariance - variance}{initialvariance}$$

Where *initialvariance* is the variance for the training data set using the mean value of the target variable as the predicted value for all rows:

$$initialvariance = \sum_{i=1}^{N} (T_i - \bar{T})^2$$

Variance is computed as shown in the previous section.

Root relative squared error [regression] – This is based on the square root of the residual variance of the fitted model divided by the initial variance. This is the recommended fitness function for regression problems.

$$fitness = \frac{1}{1 + \sqrt{\frac{variance}{initialvariance}}}$$

Number of hits and **Number of hits with precision** [classification and regression] – This is the proportion of training rows whose predicted values fall within a specified

tolerance of the actual target value. For regression problems, Number of hits and Number of hits with precision are calculated the same way:

$$fitness = \left(\frac{1}{N}\right)\sum_{i=1}^{N} \text{if } (|P_i - T_i| \leq precision) \text{ then 1 else 0}$$

Where *precision* is the "Precision (hit tolerance)" parameter on the property page.

For classification problems with a categorical target variable, the Number of hits fitness function is the proportion of the cases that have the correct predicted target value after rounding from the predicted numeric value to the closest category. Number of hits with precision is computed the same for classification as for regression.

Number of hits with penalty [classification] – This fitness function measures the number of correct classifications and penalizes the situation where there are no correct classifications for some target categories. Experiments have shown this fitness function to be highly effective; it is recommended for classification problems.

This fitness function it is based on the true positive (TP), true negative (TN), false positive (FP) and false negative (FN) counts. If a predicted value is 1 (true) and the actual class is also 1, then a TP prediction is counted. Similarly true negative (TN) predictions occur when both classes are 0. False positive and false negative predictions occur as shown in the following table:

Actual class	*Predicted class*	
	True	False
True	**TP**	**FN**
False	**FP**	**TN**

With *TP*, *TN*, *FP* and *FN* being the sum of the counts for the training data, the fitness is calculated as:

$$fitness = \text{if } (TP = 0 \text{ or } TN = 0) \text{ then 0 else } \frac{TP + TN}{N}$$

Where *N* is the total number of training cases and is equal to *TP*+*TN*+*FP*+*FN*. So if there are some correctly classified positive and negative cases the fitness is the proportion of correctly classified cases, but if there are no correct classifications for either the positive or negative cases, then the fitness is zero (i.e., the expression is unviable).

Sensitivity and specificity [classification] – In a medical context, an ideal diagnostic test would identify all patients with a suspected disease, and it would not falsely identify anyone who did not have the disease. Thus there are two types of errors: (1) failing to

identify someone with the disease and (2) incorrectly identifying someone who does not have the disease. The *sensitivity* of a test is the proportion of the people with the disease who are identified by the test. The *specificity* of the test is the proportion of the people who do not have the disease who are correctly identified as being disease-free by the test. Ideally, sensitivity and specificity would both be 1.0.

The Sensitivity and Specificity fitness function computes the fitness by multiplying the sensitivity and specificity values. This fitness function is a good choice for data with highly unbalanced distributions of target categories. Using the definitions of *TP*, *TN*, *FP* and *FN* given above, this fitness function is calculated as:

$$sensitivity = \frac{TP}{TP + FN}$$

$$specificity = \frac{TN}{TN + FP}$$

$$fitness = sensitivty \cdot specificity$$

Absolute selection range [classification and regression] – This is computed by:

$$error = |P_i - T_i|$$

$$fitness = \left(\frac{1}{N \cdot R}\right) \sum_{i=1}^{N} \text{if}(error \leq precision) \text{ then } R \text{ else}(R - error\ )$$

Where *precision* is the "Precision (hit tolerance)" parameter on the property page, and *R* is the "Selection range" parameter.

Relative selection range [classification and regression] – This is computed by:

$$error = |P_i - T_i|$$

$$fitness = \left(\frac{1}{N \cdot R}\right) \sum_{i=1}^{N} \text{if } (error \leq precision) \text{ then } R \text{ else } \left(R - 100 \cdot \frac{error}{|\mathrm{T_i}|}\right)$$

Precision and Selection Range: These two parameters are used for the fitness functions Number of Hits with Precision and Absolute Selection Range.

Expression Simplification Parameters

Do algebraic simplification: If this box is checked, DTREG will perform automatic simplification of the final expression. See page 319 for additional information about algebraic simplification.

Parsimony pressure: This parameter gives a preference to simpler expressions over more complex expressions during the evolutionary process. While simpler expressions are desirable, using parsimony pressure sometimes hinders the evolution process so that the best possible expression is not found. If you use this feature, it is best to also create a model with parsimony pressure turned off and then compare the overall quality of the fit. When parsimony pressure is used, the fitness value computed for a function is modified so that the complexity of the expression affects the fitness as follows:

$$smax = NumGenes \cdot (HeadLen + TailLen + 1) - 1$$

$$smin = 2 \cdot NumGenes - 1$$

$$fitness' = fitness \cdot \frac{1 + PP \cdot (smax - complexity)}{smax - smin}$$

Where *fitness'* is the modified fitness score, fitness is the original fitness score, *PP* is the parsimony pressure value, *NumGenes* is the number of genes in the chromosome, *HeadLen* and *TailLen* are the length of the head and tail sections of genes, and *complexity* is a count of the number of symbols in the function. See page 318 for additional information about parsimony pressure.

Try to simplify after training: If you check this box, DTREG will perform additional evolution steps after the primary training in an attempt to find a simpler function that fits the data as well or better than the function found during the primary training. During the simplification process, a simpler function will be selected over a more complex one that has the same fitness. However, quality of fit still takes priority, so if DTREG discovers a function that provides higher fitness than the function used during the primary training it will adopt that function even if it is more complex than the original function. It is recommended that this option be used, because it never results in a loss of accuracy, and it may discover a more accurate and simpler function.

Simplification generations: This is the maximum number of generations that will be evolved during the simplification process.

Generations without improvement: If the specified number of generations are evolved with no improvement in the fitness or simplicity of the function, the simplification process will stop.

Maximum minutes simplifying: Specify the maximum number of minutes of execution time that you will allow DTREG to spend on the simplification process. If this field is left blank then no time limit is imposed.

Model Testing and Validation Parameters

No validation: The mode is trained but no validation is performed. This is fast, but it is not recommended because there is no way to measure how well the model is likely to generalize to new data.

Random percent: If this option is selected, a random percentage of the rows are held out during the validation training, then those held-out rows are run through the model and their error is reported as the validation error.

Use variable to select validation rows – If you check this button, DTREG uses the hold-out control variable specified on the Validation Property Page (see page 45) to control which variables are held-out during model training and used to test the model. Rows with the specified category on the control variable are held out and used for testing; rows with any other category are used to train the model. This option is enabled only if a hold-out variable has been selected on the Validation Property Page.

V-fold cross validation: If this option is selected, *V* GEP models will be constructed with (*V*-1)/*V* proportion of the rows being used in each model. The remaining rows are then used to measure the accuracy of the model. The final model is built using all data rows. This method has the advantage of using all data rows in the final model, but the validation is performed in separately constructed models so there is some possibility that the misclassification rate for the final model may be different than the validation models.

Leave-one-out validation: This option is like the V-fold cross validation option except that *N* models are built where *N* is the number of rows in the training data set. (*N*-1) rows are used to build each model and the N^{th} remaining row is used to test the model. Because so many models are built, this option is appropriate only for small training sets.

Missing Value Parameters

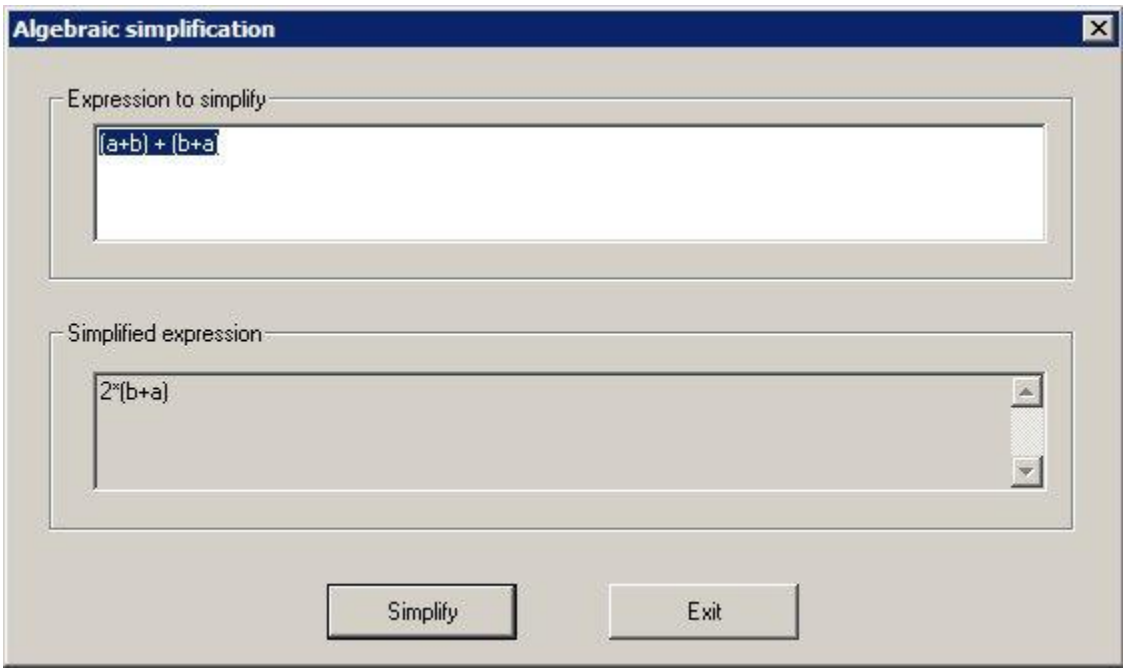
How to handle missing predictor values: DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Miscellaneous Options

Compute importance of variables: If this box is checked, DTREG will compute and display the relative importance of each predictor variable. The calculation is performed using sensitivity analysis where the values of each variable are randomized and the effect on the quality of the model is measured.

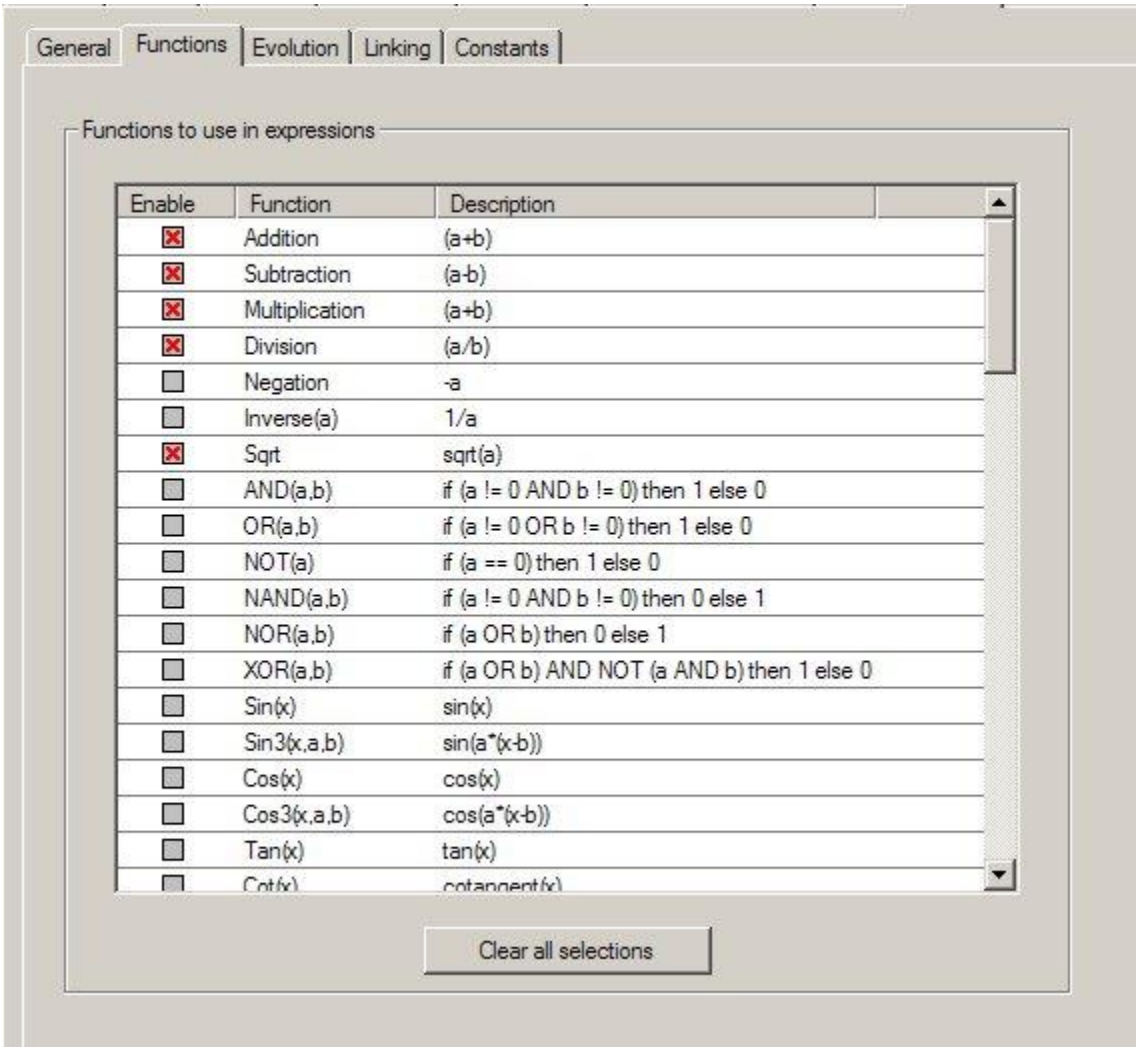
Expression Simplifier

If you click the "Expression simplifier" button, DTREG will display a screen where you can experiment with its automatic algebraic simplification.



Enter an algebraic expression in the upper window, and click the Simplify button to see how DTREG can simplify the expression. In this screen you can use any variable name that begins with a letter; it is not necessary for the variable to be in the data set for the model. Note that if you check the option box "Do algebraic simplification" DTREG will perform automatic simplification of the functions it generates, so it is not necessary to manually simplify them.

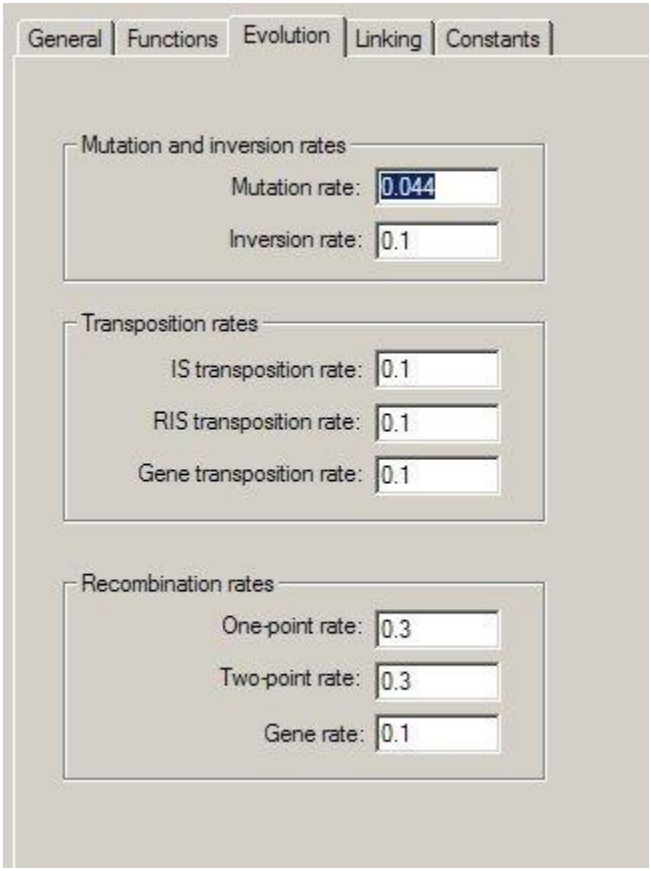
GEP Functions Property Page



The Functions property page is used to select which functions will be tried in the model during the evolution process. Check boxes next to the functions you want to include. Note that DTREG provides both mathematical functions (+,-,*,/, sqrt, sin, etc.) and logical functions (AND, OR, NOT, etc.).

GEP Evolution Property Page

The Evolution property page contains parameters that control evolution operations such as mutation and recombination.



Mutation and inversion rates

Mutation rate – This is the probability that a symbol (variable, function or constant) in a gene will be mutated during each generation. Symbols in the head of a gene can be replaced by variables, functions and constants (if constants are used); symbols in the tail of the gene can be replaced only by variables and constants.

Inversion rate – This is the probability that the inversion operation will be performed on a chromosome. Inversion selects a random starting symbol in a gene and a random ending symbol. All of the symbols between the starting and ending points are then reversed in order.

Transposition rates

Transposition is the process of moving a sequence of symbols in a gene from one location to another. Some types of transposition allow sequences of symbols to be moved from one gene to another gene in the same chromosome.

IS transposition rate – This is probability that *Insertion Sequence Transposition* will be applied to a chromosome. Source and destination genes are selected in the chromosome; the source gene may be the same as the destination. Starting and ending symbol positions are selected in the source gene. The starting point may be in the head or tail section of the gene, and the selected section may span the head and tail. The destination, insertion point is selected in the head of the destination gene, but it is not allowed to be the first (root) symbol of the gene, and the selection length is restricted so that it will remain entirely in the head of the destination gene. The selected sequence of symbols is then inserted into the destination gene, and any symbols following the insertion point that are in the head of the destination gene are moved right to make room of the insertion. Symbols shifted out of the head by the insertion are discarded.

RIS transposition rate – This is probability that *Root Insertion Sequence Transposition* will be applied to a chromosome. A random scan point is selected in the head of a gene beyond the first (root) symbol of the gene. The process then scans forward looking for a function symbol. If no function is found, RIS transposition does nothing. If a function is found, a random ending point is selected beyond the starting point but in the head of the gene. The symbols in the selected range are then inserted at the beginning (root) of the gene. Symbols pushed out of the head by the insertion are discarded.

Gene transposition rate – This is probability that *Gene Transposition* will be applied to a chromosome. A random gene that is not the first gene of a chromosome is selected. This gene is then inserted as the first gene of the chromosome. The gene being inserted is removed from its original location, and the genes preceding it are moved over to make room for the insertion at the head of the chromosome. So the length of the chromosome is not changed.

Recombination rates

During *Recombination*, two chromosomes are randomly selected, and genetic material is exchanged between them to produce two new chromosomes. It is analogous to the process that occurs when two individuals are bred, and the offspring share genetic material from both parents.

One-point rate – This is probability that *one-point recombination* will be applied to a chromosome. Two parent chromosomes are randomly selected and paired together. A split point is selected anywhere in the chromosomes (any gene and any position in a gene – head or tail). The symbols in the parents from the split point to the ends of the

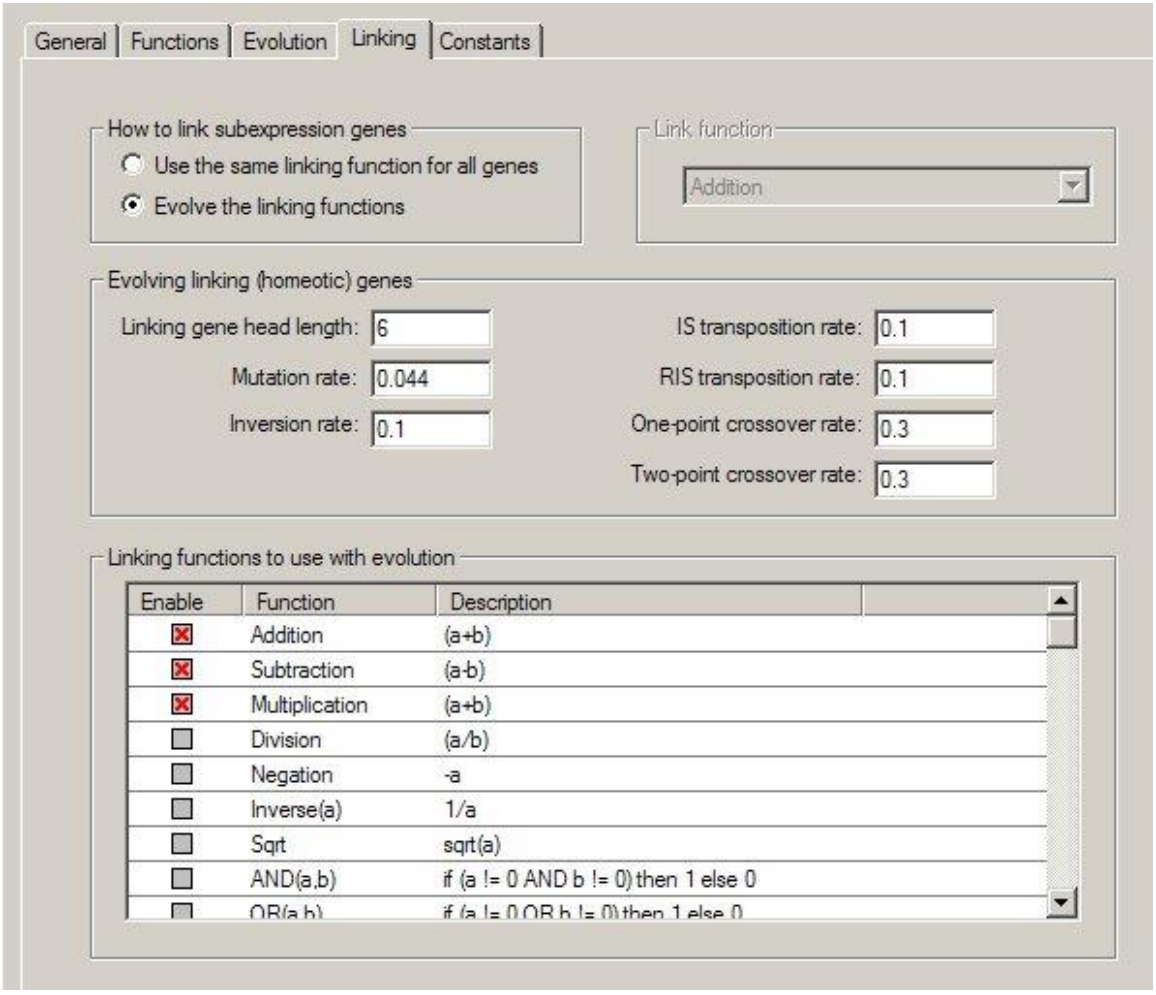
chromosomes are then exchanged between the parents. Note that all chromosomes have the same number of symbols, so no symbols are lost during the exchange.

Two-point rate – This is probability that *two-point recombination* will be applied to a chromosome. Two parent chromosomes are randomly selected and paired together. Two recombination points are selected in the chromosomes. The symbols between the starting and ending recombination points are then exchanged between the parent genes.

Gene recombination rate – This is probability that *gene recombination* will be applied to a chromosome. Two parent chromosomes are randomly selected and paired together. A random gene is selected and exchanged between the parent chromosomes.

GEP Linking Property Page

The Linking property page contains parameters that control how genes in a chromosome are linked together. See page 312 for additional information about linking.



If a chromosome has more than one gene, the expressions described by the genes must be linked together to form the full function representing the chromosome. This linking is done using a *linking function* (or operator) that has two or more arguments such as addition, logical AND and OR.

DTREG allows you to use either a *static linking function* or *homeotic genes* which are genes with linking functions that evolve. If homeotic genes are used, then different functions may be used to link different genes, and these functions are selected through evolution. See page 312 for additional information about linking functions.

How to link subexpression genes

Use the same linking function for all genes – If this option is selected, then you must select which linking function is to be used, and that function will be used to link all genes. The linking function is selected from the "Link function" dropdown list shown on the right of the screen. That list will be enabled when this option is selected.

Evolve the linking functions – If this option is selected, then a homeotic gene will be added to the chromosomes to represent the linking functions. The homeotic gene (and the linking functions it represents) will evolve in a similar manner to other genes.

Evolving (linking) homeotic genes

These parameters are only enabled if you select evolving linking functions.

Linking gene head length – This is the number of symbols in the head of the homeotic, linking gene.

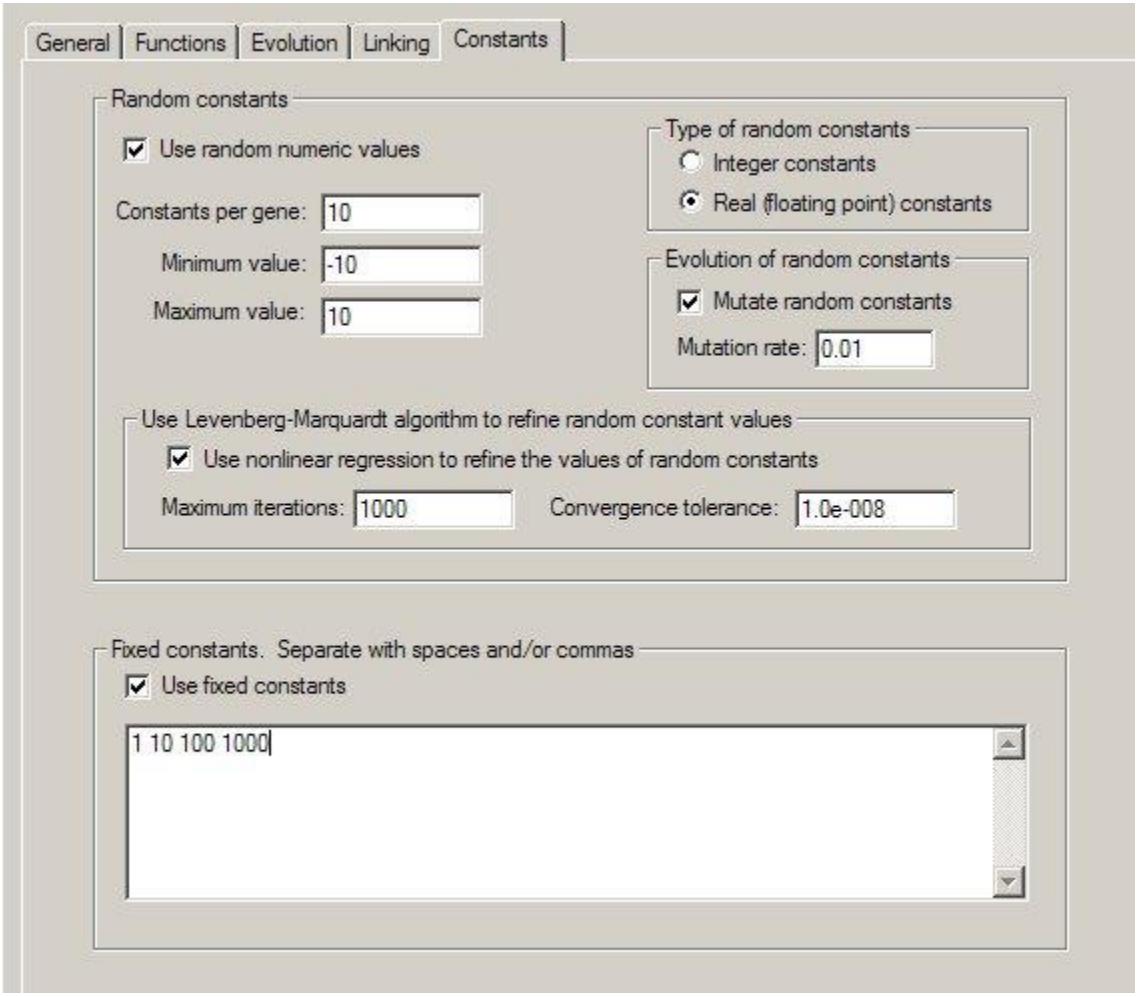
Mutation, inversion, transposition and crossover rates – These are the rates for mutation, inversion, transposition and crossover for the homeotic gene. The operations are performed in the same manner as the primary genes for the chromosome. See the descriptions above for the actions performed by each of these operations.

Linking functions to use with evolution

Check the boxes next to the functions that you want to allow to be considered as linking functions.

GEP Constants Property Page

The Constants property page contains parameters that control whether constants are to be used in the GEP functions.



The DTREG implementation of gene expression programming allows the creation of expressions with no explicit constants, with a fixed set of user-specified constants and with random constants that mutate during the evolutionary process.

Note that even if explicit constants are not enabled, constants may be developed implicitly during the evolutionary process. For example, a function may be evolved such as:

$$target = (x + x + x) + y\sqrt{y}$$

Which, of course, simplifies to

$$target = 3x + y^{1.5}$$

Constants per gene – This parameter specifies how many random constant values are to be included in each gene. Note that the inclusion of the constant values in a gene does not necessarily mean that they will actually be used in the coding region of a gene where they would be part of the expression. Just as with variables and functions, constants are moved into the functional (coding) part of a gene through mutation and selection.

Minimum value – This is the minimum value for randomly generated constants.

Maximum value – This is the maximum value for randomly generated constants.

Type of random constants – Select whether you want to generate integer or real random constants.

Mutate random constants – Check this box and specify a mutation rate if you want the values of the random constants to mutate during the evolutionary process.

Use nonlinear regression to refine the values of random constants – If this option is enabled, DTREG uses a sophisticated nonlinear regression algorithm to refine the values of the random constants. This optimization is performed after evolution has developed the functional form and linking and simplification have been performed. DTREG uses a model/trust-region technique along with an adaptive choice of the model Hessian. The algorithm is essentially a combination of Gauss-Newton and Levenberg-Marquardt methods; however, the adaptive algorithm often works much better than either of these methods alone.

Maximum iterations – This parameter controls the maximum iterations that will be performed by the nonlinear regression algorithm as it refines the constants.

Convergence tolerance – This parameter specifies an accuracy goal used by the nonlinear regression algorithm as it refines the constants.

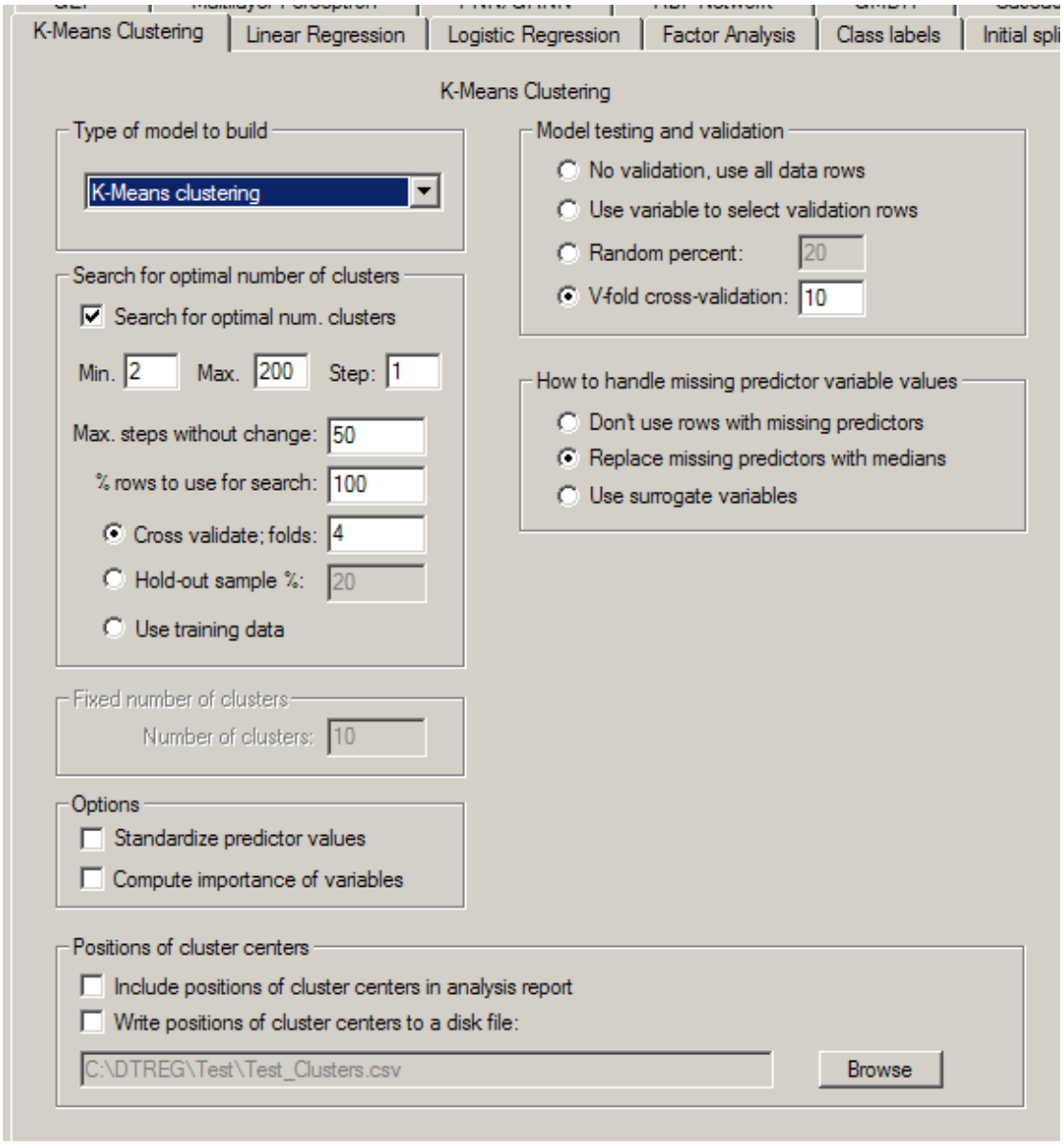
Use fixed constants – If you wish to specify a fixed, non-evolving set of constants to be considered in the functions, check this box and list the constants in the screen below. Separate the constants by spaces or put them on separate lines.

K-Means Clustering Property page

Developed between 1975 and 1977 by J. A. Hartigan and M. A. Wong (Hartigan and Wong, 1979), K-Means clustering is one of the oldest predictive modeling methods. K-Means Clustering is a relatively fast modeling method, but it is also among the least accurate models that DTREG offers.

For additional information about K-Means clustering, please see the chapter starting on page 321.

When you select the K-Means clustering property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than K-Means Clustering, all of the other controls on this screen will be disabled.

Search for optimal number of clusters: Check this box to cause DTREG to try building models with a varying number of clusters. If you don't check this box, then specify a fixed number of clusters in the "Number of clusters" field below.

Min, Max, Step: Specify the minimum number of clusters to try, the maximum and the number of clusters to add between each step.

Maximum steps without change: As DTREG builds models with progressively larger numbers of clusters it checks the validated accuracy of each model. If it tries the number of models specified by this parameter without improving the accuracy, the search stops.

% rows to use for search: If you wish, you can restrict the number of data rows used during the search process. Once the optimal size is found, the final model will be built using all data rows.

Cross validate folds, Hold out sample %, Use training data: These parameters control the method used to evaluate the accuracy of the model for each step. You can use cross-validation and specify the number of validation folds, you can hold out a certain percentage of the data and use the validation, or you can simply just the same data to train and test the model. It is highly recommended that you use either cross-validation or a hold-out sample.

Fixed number of clusters: If you do not check the box "Search for the optimal number of clusters", then specify the number of clusters to use for the model here.

Standardize predictor values: If this box is checked, the values of continuous predictor variables are standardized by subtracting the mean and dividing by the standard deviation. Selecting this option often reduces the quality of the model, so always try building a model with this option turned off.

Compute importance of variables: If this option is selected, DTREG will provide an estimate of the relative importance of each predictor variable. This is usually a fairly fast procedure unless there are a very large number of predictor variables and a lot of data.

Include position of cluster centers in analysis report: Check this option to cause DTREG to report the position of cluster centers in the analysis report.

Write positions of cluster centers to a disk file: Check this option to cause DTREG to write information about the positions of cluster centers to the specified disk file.

Testing and validation parameters – Select the type of validation you want DTREG to use to test the model. V-fold cross-validation is recommended.

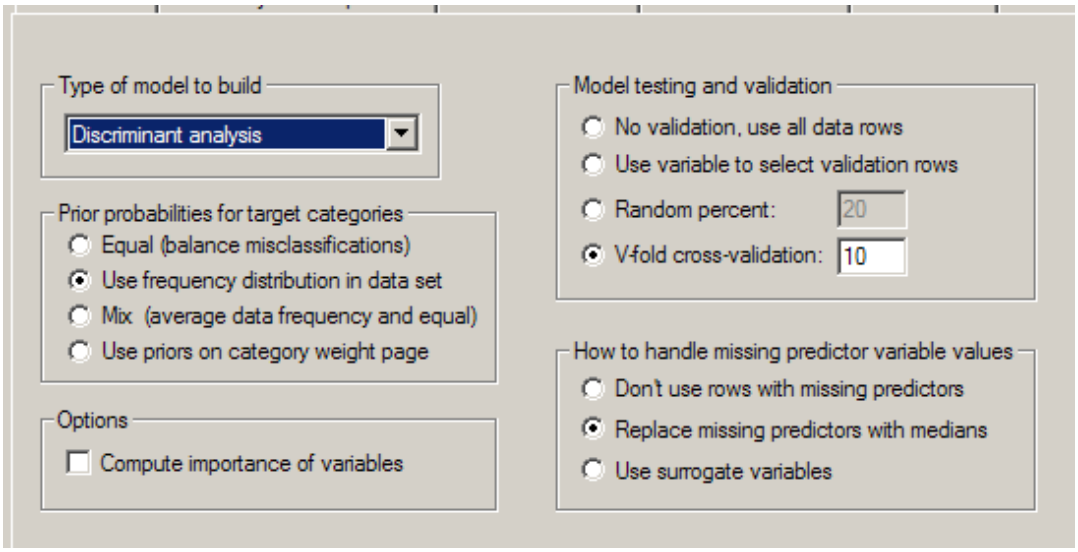
Missing value controls – DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Discriminant Analysis Property page

Discriminant analysis is a classical method of classification that usually is able to build models that rival the more sophisticated models for accuracy.

For additional information about discriminant analysis, please see the chapter starting on page 305.

When you select the discriminant analysis property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than discriminant analysis, all of the other controls on this screen will be disabled.

Prior probabilities for target categories: Select the assumed prior probability distribution for the target variable categories. Traditionally (and in most benchmarks) the distribution in the training data set is used. If you wish to specify a custom set of prior probabilities, select the option "Use priors on category weight page", and set the values of the priors on the Category weight property page (see page 128).

Compute importance of variables: If this option is selected, DTREG will provide an estimate of the relative importance of each predictor variable. This is usually a fairly fast procedure unless there are a very large number of predictor variables and a lot of data.

Model testing and validation: Select which procedure (if any) is to be used to validate the model. The recommended method is 10-fold cross validation which builds 10 models using 90% of the data for each model and 10% for validation.

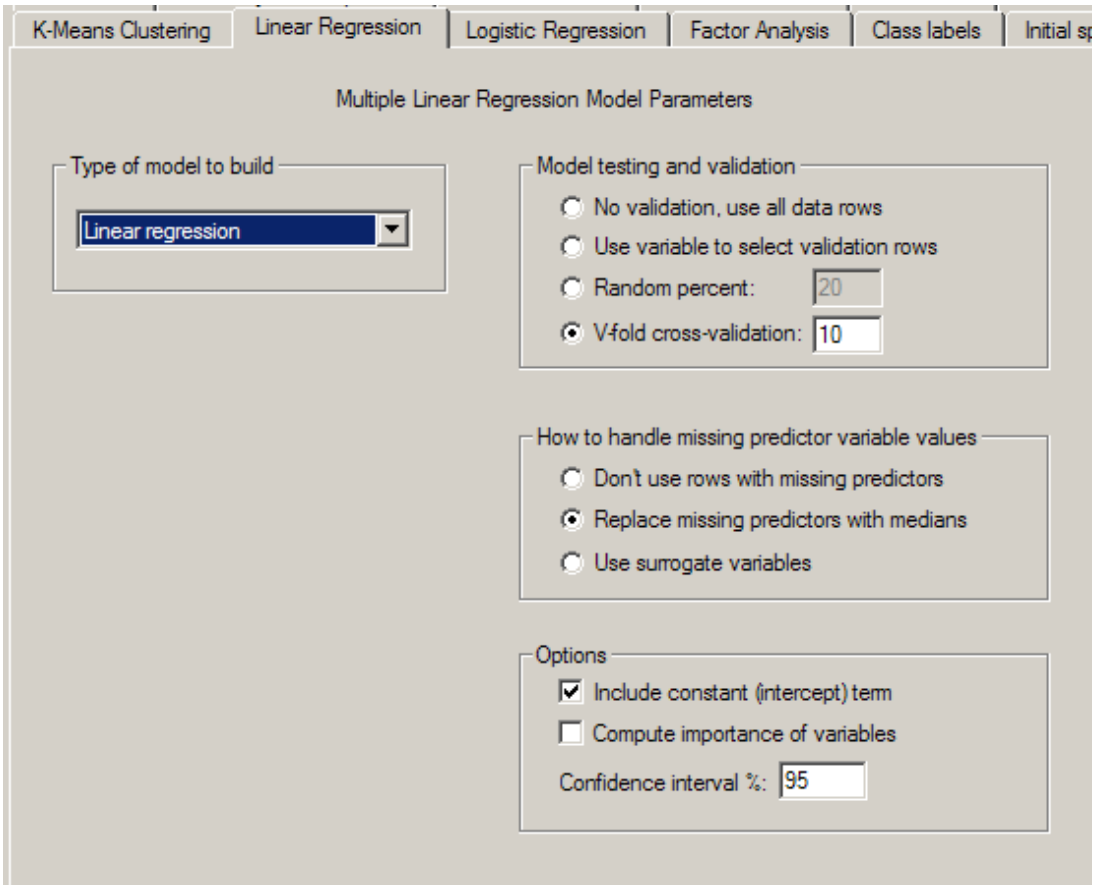
How to handle missing predictor variable values: DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those

rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Linear Regression Property Page

Linear regression is one of the most widely used modeling methods. For more technical information about linear regression, please see the chapter starting on page 331.

When you select the linear regression property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than linear regression, all of the other controls on this screen will be disabled.

Testing and validation parameters – Select the type of validation you want DTREG to use to test the model. V-fold cross-validation is recommended.

Missing value controls – DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Include constant (intercept) term – If you select this option, DTREG includes a constant value (β_0) in the model. If you don't select this option, then there is no constant, and the model consists of just computed coefficients multiplied by the predictor variables.

Compute importance of variables – If you select this option, DTREG will compute the relative importance of each predictor variable and display it in the analysis report.

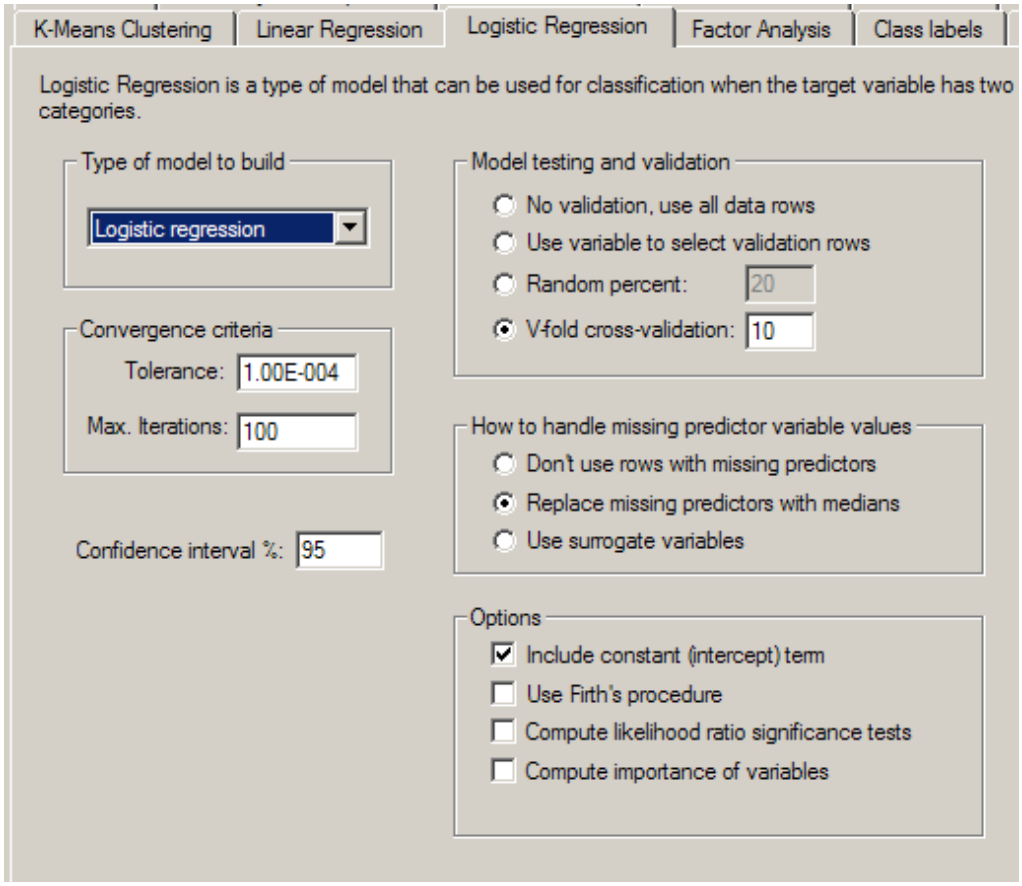
Confidence interval percent – In addition to computing the maximum likelihood values of the parameters, confidence intervals also are calculated. You can specify the percent confidence to be computed. For example, specifying a value of 95 for this parameter will cause the confidence intervals to span a range that is 95% likely to cover the true values.

Logistic Regression Property Page

Logistic regression is a popular method for modeling data that has a categorical target variable with two categories.

For more technical information about logistic regression, please see the chapter starting on page 337.

When you select the logistic regression property page, you will see a screen like this:



Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than logistic regression, all of the other controls on this screen will be disabled.

Convergence criteria – An iterative (Newton-Raphson) algorithm is used to compute the maximum likelihood values of the logistic regression parameters. Two parameters are available to control the algorithm. The **tolerance factor** is used to decide when the parameter values have converged to acceptable tolerance. If the absolute value of the maximum change of any parameter during the last iteration is less than the convergence

tolerance, then convergence is achieved. The **maximum iteration** parameter specifies a safety stop for the algorithm if convergence is not reached.

Confidence interval percent – In addition to computing the maximum likelihood values of the parameters, confidence intervals also are calculated. You can specify the percent confidence to be computed. For example, specifying a value of 95 for this parameter will cause the confidence intervals to span a range that is 95% likely to cover the true values.

Testing and validation parameters – Select the type of validation you want DTREG to use to test the model. V-fold cross-validation is recommended.

Missing value controls – DTREG offers three choices for dealing with predictor variables that have missing values. You can (1) exclude those rows from the analysis, (2) replace the missing values with the median or mode values for the variable, or (3) use surrogate variables. See page 357 for additional information about handling missing values and the use of surrogate variables.

Include constant (intercept) term – Check this box to include a constant term in the logistic regression equation. Generally, this box should be checked because regression models that contain a constant term are more accurate than those that don't.

Use Firth's procedure – Check this box to cause "Firth's procedure" to be used in the calculation of the maximum likelihood parameter values. Enabling Firth's procedure has three effects: (1) it may make it possible to converge to a solution when convergence cannot be achieved otherwise; (2) it reduces the bias of the computed parameters; (3) it significantly increases the computation time. Since the bias-correct parameter values computed using Firth's procedure may be different than those computed without Firth's procedure, be careful about comparing the parameter values with those computed by another program not using Firth's procedure. Generally, it is recommended that you do not enable Firth's procedure unless parameter convergence cannot be achieved without it.

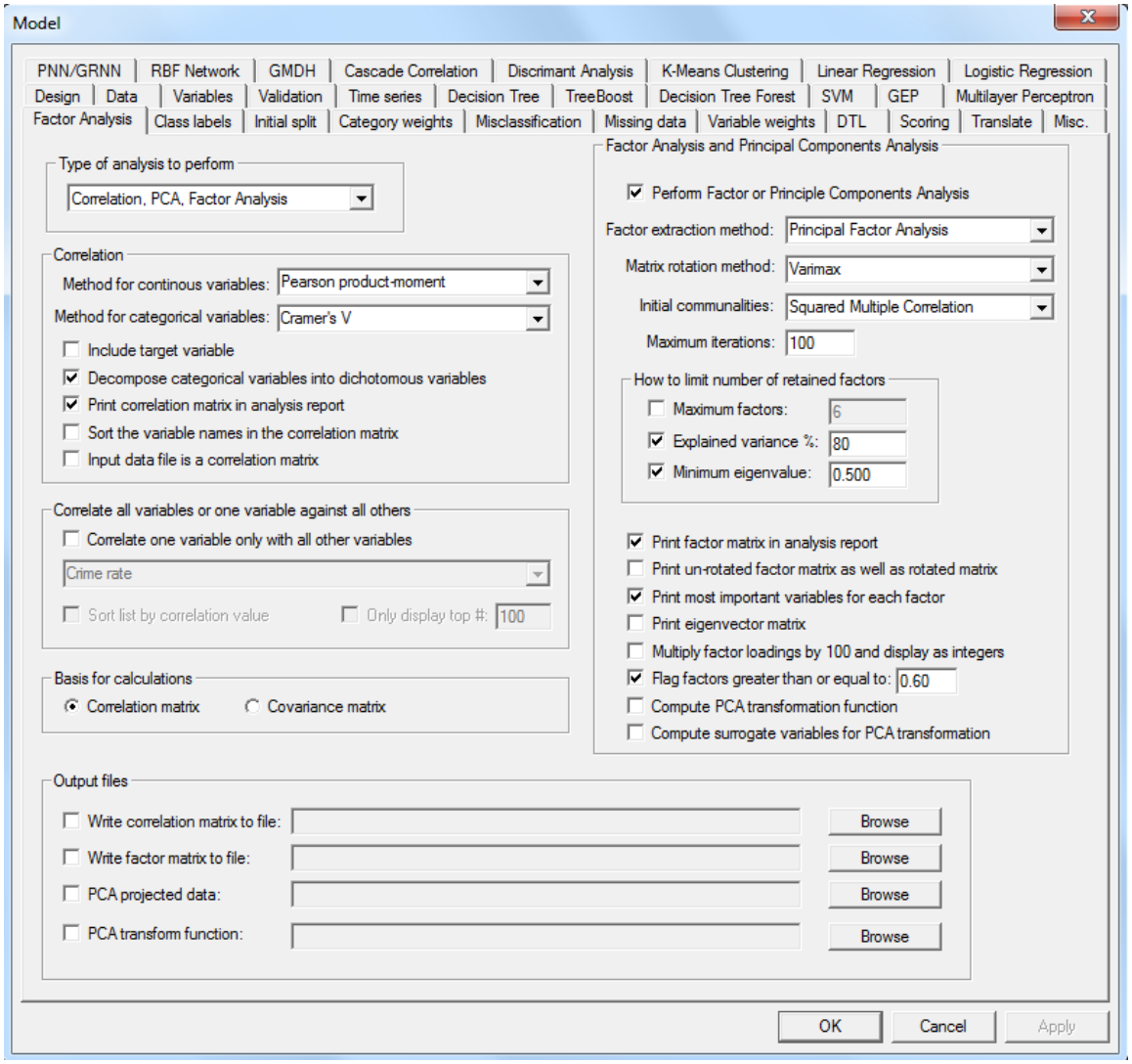
Compute likelihood ratio significance tests – Check this box to request that likelihood ratio significance tests be computed for the parameters. Likelihood ratio significance tests are a more accurate method of accessing which parameters are significant in the model than the usual Wald significance tests. The likelihood ratio significance test for an individual parameter is computed by comparing the deviance of the model including the parameter with the deviance excluding the parameter. Since the model must be recomputed with each parameter excluded, the computation time increases in direct proportion to the number of predictor variables. In the case of a predictor variable with multiple categories, the likelihood ratio is computed with the predictor included and removed rather than testing each possible category of the predictor.

Compute importance of variables – If you select this option, DTREG will compute the relative importance of each predictor variable and display it in the analysis report.

Correlation, Factor Analysis, and Principal Components Property Page

Correlation, Factor Analysis and Principal Components Analysis are exploratory analysis procedures that provide useful information about the relationship between variables. For more technical information about these procedures, please see the chapter starting on page 345.

When you select the Correlation/Factor Analysis property page, you will see a screen like this:



Type of analysis to perform – Select Correlation, PCA & Factor Analysis to enable the features on this screen.

Method for continuous variables – Two correlation methods are provided for continuous variables: (1) Pearson product moment, and (2) Spearman rank-order. Usually when the term "correlation" is used without qualification, it is referring to Pearson product moment correlation. Spearman rank-order replaces the values of the variables by their rank (sorted position order) and performs the correlation using the rank order values. This has the advantage of allowing Spearman correlation to work better with nonlinear correlations. See page 345 for more information about the types of correlation.

Method for categorical variables – Most correlation programs provide procedures only for computing correlation between continuous variables. Because DTREG allows both continuous and categorical variables, correlation becomes more complex. See page 345345 for information about the various type of correlation DTREG performs when the variables are categorical.

Include target variable – If this box is checked, the target variable is included in the analysis. If the box is not checked, only predictor variables are included.

Decompose categorical variables into dichotomous variables – Specifies that multi-category categorical variables should be decomposed into individual dichotomous variables. For example, a multi-category variable such as MaritalStatus with categorical values 0 for Single, 1 for Married, and 2 for Divorced would be converted to three dichotomous variables: MaritalStatus{0}, MaritalStatus{1}, and MaritalStatus{2}. The value of MaritalStatus{0} is 1 if the value of MaritalStatus is 0, and its value is 0 if MaritalStatus is 1 or 2. Similarly, MaritalStatus{1} is 1 if MaritalStatus is 1, and its value is 0 if MaritalStatus is 0 or 2.

Print correlation matrix in the analysis report – Check this box to cause the correlation matrix to be printed. If you have many variables so the matrix would be large and you are only interested in the Factor Analysis results, you can uncheck this box.

Sort the variable names in the correlation matrix – Check this box to cause DTREG to sort the names of the variables alphabetically in the correlation matrix. If the box is not checked, the variables are listed in the order in which they occur in the data file.

Input data is a correlation matrix – Check this box if the input data file contains a correlation matrix rather than raw data to be correlated. The correlation matrix can be provided as a full matrix, for example:

```
 V1    V2    V3
1.00  0.56  0.39
0.56  1.00  0.67
0.39  0.67  1.00
```

Or as a lower-triangular matrix like this:

```
 V1    V2    V3
1.00
0.56  1.00
0.39  0.67  1.00
```

Basis for calculations – Select whether you want principal components or factors to be based on a correlation matrix of the variables or a covariance matrix. Since the values in a covariance matrix are depending on variable units of measure (scale), it is recommended that use a correlation matrix as the basis.

Perform factor or principal components analysis – Check this box if you want to perform either factor or principal components analysis. Leave the box unchecked if you only want to compute correlations.

Factor extraction method – Select which method you want to use to extract the factors:

1. **Principal factor analysis** – Use this method to perform factor analysis where the assumption is that the correlations between variables can be explained by a set of common factors smaller in number than the number of variables. DTREG uses iterated common factor analysis to estimate the communalities.
2. **Principal components analysis** – Use this method when you want to transform a correlation matrix into a factor matrix with as many factors as variables that explain all of the variance.

Matrix rotation method – After extracting factors, you optionally can allow DTREG to rotate the factor matrix so that the factor loadings are more clearly delineated by the factors. DTREG provides two rotation methods:

1. **Varimax** – This is the most popular rotation method. It performs an orthogonal rotation of the factor matrix.
2. **Promax** – This rotation method performs oblique (non-orthogonal) rotations which allow the resulting factor axes to be correlated.

Initial communalities – When performing factor analysis, a set of initial communalities must be placed on the diagonal of the correlation matrix. The iterative factor analysis procedure will then refine these estimates. DTREG provides four ways of setting the initial communality estimates:

1. **Squared multiple correlation** – This is the squared value of the multiple correlation of each variable with all other variables. This is the default and recommended method.
2. **Maximum correlation** – The initial communality is set to the maximum correlation between the variable and any other variable.
3. **Average (SMC,MC)** – DTREG uses the average of the squared multiple correlation and the maximum correlation.
4. **1.00** – DTREG sets all initial communality values to 1.00. Note: when performing principal component analysis, communalities are always set to 1.00.

Maximum iterations – This is the maximum allowed number of iterations that the factor analysis procedure may make while refining the estimates of the communalities. It will stop before this limit if the communality values converge.

How to limit number of retained factors – Three methods are provided for determining how many significant factors will be retained. You can check some or all of the boxes. Whichever limit has the smallest value is used:

1. **Maximum factors** – Use this option if you want to explicitly specify the maximum number of factors to retain.
2. **Explained variance % --** If this option is selected, enough factors will be included so that the cumulative variance explained by them matches or exceeds the specified value.
3. **Minimum eigenvalue** – If this option is selected, a factor will be included only if its eigenvalue is at least as large as the specified value.

Print factor matrix in analysis report – Check this box if you want the factor loading matrix printed in the analysis report.

Print un-rotated factor matrix – If you request that the factor matrix be rotated (Varimax/Promax), you can check this box to have both the un-rotated and the rotated factor matrix printed in the analysis report. If the box is not checked, only the rotated factor matrix is printed.

Print most important variables for each factor – If you check this option, then DTREG will display a table showing the most important variables (i.e., variables with largest loading) for each factor. Only variables whose loadings equal or exceed the value of the parameter "Flag factors greater than equal to" (see below) are listed.

Print eigenvector matrix – If this option is selected, DTREG will include the matrix of eigenvectors in the analysis report.

Multiply factor loadings by 100 and display as integers – If you check this box, DTREG will multiply factor loadings by 100 and display them as whole integer values. So, for example, a factor loading of 0.68 would be displayed as 68. This option makes it easier to pick out significant loadings in a long list of factor loadings.

Flag factors greater than or equal to – If you check this option, then DTREG will place an asterisk to the right of any factor loading whose absolute value is equal to or greater than the specified value. This makes it easy to identify significant factor loadings.

Compute PCA transformation function – If you check this option, DTREG will compute the function to convert data values into PCA transformed scores, and it will store it with the project file so that PCA transformations can be used in input data for future models. See page 352 for additional information about using PCA transformations.

Compute surrogate variables for PCA transformation – If you check this option, DTREG will compute surrogate variable functions (see page 358) to impute the values of missing data values. The surrogate variable functions will be stored with the PCA transformation function so that they can be used for future transformations of data values into PCA transformed scores.

Write correlation matrix to file – Use this option if you want DTREG to write the correlation matrix to an external file. The file is created as a comma-separated value file.

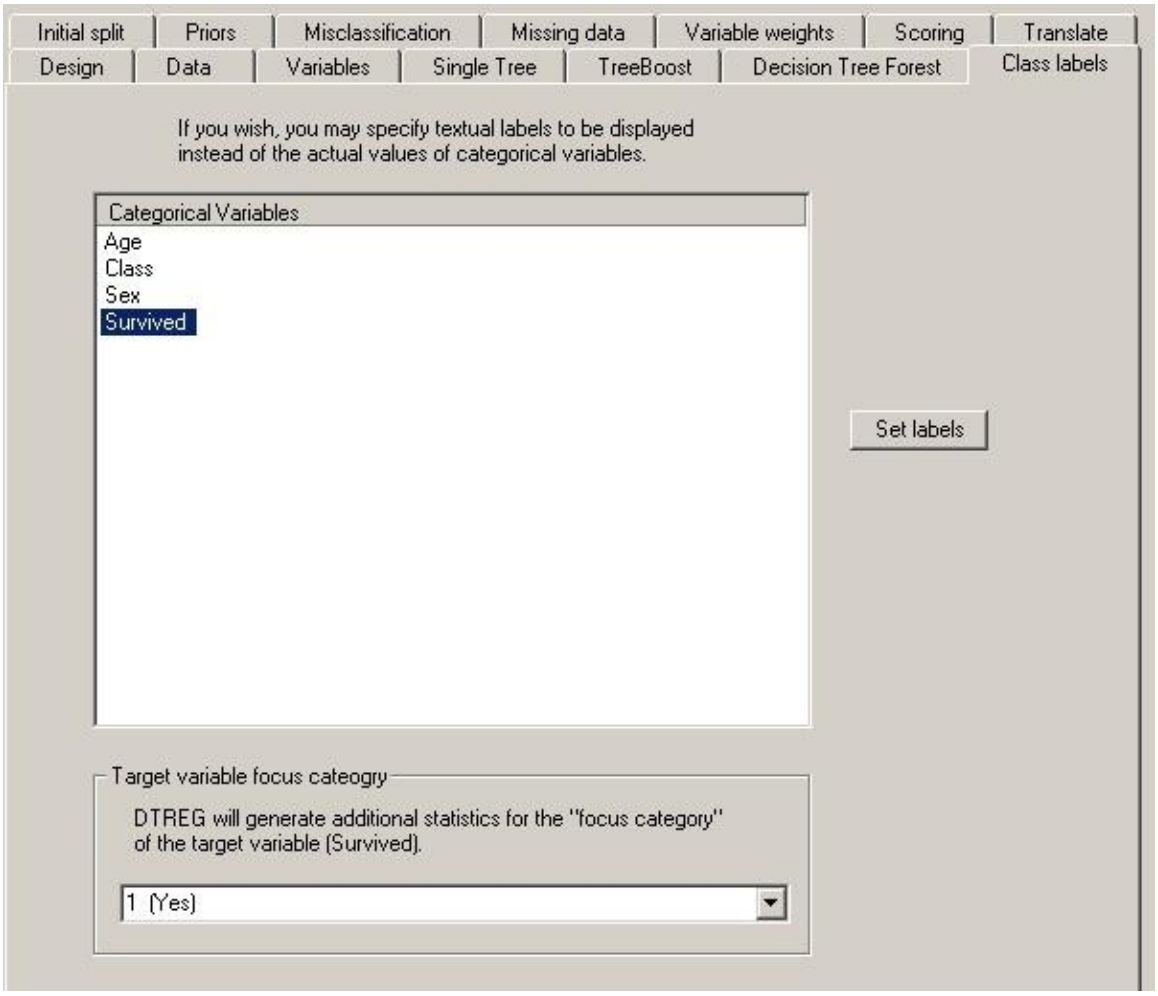
Write factor matrix to file – Use this option if you want DTREG to write the factor loading matrix to an external file. The file is created as a comma-separated value file.

PCA Projected Data – Enable this option if you want DTREG to transform the input data values into PCA scores. See page 352 for additional information about using PCA transformations. This feature is available only in the Enterprise Version of DTREG.

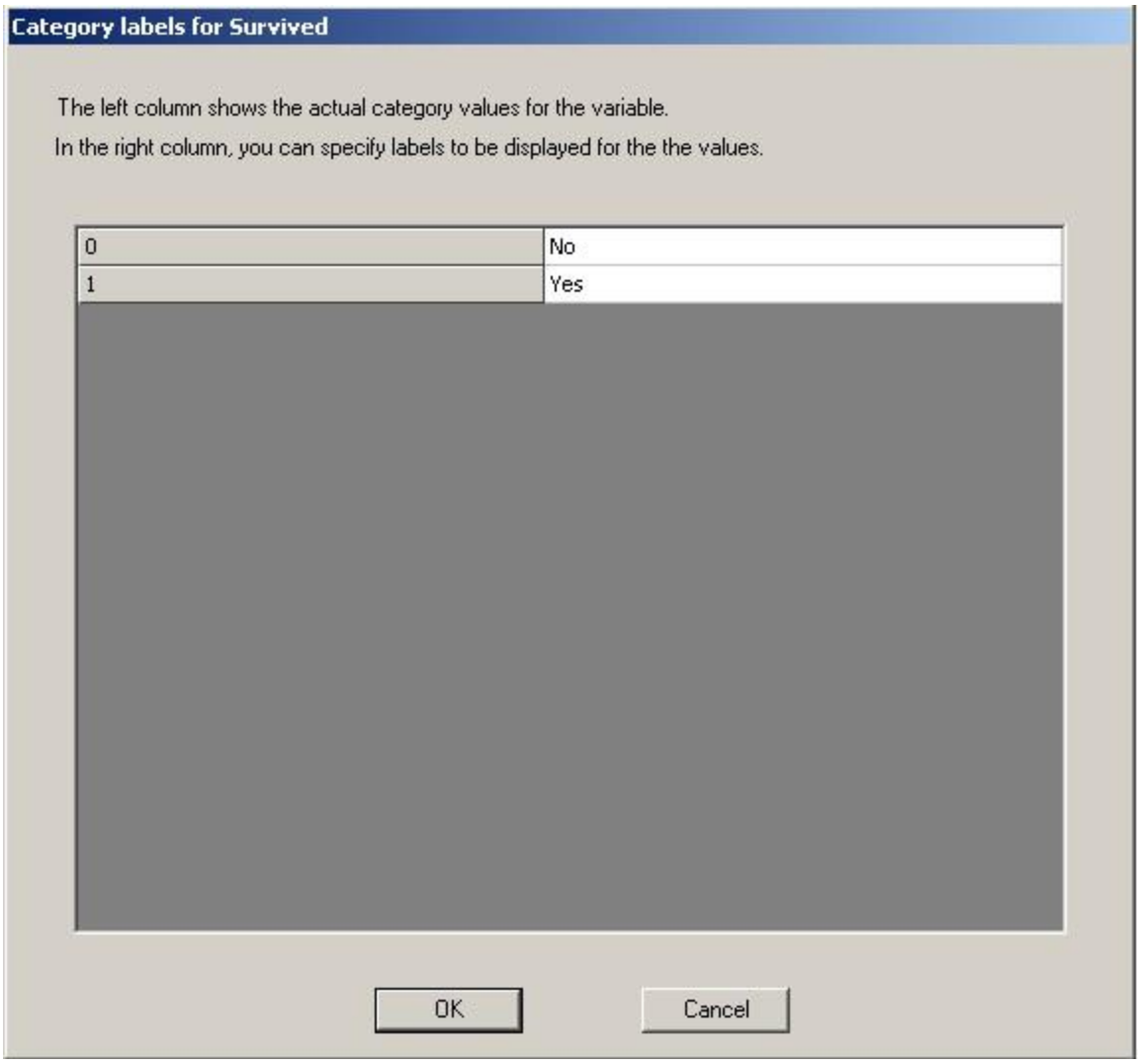
PCA Transform Function – Enable this option if you want DTREG to write the coefficients of the PCA transform functions to a data file. See page 352 for additional information about PCA transformations. This option is available only in the Enterprise Version of DTREG.

Class Labels Property Page

The Labels property page is used to specify display labels for categorical variables. Optionally, you can designate a "Focus Category" of the target variable.



The name of each categorical variable will be shown. If you wish to set display labels for the categories of a variable, select the variable and then click the "Set labels" button. A screen similar to this will be shown:



The first column displays values found in the data file for the categories of the variable. In this example, the values 1 and 2 occurred in the data file for the variable "Liver condition".

In the second column, enter text strings that you want displayed in the generated tree nodes and in the report, instead of the corresponding actual value. In this example, when the value of Liver condition is 1, the string "Normal" will be displayed, and when the value is 2, "Abnormal" will be displayed.

You can assign text labels to categorical variables that have textual values in the data file as well as those that have numeric values. For example, the values of sex might be coded as 'M' and 'F' in the data file, but by assigning labels, you could have the categories display as "Male" and "Female".

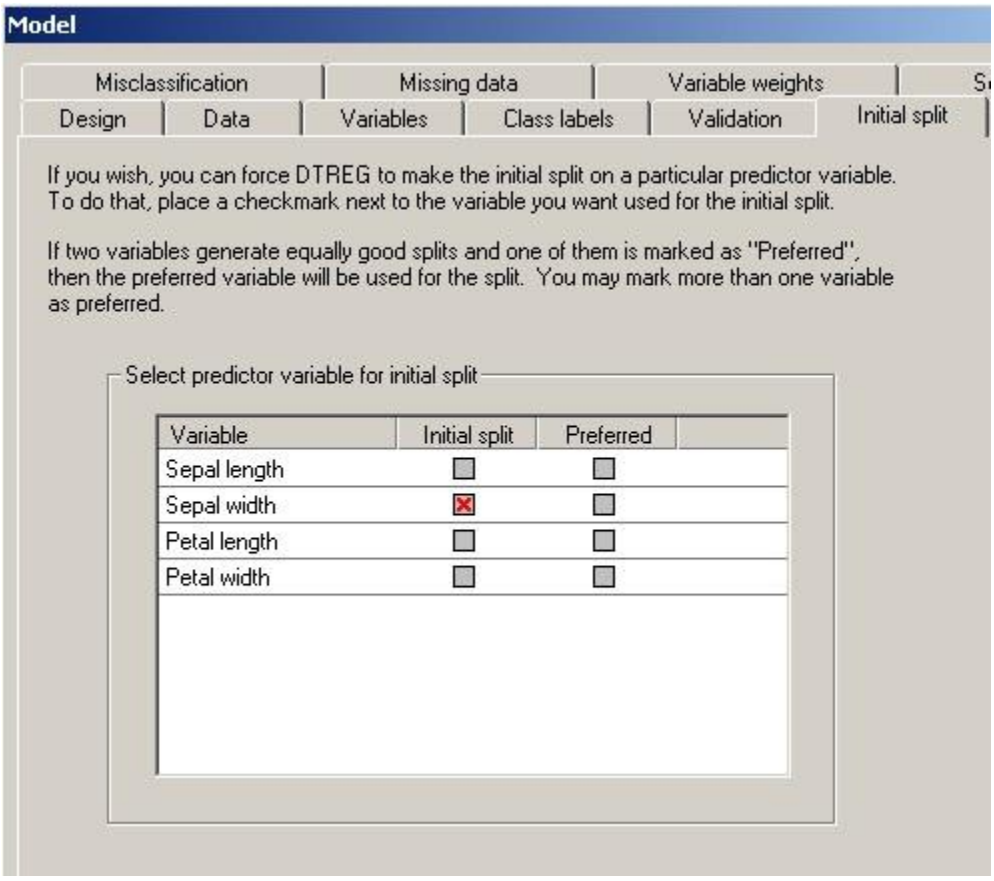
Assigned label strings are used for esthetic purposes only, and they have no effect on the generation of the model, and class labels are *not* written to the output file when data is scored.

Designating a Focus Category

In addition to setting labels for variable categories, you also can designate a "Focus Category" of the target variable. If a focus category is designated, then DTREG will collect additional information about the designated category and display them in the report and charts.

Initial Split Property Page

The Initial Split property page is used to designate a predictor variable that is to be used for the initial split and predictor variables that are to be preferred for splits.



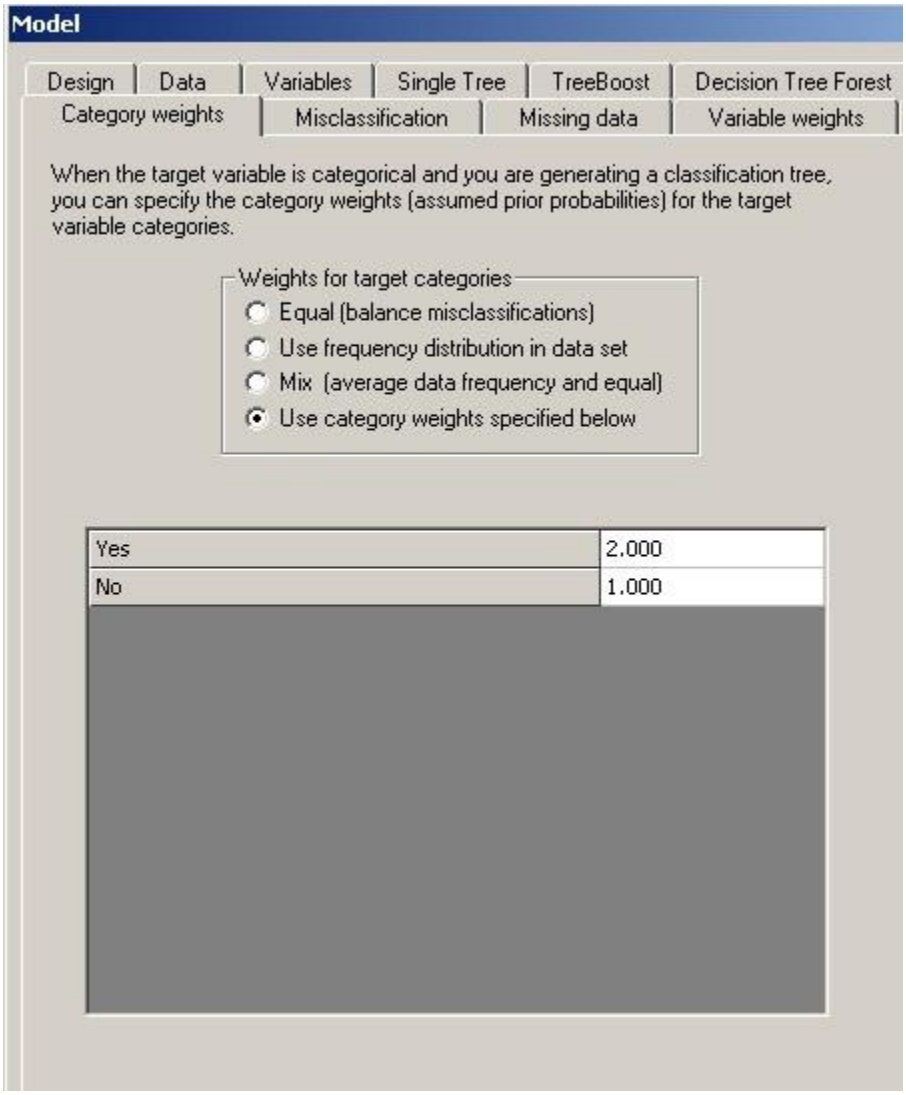
The name of each predictor variable will be shown in the list. Next to the variable names are two columns:

Initial split – If you check this box, the selected variable will be used for the initial split even if it is not the best splitting variable. This is useful if you want to force a split so as to compare the trees generated by the categories of a particular variable. For example, if sex is one of your predictor variables, you could force an initial split on it and then compare the trees generated under the male and female categories.

Preferred – If you check this box, then the selected variable will be used in preference to a non-preferred variable if they generate equally good splits. You may designate more than one variable as preferred.

Category Weights Property Page

The Category Weights property page is used to specify the weights for the categories of the target variable when you are performing a classification analysis. (Note, category weights are sometimes referred to as "priors" (*a priori*) probabilities for the categories of the target variable.)



The property page for category weights is only available when performing a classification analysis (i.e., with a categorical target variable). Category weights do not apply to regression analyses.

The category weights determine how DTREG will attempt to balance the misclassifications across the categories. The greater the weight given to a category, the fewer misclassifications it will have. If equal (balanced) category weights are selected, then DTREG will attempt to build a model so that the proportion of misclassified rows is approximately equal across the categories. If you tell DTREG to use the frequency

distribution in the data set, then categories with a higher frequency of cases will receive greater weight, and the misclassification proportions for those categories will be lower than for other, less common categories.

When category weights are set equal, the category assigned to a node is determined by the proportion of cases having each category in the node compared to the proportion in the root node. As a result, the assigned category may not be the same as the category with the most number of cases. For example, if the data from a disease treatment had 80% survival and 20% death (Live/Die target variable), then a node would be classified as death if the proportion of death cases represents more than 20% of the cases in the node – even if it is less than 50%. One surprising consequence of this is that the nodes of a binary category tree may end up with more than 50% misclassified cases.

TreeBoost and Decision Tree Forest models handle category weights by adjusting the weights of the data rows so that the sums of the weights for the rows with each target category match the proportions specified for the target category weights. For example, if equal (balanced) category weights was specified and there are twice as many rows with the "Yes" category as "No", then the weights for rows with the "No" category would be increased so that their combined weight matches the combined weight of the rows with the "Yes" category.

Category Weight Options
DTREG allows you to select several options for category weights:

Equal (balanced) – If you select this option, DTREG will attempt to build a model with roughly equal misclassification proportions for the categories. This is the default and recommended setting for category weights.

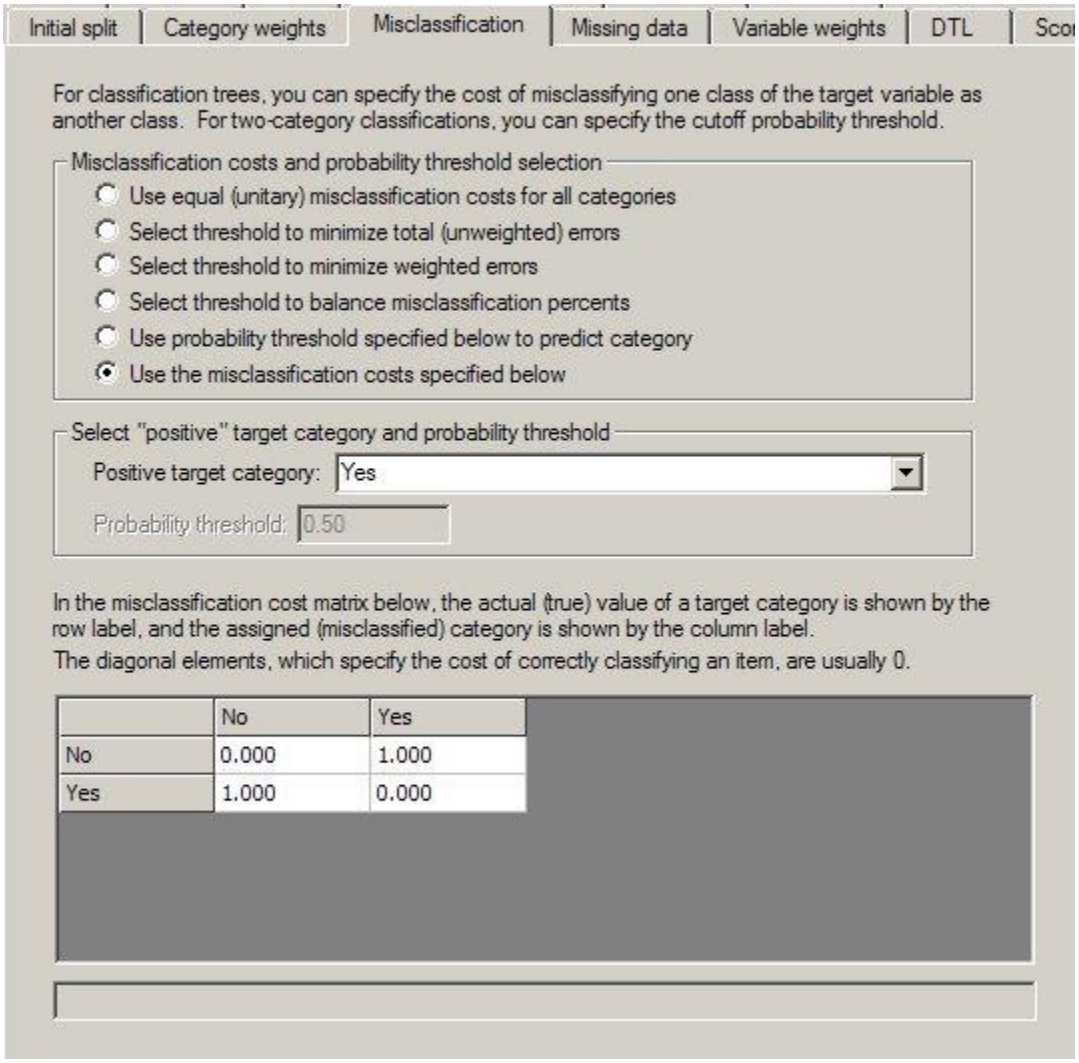
Use frequency distribution in data set – If you select this option, DTREG will compute the distribution of the categories of the target variable in the training dataset and use those proportions as the category weights. If the training sample was drawn at random from the whole population, and the category distributions are reflective of the whole population, then this is a good option to use.

Mix (average data frequency and equal) – If you select this option, DTREG sets the category weights to an average of the equal proportions and the data frequency proportions.

Use category weights specified below – If you select this option, a matrix will be displayed in the lower portion of the screen where you can enter custom category weights. Each category of the target variable will be displayed in the first column. You can enter weight values in the second column. At the beginning of an analysis, DTREG scales the category weights so their sum is 1.0; hence, only the relative values specified for each category matter.

Misclassification Cost Property Page

The Misclassification Cost property page is used to specify how much weight (cost) to give to misclassifications of categories of the target variable. It is only available when generating classification trees with categorical target variables.



In some cases, it may be more costly to misclassify some categories of the target variable than others. For example, consider a decision tree that will be used to diagnose heart attacks in patients arriving at an emergency room. Assume the target variable (Diagnosis) has several categories including heart attack, indigestion, pneumonia, bruised rib and several other possible causes of chest pain. When creating the tree, the researcher might want to assign a higher misclassification cost value to the heart attack category than the other categories, because misclassifying a heart attack is much more serious than misclassifying indigestion.

Misclassification cost and probability threshold options
You have several choices for assigning misclassification costs or selecting probability thresholds:

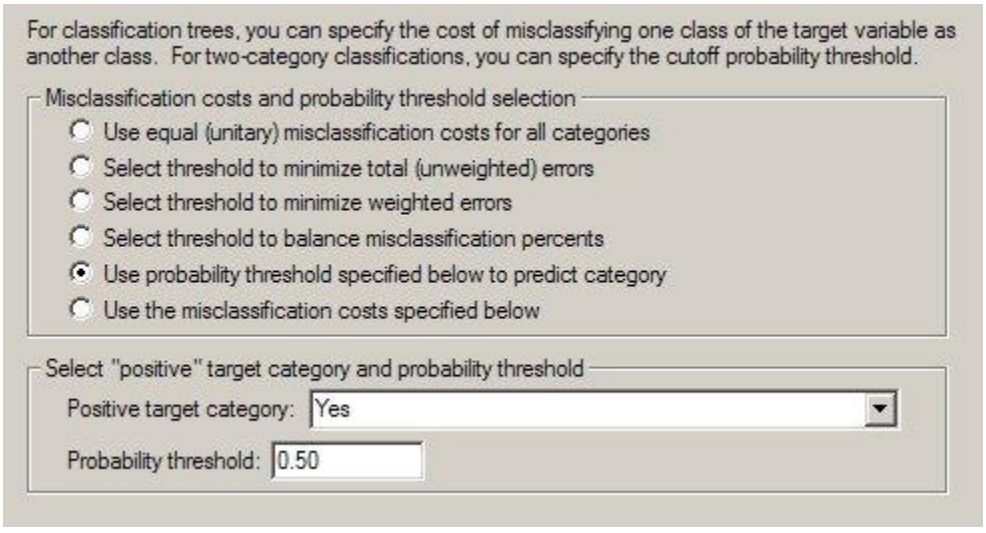
Use equal (unitary) misclassification costs for all categories – If you select this option, DTREG will use the same misclassification costs (1.00) for all categories.

Select threshold to minimize total (unweighted) errors – If this option is selected, DTREG will use a probability threshold that minimizes the total error rate for all cases. This may result in the error rates for each category being very different. This option is only available when creating a model with two target categories.

Select threshold to minimize weighted errors – If this option is selected, DTREG will use the probability threshold that minimizes the weighted misclassification errors. The weighted misclassification error is computed by multiplying the misclassification rate for each target category by a factor that corrects for the relative frequency of cases with that category in the data. Target categories that occur infrequently in the data receive a greater weight to prevent them from being overwhelmed by frequently occurring categories. This option is only available when creating a model with two target categories.

Select threshold to balance misclassification percents – If this option is selected, DTREG will use the probability threshold that approximately balances the misclassification error proportion for the target categories. This option is only available when creating a model with two target categories.

Use probability threshold to predict category – This option is enabled only if the target variable has two categories and you are creating a type of model that predicts probability scores. If you select this option, then the probability threshold section of this screen will be enabled.



Select which category of the target variable you are trying to predict and specify a probability threshold value that must be reached for a case to get assigned that category. If the probability of a case is lower than the specified threshold, then it is assigned the other category. For example, if the two target categories are Yes and No and the corresponding predicted probabilities are Pyes and Pno, then if you select Yes as the target category on this section and specify 0.60 as the threshold, a case will be assigned the Yes category if Pyes is greater than or equal to 0.60. Otherwise it will be assigned the No category. Note: Selecting Yes as the category and specifying a threshold of 0.60 is exactly the same as selecting No as the category and specifying a threshold of 0.40

The Probability Threshold Chart described on page 223 and the Probability Threshold Report described on page 197 can be used to determine how a probability threshold will affect the predictions.

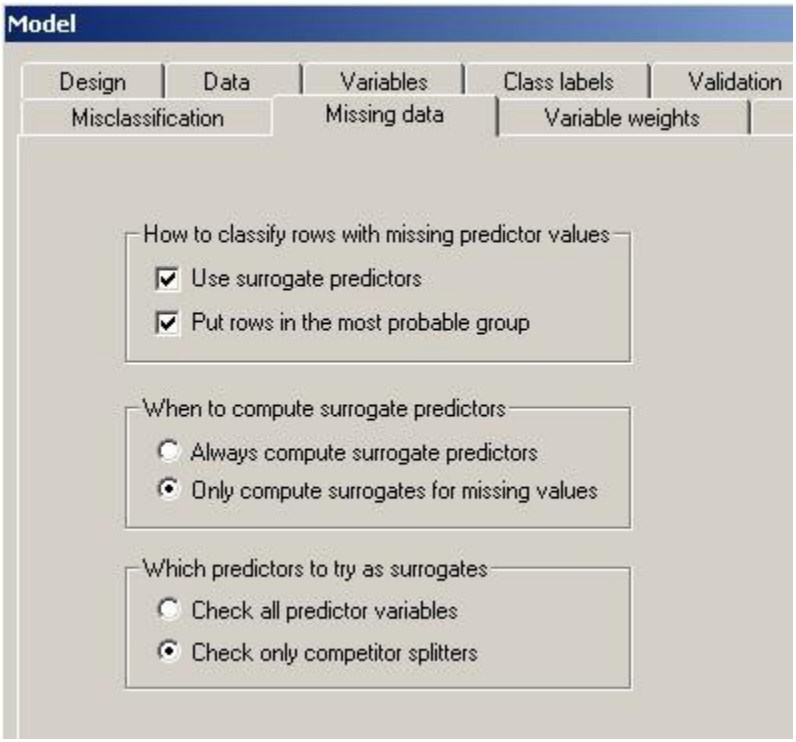
Positive Target Category – Some statistics such as Sensitivity and Specificity (see page 192) use the concept of the "positive" category of the target variable. The positive target category is specified in this field.

Use the misclassification costs specified below – If you select this option, a matrix will be displayed in the lower portion of the screen (see the example screen on the previous page). The categories of the target variable will be shown in the left column and in the top row. An entry in a specified row/column position is the cost of misclassifying the category in the selected column as the category in the selected row. The diagonal elements of the matrix are the cost of correctly classifying a category; their values are usually 0.00 since there is no misclassification cost for a correct classification.

DTREG uses the *altered priors* method to convert the specified misclassification costs into values of category weights (prior probabilities) that perform the misclassification weighting. See Breiman, Friedman, Olshen and Stone (1984) for information about the use of altered priors.

Missing Data Property Page

The Missing Data property page tells DTREG how to handle missing data values.



Missing values are an unfortunate but common occurrence in surveys and research projects: subjects refuse (or forget) to answer some questions, forms are redesigned adding or dropping questions, and subjects sometimes drop out of studies (or die) before all of the information can be collected.

If the value of the target variable or the weight variable is missing, the entire row (case) is dropped. Obviously, if all of the predictor variable values are missing, the row also must be dropped. However, if the value of the target variable is known and some of the predictor variables are available, then it is desirable to use that data rather than dropping the entire row.

Missing Data Options
DTREG offers two methods for salvaging rows with missing values on the predictor variable used for splitting a group. You may check either or both of the boxes corresponding to the methods you want DTREG to use.

DTREG attempts to use the methods in the following order. Once a method is found that can classify the row, the process stops at that point. If the row cannot be classified by any enabled method, the row is not assigned to either child group, and the last node the row ends up in becomes its terminal node.

1. Use surrogate splits – If this option is selected, DTREG attempts to classify rows by using "surrogate" splitter variables.

Surrogate splitters provide the most accurate classification of rows with missing values. This is the default and recommended method.

Surrogate splitter variables are predictor variables that are not as good at splitting a group as the primary splitter but which yield similar splits. DTREG compares which rows are sent to the left and right child groups by the primary splitter with the rows sent to the corresponding child groups by every other predictor variable. The predictors whose splits most closely mimic the split by the primary splitter are the *surrogate splitters*.

The *association* between the primary splitter and each alternate predictor is computed as a function of how closely the alternate predictor matches the primary splitter. (This roughly corresponds to a count of how many rows each predictor sends left and right, but the actual calculation is more complex.) The surrogate splitter variables are ranked in decreasing order of association.

When a row is encountered that has a missing value on the primary splitter, DTREG searches the list of surrogate splitters and uses the one with the highest association to the primary splitter that has a non-missing value for the row.

For additional information about surrogate splitters, please see page 364.

2. Put rows in the most probable group – If the value of the splitting variable is missing, the row is put into whichever child group has the greatest likelihood of receiving an unknown, random case. When this method of used, none of the predictor values for the row contribute to its classification; it is simply dumped into whichever child group has the larger probability of picking up random cases. Usually, the "most probable" group is the group with the largest number of rows assigned to it. However, the most probable group may not necessarily be the largest group if the distribution of categories is not uniform or if unequal category weight values are specified.

Always compute surrogate predictors – If you check this box, DTREG always will compute the association between the primary splitter and all other potential surrogate splitters. If you don't check this box, DTREG will only determine surrogate splitters if

they are needed because rows in a group that is being split have missing values on the primary splitter variable.

Leaving this box unchecked can significantly speed up the generation of the tree, but it has several disadvantages:

1. If you later use the generated tree to "score" a dataset that has missing values, and surrogate splitters were not generated when the tree was built, they will not be available to guide scoring of rows with missing values on splitters. If you do not plan to use the generated tree to score data, then this is not a factor.

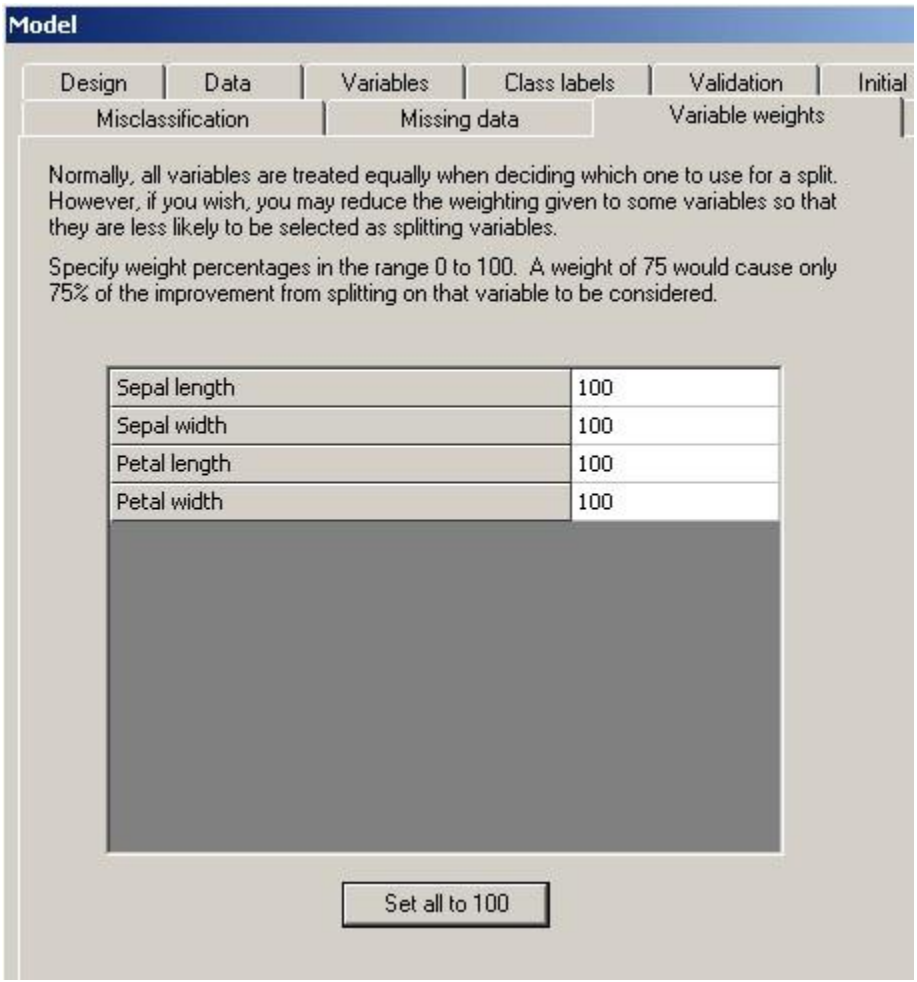
2. The association values assigned to surrogate predictors are used as a component in calculating the overall importance of variables. So if surrogate splitters are not calculated, the overall importance scores will be less accurate.

Check all predictor variables (for surrogates) – If you check this button, then DTREG will check every predictor variable to see how well it functions as a surrogate splitter for the primary splitter. If there are many predictor variables, this is a time-consuming operation, but it guarantees that the best surrogate predictors will be found.

Check only competitor splitters – If you check this button, DTREG will check only the five predictors that were the best "competitors" (runners up) to the primary splitter to see how well they function as surrogates. In about 80% of the cases, predictors that are good surrogates for the primary splitter are also good competitors to the primary splitter. Selecting this operation can dramatically speed up many analyses with minimal loss of accuracy. For example, if there are 100 predictor variables, selecting this operation would reduce the number of surrogate checks from 100 to 5. However, in some cases, predictor variables may be good surrogates without being good competitors; so it is recommended that for the final, definitive tree build, you select the option to check all predictor variables as surrogates.

Variable Weights Property Page

The Variable Weights property page allows you to assign weights to predictor variables so that the improvements derived by splitting on variables are not treated equally.



The left column of this screen shows the names of all predictor variables. The right column shows the weight values. You can assign values between 0 and 100 for weights.

If the weight values are not equal, then the improvement value computed by potentially splitting a group on a predictor is multiplied by the proportion of its weight before being compared with the possible improvements from splitting on other predictors. By reducing the weighting for a variable, you can cause it to be used as a splitter only if its improvement is better than other predictors with higher weights. Hence, DTREG is less likely to use the predictor for splitting.

Reasons for Weighting Variables

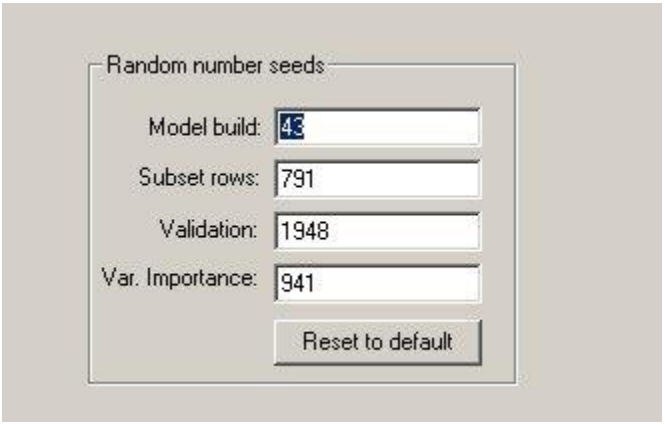
There are several reasons why you might want to use weighting to reduce the likelihood of splitting on a variable:

1. The variable may be difficult or expensive to obtain, so you don't want to have it enter the model too early. For example, the variable might correspond to the result of some unpleasant or expensive invasive medical test that you don't want to use unless it is very significant.

2. The variable may correlate with the target variable in such a way that its value tends to dominate over other predictors too much. For example, if you are analyzing sales data, the quantity of an item sold to a customer might be the target variable, and predictor variables might include the size of the customer's company, their type of business, the area of the country, etc. Since large companies tend to buy more than small companies, the company size predictor may dominate. However, it may be harder to sell to large companies than smaller ones; so, you may want to discount the value of the company size predictor so that other factors such as geographic region and company type play a more significant role in the model.

Miscellaneous Property Page

The Miscellaneous property page currently contains settings for random number seeds.



Random Number Starting Seeds

Random numbers are used for a number of stochastic processes in DTREG. If you want to test whether the random number seeds (starting values) affect the generated model, you can specify the seed values on this screen.

Model build – This is the primary random number generator used for model building. For example, it is used to select the rows and variables used for each tree in a decision tree forest.

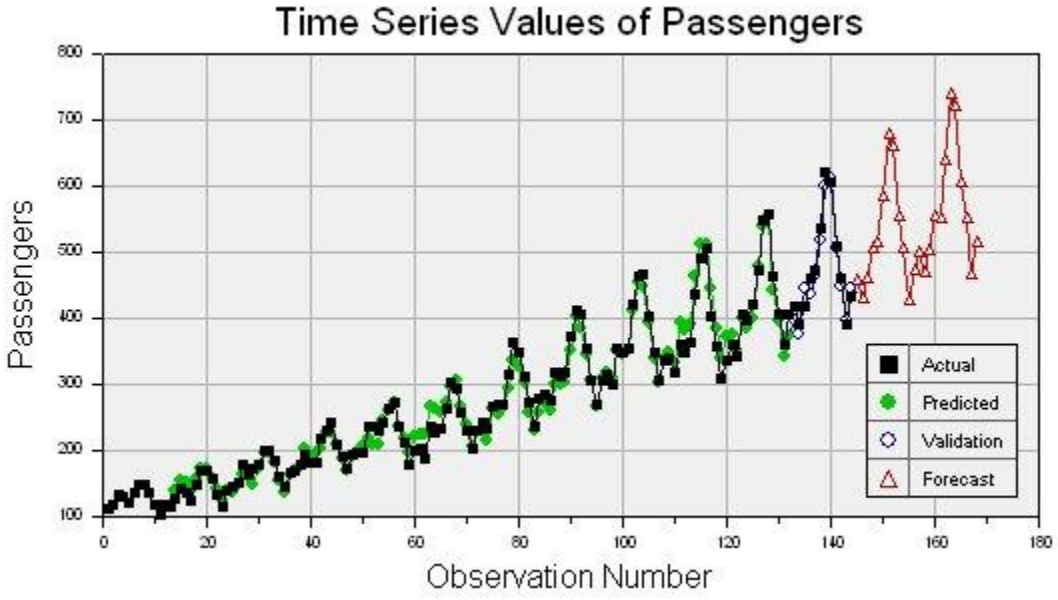
Subset rows – This random number generator is used to select rows when a subset of the rows is being used to train the model.

Validation – This random number generator is used to select the rows that go into cross validation folds. It also controls which rows are held out of the model if holdout sampling is used.

Variable importance – This random number generator is used when sensitivity analysis is being performed to estimate the relative importance of variables.

Time Series Modeling and Forecasting

"*Predicting the future is hard, especially if it hasn't happened yet.*"
– Yogi Berra



Introduction to time series analysis

A *time series* is a chronological sequence of observations on a particular variable. Usually the observations are taken at regular intervals (days, months, years), but the sampling could be irregular. Common examples of time series are the Dow Jones Industrial Average, Gross Domestic Product, unemployment rate, and airline passenger loads. A time series analysis consists of two steps: (1) building a model that represents a time series, and (2) using the model to predict (forecast) future values.

If a time series has a regular pattern, then a value of the series should be a function of previous values. If Y is the target value that we are trying to model and predict, and Y_t is the value of Y at time *t*, then the goal is to create a model of the form:

$$Y_t = f(Y_{t-1}, Y_{t-2}, Y_{t-3}, \ldots, Y_{t-n}) + e_t$$

Where Y_{t-1} is the value of Y for the previous observation, Y_{t-2} is the value two observations ago, etc., and e_t represents noise that does not follow a predictable pattern (this is called a *random shock*). Values of variables occurring prior to the current observation are called *lag values*. If a time series follows a repeating pattern, then the value of Y_t is usually highly correlated with $Y_{t-cycle}$ where *cycle* is the number of

observations in the regular cycle. For example, monthly observations with an annual cycle often can be modeled by $Y_t = f(Y_{t-12})$.

The goal of building a time series model is the same as the goal for other types of predictive models which is to create a model such that the error between the predicted value of the target variable and the actual value is as small as possible. The primary difference between time series models and other types of models is that lag values of the target variable are used as predictor variables, whereas traditional models use other variables as predictors, and the concept of a lag value doesn't apply because the observations don't represent a chronological sequence.

ARMA and modern types of models

Traditional time series analysis uses Box-Jenkins ARMA (Auto-Regressive Moving Average) models. An ARMA model predicts the value of the target variable as a linear function of lag values (this is the auto-regressive part) plus an effect from recent random shock values (this is the moving average part). While ARMA models are widely used, they are limited by the linear basis function.

In contrast to ARMA models, DTREG can create models for time series using neural networks, gene expression programs, support vector machines and other types of functions that can model nonlinear relationships. So, with a DTREG model, the function $f(.)$ in

$$Y_t = f(Y_{t-1}, Y_{t-2}, Y_{t-3}, \ldots, Y_{t-n}) + e_t$$

can be a neural network, gene expression program or other type of general model. This makes it possible for DTREG to model time series that cannot be handled well by ARMA models.

Setting up a time series analysis

Input variables

When building a normal (not time series) model, the input must consist of values for one target variable and one or more predictor variables. When building a time series model, the input can consist of values for only a single variable – the target variable whose values are to be modeled and forecast. Here is an example of an input data set:

```
Passengers
112.
118.
132.
129.
121.
```

The time between observations must be constant (a day, month, year, etc.). If there are missing values, you must provide a row with a missing value indicator for the target variable like this:

```
Passengers
112.
118.
?
129.
121.
```

For financial data like the DJIA where there are never any values for weekend days, it is not necessary to provide missing values for weekend days. However, if there are odd missing days such as holidays, then those days must be specified as missing values. It is also desirable to put in missing values for February 29 on non-leap years so that all years have 366 observations.

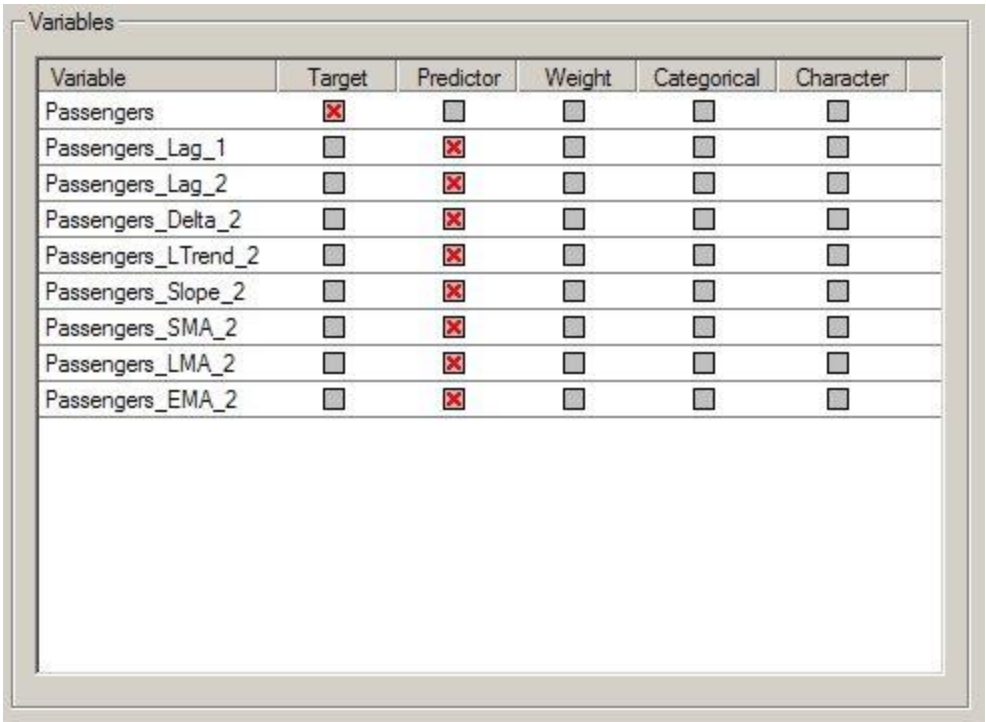
Lag variables

A *lag variable* has the value of some other variable as it occurred some number of periods earlier. For example, here is a set of values for a variable Y, its first lag and its second lag:

```
Y  Y_Lag_1  Y_Lag_2
3     ?        ?
5     3        ?
8     5        3
6     8        5
```

Notice that lag values for observations before the beginning of the series are unknown.

DTREG provides automatic generation of lag variables. On the Time Series Property page (see page 47) you can select which variables are to have lag variables generated and how far back the lag values are to run. You can also create variables for moving averages, linear trends and slopes of previous observations. Here is an example of a Variables Property Page showing lag variables generated for Passengers:



On this screen, you can select which generated variables you want to use as predictors for the model. While it is tempting to generate lots of variables and use all of them in the model, sometimes better models can be generated using only lag values that are multiples of the series' cycle period. The autocorrelation table (see page 147) provides information that helps to determine how many lag values are needed. Moving average, trend and slope variables may detract from the model, so you should always try building a model using only lag variables.

Intervention variables

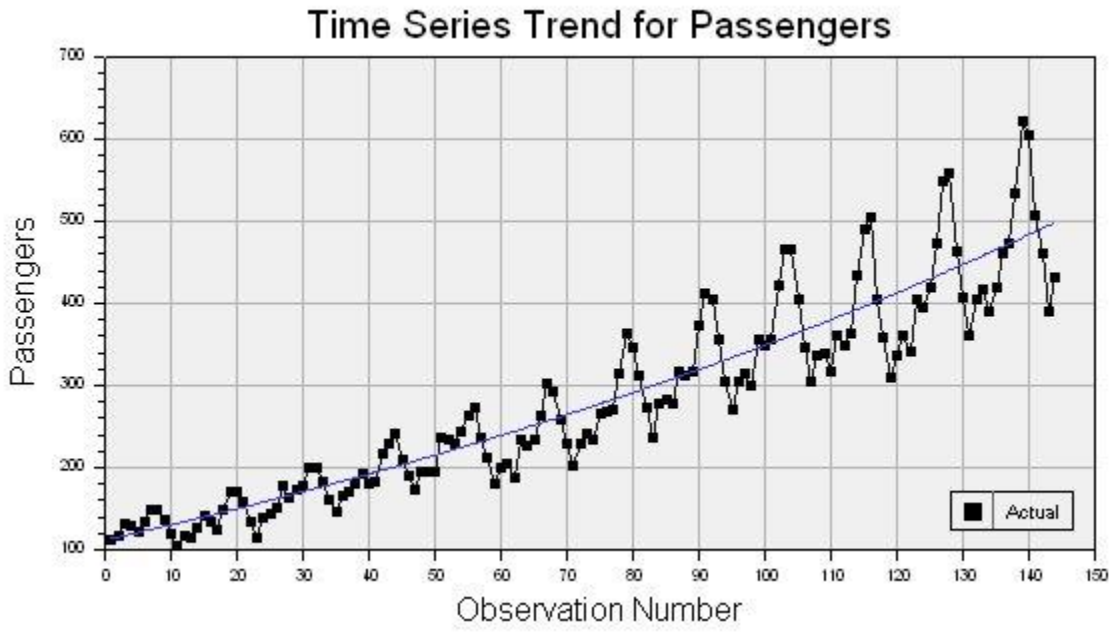
An exceptional event occurring during a time series is known as an *intervention*. Examples of interventions are a change in interest rates, a terrorist act or a labor strike. Such events perturb the time series in ways that cannot be explained by previous (lag) observations.

DTREG allows you to specify additional predictor variables other than the target variable. You could have a variable for the interest rate, the gross domestic product, inflation rate, etc. You also could provide a variable with values of 0 for all rows up to the start of a labor strike, then 1 for rows during a strike, then decreasing values following the end of a strike. These variables are called *intervention variables*; they are specified and used as ordinary predictor variables. DTREG can generate lag values for intervention variables just as for the target variable.

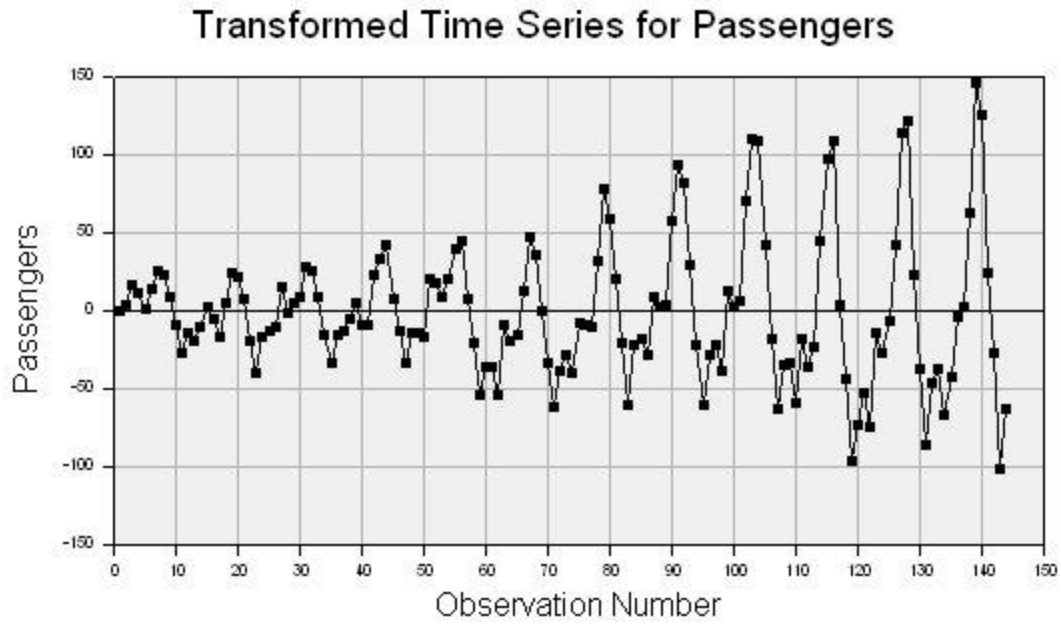
Trend removal and stationary time series

A time series is said to be *stationary* if both its mean (the value about which it is oscillating), and its variance (amplitude) remain constant through time. Classical Box-Jenkins ARMA models only work satisfactorily with stationary time series, so for those types of models it is essential to perform transformations on the series to make it stationary. The models developed by DTREG are less sensitive to non-stationary time series than ARMA models, but they usually benefit by making the series stationary before building the model. DTREG includes facilities for removing trends from time series and adjusting the amplitude.

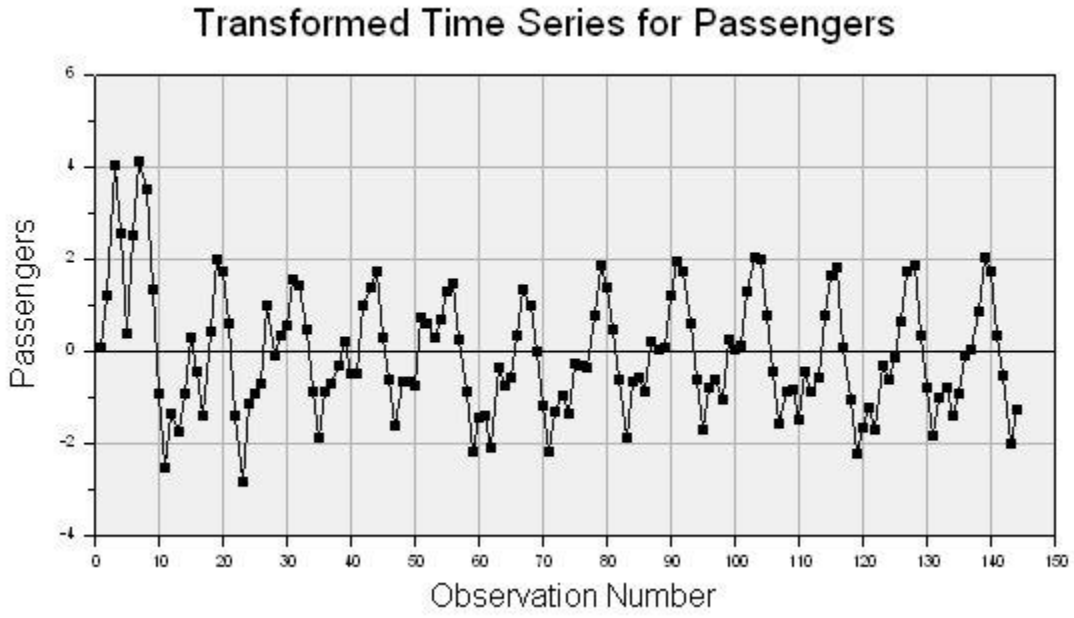
Consider this time series which has both increasing mean and variance:



If the trend removal option is enabled on the Time Series property page (see page 47), then DTREG uses regression to fit either a linear or exponential function to the data. In this example, an exponential function worked best, and it is shown as the blue line running through the middle of the data points. Once the function has been fitted, DTREG subtracts it from the data values producing a new set of values that look like this:



The trend has been removed, but the variance (amplitude) is still increasing with time. If the option is enabled to stabilize variance, then the variance is adjusted to produce this series:



This transformed series is much closer to being stable. The transformed values are then used to build the model. A reverse transformation is applied by DTREG when making forecasts using the model.

Important note: Trend removal is almost always beneficial. However, experiments show that variance stabilization (amplitude adjustment) is beneficial about 20% of the time and

harmful about 80% of the time. So you should try it both ways and use whichever is better.

Selecting the type of model for a time series

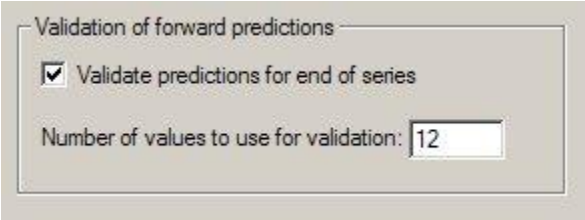
DTREG allows you to use the following types of models for time series: (1) Decision tree, (2) TreeBoost, (3) Multilayer perceptron neural network, (4) General regression neural network (GRNN), (5) RBF neural network, (6) Cascade correlation network, (7) Support vector machine (SVM), (8) Gene expression programming, (9) GMDH neural networks.

Experiments have shown that decision trees usually do not work well because they do a poor job of predicting continuous values. Gene expression programming (GEP) is an excellent method for time series because the functions generated are very general, and they can account for trends and variance changes. General regression neural networks (GRNN) and GMDH neural networks also perform very well in tests. Multilayer perceptrons sometimes work very well, but they are more temperamental to train. So the best recommendation is to always try GEP and GRNN models, and then try other types of models if you have time. If you use a GEP model, it is best to enable the feature to allow it to evolve numeric constants (see page 108).

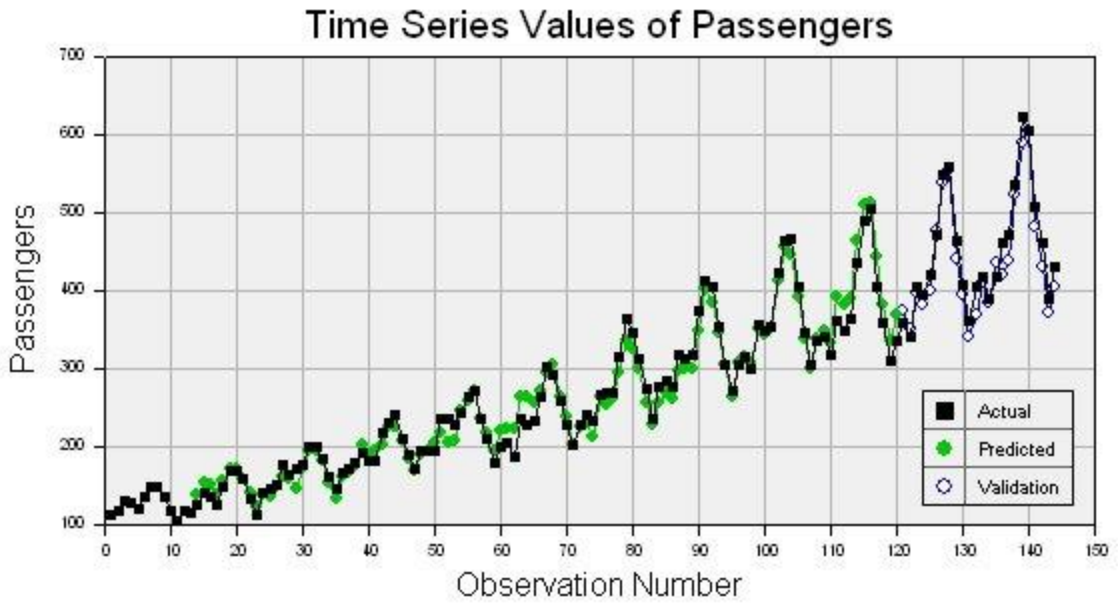
Evaluating the forecasting accuracy of a model

Before you bet your life savings on the forecasts of a model, it is nice to do some tests to evaluate the predictive accuracy of the model. DTREG includes a built-in validation system that builds a model using the first observations in the series and then evaluates (validates) the model by comparing its forecast to the remaining observations at the end of the series.

Time series validation is enabled on the Time Series property page (see page 47).



Specify the number of observations at the end of the series that you want to use for the validation. DTREG will build a model using only the observations prior to these held-out observations. It will then use that model to forecast values for the observations that were held out, and it will produce a report and chart showing the quality of the forecast. Here is an example of a chart showing the actual values with black squares and the validation forecast values with open circles:



Validation also generates a table of actual and predicted values:

```
 --- Validation Time Series Values ---

 Row     Actual    Predicted      Error      Error %
-----  ---------  ---------  ----------  --------
  133  417.00000  396.65452   20.345480     4.879
  134  391.00000  377.05068   13.949323     3.568
  135  419.00000  446.66871  -27.668706     6.604
  136  461.00000  435.56485   25.435146     5.517
  137  472.00000  462.14325    9.856747     2.088
  138  535.00000  517.45376   17.546240     3.280
  139  622.00000  599.82994   22.170064     3.564
  140  606.00000  611.68442   -5.684423     0.938
  141  508.00000  507.37890    0.621100     0.122
  142  461.00000  447.01704   13.982962     3.033
  143  390.00000  398.09507   -8.095074     2.076
  144  432.00000  444.67910  -12.679105     2.935
```

If you compare validation results from DTREG with other programs, you need to check how the other programs compute the predicted values. Some programs use actual (known) lag values when generating the predictions; this gives an unrealistically accurate prediction. DTREG uses the lag values for predicted values when forecasting: this makes validation operate like real forecasting where lag values must be based on predicted values rather than known values.

Time series model statistics report

After a model is created, DTREG produces a section in the analysis report with statistics about the model.

Hurst Exponent

The Hurst Exponent is a measure of pattern (long term memory) in a time series. In particular, it measures the relative tendency of a time series either to regress strongly to the mean or to cluster in a direction. See the description of the Hurst Exponent at Wikipedia (http://en.wikipedia.org/wiki/Hurst_exponent).

The value of the Hurst Exponent can vary from 0.0 to 1.0. A value of 0.5 indicates the series has random values. Values between 0.5 and 1.0 indicate positive autocorrelation – that is, increasing values tend to be followed by more increasing values. A Hurst Exponent value between 0.0 and 0.5 indicates negative autocorrelation – that is, increasing values tend to be followed by decreasing values. A value of 0.5 indicates that there is an equal probability of increasing or decreasing at any point. The Hurst Exponent for the Dow Jones Industrial Average (DJIA) typically varies between 0.42 and 0.68 over 4 year periods. [Qian & Rasheed, 2004]. Unfortunately, there's no way to predict what it will be for the next four years.

There are several methods for computing the Hurst Exponent; they usually produce similar values. DTREG computes the Hurst Exponent using two methods: (1) the common Rescaled Range (R/S) method, and (2) the "Dispersional Analysis" method suggested in a 2000 Ph.D. dissertation by Henrik J. Blok [Blok, 2000].

Autocorrelation and partial autocorrelation

The autocorrelation and partial autocorrelation tables provide important information about the significance of the lag variables.

Autocorrelation table

```
 ----------------------------  Autocorrelations  ------------------------------

Lag  Correlation  Std.Err.     t     -1 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 1
  1   0.70865407  0.083333    8.504  |                  .  |**************      |
  2   0.23608974  0.117980    2.001  |                .    |*****               |
  3  -0.16207088  0.121217    1.337  |               .  **|    .               |
  4  -0.41181655  0.122712    3.356  |             *******|    .               |
  5  -0.46768898  0.131961    3.544  |            ********|    .               |
  6  -0.46501203  0.143009    3.252  |            ********|     .              |
  7  -0.43595197  0.153150    2.847  |            ********|     .              |
  8  -0.36759217  0.161538    2.276  |              ******|     .              |
  9  -0.13341625  0.167246    0.798  |              .    **|      .             |
 10   0.20091610  0.167984    1.196  |              .      |****  .             |
 11   0.58898400  0.169644    3.472  |              .      |************        |
 12   0.82252315  0.183296    4.487  |              .      |****************    |
 13   0.58265202  0.207349    2.810  |            .        |************        |
 14   0.17178261  0.218423    0.786  |           .         |***     .           |
 15  -0.16852975  0.219360    0.768  |           .      **|        .           |
 16  -0.36938903  0.220257    1.677  |           .  ******|        .           |
```

An autocorrelation is the correlation between the target variable and lag values for the same variable. Correlation values range from -1 to +1. A value of +1 indicates that the two variables move together perfectly; a value of -1 indicates that they move in opposite directions. When building a time series model, it is important to include lag values that have large, positive autocorrelation values. Sometimes it is also useful to include those that have large negative autocorrelations. Examining the autocorrelation table shown above, we see that the highest autocorrelation is +0.82523155 which occurs with a lag of 12. Hence we want to be sure to include lag values up to 12 when building the model. It is best to experiment with including all lags from 1 to 12 and also ranges such as just 11 through 13.

DTREG computes autocorrelations for the maximum lag range specified on the Time Series property page, so you may want to set it to a large value initially to get the full autocorrelation table and then reduce it once you figure out the largest lag needed by the model.

The second column of the autocorrelation table shows the standard error of the autocorrelation, this is followed by the t-statistic in the third column.

The right side of the autocorrelation table is a bar chart with asterisks used to indicate positive or negative correlations right or left of the centerline. The dots shown in the chart mark the points two standard deviations from zero. If the autocorrelation bar is longer than the dot marker (that is, it covers it), then the autocorrelation should be considered significant. In this example, significant autocorrelations occurred for lags 1, 2, 11, 12 and 13.

Partial autocorrelation table

```
 ------------------------  Partial Autocorrelations  -------------------------

Lag  Correlation  Std.Err.     t     -1 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 1
  1   0.70865407  0.083333    8.504  |                   .  |**************      |
  2  -0.53454362  0.083333    6.415  |          **********|  .                  |
  3  -0.11250388  0.083333    1.350  |                  . *|  .                  |
  4  -0.19447876  0.083333    2.334  |                 ***|  .                  |
  5  -0.04801434  0.083333    0.576  |                   .  |  .                  |
  6  -0.36000273  0.083333    4.320  |               ******|  .                  |
  7  -0.23338000  0.083333    2.801  |                 ****|  .                  |
  8  -0.31680727  0.083333    3.802  |                *****|  .                  |
  9   0.14973536  0.083333    1.797  |                   .  |***                  |
 10  -0.03381760  0.083333    0.406  |                   .  |  .                  |
 11   0.54592233  0.083333    6.551  |                   .  |***********          |
 12   0.18345454  0.083333    2.201  |                   .  |****                 |
 13  -0.45227494  0.083333    5.427  |             ********|  .                  |
 14   0.16036757  0.083333    1.924  |                   .  |***                  |
```

The partial autocorrelation is the autocorrelation of time series observations separated by a lag of *k* time units with the effects of the intervening observations eliminated.

Autocorrelation and partial autocorrelation tables are also provided for the residuals (errors) between the actual and predicted values of the time series.

Measures of fitting accuracy

DTREG generates a report with several measures of the accuracy of the predicted value. The first section compares the predicted values with the actual values for the rows use the train the model. If validation is enabled, a second table is generated showing how well the predicted validation rows match the actual rows.

```
  ============  Time Series Statistics  ============

Exponential trend: Passengers = -239.952648 + 351.737895*exp(0.005155*row)
Variance explained by trend = 86.166%


 ---  Training Data  ---

Mean target value for input data = 262.49242
Mean target value for predicted values = 261.24983

Variance in input data = 11282.932
Residual (unexplained) variance after model fit = 254.51416
Proportion of variance explained by model = 0.97744  (97.744%)

Coefficient of variation (CV) = 0.060777
Normalized mean square error (NMSE) = 0.022557
Correlation between actual and predicted = 0.988944

Maximum error = 41.131548
MSE (Mean Squared Error) = 254.51416
MAE (Mean Absolute Error) = 12.726286
MAPE (Mean Absolute Percentage Error) = 5.5055268
```

If DTREG removes a trend from the time series, the table shows the trend equation, and it shows how much of the total variance of the time series is explained by the trend.

There are many useful numbers in this table, but two of them are especially important for evaluating time series predictions:

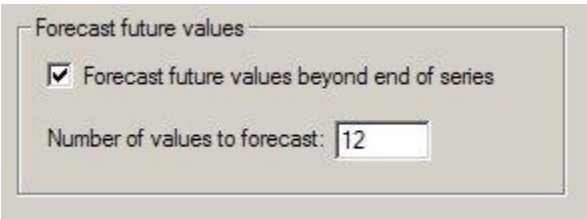
Proportion of variance explained by model – this is the best single measure of how well the predicted values match the actual values. If the predicted values exactly match the actual values, then the model would explain 100% of the variance.

Correlation between actual and predicted – This is the Pearson correlation coefficient between the actual values and the predicted values; it measures whether the actual and predicted values move in the same direction. The possible range of values is -1 to +1. A positive correlation means that the actual and predicted values generally move in the same direction. A correlation of +1 means that the actual and predicted values are synchronized; this is the ideal case. A negative correlation means that the actual and

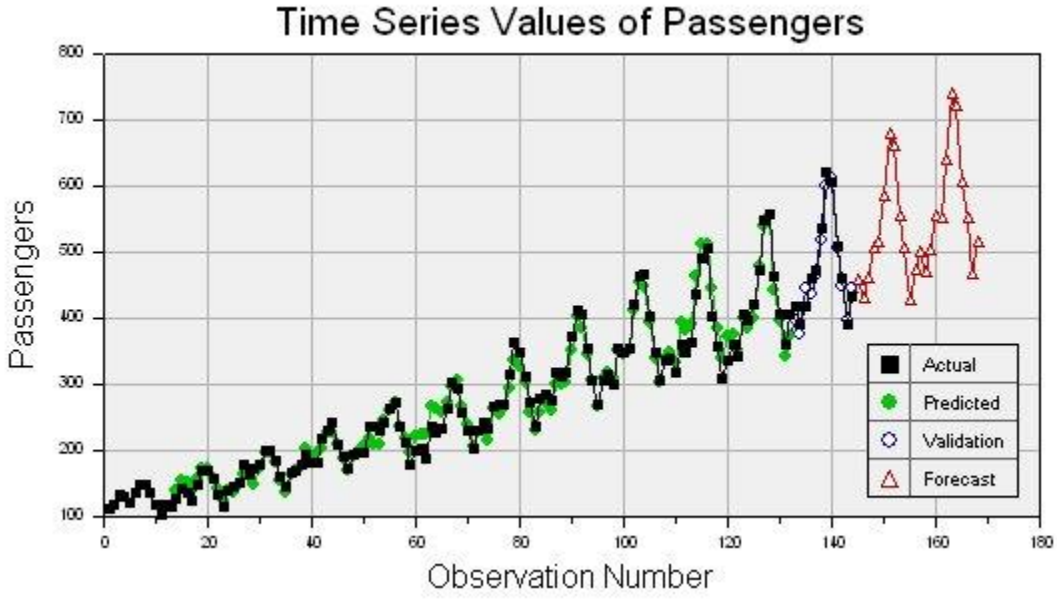
predicted values move in opposite directions. A correlation near zero means that the predicted values are no better than random guesses.

Forecasting future values

Once a model has been created for a time series, DTREG can use it to forecast future values beyond the end of the series. You enable forecasting on the Time Series property page (see page 47).



The Time Series chart displays the actual values, validation values (if validation is requested) and the forecast values.



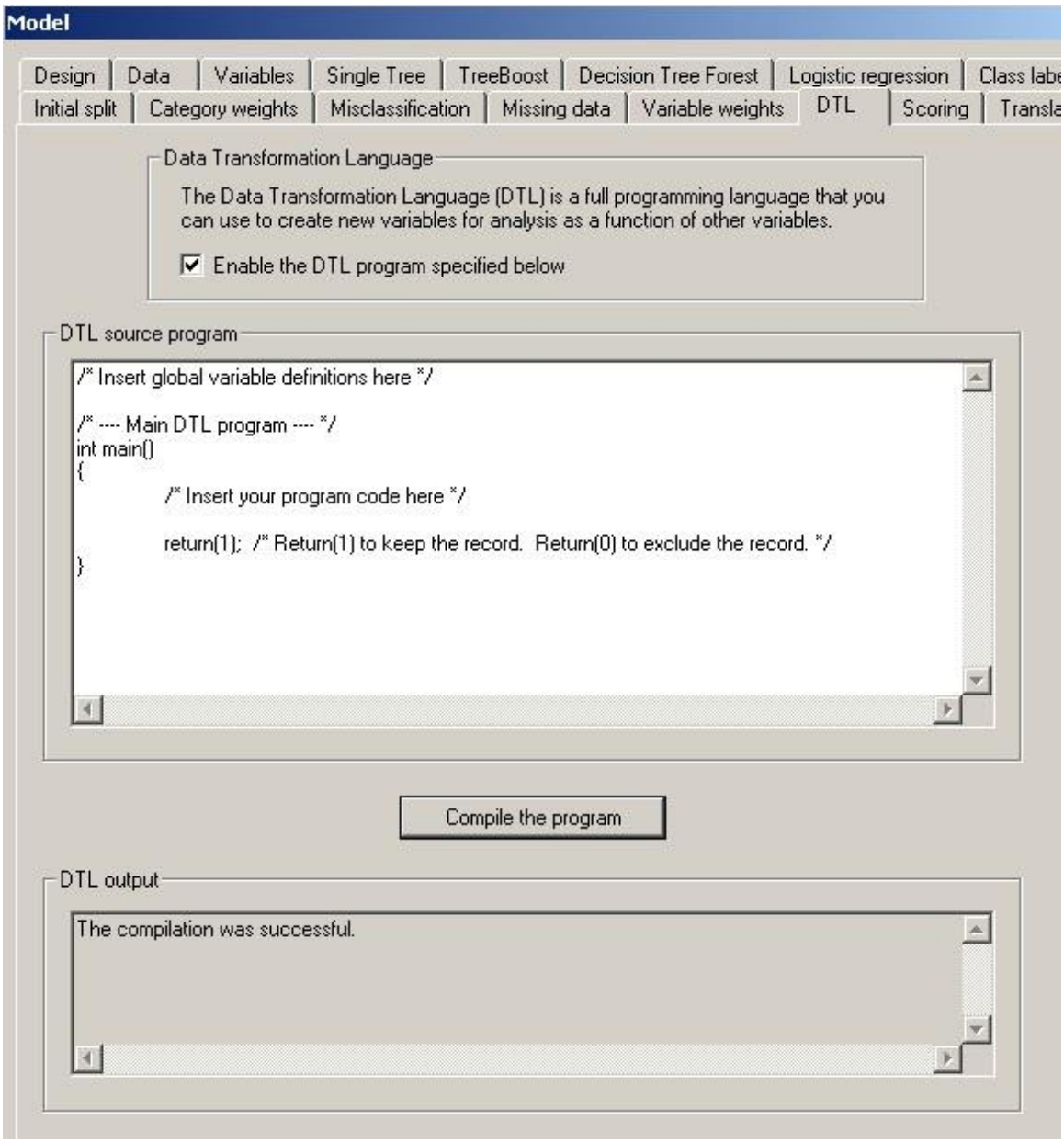
The analysis report also displays a table of forecast values:

```
 --- Forecast Time Series Values ---

 Row   Predicted
-----  ---------
  145  457.63942
  146  429.32697
  147  459.64579
  148  506.19975
  149  514.89035
  150  584.91959
```

DTL: Data Transformation Language

The Data Transformation Language (DTL) can be used to transform variables, create new variables and select which data records should be included in the analysis.

DTL Property Page



DTL is a full-featured programming language. **The full manual for DTL can be downloaded from http://www.dtreg.com/DTL.pdf** This chapter does not provide a full reference for DTL, instead it presents some typical uses of DTL with DTREG analyses.

The main() function

Every DTL program must have a main() function that is executed by DTREG for each data record. The main() function must contain a return statement that signals DTREG whether the current record is to be used in the analysis or excluded. If the return statement returns a value of 1, the record is used in the analysis. If the return statement returns a value of 0 (zero), the record is excluded from the analysis.

Here is a simple main program that accepts all records:

```
int main()
{
      return(1);
}
```

Here is an example that accepts records that have a value of "M" for Sex and rejects other records:

```
int main()
{
      if (Sex == "M") {
            return(1);
      } else {
            return(0);
      }
}
```

Here is an example that accepts records that have a value of "M" for Sex variable and a value of 65 or greater for Age:

```
int main()
{
      if (Sex == "M" && Age >= 65) {
            return(1);
      } else {
            return(0);
      }
}
```

Here is a main program that accepts about half of the records and rejects half:

```
int main()
{
      if (random() > 0.5) {
            return(1);
      } else {
            return(0);
      }
}
```

Global Variables

A global variable is a variable defined outside the scope of any function; usually, global variables are defined at the top of the program. Global variables can be accessed by any function in the DTL program. Global variables may have any of the three data types, **int**, **double** or **string**. Global variables you define are called *explicit* global variables. Global variables defined automatically by DTREG are called *implicit* global variables.

Implicit Global Variables

DTREG defines implicit global variables for each variable in the input data file. This includes *all* data variables, even variables not designated as predictor, target or weight variables. The implicit global variables are not visible in the DTL source program, but they can be used by the program.

If a variable is specified as categorical in the DTREG model, the implicit definition has type string. If the variable is specified as continuous, the implicit definition has type double. For example, if a data file contains four continuous variables, Age, BloodPressure, Height, Weight and one categorical variable Sex, then the implicit definitions (which you will not see) would be:

```
double Age;
double BloodPressure;
double Height;
double Weight;
string Sex;
```

The main() function and any other functions in the DTL program can reference these implicit global variables.

In addition to generating a global variable for each variable in the data file, DTREG also generates several other global variables:

```
int RECORDNUMBER;       /* The number of the current data record */
int DOINGSCORE;         /* 1 if scoring, 0 if analysis is being run */
double MISSINGVALUE;    /* Value used to indicate missing value */
```

Any changes your program makes to the values of implicit global variables are *not* used in the analysis. If you want to transform variables, you must define your own global variables as described below and store values into them.

Explicit Global Variables

You can define your own global variables by putting their definitions outside the scope of any function. It is recommended that they be put at the top of the DTL program before main().

Any global variable you define in a DTL program that does not have the "static" declaration will be available as a variable in the DTREG analysis. This is the way you generate transformed variables. For example, the following program generates a new variable, Size, which is the product of two input data variables, Height and Weight:

```
double Size;
int main()
{
      Size = Height * Weight;
      Return(1);
}
```

With this DTL program defined, the Size variable will be available for use in the DTREG analysis. The Height and Weight variables also are available.

Here is an example that creates a variable called Republican that is 1 if the value of PartyAffiliation is "R" and 0 if PartyAffiliation is anything else:

```
double Republican;
int main()
{
      if (PartyAffiliation == "R") {
            Republican = 1;
      } else {
            Republican = 0;
      }
      return(1);
}
```

Here is an example that creates a LogAge variable that is the natural logarithm of the Age variable:

```
double LogAge;
int main()
{
      LogAge = log(Age);
      return(1);
}
```

Here is an example that creates a variable named ZIP3 that has the first three digits of a zip code whose five-digit code is stored in ZIP5. The substring operator, **[***start***:***length***],** is used to extract the first three characters.

```
string ZIP3;
int main()
{
      ZIP3 = ZIP5[0:3];
      return(1);
}
```

Here is an example that uses the lag() function to generate variables with values of StockPrice from 1, 2 and 12 prior periods. Note, the missing value code is returned by the lag() function when a request is made for a prior value that has not yet been stored.

```
double StockPriceBack1;
double StockPriceBack2;
double StockPriceBack12;
int main()
{
      StockPriceBack1 = lag(StockPrice,1);
      StockPriceBack2 = lag(StockPrice,2);
      StockPriceBack12 = lag(StockPrice,12);
      return(1);
}
```

Sometimes missing values for numeric variables are coded with values like "999". DTREG uses a special value called "MissingValue" to indicate missing values. Here is an example DTL program that converts input data values of "999" on an Age variable to the internal missing value. The new variable with the transformed values is called NewAge.

```
double NewAge;
int main()
{
      if (Age == 999) {
            NewAge = MissingValue;
      } else {
            NewAge = Age;
      }
      return(1);
}
```

Static Global Variables

Static global variables are used to store information between calls of the main() function for each data record. They also can be used to hold information that must be accessed by multiple functions. Static global variables may *not* be used as variables in the DTREG analysis. To declare a static global variable, put the word "static" in front of the declaration like this:

```
static int FileNumber;
static int Count;
static double LastAge;
static string LastName;
```

Using the StoreData() function to generate data records

The main() function is called for each record in the input data file, and it returns 1 to keep the record or 0 to reject the record. DTL provides a StoreData() function that you can call to generate additional records. Each time you call StoreData(), the current values of the global variables are used to generate a new data record which is included in the analysis. This allows you to generate multiple records from a single input record.

Consider a data set that is to be analyzed using logistic regression. The data set measures the response of patients to varying dose levels of a drug. There are three variables in the input data file, Dose (the amount of the drug), Positive (the number of patients with positive responses) and Negative (the number of patients that did not respond). Hence the implicit global definitions generated by DTREG for the DTL program are:

```
double Dose;
double Positive;
double Negative;
```

The following DTL program defines a new variable, Response, that has the value 1 if the patient responds positively and 0 if the patient does not respond. The DTL program uses the StoreData() function to generate a separate record for each patient. After calling StoreData() the appropriate number of times, it uses the return(0) statement to reject the original record.

```
double Response;  /* Generated variable with 1 or 0 response */
int main()
{
      int count;
      /* Generate the positive response records */
      Response = 1;
      for (count=0; count<Positive; count++) {
            StoreData();
      }
      /* Generate the negative response records */
      Response = 0;
      For (count=0; count<Negative; count++) {
            StoreData();
      }
      /* Reject the original record */
      return(0);
}
```

The StartRun() and EndRun() Functions

The optional StartRun() and EndRun() functions can be used to perform initialization and cleanup in a DTL program.

If your DTL program contains a StartRun() function, it is called once at the beginning of the run before the first data record is processed. It can perform initialization.

If your DTL program contains an EndRun() function, it is called once after the last data record has been read.

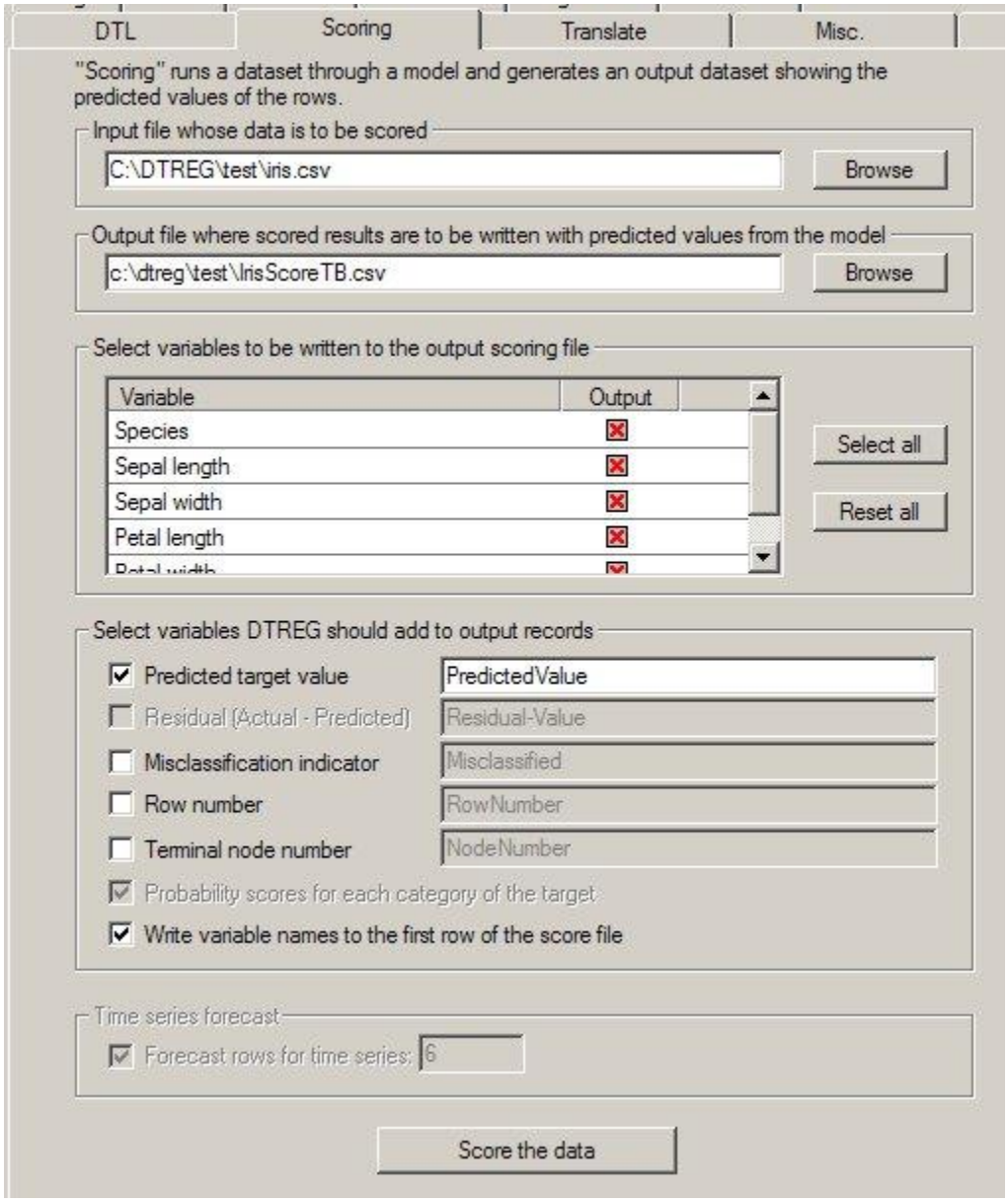
In the following example, the DTL program opens an output file in the StartRun() function, writes information about each data record in the main() function and closes the file in the EndRun() function. Note the use of a static global variable to store the file handle number between iterations.

```
static int FileHandle;

void StartRun()
{
      FileHandle = fopen(“Data.dat”,”wt”);
      return;
}

int main()
{
      fprintf(FileHandle,”%f %f\n”,x,y);
      return(1);
}

void EndRun()
{
      fclose(FileHandle);
      return;
}
```

Scoring Data Values

"Scoring" runs a set of data rows through a generated predictive model and generates a new data file showing the predicted value of the target variable and other information for each row.

Scoring Property Page
To score data, select the Scoring property page for the model.



Input and output scoring files

Input file whose data is to be scored – This is the name of the data file that is to be read and scored using the predictive model. This could be the same data file that was used to generate the model, or (more commonly) it could be some other file for which you wish to use a model to predict values.

The input data file must have the same format as an input file used to build the model:

- The first row in the file must contain the names of the variables in the file.
- Columns must be separated by the character specified by the "Character used to separate columns" parameter on the Data property page (see page 36).
- Either a period or a comma may be used as the decimal point indicator. Select which will be used on the Data property page using the parameter "Character used for a decimal point in the input data file" (see page 36).
- Missing values must be indicated by empty fields, question marks or periods.

The variables in the file being scored do *not* have to correspond to the variables in the data file that was used to build the tree. DTREG uses the first row of the file to determine which variables are present and which rows they are in. If a predictor variable is missing from the file being scored, then all of the values of that variable are treated as missing.

The target variable may be omitted (and often is) since the purpose of scoring is to predict the target value for each row. If target values are provided, they can be used to compute residual values for the prediction and misclassified rows.

Output file where scored results are to be written – This is the name of the output file that will be created by DTREG as it scores the rows in the input file. The generated output file will have the same format as the input file: the first row will have the names of the variables in the file.

Variables written to the output scoring file

Variables to be written to the output scoring file – There will be one entry in this table for each variable that was specified at the time that the tree was built. Select which variables you want written to the output file. If there are variables in the input scoring file that were not part of the input file used to construct the tree, they are written to the output file. Variables can be used to classify rows even if they are not written to the output file.

Variables DTREG should add to output records – Select which generated variables you want DTREG to add to the output file. Check the box by each variable you want DTREG to add, and specify the name of the variable in the associated box.

- **Predicted target value** – This is the predicted value of the target variable for each data row in the scoring file. The predicted target value is obtained by using the value of the predictor variables for the row to run the row through the tree until it reached a terminal node. The value of the target variable in the terminal node is used as the predicted value of the target variable for the row.
- **Residual (Actual – Predicted)** – If you are performing a regression analysis (i.e., the target variable is continuous), then this output variable is the "residual" value for the row which is the difference between the actual value of the target variable for the row and the predicted value. In order to generate this variable, values for the target variable must be included in the input scoring file.
- **Misclassification indicator** – If a classification analysis is being performed (i.e., the target variable is categorical), then this generated variable has the value 0 (zero) if the predicted value of the target variable matches the actual value. It has the value 1 (one) if the predicted value is different from the actual value (i.e., the row was misclassified). Note, in order to generate this variable, values for the target variable must be included in the input scoring file.
- **Row number** – If selected, this variable has the number of the row in the input scoring file. The first row is numbered 1.
- **Terminal node number** – If selected, this variable will have the number of the terminal node for the row. That is, the last node the row ended up in after being run through the tree.
- **Probability scores for each category of the target** – DTREG computes a posterior, likelihood probability value for each category of the target variable. The predicted category for a row is computed by selecting the most likely category adjusted by the misclassification costs (technically, the category is selected so as to minimize the misclassification cost). If you select this option, DTREG will write to the output scoring file the computed probability values for each target category. The names of the columns have the form Prob_*category* where '*category*' is the value of the category. For example, if the target variable is Sex, the probability columns might have names of Prob_Male and Prob_Female.
- **Write variable names to the first row of the scoring file** – If this box is checked then DTREG will write the names of the variables in the score file to the first record of the score file. This makes it possible to import the score file into programs like Excel that expect the variable names to be in the first row.

Forecast rows for time series – If you are performing a time series analysis, you can generate forecasts by checking this box and specifying how many observations you want DTREG to forecast beyond the end of the training data. The data set used as input for scoring must start with and include the same data that was in the training data; it may contain additional rows beyond the training data. Note that there also is an option on the Time Series property page (see page 47) where you can specify that forecast values for the training data are to be written.

Start scoring the data

Once you have specified the input and output files and selected the variables to be included in the output file, click "**Score the data**" button to begin the process.

Using scoring for validation with a test dataset

In addition to using scoring to generate predicted values for a dataset, you can use scoring to test a model against a dataset whose actual target values are known. To do this, use the normal scoring procedure with a dataset that has the target variable along with the predictors. When the scoring process finishes, DTREG displays a report showing the misclassification rate for the model applied to the dataset that was scored.

For classification models, the report looks like this:

```
Scoring was performed 23-Mar-2004 13:01:40

Input file = C:\DTREG\Test\LiverDisorder.csv
Output file = c:\DTREG\LiverDisorder2.csv

Number of observations scored = 345

             Actual   --Misclassified--
  Category    Count     Count   Percent
  --------  --------  --------  -------
         1       145        24   16.552
         2       200        53   26.500
  --------  --------  --------  -------
     Total       345        77   22.319
```

For regression models, the report looks like this:

```
Scoring was performed 23-Mar-2004 13:27:44

Input file = C:\DTREG\Test\Boston.csv
Output file = C:\DTREG\TestScore.csv

Number of observations scored = 506

Mean target value for data being scored = 22.532806
Mean target value for predicted target values = 22.532806

Average absolute error after tree fitting = 2.126722

Variance in scored data = 84.419556
Residual (unexplained) variance after tree fitting = 7.806666
Proportion of variance explained = 0.90753  (90.753%)
```

How missing values are handled during scoring

If the value of a predictor variable used at a split is missing, DTREG attempts to use the surrogate predictors for the split. It tries each surrogate splitter in the order of decreasing association values until it finds one that has a non-missing value on the row that is being scored. If it is unable to find a surrogate splitter, then the last node that the row reached (i.e., the one for which no split could be found) becomes the terminal node for that row, and the predicted value for the group of rows in that node is used as the predicted value for the row being scored. For additional information about surrogate splitters, please see page 364.

If you anticipate scoring data that has missing values, you should select the option "Always create surrogate splitters" on the Missing Data property page when the tree is built. (See page 133.) Surrogate splitters *cannot* be created when scoring is being done; they must be created at the time that the tree is constructed.

Translation: Generating Code for Scoring

"Translation" generates source code that can be compiled with an application program to perform scoring.

DTREG is capable of generating source code for the C language (this code also can be used with C++ programs) and SAS®. The Translate function can generate code for all types of models in the C language and for all types of models except for Support Vector Machine (SVM) in the SAS language. You can use the DTREG.DLL COM library module to perform scoring for other types of applications. See page 375 for information about the DTREG.DLL library module. The primary advantage of generated source code is that it executes faster than using the DLL library.

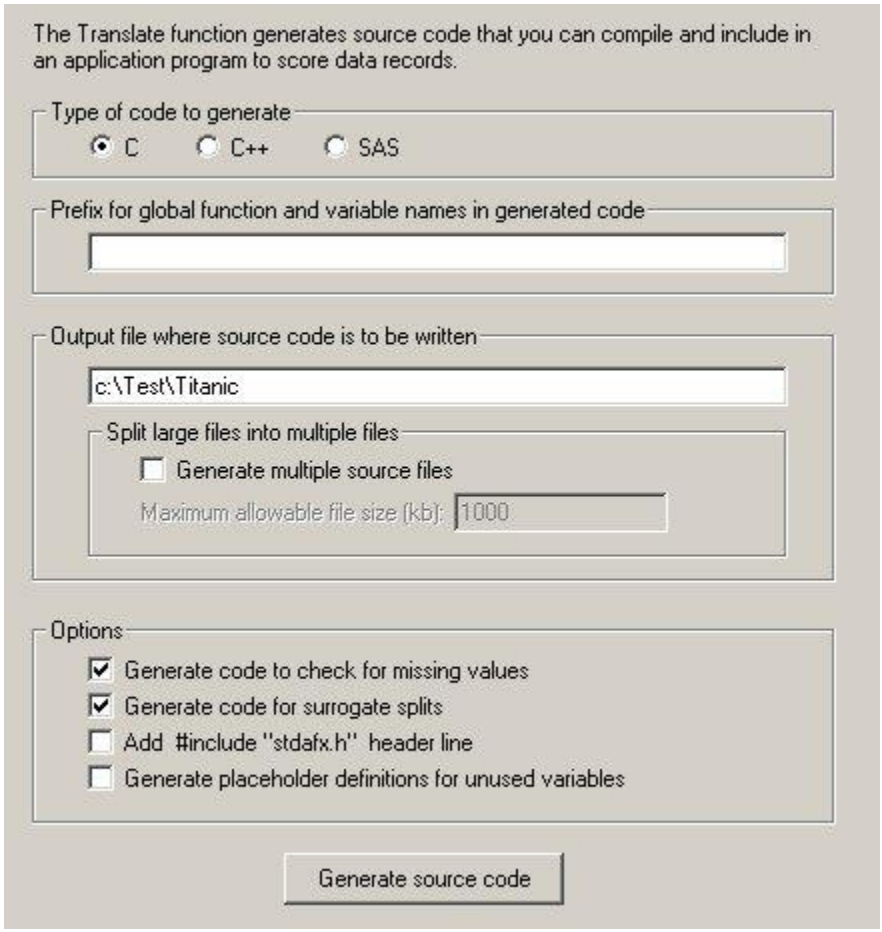
The Translate function is available only in the Enterprise Version of DTREG.

Here is an overview of the process of generating and using scoring source code:

1. Use DTREG to build a model.
2. Use the Translate function to generate source code.
3. Compile the source code with an application program you have written.
4. Run the application to read data and call the scoring function generated by DTREG.

Translate Property Page

To generate scoring source code, select the Translate property page for the model.



Type of code to generate
Check the button to select whether you want DTREG to generate a C or C++ or SAS® source file.

Prefix for global function and variable names in generated code
If you specify a string in this field, it will be added to the front of the names of all functions and global variables in the source code generated by DTREG. This is useful when you want to call generated code for two different models from the same application program. The specified string must be valid as the beginning of a variable and function name (it must begin with a letter, and it may not contain spaces).

Output file where source code is to be written
Specify the full name including device and directory where you want DTREG to write the generated source code. If you omit the extension from the file name, DTREG will add it. In addition to the .c file, DTREG also generates one or more .h header files using the same base file name. In the case of SAS code generation, DTREG generates a file named "*program*.sas" and a header file named "*program*_header.sas".

Split large files into multiple files
If the predictive model is very large, the generated source code may be too large to compile as a single unit. This problem occurs most commonly with TreeBoost and Decision Tree Forest models composed of many trees. If you turn on this option, DTREG will generate multiple source files that you can compile as separate modules and link together with the application. When multiple source files are created, DTREG appends "*_nnn*" to the end of the file name, where *nnn* is a sequence number. DTREG also generates a header file named "*basename*_Internal.h" that is used to transfer information between the generated modules; you should *not* include this header file in your application. SAS source programs cannot be split.

Maximum allowable file size
If you turn on the option to generate multiple source files, DTREG uses the size you specify in this field to control when one source file ends and the next one begins. The size is approximate since DTREG cannot split a function in the middle. The size is specified in units of K bytes, so a value of 1000 corresponds to 1000 kb which is 1 MB. The maximum allowable source file size is dependent on the compiler you use. The Microsoft Visual C++ compiler seems to be able to handle about 1.2 MB in each source file.

Generate code to check for missing values
If you turn on this option, DTREG will generate code to check for missing data values and take the appropriate action. If you do not turn on this option, it is your responsibility to make sure that no missing values are passed to the generated scoring function.

Generate code for surrogate splits
If you turn on this option, DTREG will generate code to use surrogate splits to handle missing values. In order to use this option, the model must have been created with surrogate split information, and you must turn on the option to tell DTREG to check for missing values. See page 364 for additional about surrogate splits.

Add #include "stdafx.h" header line
If you check this box, DTREG will insert the following line in each generated source file:

```
#include "stdafx.h"
```

This is necessary when you are using Microsoft Developer's Studio with the precompiled header option turned on.

Generate placeholder definitions for unused variables
If you check this box, DTREG will generate variable definitions for variables that are not used by the model. This makes it possible to select which variables are used as predictors without having to modify the application program that sets up values for all variables.

How to call the scoring function – C and C++ programs

The generated code will consist of one or more .c source files and a .h header file. The header file will contain prototypes for the generated functions and for the global variables. You must include the generated header file in the source modules of your application program that call the generated scoring function.

Generated header file

The values for predictor variables must be set in global variables prior to calling the function to perform scoring. There will be one global predictor variable for each predictor variable specified in the model. The generated .h header file contains external references to these variables. Here is an example header file:

```
#ifndef Iris_h
#define Iris_h

/*----------------------------------------------------------------
 *  Scoring header file generated by DTREG (http://www.dtreg.com)
 *  This header file should be included in applications calling the
 *  generated code.
 *  DTREG version 3.5
 *  Creation date: 21-Oct-2004 14:01:32
 *  Project file: C:\DTREG\Test\Iris.dtr
 *  Project title: Iris variety classification
 */

/*
 *  Type of model.
 */
#define MODELTYPE_TREEBOOST
/*
 *  Values used to represent missing values.
 */
extern double Missing_Continuous;   /* Continuous variables */
extern char *Missing_Category;            /* Categorical variables */
extern long Missing_Index;          /* Category index */
/*
 *  Predictor variables.
 */
/* Continuous variables */
extern double Sepal_length;
extern double Sepal_width;
extern double Petal_length;
extern double Petal_width;
```

```
/*
 *  Variable that will receive predicted value of Species.
 */
extern char PredictedValue[200];    /* Gets computed category /*
 *  Variables that will receive probability values for the
 *  categories of Species.
 */
extern double Prob_Setosa;
extern double Prob_Versicolor;
extern double Prob_Virginica;/*
*/
 *  Function prototypes.
 */
void ScoreRecord(void);

/*
 *  End of header.
 */
#endif
```

Type of model
The type of model will be defined by one of the following macros: MODELTYPE_SINGLETREE, MODELTYPE_TREEBOOST or MODELTYPE_FOREST. You can use #ifdef macros in your application program to determine which type of model was generated.

Values used to represent missing values
If you turn on the option to generate code to handle missing values, DTREG will generate references to Missing_Continuous and Missing_Cateogory. These global variables have the values that you should use to represent missing values of predictor variables.

Predictor variables
There will be an external reference to each predictor variable. If the predictor variables were specified with spaces in their names, the spaces will be converted to underscores in the generated code. Continuous predictor variables are of type double, and categorical predictor variables are of type char[200]. Note that categorical variables must be specified as character string values even if all of the values are numeric. If, for example, you had a predictor variable named sexcode that had values 1 and 2, you could use the sprintf function to format the value into the global variable:

```
    sprintf(sex,"%d",sexcode);
```

Predicted target variable
The predicted value computed by the scoring function will be returned in a global variable named PredictedValue. If the target variable is continuous, then PredictedValue will be of type double. If the target variable is categorical, then PredictedValue will be a char[200] variable. If the target variable has numeric categorical values, you can use the atol() function to convert the returned string to a long integer value.

Predicted category probabilities
DTREG will generate code to create variables that will have the probability for each category of the target variable. These variables are named Prob_*category* where *category* is the name of the category of the target variable.

Prototype for the scoring function
The function called to compute the score is named ScoreRecord. Its prototype is as follows:

```
void ScoreRecord(void);
```

Note that there are no arguments and no returned value because the predictor variable values are set in global variables before it is called, and the predicted target variable value is returned in a global variable.

Here is an outline of the procedure you should use in your application program:
1. Read values for the case you want to score.
2. Set the values of the global predictor variables.
3. Call the generated ScoreRecord() function.
4. Get the predicted target value from the PredictedValue global variable.

Generated Source File

Usually, it will not be necessary for you to edit or be concerned with the contents of the generated .c source file. You can simply compile it as a module of your application. If you turn on the option to split the source into multiple files, then you must compile each generated source file as a separate source module.

The top of a generated source file will be similar to this:

```
/*-----------------------------------------------------------------
 *  Scoring source file generated by DTREG (http://www.dtreg.com)
 *  DTREG version 3.5
 *  Creation date: 21-Oct-2004 14:44:09
 *  Project file: C:\DTREG\Test\Iris.dtr
 *  Project title: Iris variety classification
 *  Model type: Single-tree
 *  Depth of tree: 5
 *  Number of terminal nodes: 5
 *  Target variable: Species
 *  Type of analysis: Classification with 3 target classes
 */

#include <string.h>
#include <math.h>
```

```
/*
 *  Values used to represent missing values.
 */
double Missing_Continuous = -1e+035;      /* Continuous variables */
char *Missing_Category = "?";       /* Categorical variables */
long Missing_Index = -1;            /* Category index */
/*
 *  Global definitions of predictor variables.
 */
/* Continuous variables */
double Sepal_length = -1e+035;
double Sepal_width = -1e+035;
double Petal_length = -1e+035;
double Petal_width = -1e+035;
/*
 *  Define variable that will receive predicted value of Species.
 */
char PredictedValue[200] = {0};     /* Gets predicted category */

/*---------------------------------------------------------------
 *  Call this routine to compute the predicted value.
 */
void ScoreRecord(void)
{
```

How to call the scoring function – SAS® programs

SAS source code generated by DTREG consists of two parts, a header file named "*program*_header.sas" and the model evaluation code named "*program*.sas". These files should be included in the DATA proc of the program that is doing the scoring. The best way to include the files is to use the SAS %INCLUDE facility to insert the header file at the top and the evaluation code a the end after a RETURN statement. Here is the outline of a DATA proc doing this:

```
Data Titanic;
/* Include the generated header file */
%INCLUDE 'Titanic_Header.sas';

/* your statements to set up values for scoring */

length classc $1 agec $1 sexc $1;
classc = left(put(class,best12.));
agec = left(put(age,best12.));
sexc = left(put(sex,best12.));

/*
 * Use LINK to call the scoring code.
 * It will return to the statement after LINK.
 * The predicted value will be in _PredictedValue_.

LINK ScoreRecord;

/* Your statements to process the predicted value.
 * For example:
 */

DidSurvive = _PredictedValue_;

/* Output the values and return */

RETURN;

/* Put the generated scoring code here */

%INCLUDE 'Titanic.sas';
```

Data types of variables

SAS has two types of variables, numeric and character string. If the "Character" attribute is set for a variable on the variable property page (see page 41) then DTREG generates SAS code to treat the variable as a character string. Otherwise, the generated code treats the variable as a numeric value.

Generated header file

Here is an example header file:

```
/*-----------------------------------------------------------------
 *  Scoring header file generated by DTREG (http://www.dtreg.com)
 *  This header file should be included in applications calling
 * the generated code.
 *  DTREG version 4.5
 *  Creation date: 10-Nov-2005 15:01:50
 *  Project file: C:\DTREG\Test\iris.dtr
 *  Project title: Iris variety classification
 *
 *  To score a record use the statement: LINK ScoreRecord;
 *
 *  On return, the predicted value for 'Species' will be
 *  in '_PredictedValue_'.
 *  The predicted value is returned as a character string.
 *  The terminal nodel number is returned in '_Node_'.
 */

/*
 *  Declare variables.
 */
    _ModelType_ = 1;    /* Single tree */
    length _PredictedValue_ $10;
    _PredictedValue_ = '?';
    _Node_ = 0;
/*
 *  ---  End of scoring header  ---
 */
```

Type of model

The _ModelType_ variable has a value indicating what type of model was built.

Predicted target variable

The predicted value computed by the scoring function will be returned in a variable named _PredictedValue_. If the variable was declared to be of type character, then _PredictedValue_ will be declared as a character string; otherwise, it will be a numeric variable.

Terminal node number

For single-tree models, the terminal node in which a record ends is returned in the _Node_ variable. This variable is not generated for other types of models.

Predicted category probabilities

If scoring code is generated for a categorical model, there will be variables that will have the probability for each category of the target variable. These variables are named Prob_*category* where *category* is the name of the category of the target variable.

Generated Model Execution Source File

Usually, it will not be necessary for you to edit or be concerned with the contents of the generated *program*.sas source file. You can simply use a %INCLUDE statement to insert into the end of the DATA proc.

To score a record, use this statement to call the scoring code:

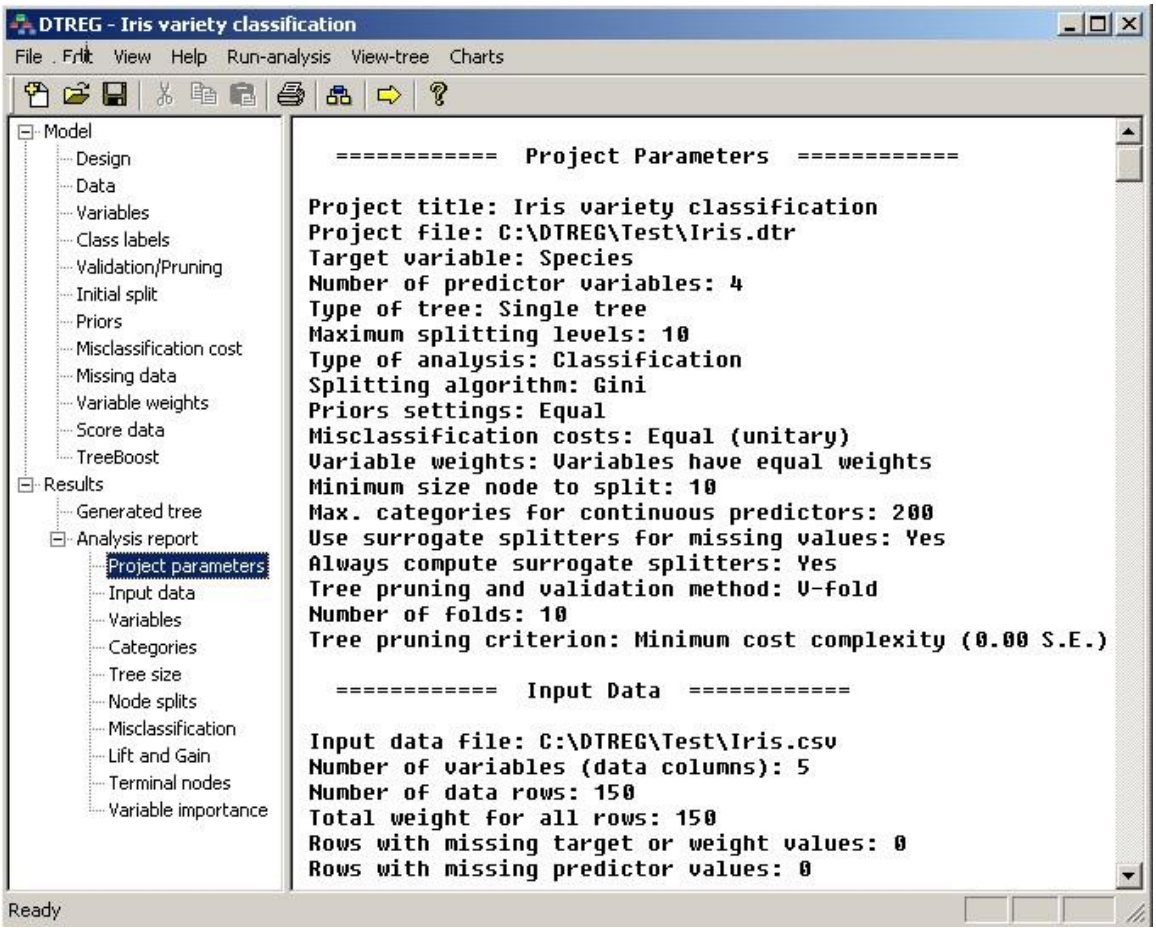
```
LINK ScoreRecord;
```

The LINK statement jumps to the ScoreRecord label in the generated code much as a GOTO statement would do. When the generated code finishes computing the predicted value, it uses a RETURN statement to return execution control to the line following the LINK statement. You can then do whatever processing is appropriate for the predicted value and then use a RETURN statement to terminate the DATA proc execution and write the record to the output dataset.

The predicted value computed by the scoring code is returned in a variable named _PredictedValue_. It will be either a character string value or a numeric value depending on whether the target variable was declared to be character or numeric.

The Output Report Generated by DTREG

Once you run an analysis, DTREG will display in the main right panel a report of the results.



There are several major sections in the report. You can use the scroll bar to move to sections, or you can click on of the section names shown under "Analysis report" in the left panel to scroll instantly to a section.

Project Parameters

```
  ============  Project Parameters  ============

Project title: Iris variety classification
Project file: C:\DTREG\Test\iris.dtr
Target variable: Species
Number of predictor variables: 4
Type of tree: Single tree
Maximum splitting levels: 10
Type of analysis: Classification
Splitting algorithm: Gini
Category weights: Equal (Balanced)
Misclassification costs: Equal (unitary)
Variable weights: Equal
Minimum size node to split: 10
Max. categories for continuous predictors: 200
Use surrogate splitters for missing values: Yes
Always compute surrogate splitters: Yes
Tree pruning and validation method: V-fold
Number of folds: 10
Tree pruning criterion: Minimum cost complexity (0.00 S.E.)
```

The Project Parameters section of the report displays a summary of the options and parameters you selected on the various property pages for the model.

Input Data

```
  ============  Input Data  ============

Input data file: C:\DTREG\iris.csv
Number of variables (data columns): 5
Number of data rows: 150
Total weight for all rows: 150
Rows with missing target or weight values: 0
Rows with missing predictor values: 0
```

The Input Data section displays information about the input data file used to construct the tree. The entry for "Rows with missing target or weight values" indicates the number of rows that were discarded because these variables had missing values.

Summary of Variables

```
  ============  Summary of Variables  ============

  Variable      Class       Type      Missing rows  Categories
------------  ---------  -----------  ------------  ----------
Species       Target     Categorical          0           3
Sepal length  Predictor  Continuous           0          35
Sepal width   Predictor  Continuous           0          23
Petal length  Predictor  Continuous           0          43
Petal width   Predictor  Continuous           0          22
```

The Summary of Variables section displays information about each variable that was present in the input dataset. The first column shows the name of the variable, the second column shows how the variable was used; the possibilities are Target, Predictor, Weight and Unused. The third column shows whether the variable is categorical or continuous, the forth column shows how many data rows had missing values on the variable, and the fifth column shows how many categories (discrete values) the variable has. In the case of continuous variables, the number of categories will be limited by the value specified for "Max. categories for predictor variables" on the model design property page.

Summary of Categories

```
  ============  Summary of Categories  ============

    ---  Predictors  ---      |  ---  Target Variable  ---

Class                         |     No      |     Yes     |
     885   40.21%  Crew       |  673   76%  |  212   24%  |
     325   14.77%  First      |  122   38%  |  203   62%  |
     285   12.95%  Second     |  167   59%  |  118   41%  |
     706   32.08%  Third      |  528   75%  |  178   25%  |

Age                           |     No      |     Yes     |
    2092   95.05%  Adult      | 1438   69%  |  654   31%  |
     109    4.95%  Child      |   52   48%  |   57   52%  |

Sex                           |     No      |     Yes     |
     470   21.35%  Female     |  126   27%  |  344   73%  |
    1731   78.65%  Male       | 1364   79%  |  367   21%  |

Survived                      |     No      |     Yes     |
    1490   67.70%  No         | 1490  100%  |    0    0%  |
     711   32.30%  Yes        |    0    0%  |  711  100%  |
```

The Summary of Categories section displays information about the categories of predictor and target variables. This section is only displayed if you select one or both of the options on the Variables property page requesting category information (see page 41).

Several items of information are displayed for each category:

1. The number of rows in the dataset having the category for the variable.
2. The percent of the rows having the category.
3. The label of the category.
4. If the target variable is categorical, a table showing the distribution of target categories for the predictor category.
5. If the target variable is continuous, the mean value of the target for the predictor category.

Surrogate Variable Report

If surrogate variables are used to impute missing predictor values, then a section is included in the report for the surrogate variables. See page 358 for information about surrogate variables. Here is an example of a surrogate variable report:

```
  ============  Surrogate Variables  ============

Predictor  #  Surrogate Association  Constant   Coeff. 1    Coeff. 2    Coeff. 3
--------- --- --------- ----------- ---------- ----------- ----------- -----------
  Cat2{0}   1     C1{0}          84     0.0000      1.0000           .           .
            2     C2{0}          82     1.0000     -1.0000           .           .
            3        X1          45     0.4393      0.0256           .           .
            4        X2          39     0.3525      0.0277      0.0002 -3.467e-005
       X1   1        X2         100    -1.3338      0.6646           .           .
            2     C1{0}          19    -3.3996      7.1541           .           .
            3   Cat2{0}          17    -2.8827      6.7654           .           .
            4     C2{0}          11     2.9238     -5.4648           .           .
       X2   1        X1         100     2.0076      1.5017           .           .
            2     C1{0}          17    -2.7628     10.3787           .           .
            3   Cat2{0}          15    -1.6381      9.6841           .           .
    C1{0}   1   Cat2{0}          84     0.0000      1.0000           .           .
            2     C2{0}          66     1.0000     -1.0000           .           .
            3        X1          51     0.4762      0.0499      0.0004     -0.0001
            4        X2          49     0.4591      0.0169           .           .
    C2{0}   1   Cat2{0}          82     1.0000     -1.0000           .           .
            2     C1{0}          66     1.0000     -1.0000           .           .
            3        X1          38     0.4966     -0.0205           .           .
            4        X2          37     0.5398     -0.0232     -0.0002  2.903e-005
```

These are the columns in the table:

Predictor – This is the primary predictor for which surrogates are associated.

– This is a sequential number showing which surrogate this line is for. The surrogate variables are listed in decreasing order of association with the primary variable.

Surrogate – This is the name of the surrogate variable.

Association – This is the association between the surrogate variable and the primary variable.

Constant – This is the constant term in the function.

Coeff 1, Coeff 2, Coeff 3 – These are the coefficients of the first, second, and third-order polynomial terms.

Tree Size and Validation Statistics

This section of the report is composed of two sub-sections: Tree Size Summary Report and Validation Statistics Report.

```
  ============  Tree Size Summary Report  ============

The full tree has 5 terminal (leaf) nodes.
The minimal cross-validated relative error occurs with 3 nodes.
The relative error value is 0.0700 with a standard error of 0.0257
You allowed up to 1 standard error for tree size reduction.
With 1.000 S.E. allowance, the optimal tree has 3 nodes.
The tree will be pruned from 5 to 3 terminal nodes.
```

The **Tree Size Summary Report** displays information about the maximum size tree that was built, and it shows summary information about the parameters that were used to prune the tree.

```
  ============  Tree Size Summary Report  ============

The full tree has 5 terminal (leaf) nodes.
The minimum validation relative error occurs with 5 nodes.
The relative error value is 0.0700 with a standard error of 0.0280
You allowed up to 1 standard error for tree size reduction.
With 1 S.E. allowance, the optimal tree has 3 nodes.
The tree will be pruned from 5 to 3 nodes.

   ------  Validation Statistics  ------

Nodes  Val cost  Val std. err.  RS cost    Complexity
-----  --------  -------------  -------  --------------
    5   0.0700        0.0280     0.0300      0.000000 <-- Min.error
    4   0.0800        0.0297     0.0400      0.006667
    3   0.0700        0.0257     0.0600      0.013333 <-- Pruned size
    2   0.5000        0.0000     0.5000      0.293333
    1   1.0000        0.0000     1.0000      0.333333
```

In order to create a tree that can be generalized to predict data values other than those in the learning dataset, DTREG builds an overly-large tree and then prunes it to the optimal size. For information about how pruning is done, please see page 366.

The **Validation Statistics** section displays information about the size of the generated tree and statistics used to prune the tree. There are five columns in the table:

1. **Nodes** – This is the number of terminal nodes in a particular pruned version of the tree. It will range from 1 up to the maximum nodes in the largest tree that was generated. The maximum number of nodes will be limited by the maximum

depth of the tree and the minimum node size allowed to be split on the Design property page for the model.

2. **Val cost** – This is the validation cost of the tree pruned to the reported number of nodes. It is the error cost computed using either cross-validation or the random-row-holdback data. The displayed cost value is the cost relative to the cost for a tree with one node. See page 369 for a detailed description of the cross-validation procedure. The validation cost is the best measure of how well the tree will fit an independent dataset different from the learning dataset. The pruned size with the minimum validation cost is marked with "←Min. validation error" in the margin. Note, if you enable DTREG to smooth the minimum values by checking the box labeled "Smooth minimum spikes" on the Validation and Pruning property page (see page **Error! Bookmark not defined.**), then the minimum value selected may not be the absolute minimum.
3. **Val std. err.** – This is the standard error of the validation cost value. If you wish, you can allow DTREG to prune to a smaller tree with a larger validation cost value than the absolute minimum by using a multiple of the validation cost standard error. See page 371 for information about controlling the pruning point. If you allow DTREG to prune the tree to a smaller size than the minimum validation cost size, the pruned size will be indicated by "←Pruned size" in the report.
4. **RS cost** – This is the resubstitution cost computed by running the learning dataset through the tree. The displayed resubstitution cost is scaled relative to the cost for a tree with one node. Since the tree is being evaluated by the same data that was used to build it, the resubstitution cost does not give an honest estimate of the predictive accuracy of the tree for other data.
5. **Complexity** – This is a "*Cost Complexity*" measure that shows the relative tradeoff between the number of terminal nodes and the misclassification cost. See Breiman, Friedman, Olshen and Stone (1984) for information about the calculation and use of the cost complexity measure.

Node Splits

The node splits section provides information about each node in the tree and how it was split to produce its child nodes. This section of the report is generated only if you check the box labeled "Generate report of tree splits" on the Single Tree property page (see page 51).

There are five subsections: (1) the node summary, (2) the distribution of categories of the target variable in the group; (3) splitting information; (4) competitor splits; (5) surrogate splits.

Node Summary

```
 ======================   Group 1   ======================
Number of rows in group: 149
Sum of weights for all rows: 149
Rows with missing values on the splitting variable: 37
Rows with missing values classified using surrogates: 37
Rows with missing values classified using target values: 0
Rows with missing values classified into most probable group: 0
Rows with missing values that stop in this node: 0
Improvement in misclassification from split: 0.251146
Complexity: 0.161633
Category of Species assigned to group: Versicolor
Misclassified rows = 66.44%
Misclassification cost = 0.6667
```

This section provides information about the node:

- **Number of rows in group** – This is the total number of rows that made it through the tree to this node.
- **Sum of weights for all rows** – This is the sum of the weights for all rows that made it to this node. If you did not specify a weight variable, all rows get a weight of 1.00, and the sum of the weights will equal the number of rows.
- **Rows with missing values on the splitting variable** – This is a count of how many rows in this node had missing values on the variable that DTREG selected to split the node. The counts that appear on the following lines show how these rows were classified.
- **Rows with missing values classified using surrogates** – This is a count of the rows that had missing values on the primary splitting variable that DTREG was able to classify using surrogate splitting variables. See the list of surrogate splitters that appears later in the node report.
- **Rows with missing values classified using target values** – This is the number of rows that could not be classified using surrogate splitters but instead were forced into the appropriate child group based on the actual value of their target variable. When the target variable is categorical, this method of assignment is used only if the actual target category for the row matches the target category assigned to one of the child rows. If the target variable is continuous (i.e., a regression tree is

being built), then the row is put in the child group whose mean value on the target variable is closest to the row's target variable value.

- **Rows with missing values classified into the most probable group** – This is the number of rows that could not be classified by either of the two methods listed above but rather were dumped into the child group that is the most probable group to receive a random row without consideration of any predictor variables. Usually, this is the child group with the most number of rows, but it could be the smaller group depending on category weight values and other factors.
- **Rows with missing values that stop in this node** – This is the number of rows with missing values on the splitting variable that could not be classified by any means, so they stopped in this node as their terminal node.

Target Category Distribution

```
  -- Distribution of categories of target variable in group --

   Category    Num. Rows   Total Weight   Category Wt.
  ----------   ---------   ------------   ------------
      Setosa *        50             50         0.3333
  Versicolor          50             50         0.3333
   Virginica          50             50         0.3333
```

If the target variable is categorical, the next section of the node report is a table showing information about the categories of the target variable occurring in the node. For each category, the table displays the category name, the number of rows with that category in the node, the total weight of the rows, and the weight that was assigned to the category. This table is not displayed if the target variable is continuous.

Node Split Information

```
--  Group 1 was split on Petal length  --

Left child group = 2.  Number of rows = 49
  A case goes left if Petal length <= 2.35

Right child group = 3.  Number of rows = 100
  A case goes right if Petal length > 2.35
```

This section displays information about how the node was split. The first line gives the number of the node being split and the name of the predictor variable that was selected as the splitting variable.

The next two parts of this section display information about the left and right child nodes generated by the split. For each child node, the number of the node is displayed along with the number of rows that were assigned to that node. In the example above, the parent node is number 1. It is split into two child nodes; the left node is number 2 and the right node is number 3.

The condition that controls whether rows are sent to the left or right node is displayed. In the example above, a row is sent to the left child node if its value on the "Petal length" predictor variable is less than or equal to 2.35. The row is sent to the right node if the value of "Petal length" is greater than 2.35.

If the predictor variable used for the split is categorical, the categories of the variable being sent to the left and right child nodes are listed. Here is an example:

```
Left child group = 2.  Number of rows = 17800
  Categories of Relationship going left: {Not-in-family,
                   Other-relative, Own-child, Unmarried}

Right child group = 3.  Number of rows = 14761
  Categories of Relationship going right: {Husband, Wife}
```

In this example, the split is being made using the "Relationship" predictor variable. Rows with values of "Not-in-family", "Other-relative", "Own-child" and "Unmarried" are sent to the left child group. Rows with values of "Husband" or "Wife" are sent to the right child group.

Competitor Predictor Variables

```
--  Competitor Splits  --

Order    Variable     Improvement   Left Categories
-----  ------------  -------------  ---------------
  1    Petal width           0.247  <= 0.8
  2    Sepal length          0.227  <= 5.45
  3    Sepal width           0.124  <= 3.35
```

For each node being split, DTREG examines all predictor variables and performs the split using the one that provides the greatest improvement. The competitor split table lists up to five predictor variables that were the runners-up splitters. They are listed in decreasing order of improvement.

Surrogate Splitters

```
--  Surrogate Splits  --

Order    Variable    Assoc  Dir   Improvement   Left Categories
-----  ------------  -----  ---  -------------  ---------------
  1    Petal width   0.748   +           0.247  <= 0.8
  2    Sepal length  0.665   +           0.227  <= 5.45
  3    Sepal width   0.427   -           0.115  <= 3.25
```

A surrogate splitter is a predictor variable that mimics the split performed by the primary splitter. That is, it sends the same rows to the left and right child groups as the primary

splitter. Surrogate splitters are used to classify rows when the value of the primary splitter is missing. For detailed information about surrogate splitters, please see page 364.

The following information is shown for each surrogate splitter:

- **Order** – This is the order of the surrogate splitters in decreasing order of association. When attempting to classify a row that has a missing value for the primary splitter, DTREG will try the surrogate splitters in the order shown until it finds one that has a non-missing value for the row.
- **Variable** – This is the name of the predictor variable that will be used for the surrogate split.
- **Association** – This is a measure of how well the surrogate split mimics the primary split. The largest possible association value is 1.0 which means the surrogate sends exactly the same set of rows to the left and right groups as the primary splitter. An association value of 0.0 means that the surrogate does no better at assigning rows than simply putting them in the most probable group.
- **Direction** – This indicates whether the split generated by the surrogate splitter assigns rows to the same or opposite child group as the primary splitter. This is roughly equivalent to variables that have a negative correlation – you can predict the value of one by going in the opposite direction on the other.
- **Improvement** – This is the improvement in misclassification that would be gained by using the surrogate split. Note that surrogate splits are not ranked by improvement but rather by association with the primary splitter.
- **Left categories** – This shows what values of the surrogate predictor send rows to the left child group. The other values of the predictor send rows to the right child group.

Note that if a predictor is listed as both a competitor and as a surrogate, the split categories and improvement values may be different. The reason for this is that when evaluated as a competitor, the split point is chosen so as to maximize the improvement, just as is done for the primary splitter. But when evaluated as a surrogate, the split point is chosen *not* to maximize the improvement, but rather to maximize the *association* between the surrogate and the primary splitter.

Analysis of Variance

The analysis of variance summary table is displayed when the target variable is continuous and a regression tree is being constructed. The variance explained by the generated tree is the best measure of how well the tree fits the data.

```
  ============  Analysis of Variance  ============

Variance in initial data sample = 84.419556
Residual (unexplained) variance after tree fitting = 7.806666
Proportion of variance explained = 0.90753  (90.753%)
Correlation between actual and predicted = 0.999610
```

The following items are displayed in the summary:

- **Variance in initial data sample** – This is the variance in the entire learning dataset before any splits have been made. The following algorithm is used to compute variance: (1) Compute the mean value of the target variable for all rows. (2) For each row, subtract the row's target value from the mean target value, square the difference and sum the squared differences. The difference between the target value of a row and the mean value of the target value is called the *residual* value for the row. The sum of the squared residuals is the *variance*.
- **Residual (unexplained) variance after tree fitting** – This is the remaining variance after the tree is applied to the data to predict the target values. This is computed by (1) computing the mean value of the target variable for all rows in a terminal node; (2) use this mean to compute the residual for each row in the node; (3) add the residuals to compute the variance within the node; (4) add the variance for all nodes. If the tree perfectly predicted the dataset, the residual variance would be 0.0.
- **Proportion of variance explained** – This is the proportion of the initial, total variance explained by the fitted tree. The larger the value, the better the tree fits and explains the data. If the tree perfectly fitted the data and exactly predicted the target value for every row, the explained variance proportion would be 1.0 (100%).
- C**orrelation between actual and predicted** – This is the Pearson correlation coefficient between the actual values and the predicted values; it measures whether the actual and predicted values move in the same direction. The possible range of values is -1 to +1. A positive correlation means that the actual and predicted values generally move in the same direction. A correlation of +1 means that the actual and predicted values are synchronized; this is the ideal case. A negative correlation means that the actual and predicted values move in opposite directions. A correlation near zero means that the predicted values are no better than random guesses.

Misclassification Summary Table

If the target variable is categorical and you are building a classification tree, then a misclassification summary table is displayed.

```
 ============  Misclassification Tables  ============

---  Training Data  ---

             --------Actual--------  -------------Misclassified-------------
  Category     Count      Weight       Count      Weight     Percent   Cost
 ----------  --------  ------------  --------  ------------  -------  ------
     Setosa        50            50         0             0    0.000   0.000
 Versicolor        50            50         3             3    6.000   0.060
  Virginica        50            50         0             0    0.000   0.000
 ----------  --------  ------------  --------  ------------  -------  ------
      Total       150           150         3             3    2.000   0.020

---  Validation Data  ---

             --------Actual--------  -------------Misclassified-------------
  Category     Count      Weight       Count      Weight     Percent   Cost
 ----------  --------  ------------  --------  ------------  -------  ------
     Setosa        50            50         0             0    0.000   0.000
 Versicolor        50            50         2             2    4.000   0.040
  Virginica        50            50         5             5   10.000   0.100
 ----------  --------  ------------  --------  ------------  -------  ------
      Total       150           150         7             7    4.667   0.047
```

There are two sections to the table – one for the misclassifications for the training dataset and one for the misclassification for the validation data (either cross-validation or random-holdback rows). See page 369 for information about how cross-validation is done.

Each category of the target variable is listed along with the following items of information:

- **Category** – The target category.
- **Actual count** – The number of rows that have this target category.
- **Actual weight** – The sum of the weights for the rows with this category.
- **Misclassified count** – The number of rows with this category that were misclassified by the tree.
- **Misclassified weight** – The sum of the weights for the rows with this category that were misclassified.
- **Misclassified percent** – The percent of the rows with this category that were misclassified.
- **Cost** – The misclassification cost for the rows with this category.

Confusion Matrix

A “Confusion Matrix” provides detailed information about how data rows are classified by the model. The matrix has a row and column for each category of the target variable. The categories shown in the first column are the actual categories of the target variable. The categories shown across the top of the table are the predicted categories. The numbers in the cells are the weights of the data rows with the actual category of the row and the predicted category of the column. Here is an example confusion matrix:

```
  ============  Confusion Matrix  ============

   --------  Training Data  --------

  Actual   : -------Predicted Category-------
 Category :   Setosa   Versicolor  Virginica
----------: ---------- ---------- ----------
    Setosa:         50          0          0
Versicolor:          0         47          3
 Virginica:          0          0         50
```

The numbers in the diagonal cells are the weights for the correctly classified cases where the actual category matches the predicted category. The off-diagonal cells have misclassified row weights. For example, the Versicolor category was misclassified as Virginica three times.

Sensitivity and Specificity Report

The Sensitivity and Specificity report is generated only for classification problems (categorical target variable). One category of the target variable is called the "positive" category, and the other is called the "negative" category. It is up to you to decide which category is positive and which is negative. You select the positive category on the Misclassification Property Page (see page 130). For example, if you are creating a model to predict if a patient has a disease, you would probably want to select the Disease category as the positive category and the No-Disease category as the negative category.

If the target variable has more than two categories, DTREG reports the sensitivity and specificity for each category. The selected category is treated as the positive category, and all other categories are grouped as the negative category.

In a medical context, an ideal diagnostic test would identify all patients with a suspected disease, and it would not falsely identify anyone who did not have the disease. Thus there are two types of errors: (1) failing to identify someone with the disease and (2) incorrectly identifying someone who does not have the disease. These errors are reported in the Confusion Matrix (see page 191) which shows the true positive (TP), true negative (TN), false positive (FP) and false negative (FN) counts. If a predicted value is 1 (true) and the actual class is also 1, then a TP prediction is counted. Similarly true negative (TN) predictions occur when both classes are 0. False positive and false negative predictions occur as shown in the following table:

Actual class	*Predicted class*	
	True	False
True	**TP**	**FN**
False	**FP**	**TN**

The ***sensitivity*** of a test is the proportion of the people with the disease who are identified by the test. The ***specificity*** of the test is the proportion of the people who do not have the disease who are correctly identified as being disease-free by the test. Ideally, sensitivity and specificity would both be 1.0.

Positive Predictive Value (PPV) is the proportion of patients with the disease who are correctly predicted to have the disease. The PPV value for a perfect model would be 1.0.

Negative Predictive Value (NPV) is the proportion of patients who do not have the disease who are correctly predicted as not having the disease. The NPV value for a perfect model would be 1.0.

Precision and Recall – These terms are most commonly used in applications related to information lookup. *Precision* is the proportion of cases selected by the model that have

the true value; precision is equal to PPV. *Recall* is the proportion of the true cases that are identified by the model; recall is equal to sensitivity.

F-Measure is the harmonic mean of precision and recall. It combines precision and recall to give an overall measure of the quality of the prediction.

Using the definitions of *TP*, *TN*, *FP* and *FN* given above, these statistics are calculated using these formulas:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$sensitivity = \frac{TP}{TP + FN}$$

$$specificity = \frac{TN}{TN + FP}$$

$$PPV = \frac{TP}{TP + FP}$$

$$NPV = \frac{TN}{TN + FN}$$

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Area under ROC curve (AUC) – This is the area under the Receive Operating Characteristic (ROC) curve for the model. This statistic is also called the "C-Statistic". The closer the value of the area is to 1.0, the better the model is. See page 215 for more information about ROC curves.

Here is an example of a sensitivity and specificity report:

```
 ============  Sensitivity & Specificity  ============

 Positive: Survived = 1  (Yes)
 Negative: Survived = 0  (No)

--------  Training Data  --------

 Accuracy = 78.33%
 Sensitivity = 50.63%
 Specificity = 91.54%
 Geometric mean of sensitivity and specificity = 68.08%
 Positive Predictive Value (PPV) = 74.07%
 Negative Predictive Value (NPV) = 79.53%
 Geometric mean of PPV and NPV = 76.76%
 Precision = 74.07%
 Recall = 50.63%
 F-Measure = 0.6015
 Area under ROC curve (AUC) = 0.768330

--------  Validation Data  --------

 Accuracy = 76.06%
 Sensitivity = 54.43%
 Specificity = 86.38%
 Geometric mean of sensitivity and specificity = 68.57%
 Positive Predictive Value (PPV) = 65.59%
 Negative Predictive Value (NPV) = 79.89%
 Geometric mean of PPV and NPV = 72.39%
 Precision = 65.59%
 Recall = 54.43%
 F-Measure = 0.5949
 Area under ROC curve (AUC) = 0.758716
```

Note that the first section of the report shows which target category DTREG is using as the "positive" category and which for the "negative" category. If it selects the wrong category, you can specify the positive category on the Misclassification Property page (see page 130).

The probability threshold used to classify predicted targets as positive or negative is shown next. The probability threshold can be specified on the Misclassification Property page (see page 130). The Sensitivity and Specificity Chart (see page 217) shows how sensitivity and specificity are changed as the probability threshold is shifted.

Probability Calibration Report

The Probability Calibration Report shows how the predicted probability of a target category is distributed and provides a means for gauging the accuracy of predicted probabilities. The probability calibration report is generated only when a classification analysis is performed and there are two target categories. Here is an example of a probability calibration report:

```
 -----   Probability calibration for Has diabetes = 1   -----

 Predicted  -------- Training Data ---------  -------- Validation Data --------
 Prob Range  # Rows  % Rows Predicted Actual   # Rows   % Rows Predicted Actual
----------- -------- ------ --------- ------  -------- ------ ---------- ------
0.00 - 0.10      186  24.22   0.0415  0.0054       172  22.40    0.0439  0.0581
0.10 - 0.20      165  21.48   0.1481  0.0788       144  18.75    0.1490  0.1250
0.20 - 0.30       86  11.20   0.2448  0.1860       109  14.19    0.2476  0.2569
0.30 - 0.40       63   8.20   0.3458  0.3333        69   8.98    0.3518  0.4058
0.40 - 0.50       58   7.55   0.4460  0.4483        78  10.16    0.4541  0.4872
0.50 - 0.60       46   5.99   0.5451  0.7391        54   7.03    0.5440  0.5556
0.60 - 0.70       30   3.91   0.6460  0.8667        39   5.08    0.6463  0.8462
0.70 - 0.80       51   6.64   0.7516  0.9412        48   6.25    0.7442  0.7708
0.80 - 0.90       36   4.69   0.8552  1.0000        36   4.69    0.8468  0.8333
0.90 - 1.00       47   6.12   0.9543  1.0000        19   2.47    0.9533  0.8421

  Average weighted probability error for training data = 0.073822
  Average weighted squared probability error for training data = 0.096968
  Average weighted probability error for validation data = 0.033229
  Average weighted squared probability error for validation data = 0.054171
```

There is one probability report table for each category of the target variable. The table shown above is for the prediction that "Diabetes = Yes".

The table has one line for each 0.1 range of predicted probability scores (0.00 to 0.10, 0.10 to 0.20, etc.). Each case in the training or validation data set is assigned to a range based on the predicted probability that the target category of the case matches the category of the table (Diabetes = Yes for this example).

Here are the columns in the table:

Predicted Probability Range – This describes a range of calculated probability values. A data case is assigned to a band based on the predicted probability that the case has the category of the table (Diabetes=Yes in this example).

 # Rows – This is the number of rows in this probability range. If a row weight variable is used, the "count" is actually the sum of the row weights.

% Rows – This is the percent of the total rows that had computed probabilities in this range.

Predicted – This is the average predicted probability for all of the data rows that had predicted probabilities in the range. Usually the average predicted probability will be about the midpoint of the range.

Actual – This is the actual proportion of the rows that had the target category for the table.

If the model is accurate, the predicted probability of an event occurring should match the actual proportion of times that the event occurs. The Probability Calibration Report provides a breakdown that allows you to make that comparison. For example, look at the second line in the table above for cases that had predicted probability scores in the range 0.10 to 0.20. There were 165 such rows which correspond to 21.48% of the total data rows. The average predicted probability for those rows is 0.1481 which is about the midpoint of the range. If the model is perfect, we would expect the actual proportion of these cases to be 0.1481. However, from the "Actual" column we see that the actual proportion is 0.0788. From this we conclude that the predicted probability for this range tends to overestimate the frequency of actual occurrence. Using this information, it is possible to develop a probability calibration correction function that maps predicted probabilities to more accurate estimates of actual probabilities.

Average weighted probability error for training data – This is the average error between the predicted probability and the actual occurrence rate weighted by the number of rows that fall in each bin. For example, for the second line of the table above describing the 0.10 to 0.2 probability range, the error was (0.1481-0.0788), and since there were 21.48% of the total rows in that range, that error contributes 21.48% of the total average error.

Average weighted squared probability error for training data – This is computed in the same way as the average error described above, except that the error is squared before being multiplied by the weight and added into the sum. After the total squared error is added up, the square root is computed, and that is the result reported for this statistic.

For problems where probability estimates are important – rather than just overall classification accuracy – the average and squared average error values are excellent overall indicators of the quality of the model.

A graphical representation of probability calibration is presented in the Probability Calibration Chart which is described on page 227.

Probability Threshold Report

The probability threshold report provides information about how different probability thresholds would affect target category assignments. The threshold report provides a convenient way to see the tradeoff between impurity and loss as the probability threshold is varied. The probability threshold report is generated only when a classification analysis is performed and there are two target categories. A graphical depiction of the probability threshold response is available in the Probability Threshold Chart described on page 223.

All classification models not only predict a specific category for each case but also generate posterior probability scores that indicate the relative likelihood for each possible category. Support Vector Machine (SVM) models can generate probability estimates if you enable the appropriate option on the SVM property page.

Usually the category with the highest probability is selected as the predicted category. In other words, the probability threshold is set at 0.5. You can specify a probability threshold to control classifications on the Misclassification Cost Property Page described on page 130.

Here is an example of a probability threshold report:

```
 -----   Threshold analysis for Liver condition = 2   -----

Probability  Proportion  Error   Impurity   Loss
-----------  ----------  ------  --------  ------
    0.00       1.0000    0.4203   0.4203   0.0000
    0.05       0.9985    0.4188   0.4194   0.0000
    0.10       0.9961    0.4164   0.4180   0.0000
    0.15       0.9571    0.3773   0.3943   0.0000
    0.20       0.8790    0.2993   0.3405   0.0000
    0.25       0.7872    0.2075   0.2636   0.0000
    0.30       0.7431    0.1634   0.2198   0.0000
    0.35       0.6972    0.1232   0.1726   0.0050
    0.40       0.6696    0.0957   0.1386   0.0050
    0.45       0.6177    0.0670   0.0850   0.0250
    0.50       0.5856    0.0581   0.0547   0.0450
    0.55       0.5503    0.0749   0.0413   0.0900
    0.60       0.5161    0.0810   0.0168   0.1247
    0.65       0.4629    0.1168   0.0000   0.2015
    0.70       0.3924    0.1873   0.0000   0.3231
    0.75       0.2817    0.2980   0.0000   0.5140
    0.80       0.1709    0.4088   0.0000   0.7052
    0.85       0.0626    0.5171   0.0000   0.8920
    0.90       0.0062    0.5735   0.0000   0.9892
    0.95       0.0000    0.5797   0.0000   1.0000
    1.00       0.0000    0.5797   0.0000   1.0000

  Area under ROC curve (AUC) for training data = 0.987897
  Threshold to minimize misclassification for training data = 0.517651
  Threshold to minimize weighted misclassification for training data = 0.517651
  Threshold to balance misclassifications for training data = 0.514761
```

For each probability threshold, several items of information are reported:

Proportion of cases – This column shows the proportion of cases that will be assigned the target category given a probability threshold. In other words, if the probability that a case has the target category exceeds the threshold, then it is assigned the category. For example, in the table shown above if the probability threshold is set to 0.20, then about 0.8790 (87.9%) of the cases will be assigned the selected target category (Liver Condition = 2 in this example). If the probability threshold is increased to 0.80, then fewer cases quality and only 0.1709 (17%) of the cases would be assigned the target category; all other cases would be assigned the other target category. Note in this example that if the default threshold of 0.50 is used, about 0.5856 (58.56%) of the cases will be assigned the target category. If the threshold is set to 0.0, all cases are assigned the target category and the proportion is 1.0. If the threshold is set to 1.0, no cases qualify.

Error – This is the proportion of cases that would be misclassified if a specified threshold is selected.

Impurity – The "impurity" is the proportion of cases whose actual (true) category is different than the selected category but which are misclassified as having the target category. In other words, it is the proportion of cases that are given the selected target

category that actually belong in the other category group. In the example table shown above, if the probability threshold is set to 0.10 then about 0.4180 (41.8%) of the cases classified as Liver Condition = 2 will actually have a different category. As the probability threshold is increased, the impurity decreases. In the example above, when the threshold is 0.50 the impurity is only 0.0547 (5%). When the probability threshold is set to 0.0 all cases are assigned to the target category, so the impurity is equal to the proportion of all cases that do not have the selected target category.

Loss – The "loss" is the proportion of cases whose actual (true) category matches the selected target category but which are assigned a different category. In the example table shown above we see that if rows are required to have a probability of 0.80 to be classified as Liver Condition = 2, then about 0.7052 (70.52%) of the cases with that actual classification will be misclassified. If the threshold is set to 0.0 then all cases are assigned the target category and the loss is 0.0. If the threshold is set to 1.0, then no cases qualify and the loss is 1.0.

Area under ROC curve – This is the area under the Receive Operating Characteristic (ROC) curve for the model. This statistic is known as Area Under Curve, "AUC"; it is also called the "C-statistic". The closer the value of the area is to 1.0, the better the model is.

Threshold to minimize misclassification for training data – This is the probability threshold that would minimize the total misclassification error for all data.

Threshold to minimize weighted misclassification for training data – This is the probability threshold that would minimize the weighted misclassification error. The weighted misclassification error is computed by multiplying the misclassification rate for each target category by a factor that corrects for the relative frequency of cases with that category in the data. Target categories that occur infrequently in the data receive a greater weight to prevent them from being overwhelmed by frequently occurring categories.

Threshold to balance misclassifications for training data – This is the probability threshold that would approximately equalize the number proportion of cases misclassified for each target category.

Focus Category Report

The Focus Category Report provides information about the "focus category" of the target variable. This section of the report is generated only if you designate a focus category on the Class Labels property page for the model (see page 124). Designating a focus category does not affect the model that DTREG generates; all it does is tell DTREG to generate additional statistics about the focus category.

Two statistics are reported for the focus category:

The **Impurity** of the focus category is the percentage of the rows predicted to be the focus category which are actually some other category. In other words, it is the percent of the misclassified cases predicted to be the focus category. If every case that is predicted to be the focus category is actually the focus category, then the impurity is 0.0.

The **Loss** of the focus category is the percentage of actual focus category cases which are misclassified as some other category. If every case of the focus category is correctly predicted to be the focus category, then the loss is 0.0.

Here is an example of the focus category model size report:

```
  ============  Focus Category Report  ============

The target variable is Species
Focus Category = Versicolor
The full tree has 5 nodes.
The minimum impurity occurs with 4 nodes.
The minimum loss occurs with 2 nodes.

   ------  Focus Category Vs. Model Size  ------

       ---- Training ----   --- Validation ---
Nodes  Impurity %  Loss %   Impurity %  Loss %
-----  ----------  ------   ----------  ------
    4       2.08    6.00        7.00    4.00 <-- Minimum impurity
    3       9.26    2.00        8.67    4.00
    2      50.00    0.00       50.00    0.00 <-- Minimum loss
```

This report shows how the impurity and loss for the focus category change with varying model sizes. For single-tree models, the model size is the number of terminal nodes in the tree. For TreeBoost and Decision Tree Forest models, the model size is the number of trees in the model. DTREG also generates charts showing the impurity and loss as a function of model size (see pages 210 and 211).

The second table in the Focus Category Report shows which categories contributed to the impurity and loss.

```
   ------  Focus Impurity and Loss Table  ------

           --- Training ---   -- Validation --
 Category  Impurity %  Loss %   Impurity %  Loss %
---------  ----------  ------   ----------  ------
   Setosa       0.00    0.00         0.00    0.00
Virginica       0.00    6.00         4.17    8.00
```

In this example, the focus category is Versicolor, so all of the categories other than Versicolor are listed. This table shows that the validation data for the model had 4.17% impurity due to Virginica cases that were misclassified as Versicolor. The focus category had an 8% loss due to Versicolor cases being misclassified as Virginica.

Lift and Gain Table

The lift and gain table is a useful tool for measuring the value of a predictive model. Lift and gain values are especially useful when a model is being used to target (prioritize) marketing efforts. Here is an example of a Lift and Gain table:

Lift/Gain for Survived = Yes

Bin Index	Cutoff Probability	Class % of bin	Cum % Population	Cum % of class	Cum Gain	% of Population	% of Class	Lift
1	0.73191	72.40	10.04	22.50	2.24	10.04	22.50	2.24
2	0.73191	73.76	20.08	45.43	2.26	10.04	22.93	2.28
3	0.21202	18.10	30.12	51.05	1.69	10.04	5.63	0.56
4	0.21202	28.51	40.16	59.92	1.49	10.04	8.86	0.88
5	0.21202	17.65	50.20	65.40	1.30	10.04	5.49	0.55
6	0.21202	34.84	60.25	76.23	1.27	10.04	10.83	1.08
7	0.21202	58.82	70.29	94.51	1.34	10.04	18.28	1.82
8	0.21202	2.26	80.33	95.22	1.19	10.04	0.70	0.07
9	0.21202	3.17	90.37	96.20	1.06	10.04	0.98	0.10
10	0.00000	12.74	100.00	100.00	1.00	9.63	3.80	0.39

Average gain = 1.485
Percent of cases with Survived = Yes: 32.30%

The lift and gain tables for a single-tree model have an entry for each terminal node. The lift and gain charts for other types of models have a fixed number of bins – usually 10, but you can change the number of bins on the Design Property Page (see page 33).

The basic idea of lift and gain is to sort the predicted target values in decreasing order of purity on some target category (probability of Survived=Yes in the example above) and then compare the proportion of cases with the category in each bin with the overall proportion. In the case of a model with a continuous target variable, the predicted target values are sorted in decreasing target value order and then compared with the mean target value. The lift and gain values show how much improvement the model provides in picking out the best 10%, 20%, etc. of the cases.

Most of the numbers in the table are relative to the overall percentage of cases with the selected target category. This value is shown below the table (for example, "Percent of cases with Survived = Yes: 32.30%"). Note that this percentage is calculated from the data rows used to build the table, so the percentage for the training and validation data may differ slightly.

Bin index – Bins are numbered from 1 up to the maximum number specified on the Design Property Page. The first bin represents the data rows that have the highest predicted probability for the specified target category (Survived=Yes for this example).

Cutoff Probability – This is the smallest predicted probability of data rows falling in this bin or earlier bins.

Class % of bin – This is the percentage of the cases in the bin that have the specified category of the target variable. In the example above, the target variable is "Survived" and this lift/gain table is for category "Yes" of Survived.

Cumulative % population – This is the cumulative percentage of the total cases (with any category) falling in bins up to and including the current one.

Cumulative % of class – This is the cumulative percentage of the rows with the specified category (Survived=Yes in this example) falling in bins up to and including the current one. In the example above, the first two bins have 48.38% of all of the Survived=Yes cases.

Cumulative gain – This is the ratio of the cumulative percent of class divided by the cumulative percent of the population. In the example above, the cumulative gain for bin 2 is 2.26 which is calculated by dividing 45.43 by 20.08.

% of population – This is the percentage of the total cases falling in the bin. This will be approximately 100/number-of-bins.

% of class – This is percent of the cases with the specified category (Survived=Yes in this example) that were placed in this bin. In this example, 22.50% of all the cases with category Yes ended up in the first bin.

The **Lift** value (last column) is calculated by dividing the percent of rows in a bin with the specified target category (% of Class) by the total percent of cases in the bin (% of Population). In the table above, the lift for the first row is calculated as 2.24 = 22.50/10.04.

To understand lift and gain, consider the example of a company that wants to do a mail marketing campaign. The company has a database of 100,000 potential customers, and they calculate that each mailed advertisement will cost $1.00. Prior experience has shown that the average response rate is 10%. So if they send the advertisement to all of the prospects, they will incur an expense of $100,000 and they will likely receive approximately 10,000 sales.

Hoping to improve their return on investment (ROI), the company uses DTREG to build a predictive model using data from previous campaigns with Sale/No-sale as the target variable and various demographic variables as predictors. The predictive model is used to prioritize the prospects so that they can be sorted in decreasing order of expected sales (i.e., the best sales candidates are sorted to the front of the list).

Using the "Cum % Population", "Cum % of class", "Cum Gain" and "Lift" columns from the Lift/Gain chart, the marketing director of the company prepares the following table:

Ads Mailed	Cum. % Class	Expected Sales	Cum. Gain	Lift
10000	30	3000	3.00	3.00
20000	50	5000	2.50	2.00
30000	65	6500	2.17	1.50
40000	72	7200	1.80	0.70
50000	80	8000	1.60	0.80
60000	85	8500	1.42	0.50
70000	90	9000	1.29	0.50
80000	95	9500	1.19	0.50
90000	98	9800	1.09	0.30
100000	100	10000	1.00	0.20

The table divides the total prospect set into 10 bins with the best 10% of the prospects (as predicted by DTREG) in the first bin, the second-best 10% in the second bin, and so forth. The table has five columns:

Ads mailed – This is the cumulative number of ads mailed starting with the best prospects and advancing to less well qualified prospects.

Cum. % class – This is the cumulative percentage of the sales expected from ads sent to prospects in the bins up to and including the one with the percentage. For example, we expect to receive 50% of total sales from ads sent to the prospects in the two highest-priority bins.

Expected sales – This is the total number of sales that can be expected from the cumulative number of ads mailed to customers in bins up to and including the current one. In this example, it is believed that of the total population (100,000) about 10% will respond resulting in sales of 10,000 units if all customers are targeted. So the expected cumulative sales for a bin are calculated by multiplying the expected total sales (10,000) by the cumulative percentage of the class up to and including the bin ("Cum. % class"). For example, if ads are mailed to customers falling in bins 1 and 2, then about 50% of the 10,000 expected sales will be achieved resulting in cumulative expected sales of 5,000 units.

Cum. Gain – This is the ratio of the expected sales using the model to prioritize the prospects divided by the expected sales if a random mailing was done. In this example we see that by targeting the customers in bins 1 and 2, we will get about 2.50 times as many sales as if we mailed the same number of ads to a random set of customers. Thus our return on investment (ROI) is increased by 2.5 if we target this group. Note that if we increase the number of ads mailed to include less qualified customers in higher bins, the gain decreases because we are now mailing to people who are less likely to respond. If we send ads to all 100,000 potential customers then the gain is 1.00 because are not doing any selective targeting.

Lift – This is the ratio of the expected sales for the prospects in a bin ("% of class") divided by the percent of the population in the bin ("% of population"). As you send ads

to less well qualified customers the number of proportion of sales decreases; this is reflected by the lift decreasing in higher bins.

What we learn from the table is that by targeting the campaign at the best 20% of the prospects (i.e., the prospects falling in the first two bins), we can expect 5000 sales which constitute 50% of the total expected sales. By targeting the best 50000 prospects, we can expect 8000 sales which constitute 80% of the total. The mailings done to the 10,000 prospects in the last (worst) bin are likely to yield only 200 sales for a return of 2%.

How Lift and Gain Values are calculated

Using the predictive model generated by DTREG, predicted target values are calculated for each row. A one-dimensional array (i.e., a "vector") is allocated with an entry for each row, and predicted target values are stored for each row. In the case of a classification problem (categorical target variable), the value is set to 1 if the predicted target category for a row matches the target value selected for the table (a separate Lift/Gain table is generated for each target category). A value of 0 is stored for rows where the predicted category is different from the target value selected for the table. For a regression analysis (target variable is continuous), the predicted value for each row is stored in the vector.

The vector of row values is then sorted in decreasing order. In the case of a classification problem, the rows that were assigned 1 because their predicted category matches the category of the table get sorted to the front of the list. In the case of a regression problem, the rows with the largest predicted target values get sorted the front of the list. The sort is done in a manner so that the row numbers that correspond to the sorted values are also rearranged; so we know which row has the largest value, which row has the smallest value, etc.

Another one-dimensional array is allocated with an entry for each bin in the lift/gain table. Usually there are 10 bins. The sorted row index numbers computed in the previous step are divided into *n* partitions, where *n* is the number of bins (it is actually a little more complex than this because row weights are factored into the partitioning). So the first bin has the set of rows whose predicted values are the ones that best match the target category for classification models or the largest numerical values for regression models.

Values are then calculated for each bin using the rows that were partitioned into the bin.

For classification trees, the **Lift** for the bin is the ratio of the weight of rows whose *predicted* target categories match the category of the table divided by the weight of the rows in the bin whose *actual* target category matches the category for the Lift/Gain table. For regression trees, the **Lift** for the bin is the ratio of the sum of *predicted* target values in the bin divided by the sum of the *actual* target values for the bin.

Since the row values were sorted in decreasing value, the first bins are likely to have the best predicted values, so their lift values will usually be greater than 1.00. Bins at the bottom of the table have rows that were not predicted well (or which had small predicted values), and their lift will usually be less than 1.00. If the model simply generated random predictions, the lift values for all bins would be approximately 1.00.

The **Cumulative Gain** for each bin is the ratio of the proportion of all rows with predicted categories matching the table category up to and including the bin divided by the proportion of rows with the actual target category of the table up through the current bin. Or, for regression trees, it is the proportion of the total predicted values for all rows up to the bin divided by the proportion of the actual target values up through the bin. The Cumulative Gain for the final bin will always be 1.00 because the proportion of the predicted values for the entire set of rows is 1.00 as is the proportion of the actual values.

Here is a summary of how lift/gain values are calculated:

Let:

ActualTarget = The actual value of the target variable for each row.

PredictedTarget = The predicted value of the target variable for each row as predicted by the model.

NumBins = Number of bins that will be in the lift/gain chart (specified on the Design Property page).

1. Sort the data rows in descending order of **PredictedTarget**.

2. Divide the sorted rows into **NumBins** bins with approximately the same number of rows in each bin. For a single-tree model, the bins contain the rows in each terminal node.

3. Calculate and report the following values:

Mean Target = For a regression model, this is the weighted mean of **ActualTarget** values in the bin. The bins are sorted in decreasing order on this column.

Class % of bin = For a classification model, this is the percentage of the rows in the bin that have the selected category. For example, if "Purchased-Product" is the selected category, then the value shown in this column is the number of rows representing people who purchased the product. The bins are sorted in decreasing order on this column, so the top row in the table has the purest set of rows for the category.

Cum. % Population = This is the cumulative percentage of the rows in all bins up to and including the current bin.

Cum % Target = For a regression model, this is the cumulative percent of the sum of the weighted target values (**ActualTarget**) occurring in the bins up to and including the current bin. (The percentage is relative to the total weighted sum of **ActualTarget** values in all rows.)

Cum % Class = For a classification model, this is the cumulative percent of the total rows having the selected category (**ActualTarget**) that fall in bins up to and including the bin.

Cum Gain = **Cum % Target** divided by **Cum % Population**. The gain shows how much of an improvement is provided by the model by using the high priority bins up to the one with the value.

% of Population = Percent of the total rows that are included in the bin.

% of Target = For a regression model, this is the sum of the **ActualTarget** values in the bin divided by the total sum of **ActualTarget** values for the population times 100.

% of Class = For a classification model, this is the number of rows having the designated category in the bin divided by the total number of rows having the designated category times 100.

Lift = **% of Target** (or **% of Class**) divided by **% of Population** times 100.

See page 212 for information about generating lift and gain charts.

Terminal Node Table

The terminal node table displays summary statistics about each terminal node in a single decision tree model. This section of the report is generated only if you check the box labeled "Generate report of tree splits" on the Single Tree property page (see page 51).

```
  ============  Terminal Nodes  ============

Terminal (leaf) tree nodes sorted by target category

Category   Node   Misclassification  Num. Rows     Weight
--------  ------  -----------------  ---------  ------------
1              5        25.00%             80            80
1              7        31.25%             16            16
1             58        33.33%             27            27
1              8        33.33%              6             6
1             77        34.29%             35            35
1             78        40.00%             10            10
2              9        10.53%             38            38
2             42        11.48%             61            61
2             57        14.71%             34            34
2             79        16.67%             24            24
2             59        21.43%             14            14
```

The terminal nodes are ordered by the categories of the target variable. For each category, the table shows each terminal node that predicts that category and the misclassification rate. Within a category, the nodes are ordered by increasing misclassification rate: so, the first terminal node listed for a category is the node that has the lowest misclassification rate for the category (i.e., it is the purest node for the category).

If the target variable is continuous, then the target node table has this format:

Terminal (leaf) tree nodes sorted by Sales value

Node	Target mean	Target std.dev.	Num. rows	Weight
93	9.91364	2.485375	44	44
92	13.92222	2.044384	18	18
65	14.04167	2.803854	24	24
119	14.40000	3.050683	3	3
86	16.63333	4.313416	12	12

In this case, the node number is shown in the first column, the mean value of the target variable for rows in the node is shown next followed by the standard deviation of the target mean then the number of rows and their weight. The nodes are ordered by increasing value of the target variable means.

The terminal node table is very useful for identifying focus groups. For example, if the target variable is customer sales and you are trying to identify the type of customers who are most likely to buy a product, then you would focus your attention on the terminal nodes that have the highest mean value on the customer sales target variable.

Variable Importance Table

The variable importance table gives a ranking of the overall importance of the predictor variables.

```
  ============  Overall Importance of Variables  ============

     Variable          Importance
-------------------    ----------
Lower status              100.000
Num. rooms                 88.439
Distance                   28.388
Pupil-teacher ratio        24.965
Nitric oxides              24.739
Industrial                 22.049
Tax rate                   19.691
Old houses                 15.584
Crime rate                 12.341
Large lots                 11.772
Radial highways             4.867
Black                       1.648
Charles River               0.509
```

Importance scores are computed by using information about how variables were used as primary splitters and also as surrogate splitters. Obviously, a variable that is selected as a primary splitter early in the tree is important. What is less obvious is that surrogate splitters that closely mimic the primary splitter are also important because they may be nearly as good as the primary splitter in producing the tree. If a primary splitter is slightly better than a surrogate, then the primary splitter may "mask" the significance of the other variable. By considering surrogate splits, the importance measure calculated by DTREG gives a more accurate measure of the actual and potential value of a predictor.

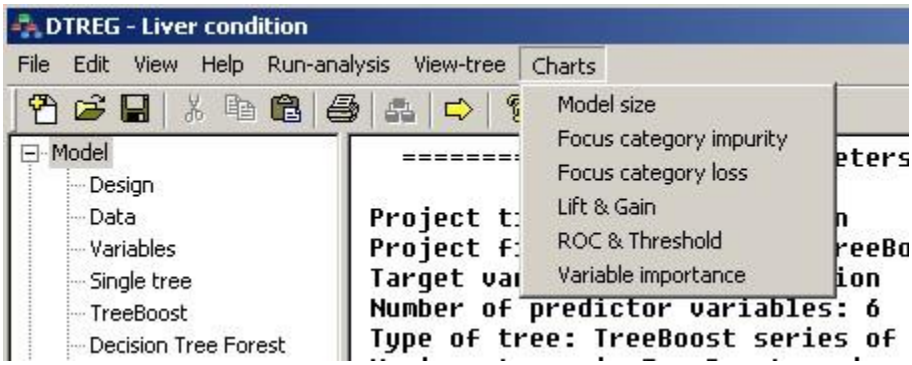
To get the most accurate measure of importance, you should select the option "Always compute surrogate predictors" on the Missing Data property page (see page 133).

The importance score for the most important predictor is scaled to a value of 100.00. Other predictors will have lower scores. Only predictors with scores greater than zero are shown in the table.

See page 228 for information about displaying a chart of variable importance.

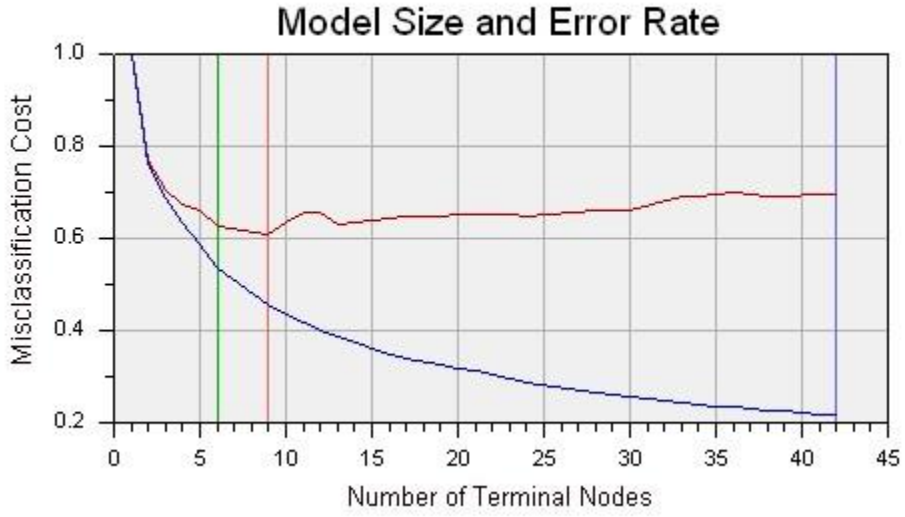
Charts and Graphs

DTREG generates a number of charts and graphs to show statistics for models. To view a chart, click "Charts" on the main menu, and select the desired chart from the drop-down menu.



Each of the charts is described below.

Model Size Chart

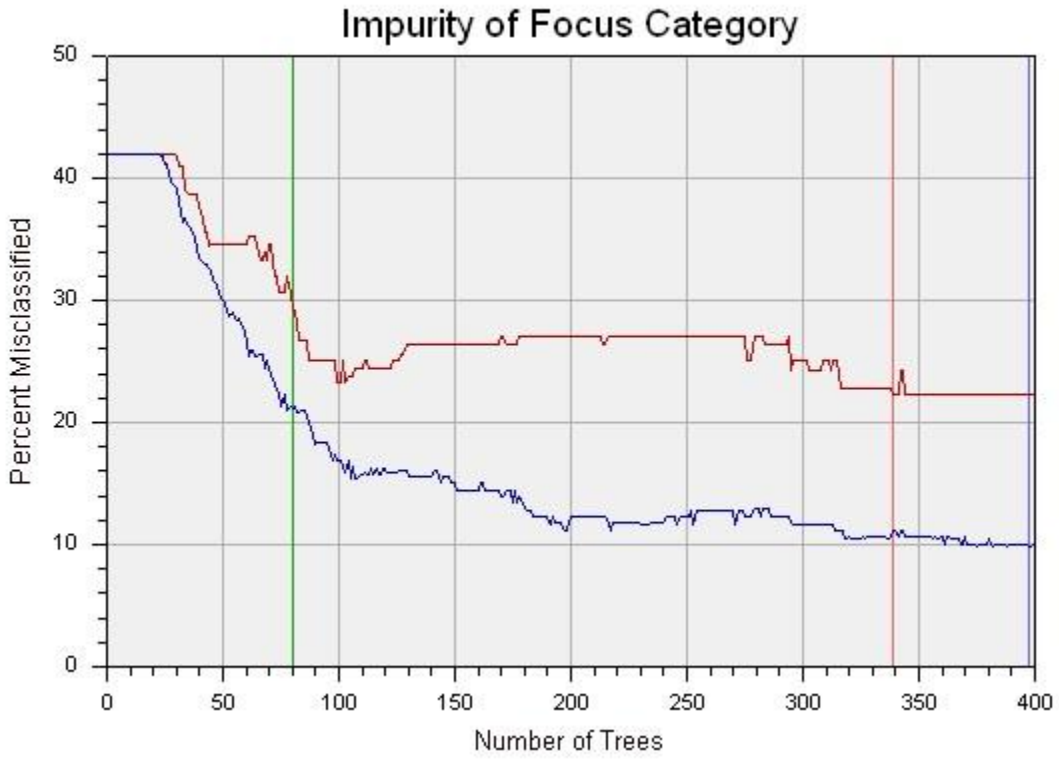


The Model Size chart shows how the error rate (residual or misclassifications) change with the size of the model. For a single-tree model, the model size is the number of terminal nodes in the tree. For a TreeBoost model, the model size is the number of trees in the TreeBoost model series. For a Decision Tree Forest mode, the model size is the number of trees in the forest. For multilayer perceptron neural networks where DTREG has automatically found the optimal number of neurons, the model size chart shows the

model error as a function of the number of neurons in the hidden layer. For PNN/GRNN neural networks, the chart shows the error as a function of the number of neurons.

The blue line on the chart represents the error rate for the training data. The red line shows the error rate for the validation (test) data. A blue vertical line shows the size with the minimum error on the training data line; a red vertical line shows the size with the minimum error for the validation data. A green vertical line shows the size to which the tree is pruned.

Focus Category Impurity Chart



The Focus Category Impurity Chart shows the impurity of the designated focus category of the target variable as a function of the size of the model. For a single-tree model, the model size is the number of terminal nodes in the tree. For a TreeBoost model, the model size is the number of trees in the TreeBoost model series. For a Decision Tree Forest mode, the model size is the number of trees in the forest.

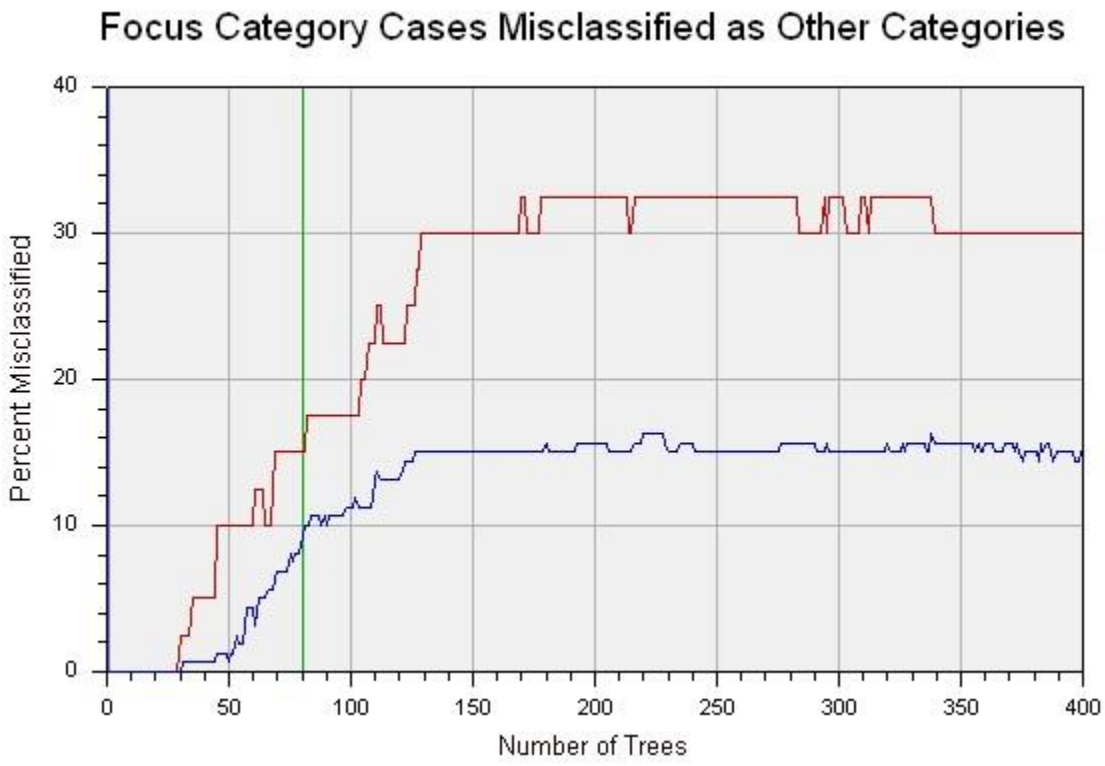
The blue line on the chart represents the impurity percentage for the training data. The red line shows the impurity for the validation (test) data. A blue vertical line shows the size with the minimum impurity on the training data line; a red vertical line shows the

size with the minimum impurity for the validation data. A green vertical line shows the size to which the tree is pruned.

The **Impurity** of the focus category is the percentage of the rows predicted to be the focus category which are actually some other category. In other words, it is the percent of the misclassified cases predicted to be the focus category. If every case that is predicted to be the focus category is actually the focus category, then the impurity is 0.0.

A Focus Category Impurity chart is generated only if you designate a focus category on the Class Table property page (see page 124).

Focus Category Loss Chart



The Focus Category Loss Chart shows the loss of the designated focus category of the target variable as a function of the size of the model. For a single-tree model, the model size is the number of terminal nodes in the tree. For a TreeBoost model, the model size is the number of trees in the TreeBoost model series. For a Decision Tree Forest mode, the model size is the number of trees in the forest.

The blue line on the chart represents the loss for the training data. The red line shows the loss for the validation (test) data. A blue vertical line shows the size with the minimum

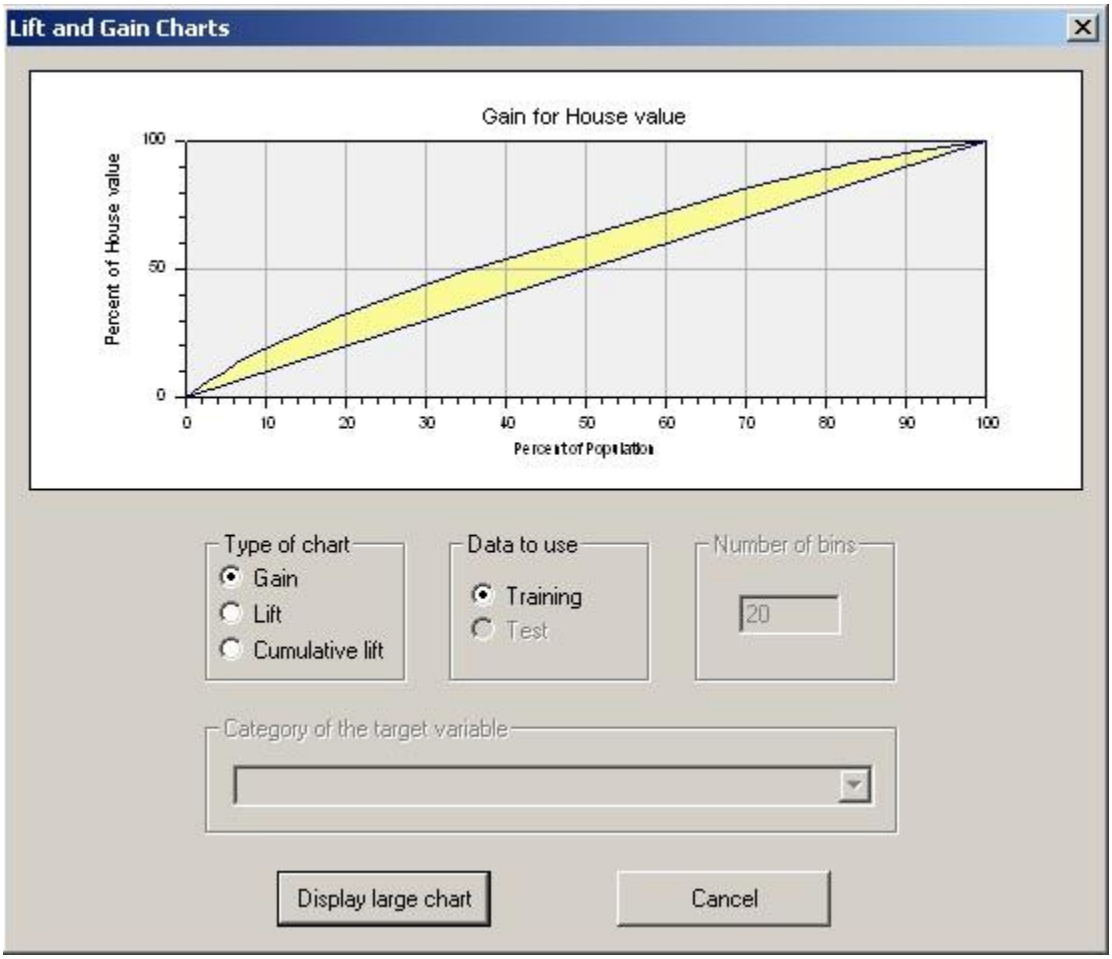
loss on the training data line; a red vertical line shows the size with the minimum loss for the validation data. A green vertical line shows the size to which the tree is pruned.

The **Loss** of the focus category is the percentage of actual focus category cases which are misclassified as some other category. If every case of the focus category is correctly predicted to be the focus category, then the loss is 0.0.

A Focus Category Loss chart is generated only if you designate a focus category on the Class Table property page (see page 124).

Lift and Gain Chart

When you select the "Lift & Gain" chart item, DTREG displays a screen with options related to these charts. See page 204 for information about how Lift and Gain values are calculated.

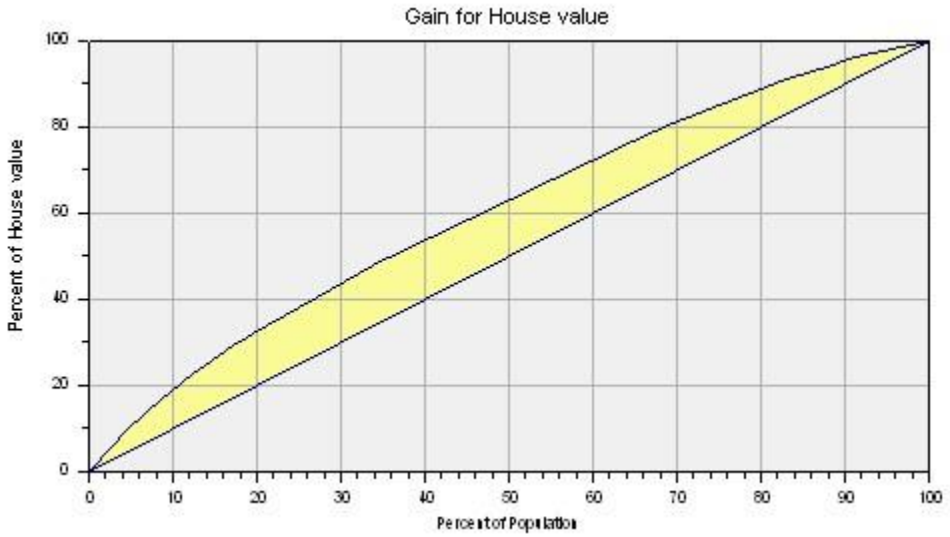


Select the type of chart you want to view (Gain, Lift or Cumulative lift) and the data to be used for the chart (Training or Test). You also can select the number of bins to divide the

data into. For classification models, select which category of the target variable the lift/gain is to be calculated for. See page 201 for information about how lift and gain values are computed and used.

Gain Chart

A gain chart displays cumulative percent of the target value on the vertical axis and cumulative percent of population on the horizontal axis. The straight, diagonal line shows the expected return if no model is used for the population. The curved line shows the expected return using the model. The shaded area between the lines shows the improvement (gain) from the model.

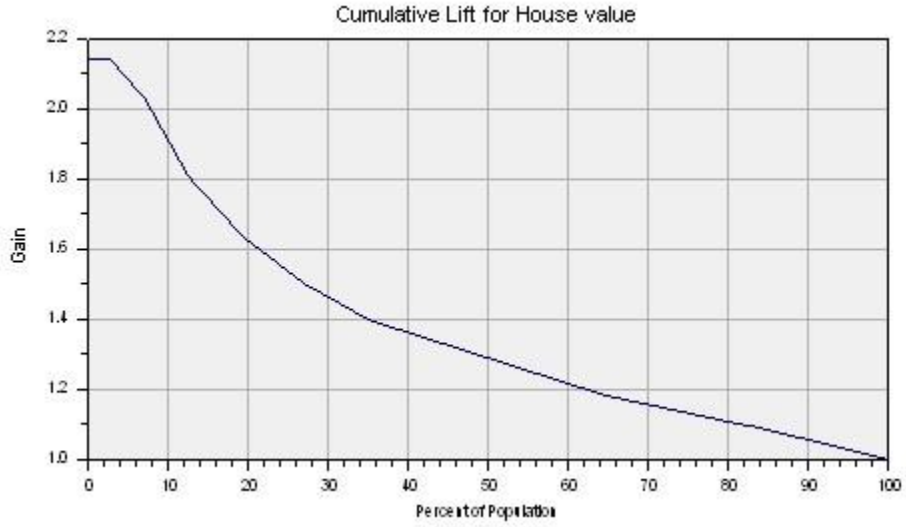


Lift Chart

A lift chart displays the lift for each bin on the vertical axis and the cumulative population on the horizontal axis.

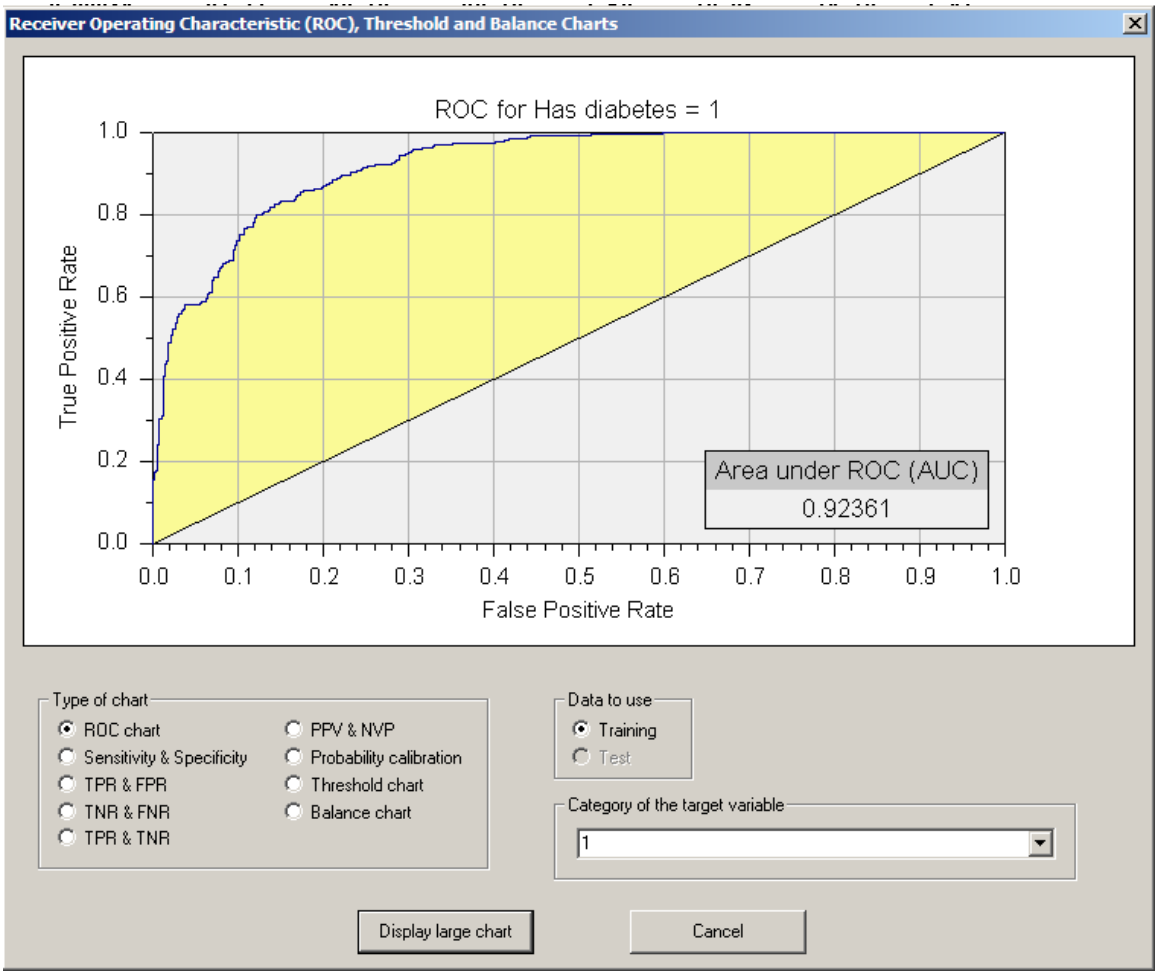
Cumulative Lift Chart

A cumulative lift chart displays gain on the vertical axis and percent of population on the horizontal axis.



ROC Chart

A *Receiver Operating Characteristic* (ROC) chart is available when a classification analysis has been and the target variable has two categories. ROC charts are not available for regression or for classification models where there are more than two target categories.



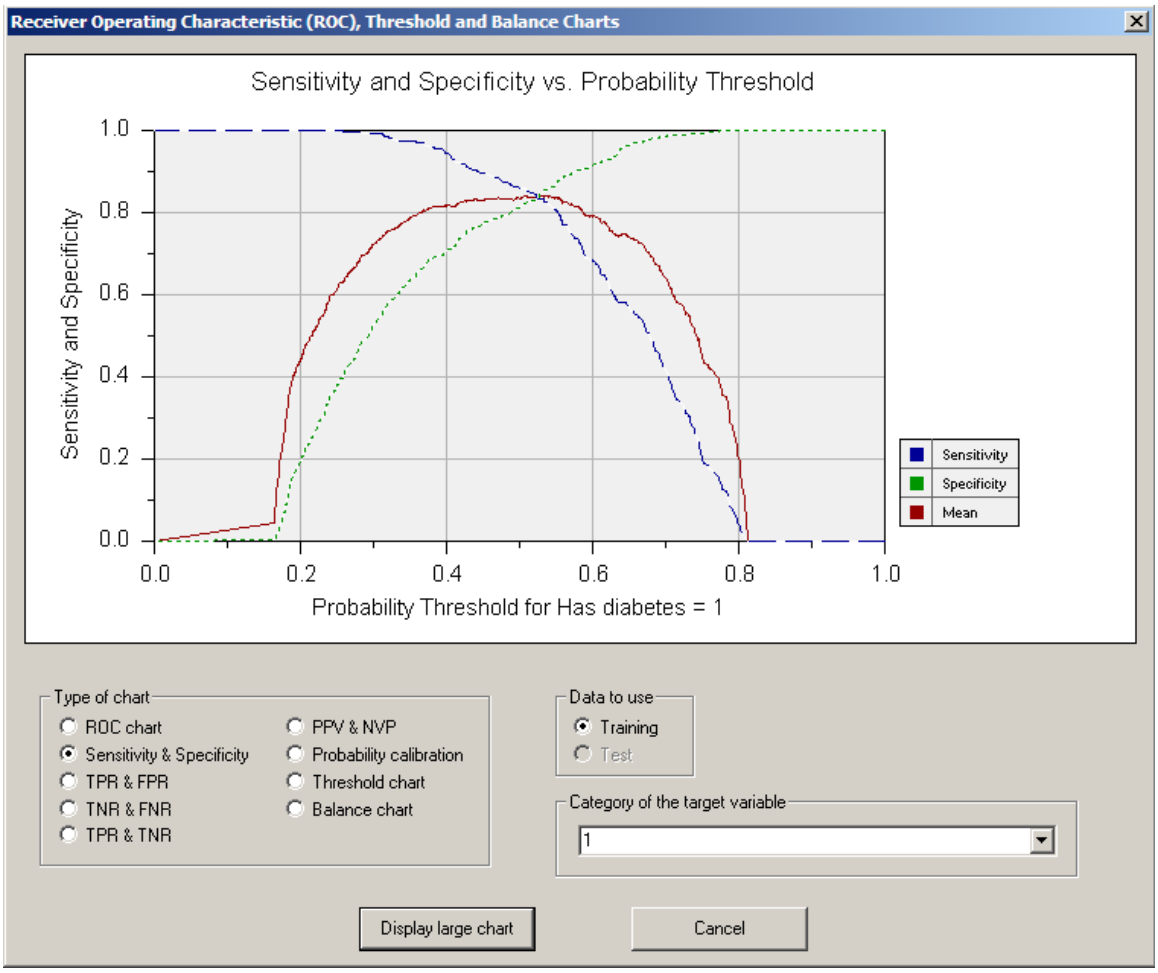
Classification models not only predict a specific category for each case but also generate posterior probability scores that indicate the relative likelihood for each possible category. Usually the category with the highest probability is selected as the predicted category.

A Receiver Operating Characteristic (ROC) chart displays the True Positive Rate (TPR) for predictions of a specific category on the vertical (Y) axis and the False Positive Rate (FPR) on the horizontal (X) axis. An ROC chart shows the trade-off between missed classifications (low TPR) and false classifications (high FPR) as different probability thresholds are considered. See also the description of the TPR/FPR chart that displays TPR and FPR curves relative to probability thresholds.

The (0,1) point in the upper left corner represents perfect classification – the true classification rate is 1.0 and the false classification rate is 0.0. The closer the ROC curve gets to the upper left corner of the chart, the better it is. The (0,0) point is reached when the probability threshold is set so high that that no cases are assigned the category, and no other categories are misclassified as the designated category. The (1,1) point is reached when the probability threshold is set so low that all cases receive the category classification even if their actual category is something else. The diagonal line from (0,0) to (1,1) represents the response that would be expected from randomly assigning the category. The yellow area between the diagonal line and the ROC line is the benefit gained by the model. The larger the yellow area, the better job the model is doing.

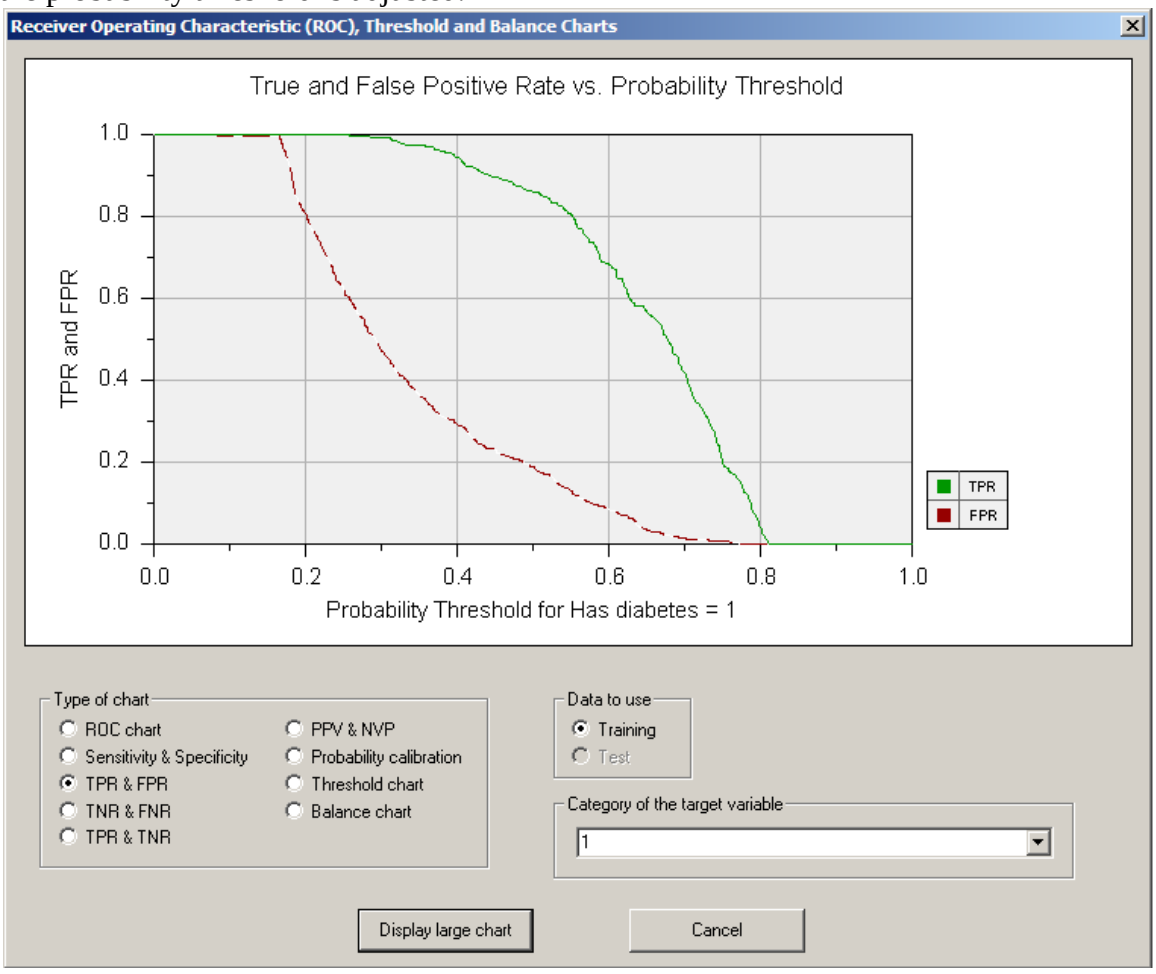
Sensitivity and Specificity Chart

A Sensitivity and Specificity Chart is available when a classification analysis has been run with two target categories and probabilities calculated. This chart shows how sensitivity and specificity can be adjusted by shifting the probability threshold for classifying cases as positive or negative. The probability threshold is specified on the Misclassification Cost property page (see page 130).



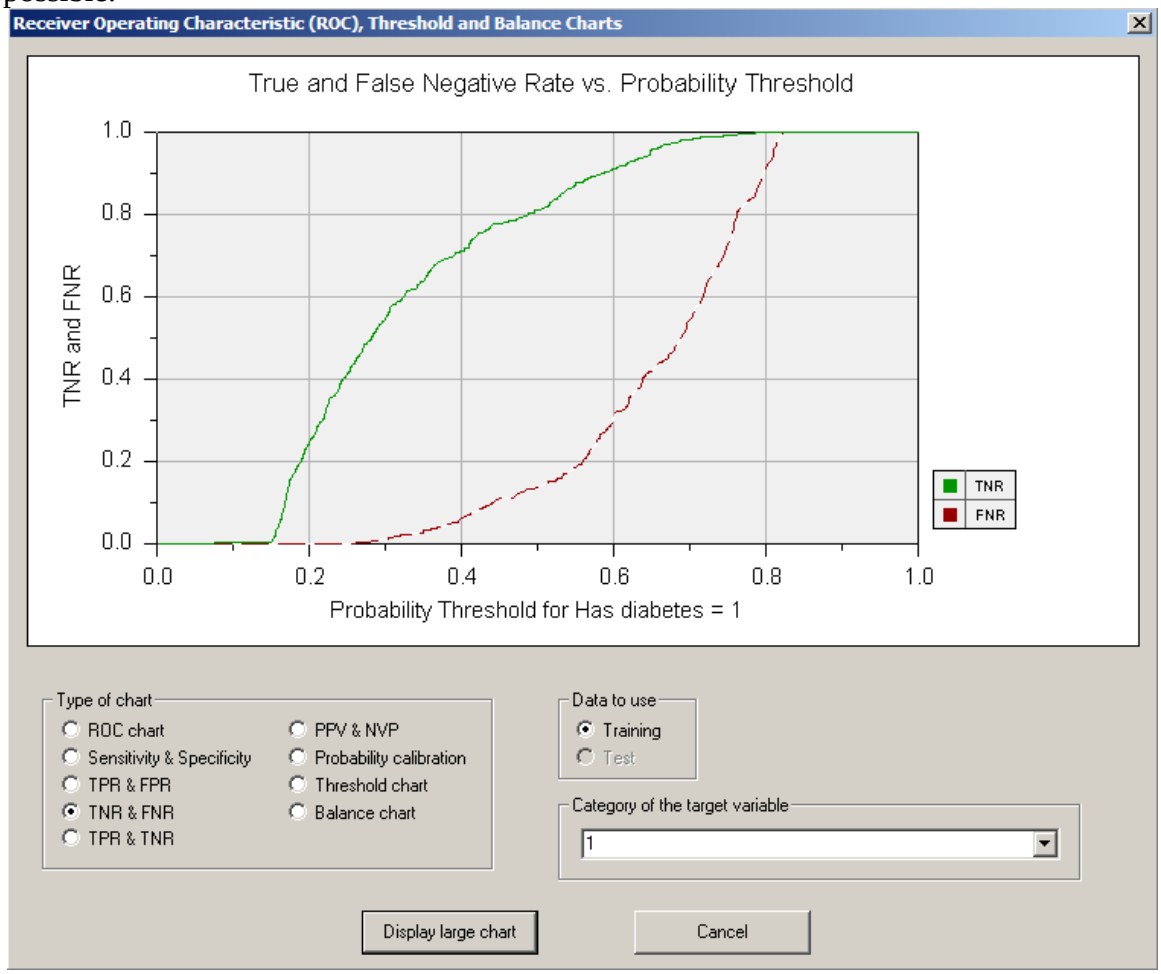
True Positive/False Positive Rate (TPR/FPR) Chart

This chart shows how True Positive Rate (TPR) and False Positive Rate (FPR) can be adjusted by shifting the probability threshold for classifying cases as positive or negative. The probability threshold is specified on the Misclassification Cost property page (see page 130). It is desirable that TPR be as large as possible and FPR be as small as possible. This chart is similar to the ROC chart described on page 215 in that they both display TPR and FPR values. However, this chart shows how the TPR and FPR vary as the probability threshold is adjusted.



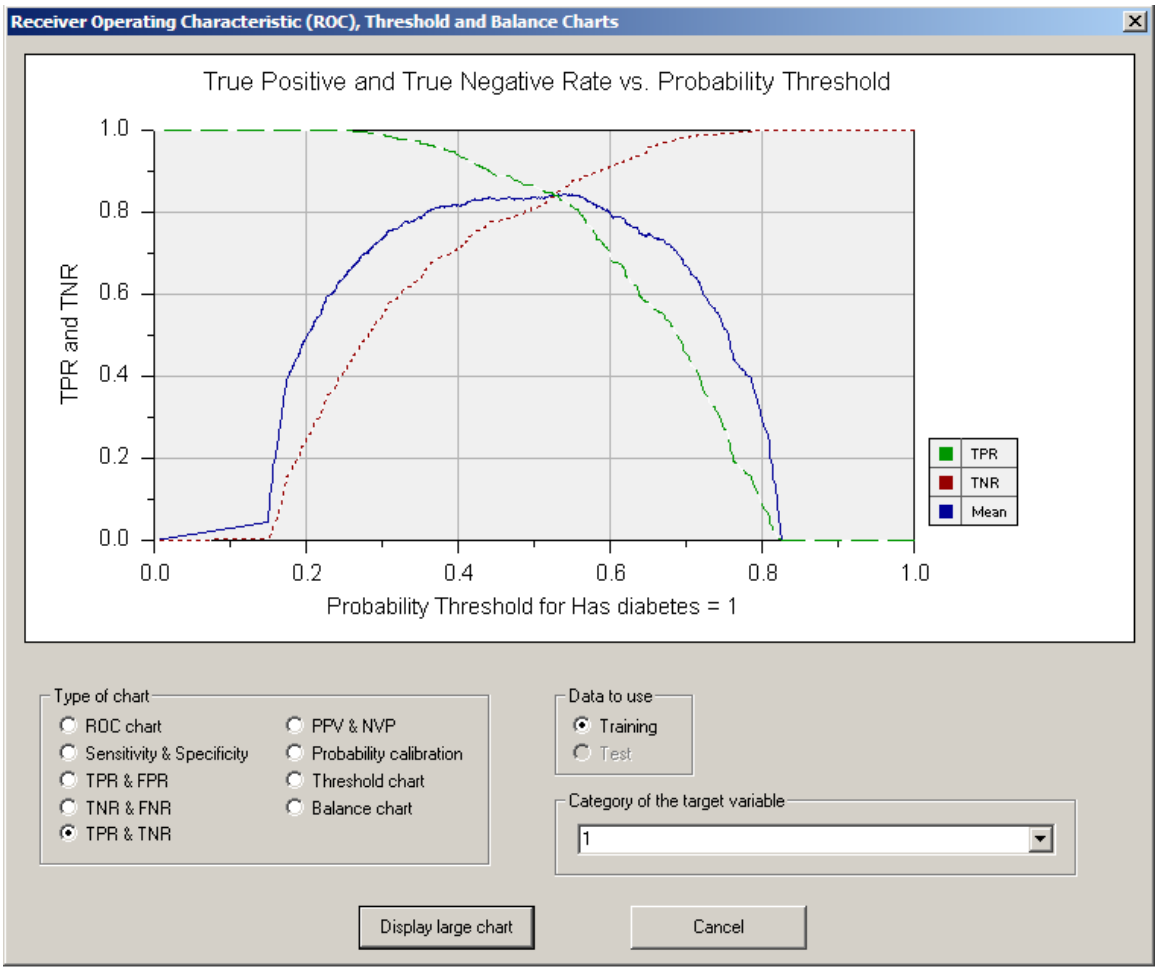
True Negative/False Negative Rate (TNR/FNR) Chart

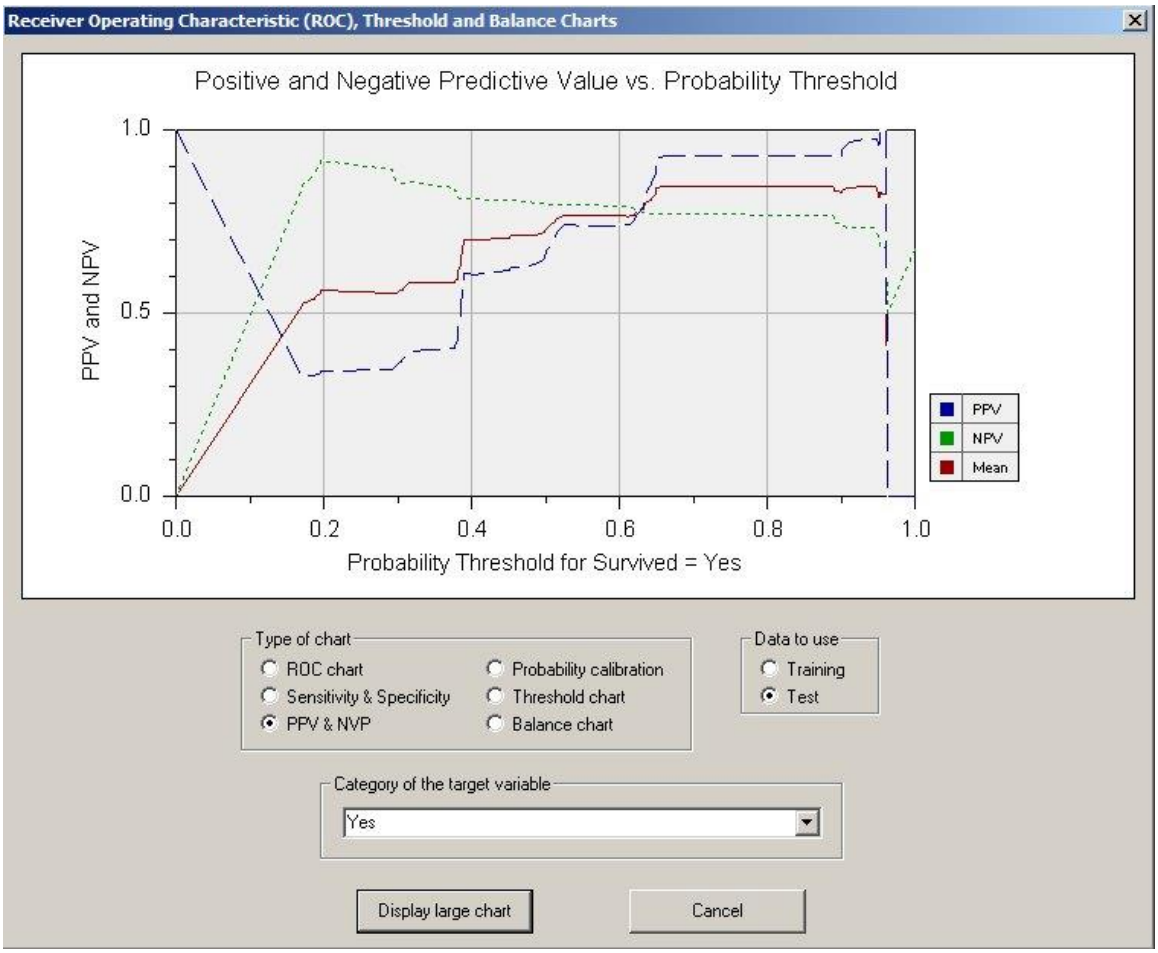
This chart shows how True Negative Rate (TNR) and False Negative Rate (FNR) can be adjusted by shifting the probability threshold for classifying cases as positive or negative. The probability threshold is specified on the Misclassification Cost property page (see page 130). It is desirable that TNR be as large as possible and FNR be as small as possible.



True Positive/True Negative Rate (TPR/TNR) Chart

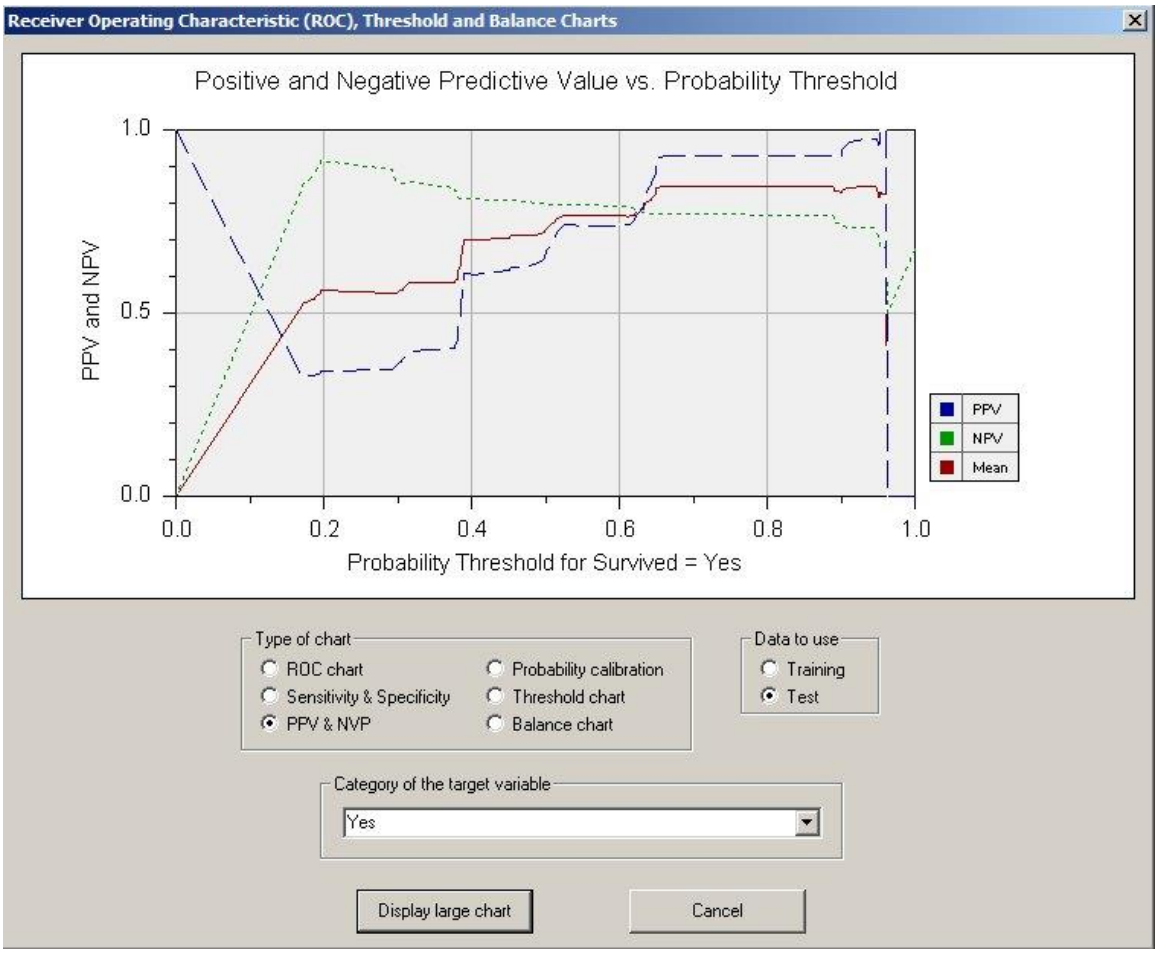
This chart shows how True Positive Rate (TPR) and True Negative Rate (TNR) can be adjusted by shifting the probability threshold for classifying cases as positive or negative. The probability threshold is specified on the Misclassification Cost property page (see page 130). It is desirable that TPR and TNR be as large as possible. The geometric mean value of TPR and TNR is shown as the blue line.





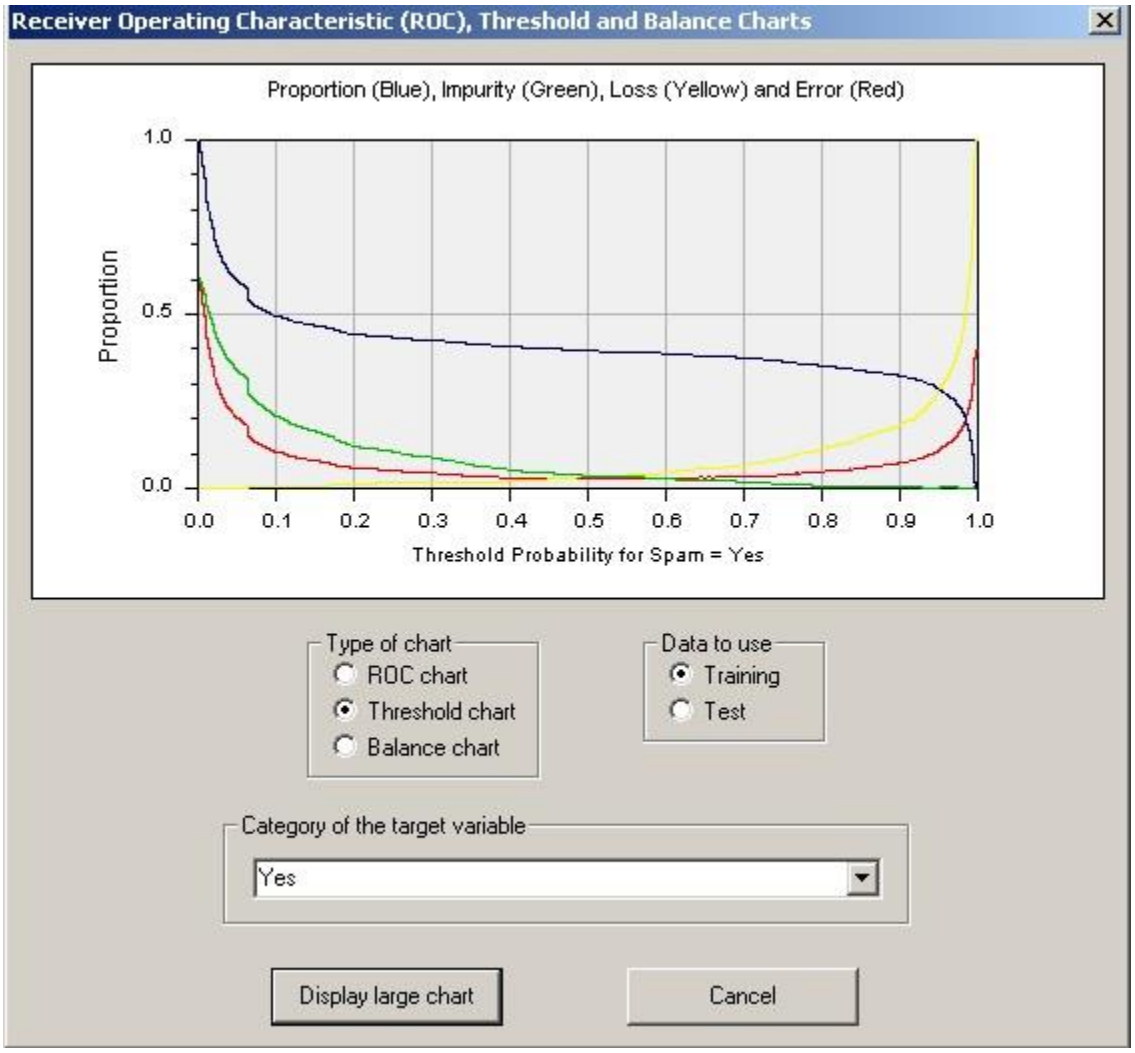
Positive and Negative Predictive Value Chart

A Positive and Negative Predictive Value Chart is available when a classification analysis has been run with two target categories and probabilities calculated. This chart shows how PPV and NPV can be adjusted by shifting the probability threshold for classifying cases as positive or negative. The probability threshold is specified on the Misclassification Cost property page (see page 130).


Receiver Operating Characteristic (ROC), Threshold and Balance Charts
Positive and Negative Predictive Value vs. Probability Threshold
PPV and NPV
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Probability Threshold for Survived = Yes
PPV
NPV
Mean
Type of chart
ROC chart
Sensitivity & Specificity
PPV & NVP
Probability calibration
Threshold chart
Balance chart
Data to use
Training
Test
Category of the target variable
Yes
Display large chart
Cancel

Probability Threshold Chart

A Probability Threshold Chart is available when a classification analysis has been run and the target variable has two categories. Threshold charts are not available for regression or for models where the target variable has more than two categories. A table showing the probability threshold response is generated in the analysis report. See page 197 for a description of the Probability Threshold Report.



Classification methods such as TreeBoost, SVM, Discriminant Analysis and Logistic Regression not only predict a specific category for each case but also generate probability scores that indicate the relative likelihood for each possible category. Usually the category with the highest probability is selected as the predicted category. In other words, the probability threshold is set at 0.5.

A Probability Threshold Chart shows how varying probability threshold values would affect the proportion of cases assigned the selected target category. The horizontal (X)

axis of the threshold chart has probability threshold values varying from 0.0 to 1.0. The vertical (Y) axis shows a proportion value. Three colored lines are shown on the chart:

Blue line, proportion of cases – The blue line shows the proportion of cases that will be assigned the target category given a probability threshold. In other words, if the probability that a case has the target category exceeds the threshold, then it is assigned the category. For example, in the chart shown above if the probability threshold is set to 0.2, then about 0.88 (88%) of the cases will be assigned the selected target category (Liver Condition = 2 in this example). If the probability threshold is increased to 0.8, then fewer cases quality and only 0.17 (17%) of the cases would be assigned the target category; all other cases would be assigned the other target category. Note in this example that if the default threshold of 0.5 is used, about 0.59 (59%) of the cases will be assigned the target category. If the threshold is set to 0.0, all cases are assigned the target category and the proportion is 1.0. If the threshold is set to 1.0, no cases qualify.

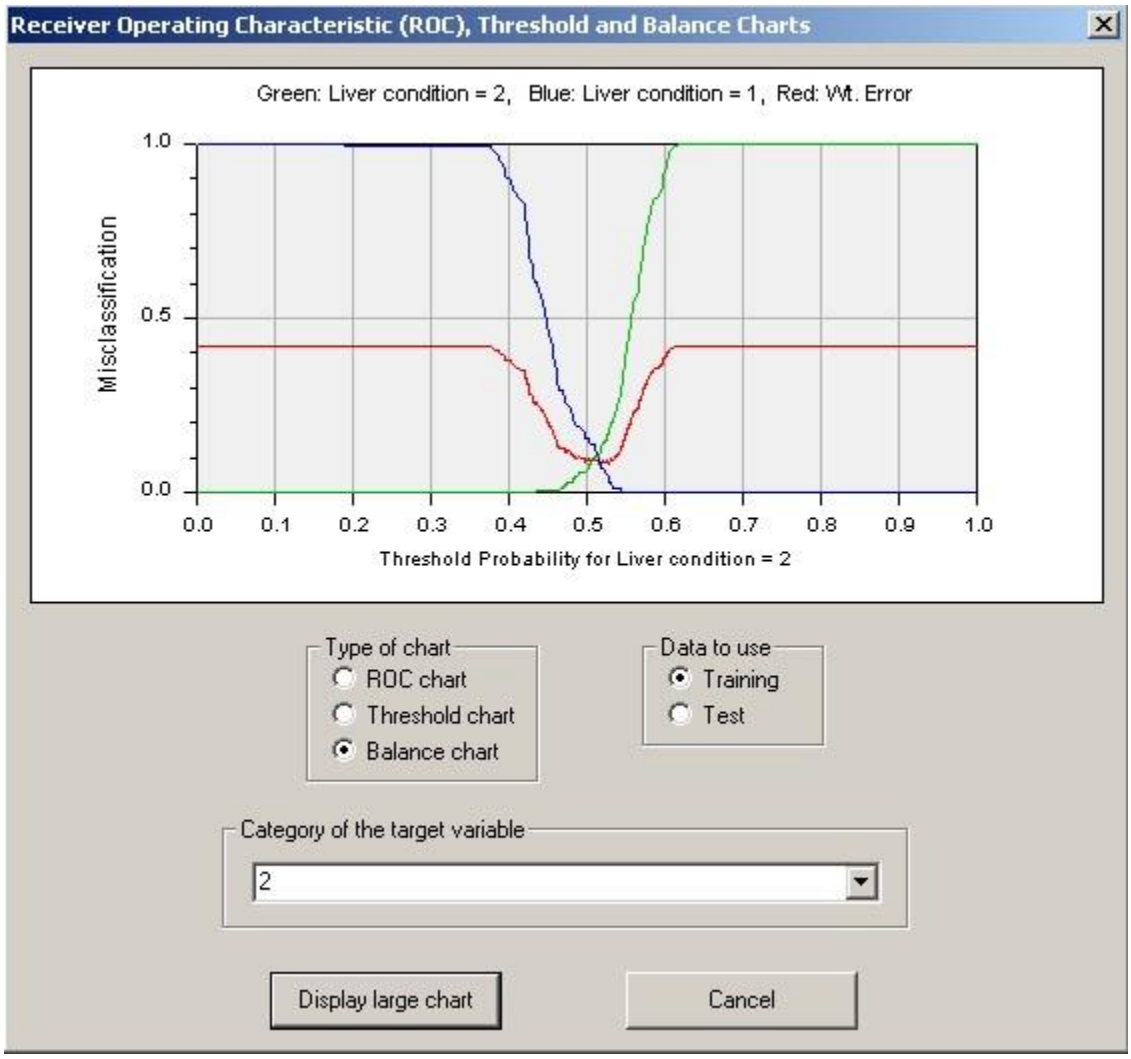
Green line, impurity – The "impurity" is the proportion of cases whose actual (true) category is different than the selected category but which are misclassified as having the target category. In other words, it is the proportion of cases that are given the selected target category that actually belong in the other category group. In the example chart shown above, if the probability threshold is set to 0.1 then about 0.42 (42%) of the cases classified as Liver Condition = 2 will actually have a different category. As the probability threshold is increased, the impurity decreases. In the example above, when the threshold is 0.5 the impurity is only 0.05 (5%). When the probability threshold is set to 0.0 all cases are assigned to the target category, so the impurity is equal to the proportion of all cases that do not have the selected target category.

Yellow line, loss – The "loss" is the proportion of cases whose actual (true) category matches the selected target category but which are assigned a different category. In the example chart shown above we see that if rows are required to have a probability of 0.8 to be classified as Liver Condition = 2, then about 0.71 (71%) of the cases with that actual classification will be misclassified. If the threshold is set to 0.0 then all cases are assigned the target category and the loss is 0.0. If the threshold is set to 1.0, then no cases qualify and the loss is 1.0.

The probability threshold chart provides a convenient way to see the tradeoff between impurity and loss as the probability threshold is varied. You can specify the probability threshold to use for classifications on the Misclassification Cost Property Page described on page 130.

Threshold Balance Chart

The Threshold Balance Chart shows how the misclassification error rate for each category is affected by varying probability thresholds. A Threshold Balance Chart is available when a classification analysis has been and the target variable has two categories. Threshold balance charts are not available for regression analyses or for models with more than two categories of the target variable. A table showing the probability threshold response is generated in the analysis report. See page 197 for a description of the Probability Threshold Report.



A Threshold Balance Chart shows how varying probability threshold values would affect the misclassification proportion for cases with each target category. The horizontal (X) axis of the threshold chart has probability threshold values varying from 0.0 to 1.0. The vertical (Y) axis shows a misclassification proportion value. Three colored lines are shown on the chart:

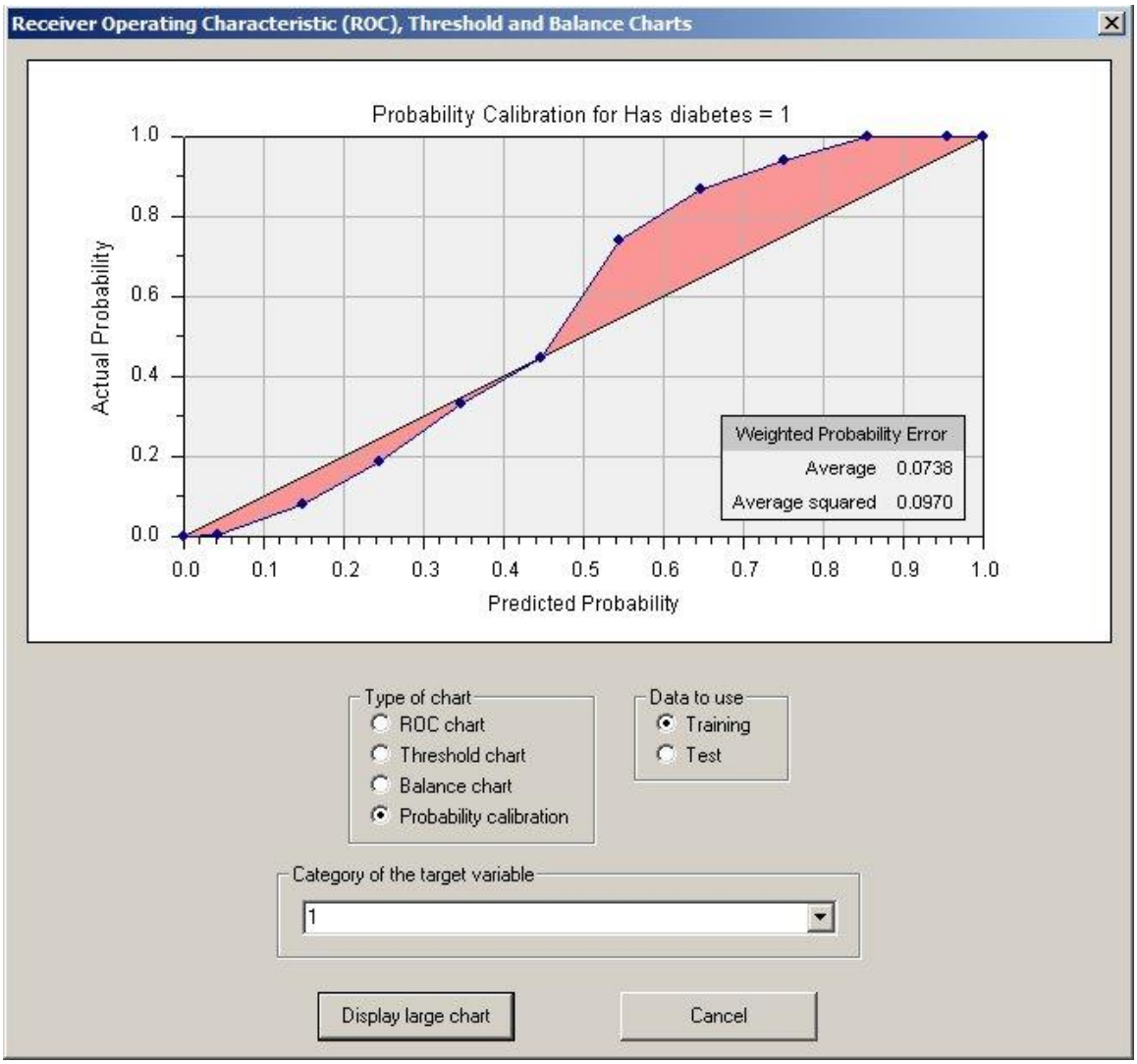
Green line – Proportion of cases misclassified for one of the target categories.

Blue line – Proportion of cases misclassified for the other target category.

Red line – Weighted misclassification rate. The weighted misclassification error is computed by multiplying the misclassification rate for each target category by a factor that corrects for the relative frequency of cases with that category in the data. Target categories that occur infrequently in the data receive a greater weight to prevent them from being overwhelmed by frequently occurring categories.

Probability Calibration Chart

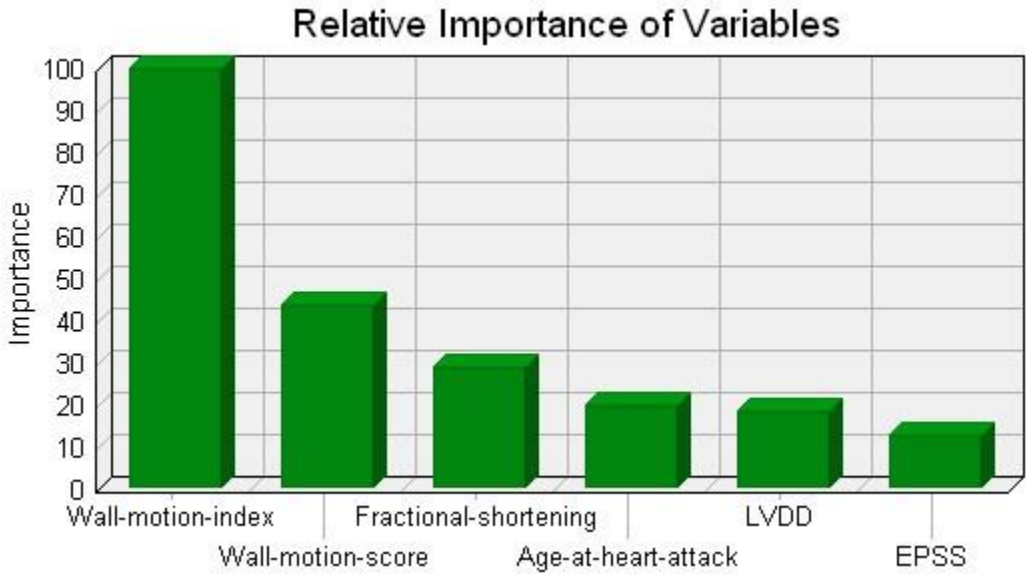
The Probability Calibration Chart shows how the predicted probability for a target category is distributed and provides a means for gauging the accuracy of predicted probabilities. The probability calibration chart is generated only when a classification analysis is performed and there are two target categories. Here is an example of a probability calibration chart:



The horizontal axis has the predicted probability for the observations. The vertical axis has the actual probability based on the frequency of occurrence. For example, in the chart above the average predicted probability for cases between 0.6 and 0.7 was about 0.65; the actual probability based on the rate of occurrence for those cases was about 0.87. If the predicted probabilities match the actual probabilities, the points fall on the diagonal line. The red shaded area shows the error which is the difference between the predicted and actual probabilities. For additional information, see the description of the Probability Calibration Report on page 195.

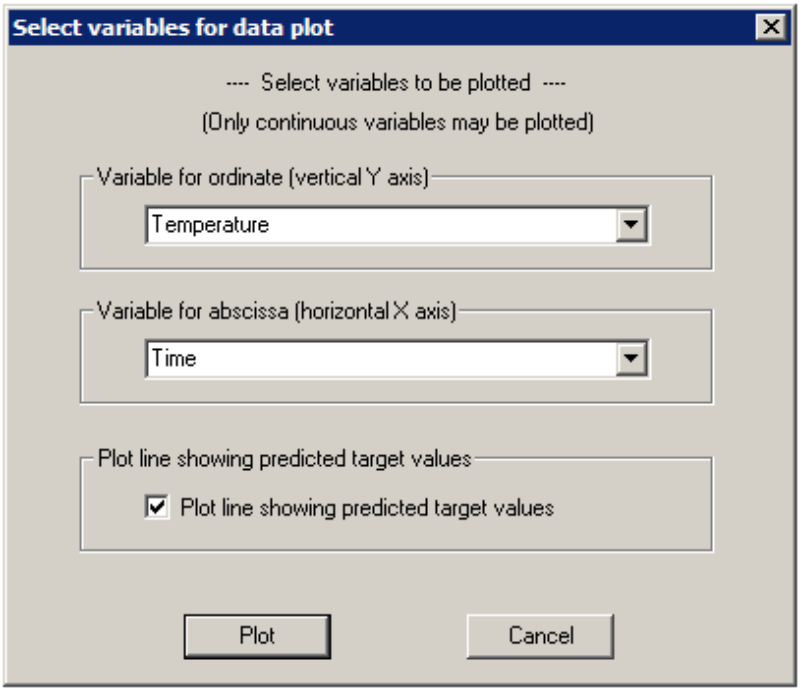
Variable Importance Chart

The Variable Importance chart is a bar chart showing the relative importance for the 10 most important variables.



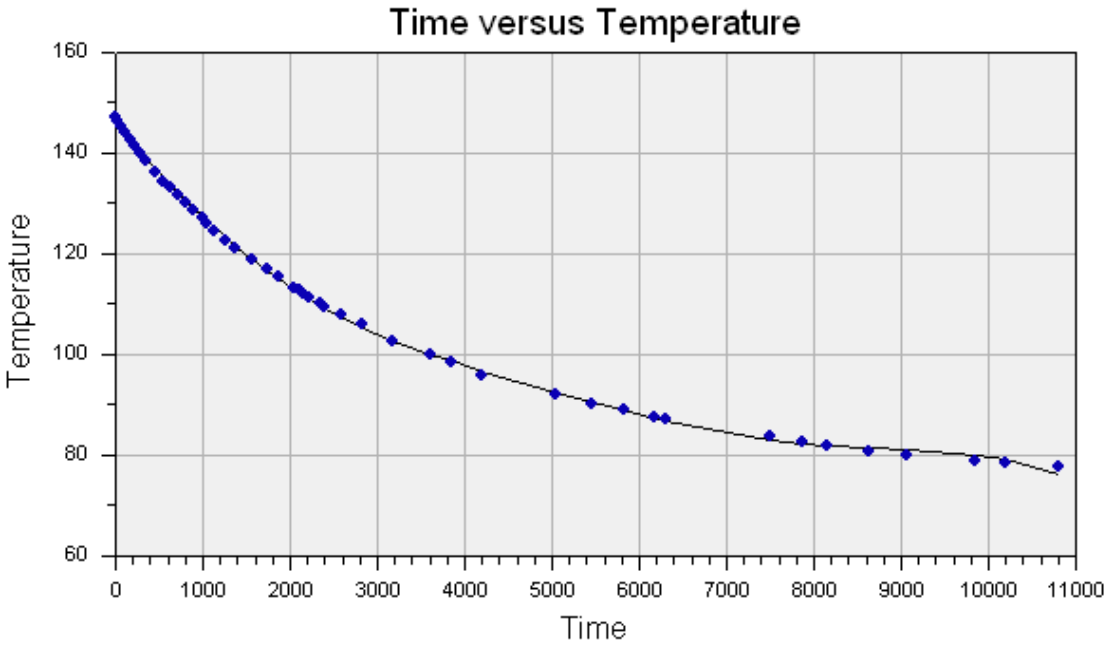
X-Y Data Plot

The X-Y Data Plot chart displays the values of two continuous variables on a Cartesian plot. When you select this type of chart, DTREG will display a screen where you select which variables you want to plot. Here is an example:



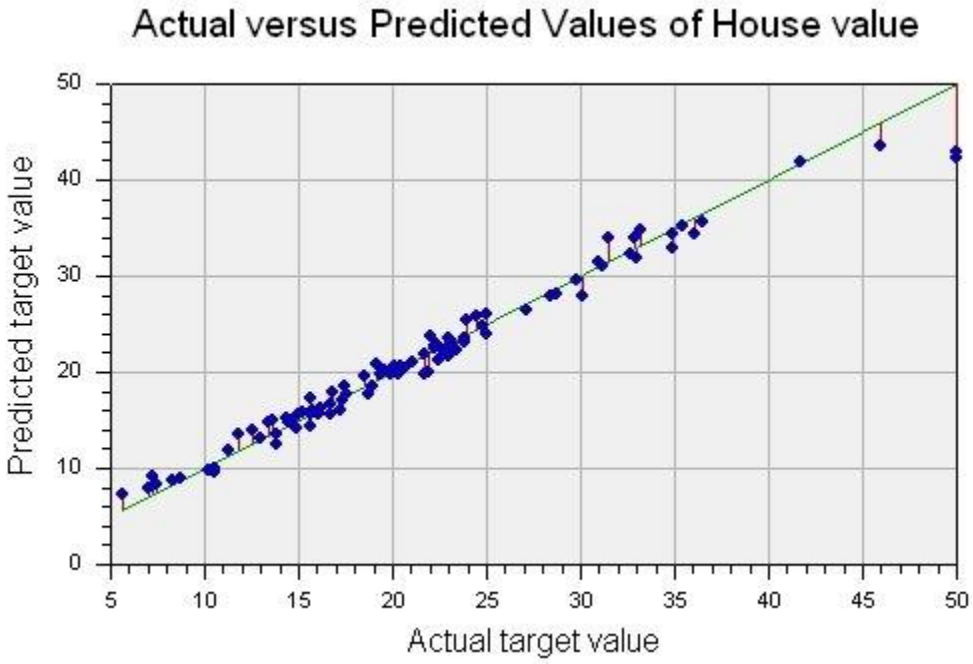
In the top field, select the variable to be displayed on the vertical Y axis (ordinate); in the lower field, select the variable to be displayed on the horizontal X axis (abscissa). Only continuous variables may be selected.

If you have created a model, the target variable is continuous, and you select the target variable to be displayed on the Y axis, then the "Plot line showing predicted target values" option will be enabled. Check this box to display the predicted values of the target variable on the plot. Here is an example of an X-Y data plot showing both the actual data points and the fitted function:


Time versus Temperature
Temperature
160
140
120
100
80
60
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
Time

Residual (Actual versus Predicted) Chart

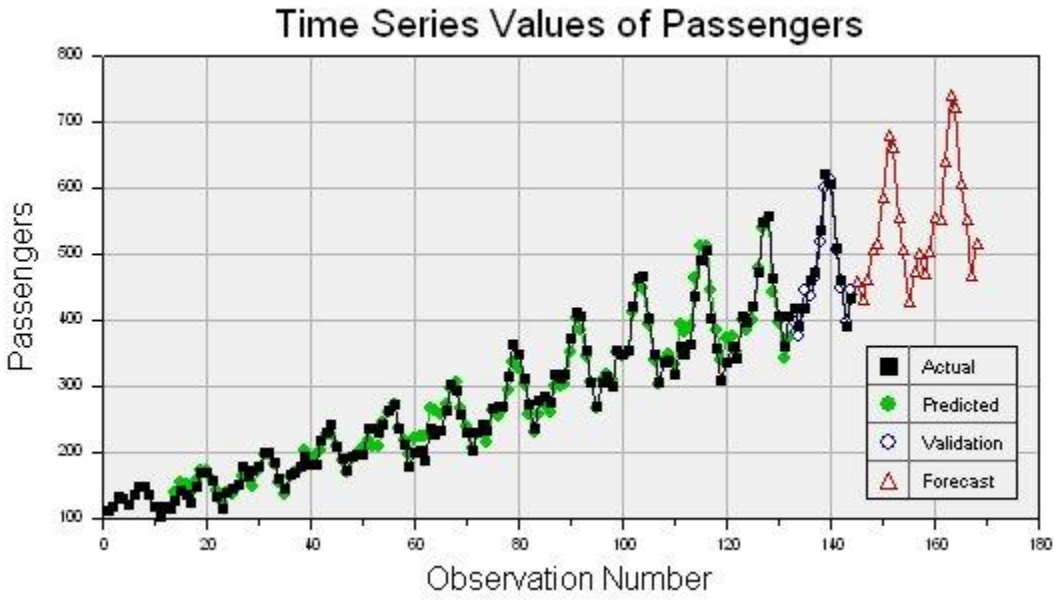
The Actual versus Predicted chart is available only after building a model where the target variable is continuous. It displays a point for each data row. The X coordinate of a point is the actual target value. The Y coordinate of the point is the corresponding predicted target value. This type of chart is sometimes called a Residual Chart. With a perfect model, the predicted values would equal the actual values, the X and Y coordinates for each point would be equal, and all points would be located on the diagonal line where X=Y. When the predicted value differs from the actual value, the points are offset from the diagonal line, and the vertical distance from the line to the point corresponds to the error (residual). The error is denoted by red vertical lines.



Time Series Chart

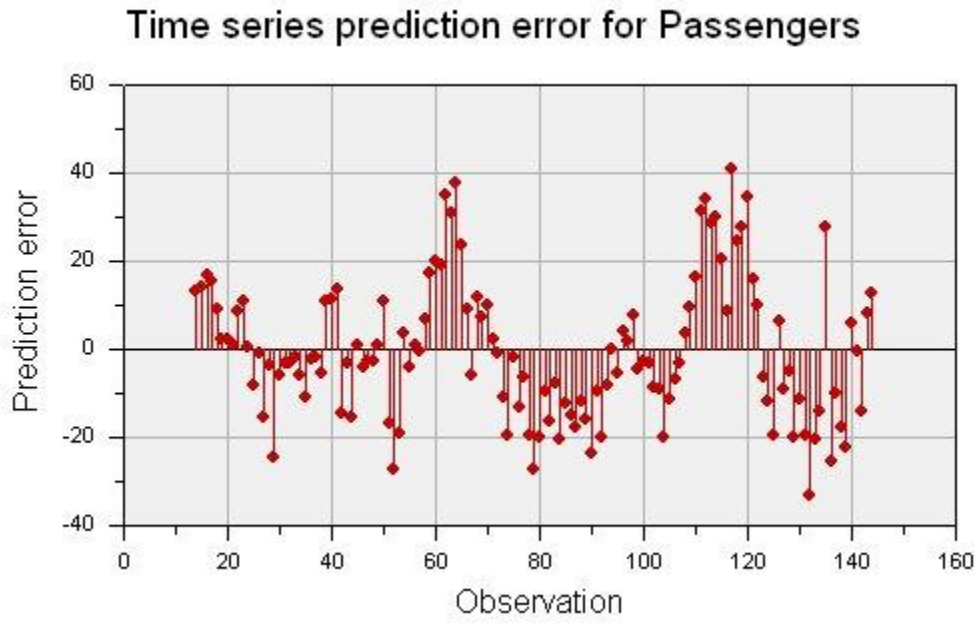
The Time Series chart displays up to four lines:

- Black square – Actual values of the time series
- Green square – Predicted values for points corresponding to training points
- Open blue circle – Predicted values for validation rows not used for training
- Open red circle – Forecast values beyond the end of the time series



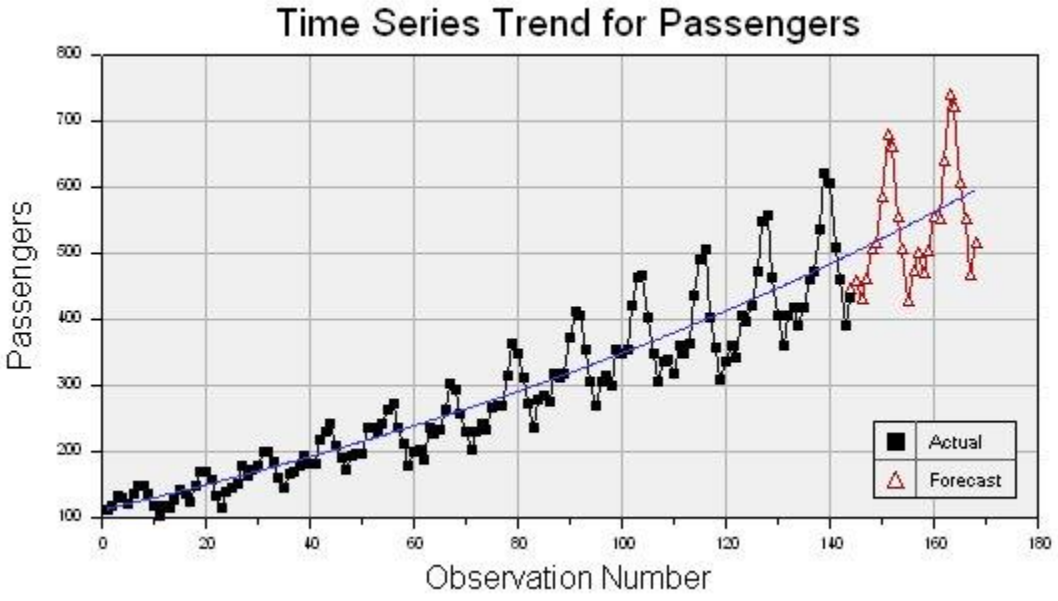
Time Series Residuals Chart

The Time Series Residual chart shows the residuals (errors) of the predicted values minus the actual values.



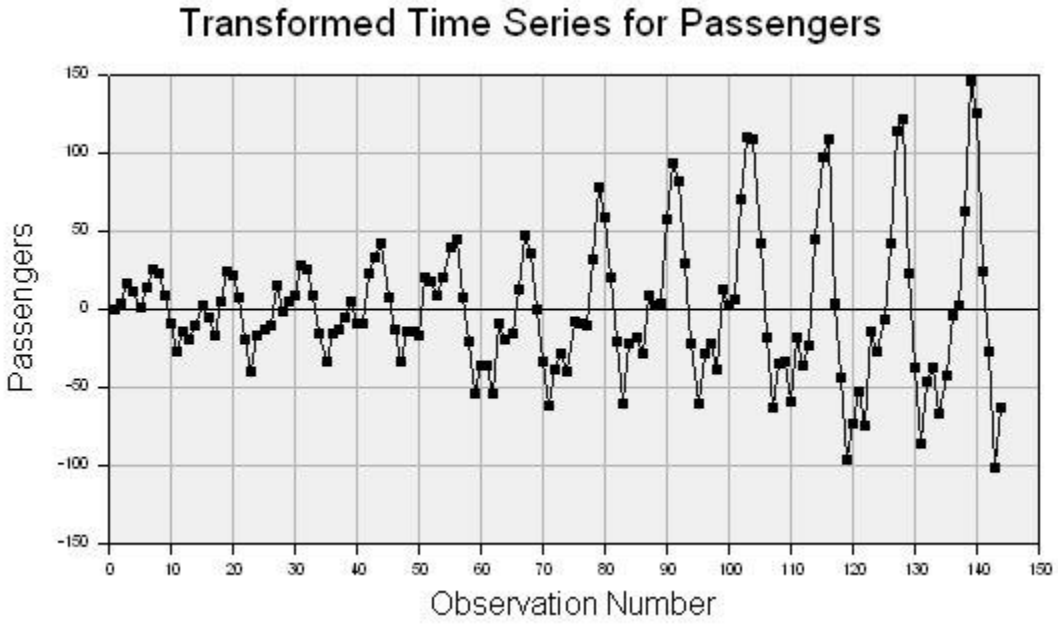
Time Series Trend Chart

The Time Series Trend chart shows the actual values and a trend line fitted to the series.



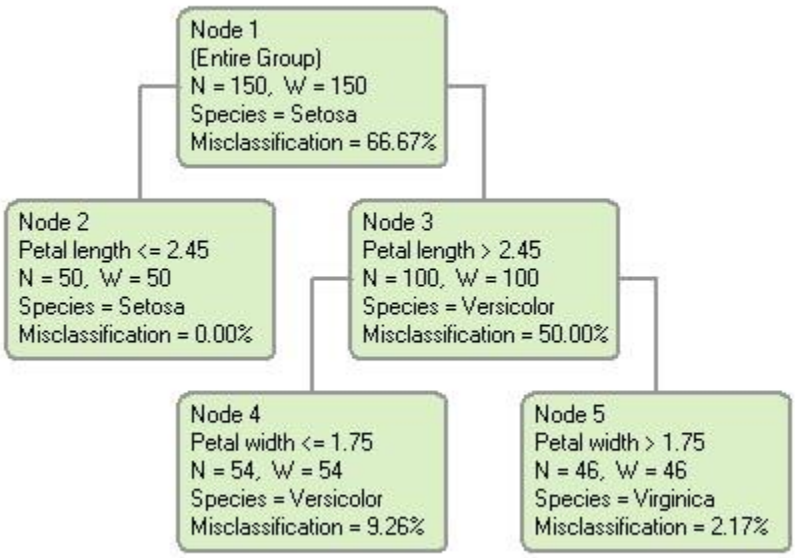
Time Series Transformed Chart

The Time Series Transformed chart shows the time series after DTREG has removed the trend and stabilized the variance (if requested).



Decision Trees

A decision tree is a logical model represented as a binary (two-way split) tree that shows how the value of a *target variable* can be predicted by using the values of a set of *predictor variables*. An example of a decision tree is shown below:



Decision Tree Nodes

The rectangular boxes shown in the tree are called "*nodes*". Each node represents a set of records (rows) from the original dataset. Nodes that have child nodes (nodes 1 and 3 in the tree above) are called "*interior*" nodes. Nodes that do not have child nodes (nodes 2, 4 and 5 in the tree above) are called "*terminal*" or "*leaf*" nodes. The topmost node (node 1 in the example) is called the "*root*" node. (Unlike a real tree, decision trees are drawn with their root at the top). The root node represents all the rows in the dataset.

In the top of the node box is the node number. Use the node number to find information about the node in the reports generated by DTREG. The "N = *nn*" line shows how many rows (cases) fall in the node. The "W = *nn*" line shows the sum of the weights of the rows in the node. For details on the information presented in each node, see "What's in a node" on page 241.

Splitting Nodes

A decision tree is constructed by a binary split that divides the rows in a node into two groups (child nodes). The same procedure is then used to split the child groups. This process is called "recursive partitioning". The split is selected to construct a tree that can be used to predict the value of the target variable.

For each split, two decisions are made by DTREG: (1) which predictor variable to use for the split (this is called the "*splitting variable*"), and (2) which set of values of the predictor variable go into the left child node and which set go into the right child node; this is called the "*split point*". The same predictor variable can be used to split many nodes. For a more detailed explanation of how trees are built, please see page 361.

The name of the predictor variable used to construct a node is shown in the node box below the node number. For example, in the tree shown on page 235, nodes 2 and 3 were formed by splitting node 1 on the predictor variable "Petal length". The split point is 2.45. If the splitting variable is continuous (numeric) as in this split, the values going into the left and right child nodes will be shown as values less than or greater than some split point (2.45 in this example). Node 2 consists of all rows with the value of "Petal length" less than or equal to 2.45, whereas node 3 consists of all rows with Petal length greater than 2.45. If the splitting variable is categorical, the categories of the splitting variable going into each node will be listed.

Building and Using a Decision Tree Model

There are two steps to making productive use of decision trees (1) building a decision tree model, and (2) using the decision tree to draw inferences and make predictions. The following sections provide an overview of how decision trees are built and used.

Overview of the Tree Building Process

The first step in building a decision tree is to collect a set of data values that DTREG can analyze. This data is called the *learning* or *training* dataset because it is used by DTREG to learn how the value of a target variable is related to the values of predictor variables. This dataset must have instances for which you know the actual value of the target variable and the associated predictor variables. You might have to perform a study or survey to collect this data, or you might be able to obtain it from previously-collected historical records.

Each entry in the learning dataset provides values for the target and predictor variables for a specific customer, patient, company, etc. Each entry is known as a "*case*," "*row*," "*record*," "*observation*" or "*vector*". See page 36 for information about the format of datasets.

The question "How much data is required for the learning dataset?" is answered by addressing the level of precision you desire in the resulting tree. In general, DTREG will not split a node with fewer than 10 rows. So, a tree with three levels and four terminal nodes must have an absolute minimum of 20 records, but the predictive accuracy would be greatly improved by having four or more times that many records. DTREG is designed to handle tens of thousands of records.

Once you obtain enough data for the learning dataset, this data is fed into DTREG which performs a complex analysis on it and builds a decision tree that models the data. See page 361 for additional information about the tree building process.

Overview of Using Decision Trees

Once DTREG has created a decision tree, you can use it in the following ways:

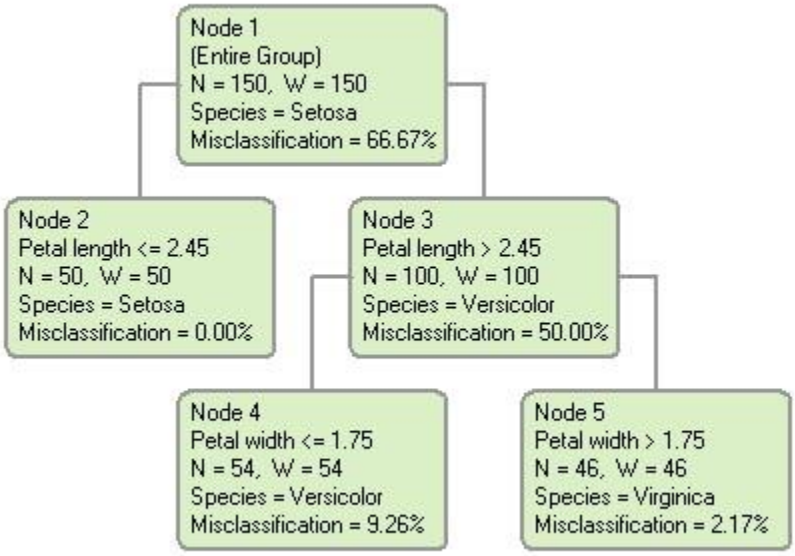
- You can use the tree to make inferences that help you understand the "big picture" of the model. One of the great advantages of decision trees is that they are easy to interpret even by non-technical people. For example, if the decision tree models product sales, a quick glance might tell you that men in the South buy more of your product than women in the North. If you are developing a model of health risks for insurance policies, a quick glance might tell you that smoking and age are important predictors of health.

- You can use the decision tree to identify target groups. For example, if you are looking for the best potential customers for a product, you can identify the terminal nodes in the tree that have the highest percentage of sales, and then focus your sales effort on individuals described by those nodes.

- You can predict the target value for specific cases where you know only the predictor variable values. This is known as "scoring". Scoring is described in the following section and, in more detail, on page 163.

Using a Decision Tree to Predict Target Variable Values (Scoring)

A decision tree can be used to predict the values of the target variable based on values of the predictor variables.



To determine the predicted value of a row, begin with the root node (node 1 above). Then decide whether to go into the left or right child node based on the value of the splitting variable. Continue this process using the splitting variable for successive child nodes until you reach a terminal, leaf node. The value of the target variable shown in the leaf node is the predicted value of the target variable.

For example, let's use the decision tree shown above to classify a case that has the following predictor values:

Petal length = 3.5
Petal width = 2.1

Begin the analysis by starting in the root node, node 1. The first split is made using the Petal length predictor. Since the value of Petal length in our case is 3.5, which is greater than the split point of 2.45, we move from node 1 into node 3. If we stopped at that point, the best estimate of Species would be Versicolor. Node 3 is split on a different predictor variable, Petal width. Our value of Petal width is 2.1, which is greater than the split point of 1.75, so we move into node 5. This is a terminal node, so we classify the species as Virginica, which is the category assigned to the terminal node.

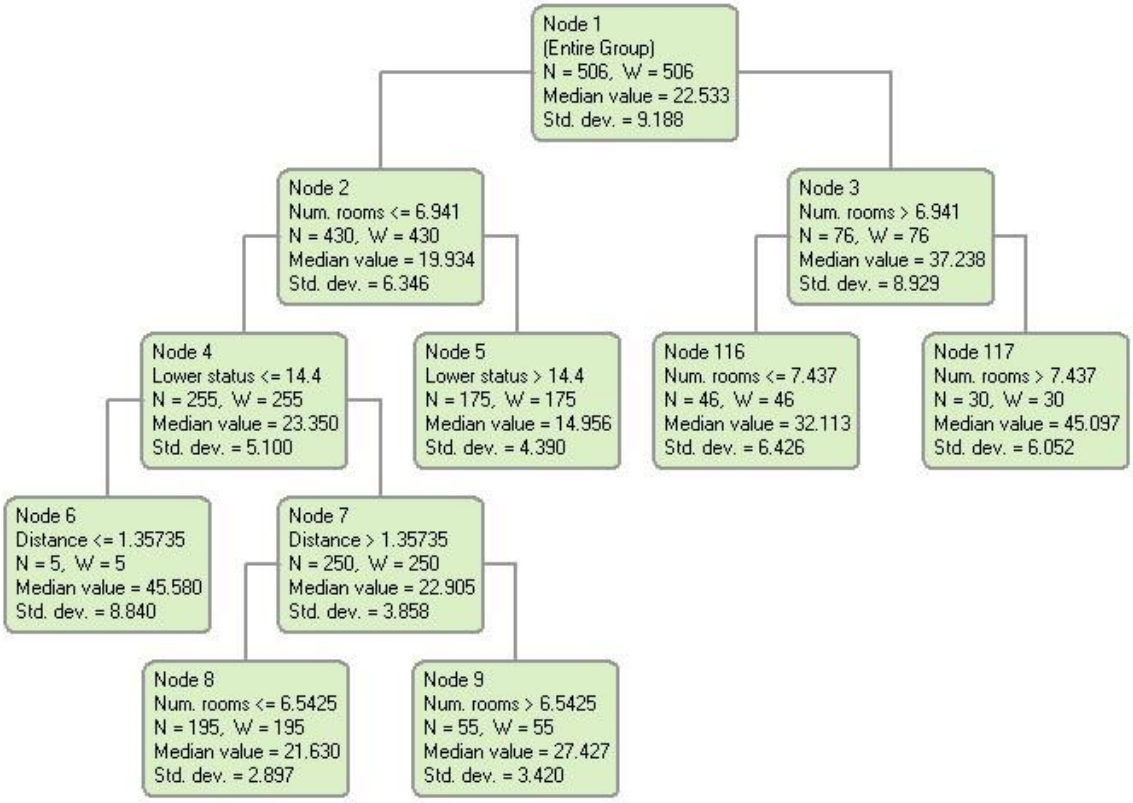
In the case of regression trees where the target variable is continuous, the mean value of the target variable for the rows falling in a leaf node is used as the predicted value of the target variable.

Regression and Classification Models

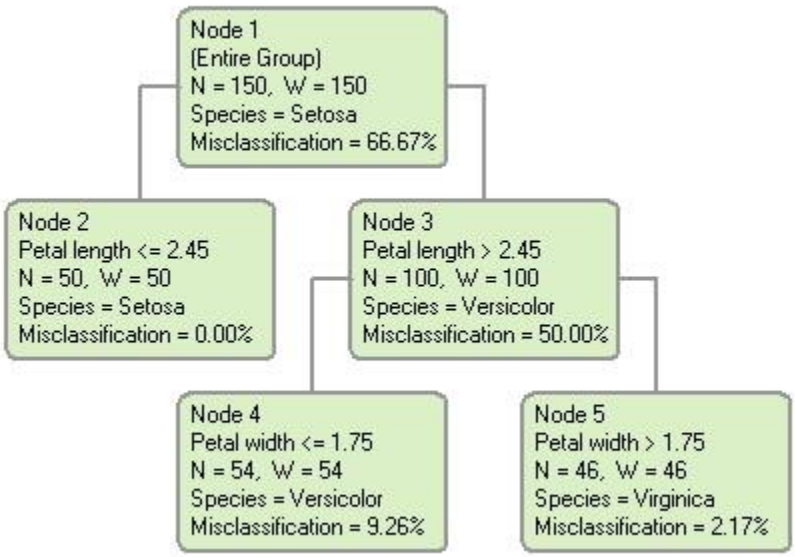
DTREG will generate a regression model or a classification model depending on whether the target variable is continuous or categorical.

Regression Models -- If the target variable is continuous, a regression model is generated. When using a regression tree to predict the value of the target variable, the mean value of the target variable of the rows falling in a terminal (leaf) node of the tree is the predicted value.

An example of a regression tree is shown below. In this example, the target variable is "Median value". From the tree we see that if the value of the predictor variable "Num. rooms" is greater than 6.941, then the estimated (average) value of the target variable is 37.238; whereas, if the number of rooms is less than or equal to 6.941, then the average value of the target variable is 19.934.



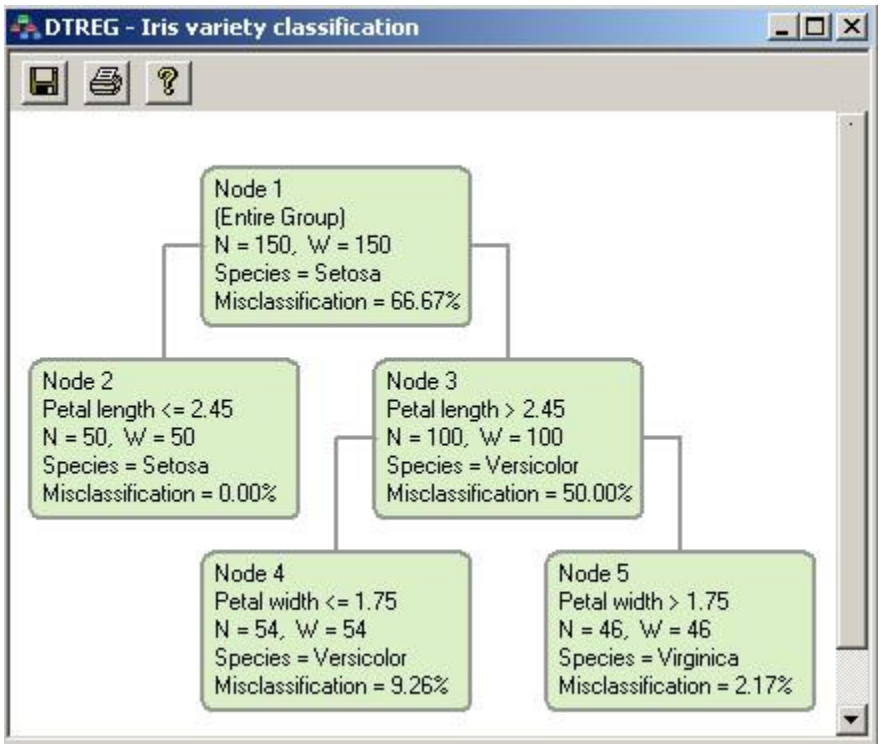
Classification Models -- If the target variable is categorical, then a classification model is generated. To predict the value (category) of the target variable using a classification tree, use the values of the predictor variables to move through the tree until you reach a terminal (leaf) node, then predict the category shown for that node. An example of a classification tree is shown below. The target variable is "Species", the species of Iris. We can see from the tree that if the value of the predictor variable "Petal length" is less than or equal to 2.45 the species is Setosa. If the petal length is greater than 2.45, then additional splits are required to classify the species.



Viewing a Decision Tree

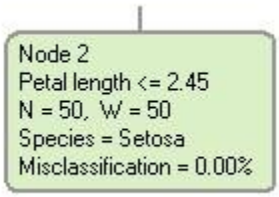
I think that I shall never see a poem lovely as a tree.
– Joyce Kilmer

Once an analysis has been completed, you can view the generated decision tree by clicking the toolbar icon or by clicking "View-tree" on the main menu.



What's in a node – Classification tree

The information displayed in each node depends on whether it is part of a classification tree (categorical target variable) or a regression tree (continuous target variable). Here is an example of a node from a classification tree:

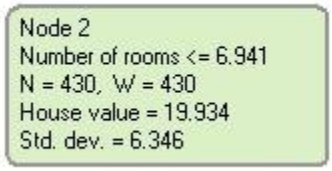


Five lines of information are presented in this node:

1. **Node number** – The top line displays the number of the node. This number allows you to match the node to the textual report for the analysis.
2. **Predictor variable used for split** – The second line displays the name of the predictor variable that was used to generate the split from the parent node (i.e., the split that generated this node). In this example, the parent node was split on "Petal length". Following the name of the predictor variable is either a "<=" or ">" sign indicating if values less than or equal or greater than the split point go into this node. In this example, it shows that records with values of Petal length less than or equal to 2.45 were placed in this node. The sibling node received records with Petal length greater than 2.45. If the predictor variable is categorical, the categories of the variable that were placed in this node are shown after the variable name.
3. **Record and weight counts** – The "N=*nn*" and "W=*nn*" values show how many rows (N) were placed in this node and the sum of the row weights (W). If no weight variable was specified, or all weights are 1.0, and the sum of the weights will equal the number of rows.
4. **Target variable category** – This line displays the name of the target variable ("Species") and the category of it that was assigned to this node ("Setosa"). See page 364 for information about how target categories are assigned to nodes.
5. **Misclassification percent** – This is the percentage of the rows in this node that had target variable categories different from the category that was assigned to the node. In other words, it is the percentage of rows that were misclassified.

What's in a node – Regression tree

The information shown in a node for a regression tree is illustrated below:



In his example, this node was produced by splitting its parent node on the predictor variable "Number of rooms". There were 430 rows with values of "Number of rooms" less than or equal to 6.941 that were assigned to this node.

The bottom two lines are different for regression trees than classification trees. The next-to-bottom line displays the name of the target variable ("House value") and the mean value of the target variable for all rows in this node. So, in this example, the mean value of "House value" is 19.934, and this would be the best predicted value for the target variable for rows falling in this node.

The bottom line displays the standard deviation for the mean target value.

The History of Decision Tree Analysis

The first widely-used program for generating decision trees was "AID" (Automatic Interaction Detection) developed in 1963 by J. N. Morgan and J. A. Sonquist[1]. Written in FORTRAN and limited by the hardware of the time, AID was suitable only for small to medium size data sets, and it could generate only regression trees. None the less, this pioneering program was well received and widely used during the 1960's and 70's.

AID was followed by many other decision tree generators including THAID by Morgan and Messenger in 1973[2], and ID3 and, later, C4.5 by J. Ross Quinlan[3].

The theoretical underpinning of decision tree analysis was greatly enhanced by the research done by Leo Breiman, Jerome Friedman, Richard Olshen and Charles Stone that was published in their book *Classification And Regression Trees*[4]. Much of their research was embedded in a program they developed called "CART"[5].

Recent advancements in decision tree analyses include the TreeBoost method developed by Jerome Friedman (Friedman, 1999b) and Decision Tree Forests developed by Leo Breiman (Breiman, 2001). Both of these methods use ensembles of trees to increase the predictive accuracy over a single-tree model. DTREG can generate single-tree, TreeBoost and Decision Tree Forest models.

---

[1] Morgan & Sonquist (1963) "Problems in the analysis of survey data and a proposal", JASA, 58, 415-434. (Original AID)

[2] Morgan & Messenger (1973) *THAID -- A sequential analysis program for the analysis of nominal scale dependent variables*, Survey Research Center, U of Michigan.

[3] Quinlan, J.R. (1993), *C4.5: Programs for Machine Learning*, Morgan Kaufman: San Mateo, CA.

[4] Breiman, L., Friedman, J.H., Olshen, R.A. & Stone, C.J. (1984), *Classification and Regression Trees*, Wadsworth: Belmont, CA.

[5] CART® is a registered trademark of Salford Systems.

TreeBoost – Stochastic Gradient Boosting

"**Boosting**" is a technique for improving the accuracy of a predictive function by applying the function repeatedly in a series and combining the output of each function with weighting so that the total error of the prediction is minimized. In many cases, the predictive accuracy of such a series greatly exceeds the accuracy of the base function used alone.

See page 54 for the TreeBoost property page where you select TreeBoost models and set parameters.

The **TreeBoost** algorithm used by DTREG was developed by Jerome H. Friedman (Friedman 1999) and is optimized for improving the accuracy of models built on decision trees. Research has shown that models built using TreeBoost are among the most accurate of any known modeling technique. TreeBoost is also known as "**Stochastic Gradient Boosting**" and "**Multiple Additive Regression Trees**" (**MART**).

The TreeBoost algorithm is functionally similar to **decision tree forests** because it creates a tree ensemble, but a TreeBoost model consists of a series of trees whereas a decision tree forest consists of a collection of trees grown in parallel. See the following chapter for information about decision tree forests.

Mathematically, a TreeBoost model can be described as:

$$PredictedTarget = F_0 + B_1 * T_1(X) + B_2 * T_2(X) + \ldots + B_M * T_M(X)$$

Where F_0 is the starting value for the series (the median target value for a regression model), X is a vector of "pseudo-residual" values remaining at this point in the series, $T_1(X)$, $T_2(X)$ are trees fitted to the pseudo-residuals and B_1, B_2, etc. are coefficients of the tree node predicted values that are computed by the TreeBoost algorithm.

Graphically, a TreeBoost model can be represented like this:

The first tree is fitted to the data. The residuals (error values) from the first tree are then fed into the second tree which attempts to reduce the error. This process is repeated through a series of successive trees. The final predicted value is formed by adding the weighted contribution of each tree.

Usually, the individual trees are fairly small (typically 3 levels deep with 8 terminal nodes), but the full TreeBoost additive series may consist of hundreds of these small trees.

Features of TreeBoost Models

- TreeBoost models often have a degree of accuracy that cannot be obtained using a large, single-tree model. TreeBoost models are often equal to or superior to any other predictive functions including neural networks.
- TreeBoost models have been shown to produce more accurate results than competing composite-tree methods such as bagging or boosting using other methods such as AdaBoost.
- TreeBoost models are as easy to create as single-tree models. By simply setting a control button, you can direct DTREG to create a single-tree model or a TreeBoost model for the same analysis.
- TreeBoost models can handle hundreds or thousands of potential predictor variables.
- Irrelevant predictor variables are identified automatically and do not affect the predictive model.
- TreeBoost uses the Huber M-regression loss function (Huber, 1964) which makes it highly resistant to outliers and misclassified cases.
- The sophisticated and accurate method of surrogate splitters is used for handling missing predictor values.
- The stochastic (randomization) element in the TreeBoost algorithm makes it highly resistant to over fitting.
- Cross-validation and random-row-sampling methods can be used to evaluate the generalization of a TreeBoost model and guard against over fitting.
- TreeBoost can be applied to regression models and *k*-class classification problems.
- TreeBoost can handle both continuous and categorical predictor and target variables. Variables with textual values like "Male" and "Female" can be used as well as numeric variables.
- TreeBoost models are grown quickly – in some cases up to 100 times as fast as neural networks.
- The TreeBoost algorithm achieves the accuracy of other boosting methods such as AdaBoost with much lower sensitivity to misclassified cases and outliers.

The primary *dis*advantage of TreeBoost is that the model is complex and cannot be visualized like a single tree. It is more of a "black box" like a neural network. Because of this, it is advisable to create both a single-tree and a TreeBoost model. The single-tree model can be studied to get an intuitive understanding of how the predictor variables relate, and the TreeBoost model can be used to score the data and generate highly accurate predictions.

How TreeBoost Models Are Created

Here is an outline of the TreeBoost algorithm for regression models. For more details, see Friedman (1999).

1. Find the median value of the target variable. This is the starting value for the series (F_0 in the mathematical description above).
2. Determine which rows will be used to build the next tree in the series. A specified proportion of the rows are chosen randomly, with the target variable values stratified. (In the case of a classification model, *influence trimming* may reduce the set of rows by removing insignificant ones.)
3. Sort the residual values for the rows being used and find the quantile cutoff point for the Huber-M loss function. The quantile cutoff point is specified as a TreeBoost parameter. The residual values are then transformed by Huber's method to reduce the effect of outliers. The transformed residual values are known as "pseudo residuals".
4. Fit a tree (T_1) to the pseudo residual values.
5. Compute the median of the pseudo residual values for the rows ending in each terminal node of the tree. This median becomes the predicted value for the terminal node. (In a single-tree model, the *mean* value of the target variable for rows ending in a node is the predicted value for the node.)
6. Sum the differences (residuals) between the predicted node value and the pseudo residuals that went into the tree build (with Huber's adjustment for outliers). Then compute the mean value of these residuals.
7. Compute the boost coefficient (B_1) for the node based on the difference between the mean residual values for the node and the median (predicted) value for the node.
8. Multiply the boost coefficient by the shrink factor to reduce the rate of learning.

For 2-category classification models, the TreeBoost method is essentially the same as for regression except logit (probability) values are fitted rather than raw target values. At the end of the process, the category that minimizes the misclassification cost is chosen as the predicted value.

K-category classification is more complex: In this case, the algorithm builds *K* parallel TreeBoost series to model the probability of each possible category. At the end of the process, the probability values for the categories are compared and the one that minimizes misclassification cost is chosen as the best predicted category. Since *K* TreeBoost series must be built in parallel, this process is computationally expensive if the target variable has many categories.

The TreeBoost algorithm generates the most accurate models with minimum over fitting if only a portion of the data rows are used to build each tree in the series (Friedman, 1999). This is the *stochastic* part of stochastic gradient boosting. You can specify the proportion of the rows used for each tree on the TreeBoost parameter screen (see page 54).

Research has shown (Friedman, 2001) that the predictive accuracy of a TreeBoost series can be improved by apply a weighting coefficient that is less than 1 ($0 < v < 1$) to each tree as the series is constructed. This coefficient is called the "shrinkage factor". The effect is to retard the learning rate of the series, so the series has to be longer to compensate for the shrinkage but its accuracy is better. Tests have shown that small shrinkage factors in the range of 0.1 yield dramatic improvements over TreeBoost series built with no shrinkage ($v = 1$). The tradeoff in using a small shrinkage factor is that the TreeBoost series is longer and the computational time increases. You can select the shrinkage factor on the TreeBoost parameter screen.

Decision Tree Forests

You can't see the forest for the trees.
– Anon.

A **Decision Tree Forest** consists of an ensemble (collection) of decision trees whose predictions are combined to make the overall prediction for the forest. A decision tree forest is similar to a TreeBoost model in the sense that a large number of trees are grown. However, TreeBoost generates a series of trees with the output of one tree going into the next tree in the series. In contrast, a decision tree forest grows a number of independent trees in parallel, and they do not interact until after all of them have been built.

Both TreeBoost and decision tree forests produce high accuracy models. Experiments have shown that TreeBoost works better with some applications and decision tree forests with others, so it is best to try both methods and compare the results.

The Decision Tree Forest technique used by DTREG is an implementation of the "Random Forest"™ algorithm developed by Leo Breiman (Breiman, 2001).[6]

Features of Decision Tree Forest Models

- Decision tree forest models often have a degree of accuracy that cannot be obtained using a large, single-tree model. Decision tree forest models are among the most accurate models yet invented.
- Decision tree forest models are as easy to create as single-tree models. By simply setting a control button, you can direct DTREG to create a single-tree model or a decision tree forest model or a TreeBoost model for the same analysis.
- Decision tree forests use the "out of bag" data rows for validation of the model. This provides an independent test without requiring a separate data set or holding back rows from the tree construction.
- Decision tree forest models can handle hundreds or thousands of potential predictor variables.
- The sophisticated and accurate method of surrogate splitters is used for handling missing predictor values.
- The stochastic (randomization) element in the decision tree forest algorithm makes it highly resistant to over fitting.
- Decision tree forests can be applied to regression and classification models.

The primary *dis*advantage of decision tree forests is that the model is complex and cannot be visualized like a single tree. It is more of a "black box" like a neural network.

[6] "Random Forest" is a trademark of Leo Breiman and Adele Cutler and is licensed exclusively to Salford Systems, San Diego, CA.

Because of this, it is advisable to create both a single-tree and a decision tree forest model. The single-tree model can be studied to get an intuitive understanding of how the predictor variables relate, and the decision tree forest model can be used to score the data and generate highly accurate predictions.

How Decision Tree Forests Are Created

Here is an outline of the algorithm used to construct a decision tree forest:

Assume the full data set consists of *N* observations.

1. Take a random sample of *N* observations from the data set with replacement (this is called "bagging"). Some observations will be selected more than once, and others will not be selected. On average, about 2/3 of the rows will be selected by the sampling. The remaining 1/3 of the rows are called the "out of bag (OOB)" rows. A new random selection of rows is performed for each tree constructed.

2. Using the rows selected in step 1, construct a decision tree. Build the tree to the maximum size, and do not prune it. As the tree is built, allow only a subset of the total set of predictor variables to be considered as possible splitters for each node. Select the set of predictors to be considered as a random subset of the total set of available predictors. For example, if there are ten predictors, choose a random five as candidate splitters. Perform a new random selection for each split. Some predictors (possibly the best one) will not be considered for each split, but a predictor excluded from one split may be used for another split in the same tree.

3. Repeat steps 1 and 2 a large number of times constructing a forest of trees.

4. To "score" a row, run the row through each tree in the forest and record the predicted value (i.e., terminal node) that the row ends up in (just as you would score using a single-tree model). For a regression analysis, compute the average score predicted by all of the trees. For a classification analysis, use the predicted categories for each tree as "votes" for the best category, and use the category with the most votes as the predicted category for the row.

Decision tree forests have two stochastic (randomizing) elements: (1) the selection of data rows used as input for each tree, and (2) the set of predictor variables considered as candidates for each node split. For reasons that are not well understood, these randomizations along with combining the predictions from the trees significantly improve the overall predictive accuracy.

No Over fitting or Pruning

"Over fitting" is a problem in large, single-tree models where the model begins to fit noise in the data. When such a model is applied to data not used to build the model, the model does not perform well (i.e., it does not generalize well). To avoid this problem, single-tree models must be pruned to the optimal size. In nearly all cases, decision tree forests do not have a problem with over fitting, and there is no need to prune the trees in the forest. Generally, the more trees in the forest, the better the fit.

Internal Measure of Test Set (Generalization) Error

When a decision tree forest is constructed using the algorithm outlined above, about 1/3 of data rows are excluded from each tree in the forest. The rows that are excluded from a tree are called the "out of bag (OOB)" rows for the tree; each tree will have a different set of out-of-bag rows. Since the out of bag rows are (by definition) not used to build the tree, they constitute an independent test sample for the tree.

To measure the generalization error of the decision tree forest, the out of bag rows for each tree are run through the tree and the error rate of the prediction is computed. The error rates for all of the trees in the forest are then averaged to give the overall generalization error rate for the entire forest.

There are several advantages to this method of computing generalization error: (1) all of the rows are used to construct the model, and none have to be held back as a separate test set, (2) the testing is fast because only one forest has to be constructed (as compared to *V*-fold cross-validation where additional trees have to be constructed).

See page 60 for the Decision Tree Forest property page where you select Decision Tree Forest models and set parameters.

Multilayer Perceptron Neural Networks

Call it a network, call it a tribe, call it a family. Whatever you call it, whoever you are, you need one.
– Jane Howard, "*Families*"

Neural networks are predictive models loosely based on the action of biological neurons. The following diagram by Jonathan Rosen illustrates the design and operation of a neural network:

Just kidding!

A Brief History of Neural Networks

The selection of the name "neural network" was one of the great PR successes of the Twentieth Century. It certainly sounds more exciting than a technical description such as "A network of weighted, additive values with nonlinear transfer functions". However, despite the name, neural networks are far from "thinking machines" or "artificial brains". A typical artificial neural network might have a hundred neurons. In comparison, the human nervous system is believed to have about $3x10^{10}$ neurons. We are still light years from "Data" on *Star Trek*.

The original "Perceptron" model was developed by Frank Rosenblatt in 1958. Rosenblatt's model consisted of three layers, (1) a "retina" that distributed inputs to the second layer, (2) "association units" that combine the inputs with weights and trigger a threshold step function which feeds to the output layer, (3) the output layer which combines the values. Unfortunately, the use of a step function in the neurons made the perceptions difficult or impossible to train. A critical analysis of perceptrons published in 1969 by Marvin Minsky and Seymore Papert pointed out a number of critical weaknesses of perceptrons, and, for a period of time, interest in perceptrons waned.

Interest in neural networks was revived in 1986 when David Rumelhart, Geoffrey Hinton and Ronald Williams published "Learning Internal Representations by Error Propagation". They proposed a multilayer neural network with nonlinear but differentiable transfer functions that avoided the pitfalls of the original perceptron's step functions. They also provided a reasonably effective training algorithm for neural networks.

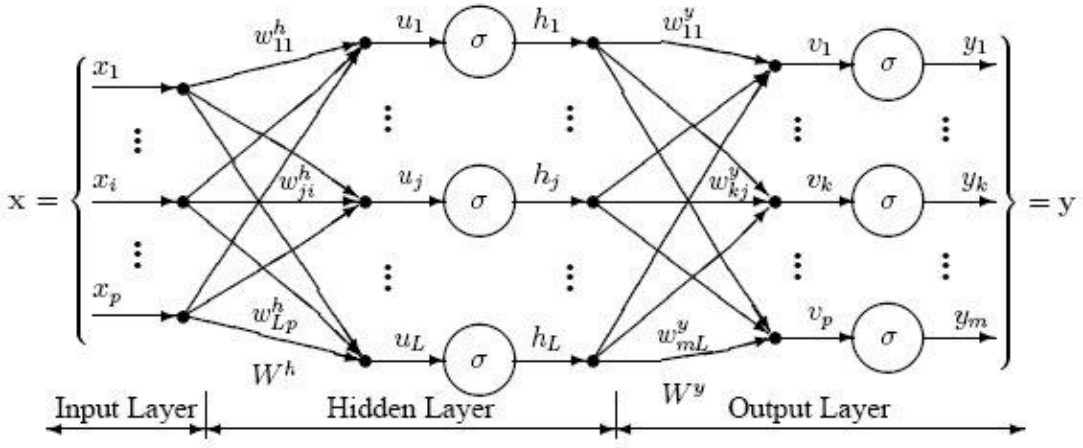
Types of Neural Networks

When used without qualification, the terms "Neural Network" (NN) and "Artificial Neural Network" (ANN) usually refer to a *Multilayer Perceptron Network* (MLP). However, there are many other types of neural networks including Probabilistic Neural Networks, General Regression Neural Networks, Radial Basis Function Networks, Polynomial Neural Networks (GMDH), Cascade Correlation, Functional Link Networks, Kohonen networks, Gram-Charlier networks, Learning Vector Quantization, Hebb networks, Adaline networks, Heteroassociative networks, Recurrent Networks and Hybrid Networks.

DTREG implements the most widely used types of neural networks: Multilayer Perceptron Networks (MLP), Probabilistic Neural Networks (PNN) and General Regression Neural Networks (GRNN), Radial Basic Function (RBF) networks, Polynomial Neural Networks (GMDH), and Cascade Correlation networks. This chapter describes Multilayer Perception Networks.

The Multilayer Perceptron Neural Network Model

The following diagram illustrates a perceptron network with three layers:



This network has an **input layer** (on the left) with three neurons, one **hidden layer** (in the middle) with three neurons and an **output layer** (on the right) with three neurons.

There is one neuron in the input layer for each predictor variable ($x_1 \ldots x_p$). In the case of categorical variables, N-1 neurons are used to represent the N categories of the variable.

Input Layer

A vector of predictor variable values ($x_1 \ldots x_p$) is presented to the input layer. The input layer (or processing before the input layer) standardizes these values so that the range of each variable is -1 to 1. The input layer distributes the values to each of the neurons in the hidden layer. In addition to the predictor variables, there is a constant input of 1.0, called the *bias* that is fed to each of the hidden layers; the bias is multiplied by a weight and added to the sum going into the neuron.

Hidden Layer

Arriving at a neuron in the hidden layer, the value from each input neuron is multiplied by a weight (w_{ji}), and the resulting weighted values are added together producing a combined value u_j. The weighted sum (u_j) is fed into a transfer function, σ, which outputs a value h_j. The outputs from the hidden layer are distributed to the output layer.

Output Layer

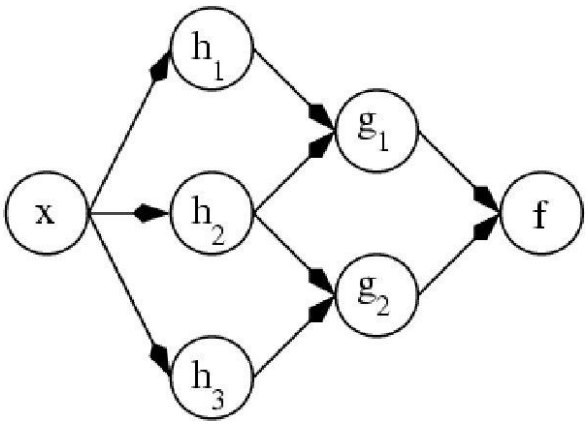
Arriving at a neuron in the output layer, the value from each hidden layer neuron is multiplied by a weight (w_{kj}), and the resulting weighted values are added together producing a combined value v_j. The weighted sum (v_j) is fed into a transfer function, σ, which outputs a value y_k. The y values are the outputs of the network.

If a regression analysis is being performed with a continuous target variable, then there is a single neuron in the output layer, and it generates a single y value. For classification problems with a binary-value categorical target variable, there is a single output neuron whose value determines whether the output category is predicted to be 1 or 0. For classification problems with more than two target categories, there are N neurons in the output layer producing N values, one for each of the N categories of the target variable.

Multilayer Perceptron Architecture

The network diagram shown above is a full-connected, three layer, feed forward, perceptron neural network. "Fully connected" means that the output from each input and hidden neuron is distributed to all of the neurons in the following layer. "Feed forward" means that the values only move from input to hidden to output layers; no values are fed back to earlier layers (a Recurrent Network allows values to be fed backward).

All neural networks have an input layer and an output layer, but the number of hidden layers may vary. Here is a diagram of a perceptron network with two hidden layers and four total layers:



When there is more than one hidden layer, the output from one hidden layer is fed into the next hidden layer and separate weights are applied to the sum going into each layer.

Training Multilayer Perceptron Networks

The goal of the training process is to find the set of weight values that will cause the output from the neural network to match the actual target values as closely as possible.

There are several issues involved in designing and training a multilayer perceptron network:

- Selecting how many hidden layers to use in the network.
- Deciding how many neurons to use in each hidden layer.
- Finding a globally optimal solution that avoids local minima.
- Converging to an optimal solution in a reasonable period of time.
- Validating the neural network to test for over fitting.

Selecting the Number of Hidden Layers

For nearly all problems, one hidden layer is sufficient. Two hidden layers are required for modeling data with discontinuities such as a saw tooth wave pattern. Using two hidden layers rarely improves the model, and it may introduce a greater risk of converging to a local minima. There is no theoretical reason for using more than two hidden layers. DTREG can build models with one or two hidden layers. Three layer models with one hidden layer are recommended.

Deciding how many neurons to use in the hidden layers

One of the most important characteristics of a multilayer perceptron network is the number of neurons in the hidden layer(s). If an inadequate number of neurons are used, the network will be unable to model complex data, and the resulting fit will be poor.

If too many neurons are used, the training time may become excessively long, and, worse, the network may over fit the data. When over fitting occurs, the network will begin to model random noise in the data. The result is that the model fits the training data extremely well, but it generalizes poorly to new, unseen data. Validation must be used to test for this.

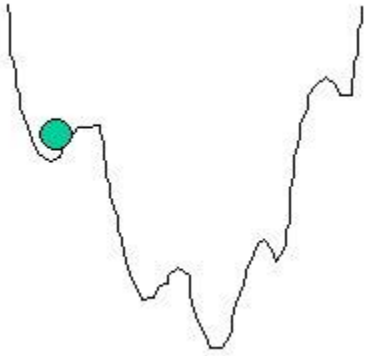
DTREG includes an automated feature to find the optimal number of neurons in the hidden layer. (See page 64 for details.) You specify the minimum and maximum number of neurons you want it to test, and it will build models using varying numbers of neurons and measure the quality using either cross validation or hold-out data not used for training. This is a highly effective method for finding the optimal number of neurons, but it is computationally expensive, because many models must be built, and each model has to be validated. If you have a multiprocessor computer, you can configure DTREG to use multiple CPU's during the process. See page 16 for additional information.

The automated search for the optimal number of neurons only searches the first hidden layer. If you select a model with two hidden layers, you must manually specify the number of neurons in the second hidden layer.

Finding a globally optimal solution

A typical neural network might have a couple of hundred weighs whose values must be found to produce an optimal solution. If neural networks were linear models like linear regression, it would be a breeze to find the optimal set of weights. But the output of a neural network as a function of the weights is often highly nonlinear; this makes the optimization process complex.

If you plotted the error as a function of the weights, you would likely see a rough surface with many local minima such as this:



This picture is highly simplified because it represents only a single weight value (on the horizontal axis). With a typical neural network, you would have a 200-dimension, rough surface with many local valleys.

Optimization methods such as steepest descent and conjugate gradient are highly susceptible to finding local minima if they begin the search in a valley near a local minimum. They have no ability to see the big picture and find the global minimum.

DTREG uses the Nguyen-Widrow algorithm (Nguyen, 1990) to select the initial range of starting weight values. It then uses the conjugate gradient algorithm to optimize the weights. Conjugate gradient usually finds the optimum weights quickly, but there is no guarantee that the weight values it finds are globally optimal. So it is useful to allow DTREG to try the optimization multiple times with different sets of initial random weight values. The number of tries allowed using randomly selected starting weights is specified on the Multilayer Perceptron property page (see page 67).

Converging to the Optimal Solution – Conjugate Gradient

Given a set of randomly-selected starting weight values, DTREG uses the conjugate gradient algorithm to optimize the weight values.

Most training algorithms follow this cycle to refine the weight values:

1. Run the predictor values for a case through the network using a tentative set of weights.
2. Compute the difference between the predicted target value and the actual target value for the case. This is the error of the prediction.
3. Average the error information over the entire set of training cases.
4. Propagate the error backward through the network and compute the gradient (vector of derivatives) of the change in error with respect to changes in weight values.
5. Make adjustments to the weights to reduce the error.

Each cycle is called an *epoch*.

Because the error information is propagated backward through the network, this type of training method is called *backward propagation* or "backprop".

The *backpropagation* training algorithm was first described by Rumelhart and McClelland in 1986; it was the first practical method for training neural networks. The original procedure used the *gradient descent* algorithm to adjust the weights toward convergence using the gradient. Because of this history, the term "backpropagation" or "backprop" often is used to denote a neural network training algorithm using gradient descent as the core algorithm. That is somewhat unfortunate since backward propagation of error information through the network is used by nearly all training algorithms, some of which are much better than gradient descent.

Backpropagation using gradient descent often converges very slowly or not at all. On large-scale problems its success depends on user-specified *learning rate* and *momentum* parameters. There is no automatic way to select these parameters, and if incorrect values are specified the convergence may be exceedingly slow, or it may not converge at all. While backpropagation with gradient descent is still used in many neural network programs, it is no longer considered to be the best or fastest algorithm.

DTREG uses the *conjugate gradient* algorithm to adjust weight values using the gradient during the backward propagation of errors through the network. Compared to gradient descent, the conjugate gradient algorithm takes a more direct path to the optimal set of weight values. Usually, conjugate gradient is significantly faster and more robust than gradient descent. Conjugate gradient also does not require the user to specify learning rate and momentum parameters.

The traditional conjugate gradient algorithm uses the gradient to compute a search direction. It then uses a line search algorithm such as *Brent's Method* to find the optimal step size along a line in the search direction. The line search avoids the need to compute the Hessian matrix of second derivatives, but it requires computing the error at multiple points along the line. The conjugate gradient algorithm with line search (CGL) has been used successfully in many neural network programs, and is considered one of the best methods yet invented.

DTREG provides the traditional conjugate gradient algorithm with line search, but it also offers a newer algorithm, *Scaled Conjugate Gradient* (see Moller, 1993).

The scaled conjugate gradient algorithm uses a numerical approximation for the second derivatives (Hessian matrix), but it avoids instability by combining the model-trust region approach from the Levenberg-Marquardt algorithm with the conjugate gradient approach. This allows scaled conjugate gradient to compute the optimal step size in the search direction without having to perform the computationally expensive line search used by the traditional conjugate gradient algorithm. Of course, there is a cost involved in estimating the second derivatives.

Tests performed by Moller show the scaled conjugate gradient algorithm converging up to twice as fast as traditional conjugate gradient and up to 20 times as fast as backpropagation using gradient descent. Moller's tests also showed that scaled conjugate gradient failed to converge less often than traditional conjugate gradient or backpropagation using gradient descent.

Avoiding Over fitting

"Over fitting" occurs when the parameters of a model are tuned so tightly that the model fits the training data well but has poor accuracy on separate data not used for training. Multilayer perceptrons are subject to over fitting as are most other types of models.

DTREG has two methods for dealing with over fitting: (1) by selecting the optimal number of neurons as described above, and (2) by evaluating the model as the parameters are being tuned and stopping the tuning when over fitting is detected. This is known as "early stopping".

If you enable the early-stopping option, DTREG holds out a specified percentage of the training rows and uses them to check for over fitting as model tuning is performed. The tuning process uses the training data to search for optimal parameter values. But as this process is running, the model is evaluated on the hold-out test rows, and the error from that test is compared with the error computed using previous parameter values. If the error on the test rows does not decrease after a specified number of iterations then DTREG stops the training and uses the parameters which produced the lowest error on the test data.

See page 67 for information about setting the parameters for the conjugate gradient algorithm.

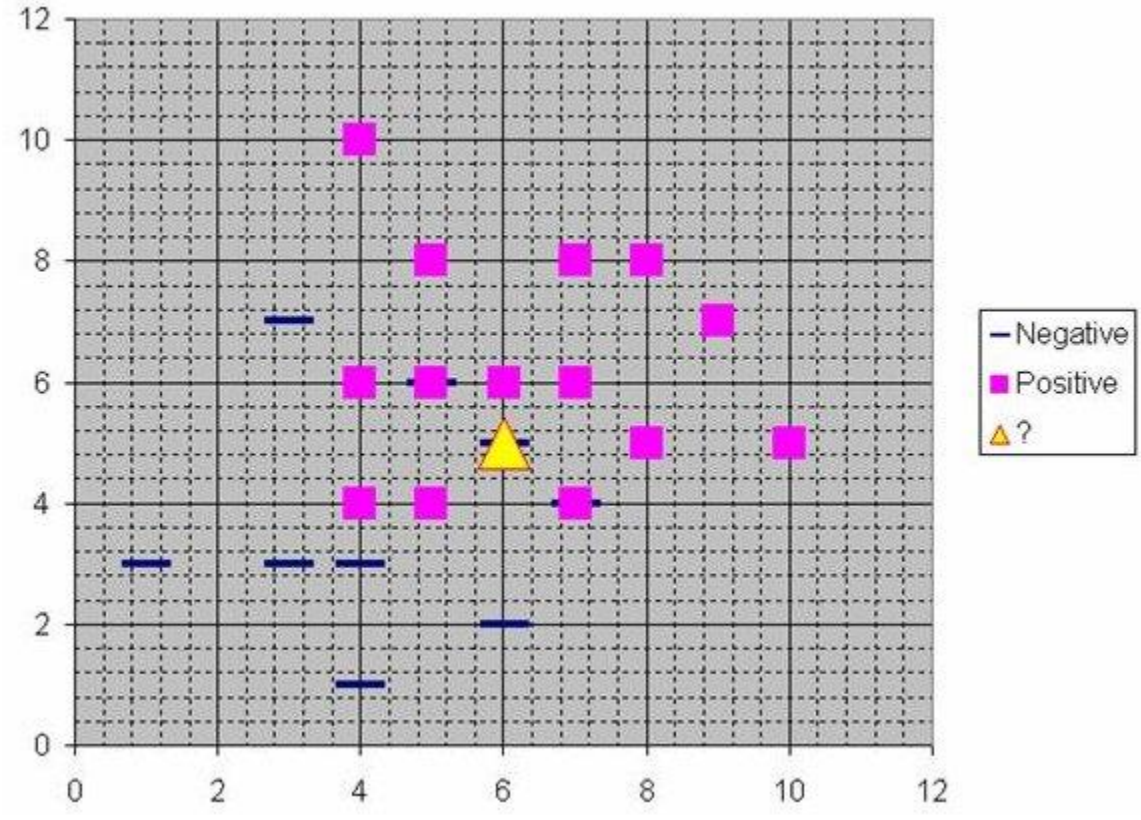
Radial Basis Function (RBF) Neural Networks

A Radial Basis Function (RBF) neural network has an input layer, a hidden layer and an output layer. The neurons in the hidden layer contain Gaussian transfer functions whose outputs are inversely proportional to the distance from the center of the neuron.

RBF networks are very similar to PNN/GRNN networks (see page 279). The main difference is that PNN/GRNN networks have one neuron for each point in the training file, whereas RBF networks have a variable number of neurons that is usually much less than the number of training points. For problems with small to medium size training sets, PNN/GRNN networks are usually more accurate than RBF networks, but PNN/GRNN networks are impractical for large training sets.

How RBF networks work

Although the implementation is very different, RBF neural networks are conceptually similar to *K-Nearest Neighbor* (k-NN) models. The basic idea is that a predicted target value of an item is likely to be about the same as other items that have close values of the predictor variables. Consider this figure:



Assume that each case in the training set has two predictor variables, *x* and *y*. The cases are plotted using their *x,y* coordinates as shown in the figure. Also assume that the target

variable has two categories, *positive* which is denoted by a square and *negative* which is denoted by a dash. Now, suppose we are trying to predict the value of a new case represented by the triangle with predictor values x=6, y=5.1. Should we predict the target as positive or negative?

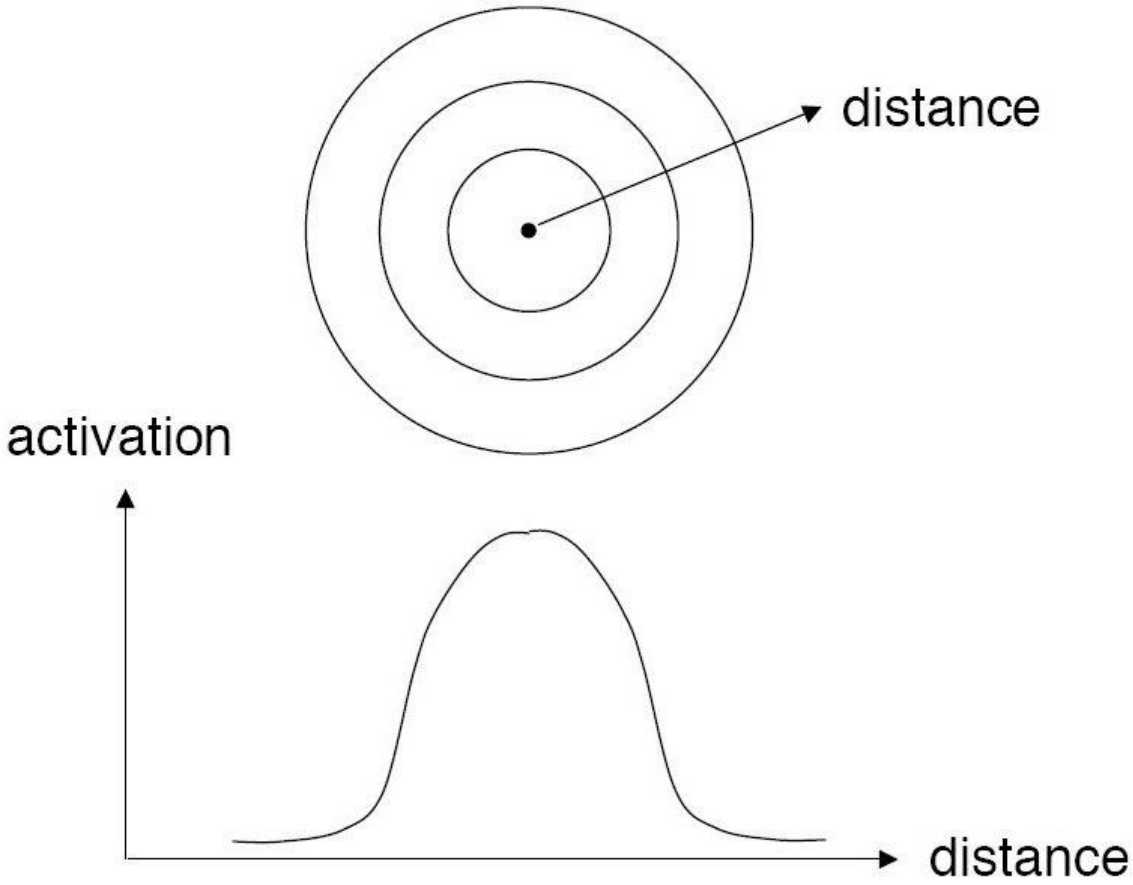
Notice that the triangle is position almost exactly on top of a dash representing a negative value. But that dash is in a fairly unusual position compared to the other dashes which are clustered below the squares and left of center. So it could be that the underlying negative value is an odd case.

The nearest neighbor classification performed for this example depends on how many neighboring points are considered. If 1-NN is used and only the closest point is considered, then clearly the new point should be classified as negative since it is on top of a known negative point. On the other hand, if 9-NN classification is used and the closest 9 points are considered, then the effect of the surrounding 8 positive points may overbalance the close negative point.

An RBF network positions one or more RBF neurons in the space described by the predictor variables (x,y in this example). This space has as many dimensions as there are predictor variables. The Euclidean distance is computed from the point being evaluated (e.g., the triangle in this figure) to the center of each neuron, and a *radial basis function* (RBF) (also called a *kernel function*) is applied to the distance to compute the weight (influence) for each neuron. The radial basis function is so named because the radius distance is the argument to the function.

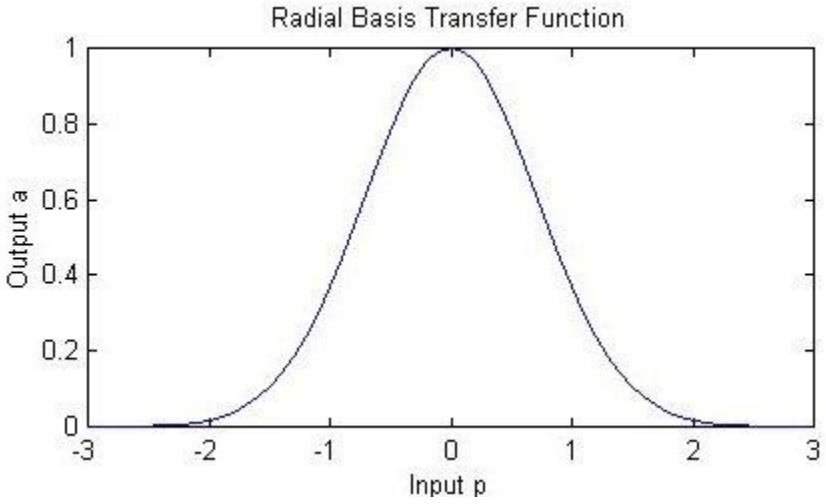
$$\textit{Weight} = \text{RBF}(\textit{distance})$$

The further a neuron is from the point being evaluated, the less influence it has.

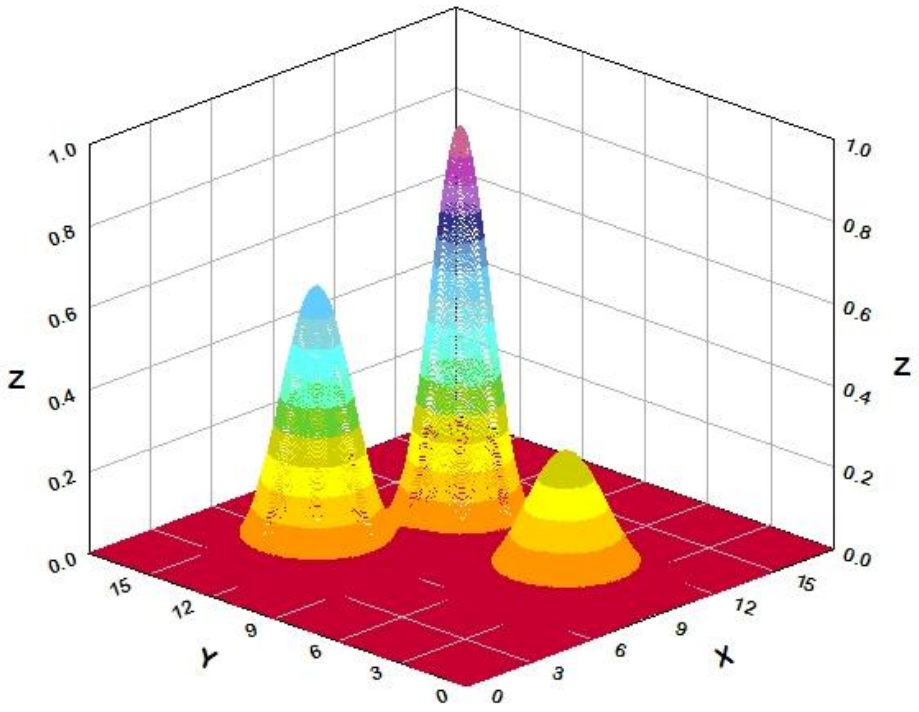


Radial Basis Function

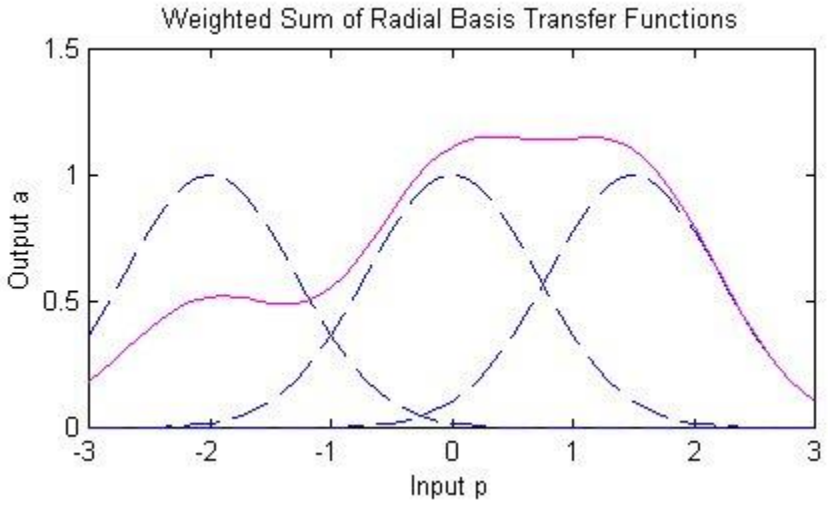
Different types of radial basis functions could be used, but the most common is the Gaussian function:



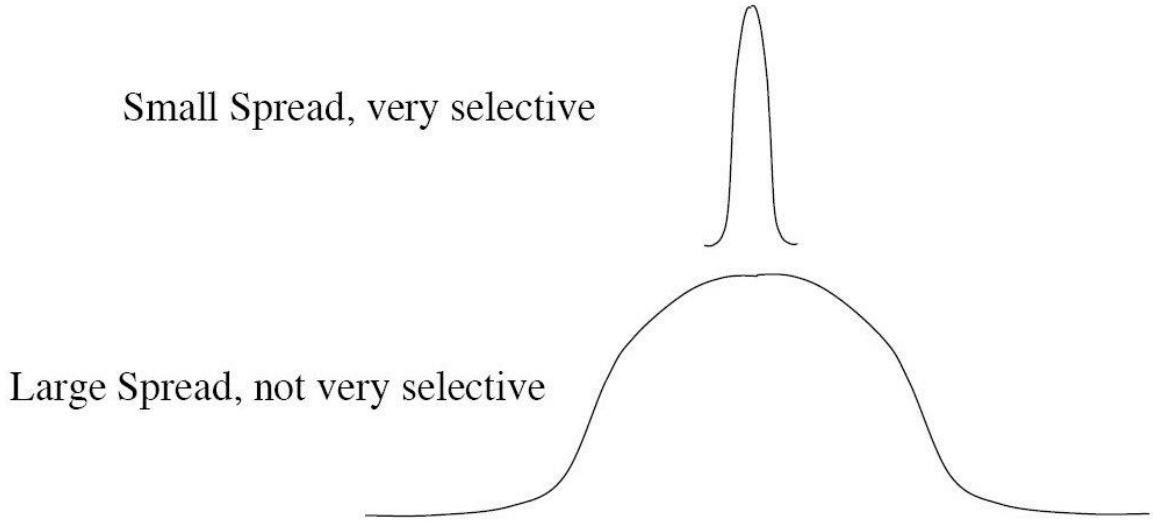
If there is more than one predictor variable, then the RBF function has as many dimensions as there are variables. The following picture illustrates three neurons in a space with two predictor variables, X and Y. Z is the value coming out of the RBF functions:



The best predicted value for the new point is found by summing the output values of the RBF functions multiplied by weights computed for each neuron.

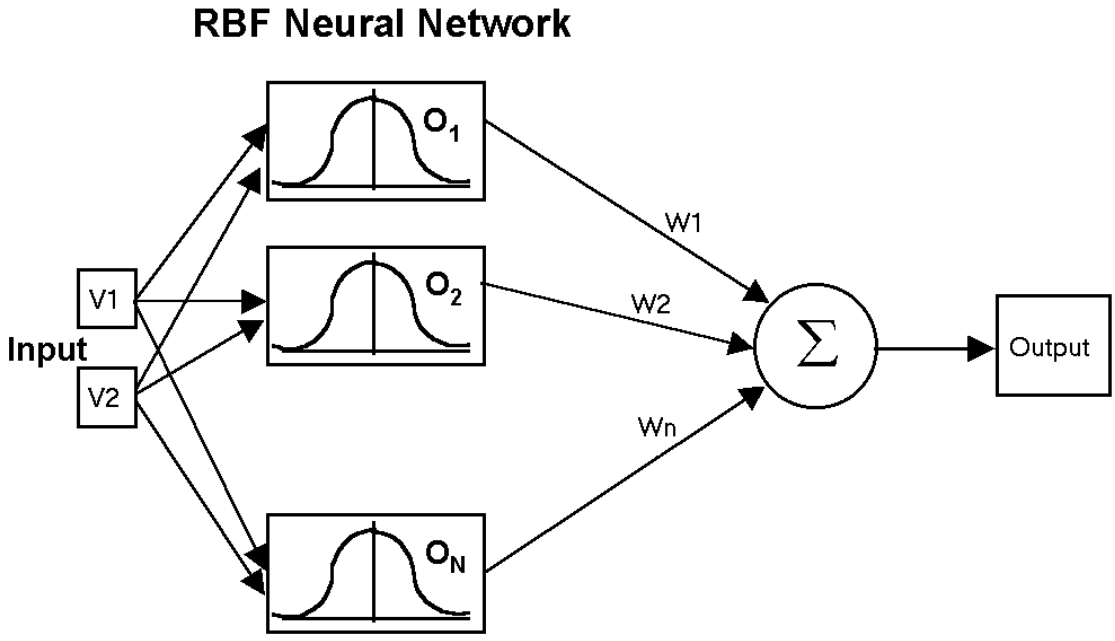


The radial basis function for a neuron has a center and a radius (also called a spread). The radius may be different for each neuron, and, in RBF networks generated by DTREG, the radius may be different in each dimension.



With larger spread, neurons at a distance from a point have a greater influence.

RBF Network Architecture



RBF networks have three layers:

Input layer – There is one neuron in the input layer for each predictor variable. In the case of categorical variables, *N*-1 neurons are used where *N* is the number of categories. The input neurons (or processing before the input layer) standardizes the range of the values by subtracting the median and dividing by the interquartile range. The input neurons then feed the values to each of the neurons in the hidden layer.

Hidden layer – This layer has a variable number of neurons (the optimal number is determined by the training process). Each neuron consists of a radial basis function centered on a point with as many dimensions as there are predictor variables. The spread (radius) of the RBF function may be different for each dimension. The centers and spreads are determined by the training process. When presented with the ***x*** vector of input values from the input layer, a hidden neuron computes the Euclidean distance of the test case from the neuron's center point and then applies the RBF kernel function to this distance using the spread values. The resulting value is passed to the the summation layer.

Summation layer – The value coming out of a neuron in the hidden layer is multiplied by a weight associated with the neuron (W1, W2, ...,W*n* in this figure) and passed to the summation which adds up the weighted values and presents this sum as the output of the network. Not shown in this figure is a bias value of 1.0 that is multiplied by a weight W0 and fed into the summation layer. For classification problems, there is one output (and a separate set of weights and summation unit) for each target category. The value output for a category is the probability that the case being evaluated has that category.

Training RBF Networks

The following parameters are determined by the training process:

1. The number of neurons in the hidden layer.
2. The coordinates of the center of each hidden-layer RBF function.
3. The radius (spread) of each RBF function in each dimension.
4. The weights applied to the RBF function outputs as they are passed to the summation layer.

Various methods have been used to train RBF networks. One approach first uses K-means clustering to find cluster centers which are then used as the centers for the RBF functions. However, K-means clustering is a computationally intensive procedure, and it often does not generate the optimal number of centers. Another approach is to use a random subset of the training points as the centers.

DTREG uses a training algorithm developed by Sheng Chen, Xia Hong and Chris J. Harris (Chen, Hong, Harris, 2005). This algorithm uses an evolutionary approach to determine the optimal center points and spreads for each neuron. It also determines when

to stop adding neurons to the network by monitoring the estimated leave-one-out (LOO) error and terminating when the LOO error beings to increase due to over fitting.

The computation of the optimal weights between the neurons in the hidden layer and the summation layer is done using ridge regression.. An iterative procedure developed by Mark Orr (Orr, 1966) is used to compute the optimal regularization Lambda parameter that minimizes generalized cross-validation (GCV) error.

See page 69 for information about parameters that control the training process.

GMDH Polynomial Neural Networks

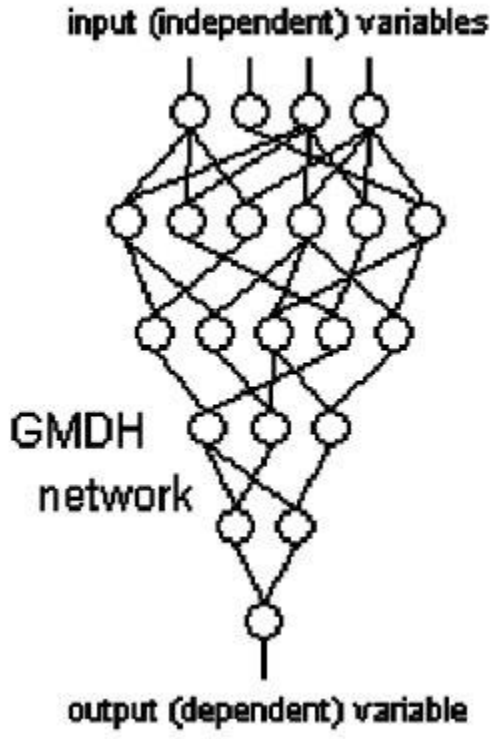
Group Method of Data Handling (GMDH) polynomial neural networks are "self organizing" networks. The network begins with only input neurons. During the training process, neurons are selected from a pool of candidates and added to the hidden layers.

GMDH networks were originated in 1968 by Prof Alexey G. Ivakhnenko at the Institute of Cybernetics in Kyiv (Ukraine).

Structure of a GMDH network

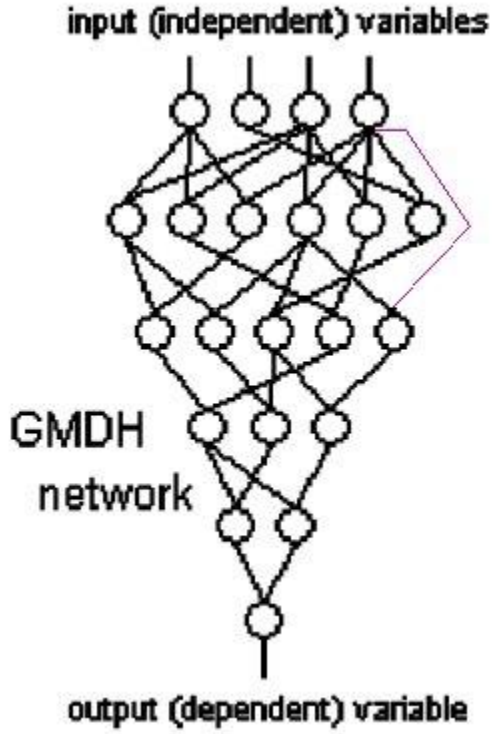
GMDH networks are *self organizing*. This means that the connections between neurons in the network are not fixed but rather are selected during training to optimize the network. The number of layers in the network also is selected automatically to produce maximum accuracy without over fitting.

The following figure from Kordik, Naplava, Snorek illustrates the structure of a basic GMDH network using polynomial functions of two variables:



The first layer (at the top) presents one input for each predictor variable. Each neuron in the second layer draws its inputs from two of the input variables. The neurons in the third layer draw their inputs from two of the neurons in the previous layer; this progresses through each layer. The final layer (at the bottom) draws its two inputs from the previous layer and produces a single value which is the output of the network.

Inputs to neurons in GMDH networks can skip layers and come from the original variables or layers several layers earlier as illustrated by this figure:



In this network, the neuron at the right end of the third layer is connected to an input variable rather than the output of a neuron on the previous layer.

Traditional GMDH neural networks use complete quadratic polynomials of two variables as transfer functions in the neurons. These polynomials have the form:

$$y = p_0 + p_1 x_1 + p_2 x_2 + p_3 x_1^2 + p_4 x_2^2 + p_5 x_1 x_2$$

DTREG extends GMDH networks by allowing you to select which functions may be used in the network. See the GMDH property page description on page 73 for information about selecting functions.

GMDH Training Algorithm

Two sets of input data are used during the training process: (1) the primary *training data*, and (2) the *control data* which is used to stop the building process when over fitting occurs. The control data typically has about 20% as many rows as the training data. The percentage is specified as a training parameter.

The GMDH network training algorithm proceeds as follows:

1. Construct the first layer which simply presents each of the input predictor variable values.
2. Using the allowed set of functions, construct all possible functions using combinations of inputs from the previous layer. If only two-variable polynomials are enabled, there will be $n*(n-1)/2$ candidate neurons constructed where n is the number of neurons in the previous layer. If the option is selected to allow inputs from the previous layer and the input layer, then n will the sum of the number of neurons in the previous layer and the input layer. If the option is selected to allow inputs from any layer, then n will the sum of the number of input variables plus the number of neurons in all previous layers.
3. Use least squares regression to compute the optimal parameters for the function in each candidate neuron to make it best fit the *training* data. Singular value decomposition (SVD) is used to avoid problems with singular matrices. If nonlinear functions are selected such as logistic or asymptotic, a nonlinear fitting routine based on Levenberg-Marquardt method is used.
4. Compute the mean squared error for each neuron by applying it to the *control* data. Note, the control data is different from the training data.
5. Sort the candidate neurons in order of increasing error.
6. Select the best (smallest error) neurons from the candidate set for the next layer. A model-building parameter specifies how many neurons are used in each layer.
7. If the error for the best neuron in the layer as measured with the control data is better than the error from the best neuron in the previous layer, and the maximum number of layers has not been reached, then jump back to step 2 to construct the next layer. Otherwise, stop the training. Note, when over fitting begins, the error as measured with the control data will being to increase, thus stopping the training.

If you are running on a multi-core CPU system, DTREG will perform GMDH training in parallel using multiple CPU's. See page 16 for information about setting how many CPU's to use.

Output Generated for GMDH Networks

In addition to the usual information reported for a model, DTREG displays the actual GMDH polynomial network generated. Here is an example:

```
  ============  GMDH Model  ============

N(3) = 0.650821+5.931812e+012*Age{Adult}-
5.931812e+012*Age{Adult}^2+4.685991e+015*Class{Second}-
4.685991e+015*Class{Second}^2+0.048843*Age{Adult}*Class{Second}

N(1) = 13.82502-2.390281e+012*Class{Crew}+4.78725e+011*Class{Crew}^2-
28.1043*N(3)+11.9577*N(3)^2+2.959348e+012*Class{Crew}*N(3)
```

```
N(7) = 0.325439+9.737558e+013*Sex{Male}-
9.737558e+013*Sex{Male}^2+9.00077e+015*Class{Second}-
9.00077e+015*Class{Second}^2+1.081648*Sex{Male}*Class{Second}

N(9) = 0.372317+2.225363e+013*Sex{Male}-2.225363e+013*Sex{Male}^2-
3.960104e+015*Class{First}+3.960104e+015*Class{First}^2-
0.244027*Sex{Male}*Class{First}

N(6) = -0.263746+1.221826*N(7)+0.268249*N(7)^2+1.636075*N(9)-
0.172757*N(9)^2-2.019208*N(7)*N(9)

Survived{Yes} = -0.008121+1.631212*N(1)-2.485552*N(1)^2-
0.186281*N(6)+0.126492*N(6)^2+1.720465*N(1)*N(6)
```

Output from neuron *i* is shown as N(*i*). Categorical predictor variables such as Sex are shown with the activation category in braces. For example, "Sex{Male}" has the value 1 if the value of Sex is "Male", and it has the value 0 if Sex is any other category. The final line shows the output of the network. In this case, the probability of Survived being Yes is predicted. Note how the inputs to each neuron are drawn from the outputs of neurons in lower levels of the network. This example uses only two-variable quadratic functions.

Cascade Correlation Neural Networks

Cascade correlation neural networks (Fahlman and Libiere, 1990) are "self-organizing" networks. The network begins with only input and output neurons. During the training process, neurons are selected from a pool of candidates and added to the hidden layer.

Cascade correlation networks have several advantages over multi-layer perceptron (MLP) neural networks:

1. Because they are self-organizing and grow the hidden layer during training, you do not have to be concerned with the issue of deciding how many layers and neurons to use in the network.
2. Training time is very fast – often 100 times as fast as a multilayer perceptron network. This makes cascade correlation networks suitable for large training sets.
3. Typically, cascade correlation networks are fairly small, often having fewer than a dozen neurons in the hidden layer. Contrast this to probabilistic neural networks which require a hidden-layer neuron for each training case.
4. Cascade correlation network training is quite robust, and good results usually can be obtained with little or no adjustment of parameters.
5. Cascade correlation is less likely to get trapped in local minima than MLP networks.

As with all types of models, there are some disadvantages to cascade correlation networks:

1. They have an extreme potential for over fitting the training data; this results in excellent accuracy on the training data but poor accuracy on new, unseen data. DTREG includes an over fitting control facility to prevent this.
2. Cascade correlation networks usually are less accurate than probabilistic and general regression neural networks on small to medium size problems (i.e., fewer than a couple of thousand training rows). But cascade correlation scales up to handle large problems far better than probabilistic or general regression networks.

Cascade Correlation Network Architecture

A cascade correlation network consists of a cascade architecture, in which hidden neurons are added to the network one at a time and do not change after they have been added. It is called a *cascade* because the output from all neurons already in the network feed into new neurons. As new neurons are added to the hidden layer, the learning algorithm attempts to maximize the magnitude of the correlation between the new neuron's output and the residual error of the network which we are trying to minimize.

A cascade correlation neural network has three layers: input, hidden and output.

Input Layer

A vector of predictor variable values ($x_1 \ldots x_p$) is presented to the input layer. The input neurons perform no action on the values other than distributing them to the neurons in the hidden and output layers. In addition to the predictor variables, there is a constant input of 1.0, called the *bias* that is fed to each of the hidden and output neurons; the bias is multiplied by a weight and added to the sum going into the neuron.

Hidden Layer

Arriving at a neuron in the hidden layer, the value from each input neuron is multiplied by a weight (w_{ji}), and the resulting weighted values are added together producing a combined value u_j. The weighted sum (u_j) is fed into a transfer function, σ, which outputs a value h_j. The outputs from the hidden layer are distributed to the output layer.

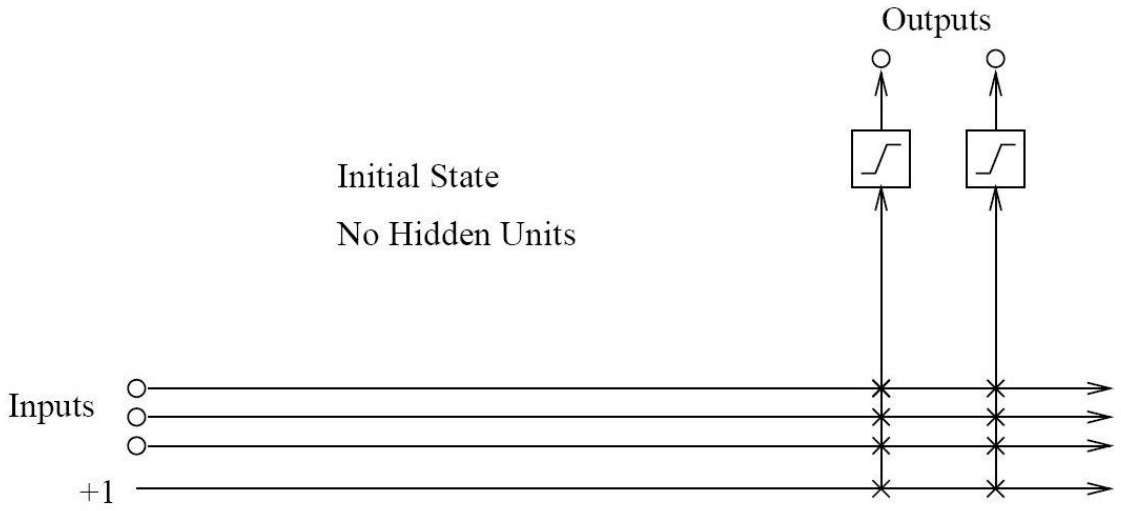
Output Layer

For regression problems, there is only a single neuron in the output layer. For classification problems that have binary outcomes, there is a single output neuron whose value varies from 0 to 1 with the outcome class being determined by whether the value is closer to 1 or 0. For classification problems with more than two target categories, there is a neuron for each category of the target variable, and the output of a neuron represents the probably of the corresponding category.

Each output neuron receives values from all of the input neurons (including the bias) and all of the hidden layer neurons. Each value presented to an output neuron is multiplied by a weight (w_{kj}), and the resulting weighted values are added together producing a combined value v_j. The weighted sum (v_j) is fed into a transfer function, σ, which outputs a value y_k. The y values are the outputs of the network. For regression problems, a linear transfer function is used in the output neurons. For classification problems, a sigmoid transfer function is used.

Training Algorithm for Cascade Correlation Networks

Initially, a cascade correlation neural network consists of only the input and output layer neurons with no hidden layer neurons. Every input is connected to every output neuron by a connection with an adjustable weight, as shown below:

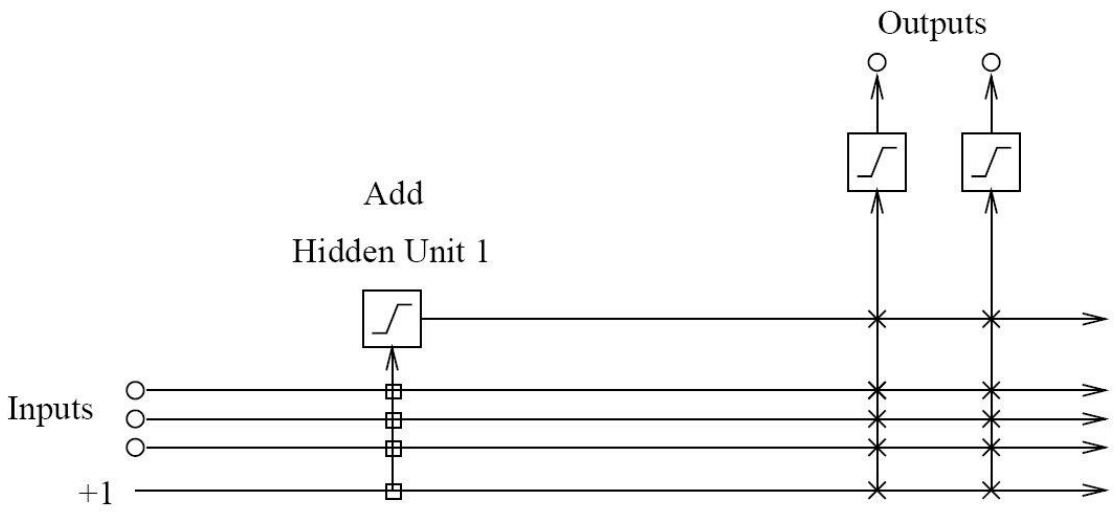


Each 'x' represents a weight value between the input and the output neuron. Values on a vertical line are added together after being multiplied by their weights. So each output neuron receives as its input a weighted sum from all of the input neurons including the bias. The output neuron sends this weighted input sum through its transfer function to produce the final output.

Even a simple cascade correlation network with no hidden neurons has considerable predictive power. For a fair number of problems, a cascade correlation network with just input and output layers provides good predictions.

Neurons are added to the hidden layer one by one. Each new hidden neuron receives a connection from each of the network's original inputs and also from every pre-existing hidden neuron (hence it is a cascade architecture). The hidden neuron's input weights are trained and then frozen at the time the unit is added to the net; only the output connection weights are trained repeatedly. Each new neuron therefore adds a new one-unit "layer" to the network. This leads to the creation of very powerful high-order feature detectors; it also may lead to very deep networks with a large number of inputs to the output neurons.

After the addition of the first hidden neuron, the network would have this structure:



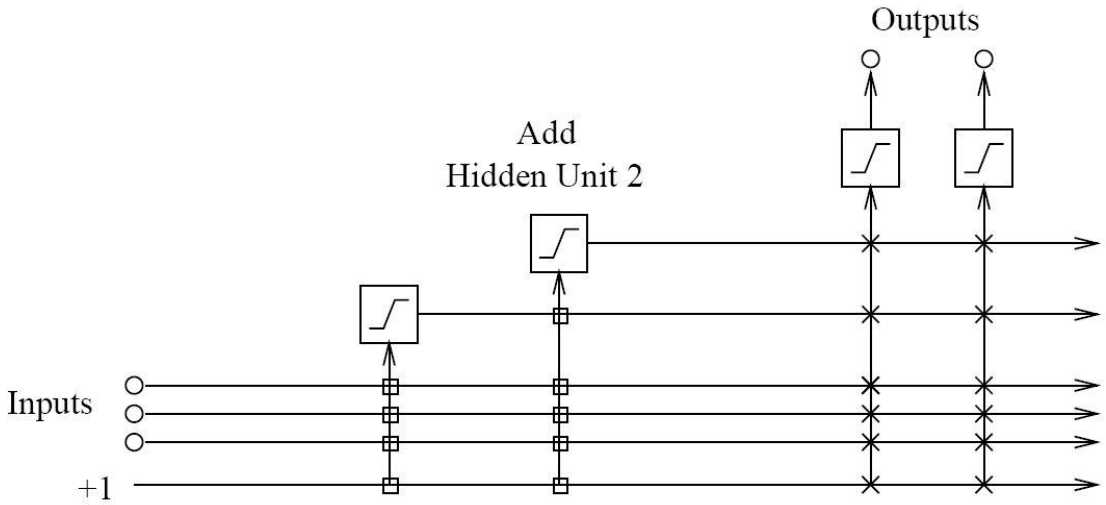
The input weights for the hidden neuron are shown as square boxes to indicate that they are fixed once the neuron has been added. Weights for the output neurons shown as 'x' continue to be adjusted during the training process.

To create a new hidden neuron, we begin with a candidate neuron that receives trainable input connections from all of the network's external inputs and from all pre-existing hidden neurons. The output of this candidate neuron is not yet connected to the active network. We run a number of passes over the examples in the training set, adjusting the candidate neuron's input weights after each pass. The goal of this adjustment is to maximize the sum over all output neurons of the magnitude of the correlation between the candidate neuron's value and the residual output error observed at the outputs.

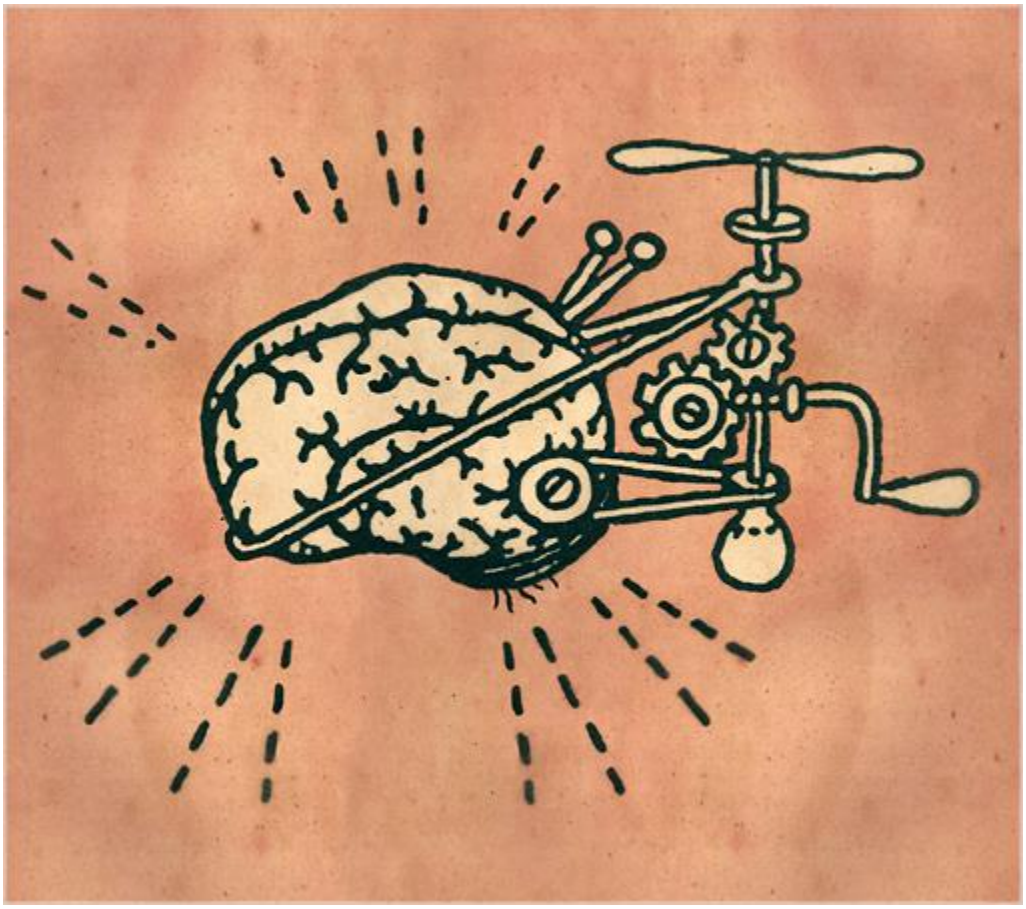
A candidate neuron cares only about the magnitude of its correlation with the error at a given output, and not about the sign of the correlation. As a rule, if a hidden neuron correlates positively with the error at a given output neuron, it will develop a negative connection weight to that neuron, attempting to cancel some of the error; if the correlation is negative, the output weight will be positive. Since a neuron's weights to different output neurons may be of mixed sign, a neuron can sometimes server two purposes by developing a positive correlation with the error at one output and a negative correlation with the error at another.

Instead of simply training a single candidate neuron, DTREG uses a pool of candidate neurons, each with a different set of random initial weights. If allowed, the candidate neurons also may have a mixture of transfer functions (sigmoid and Gaussian). All candidates receive the same input signals and see the same residual error for each training case. After all of the candidate neurons have been training to have maximum correlation with the output error, the candidate with the highest correlation is selected from the pool and added to the hidden layer. The output neuron weights are then trained using the all of their inputs including the output from the new hidden neuron. Note that the input weights for the other hidden neurons that are already part of the network are not retrained.

Here is a schematic of a network with two hidden neurons. Note how the second neuron receives inputs from the external inputs and pre-existing hidden neurons.



Probabilistic and General Regression Neural Networks



Probabilistic and General Regression Neural Networks have similar architectures, but there is a fundamental difference: Probabilistic networks perform classification where the target variable is categorical, whereas general regression neural networks perform regression where the target variable is continuous. If you select a PNN/GRNN network, DTREG will automatically select the correct type of network based on the type of target variable.

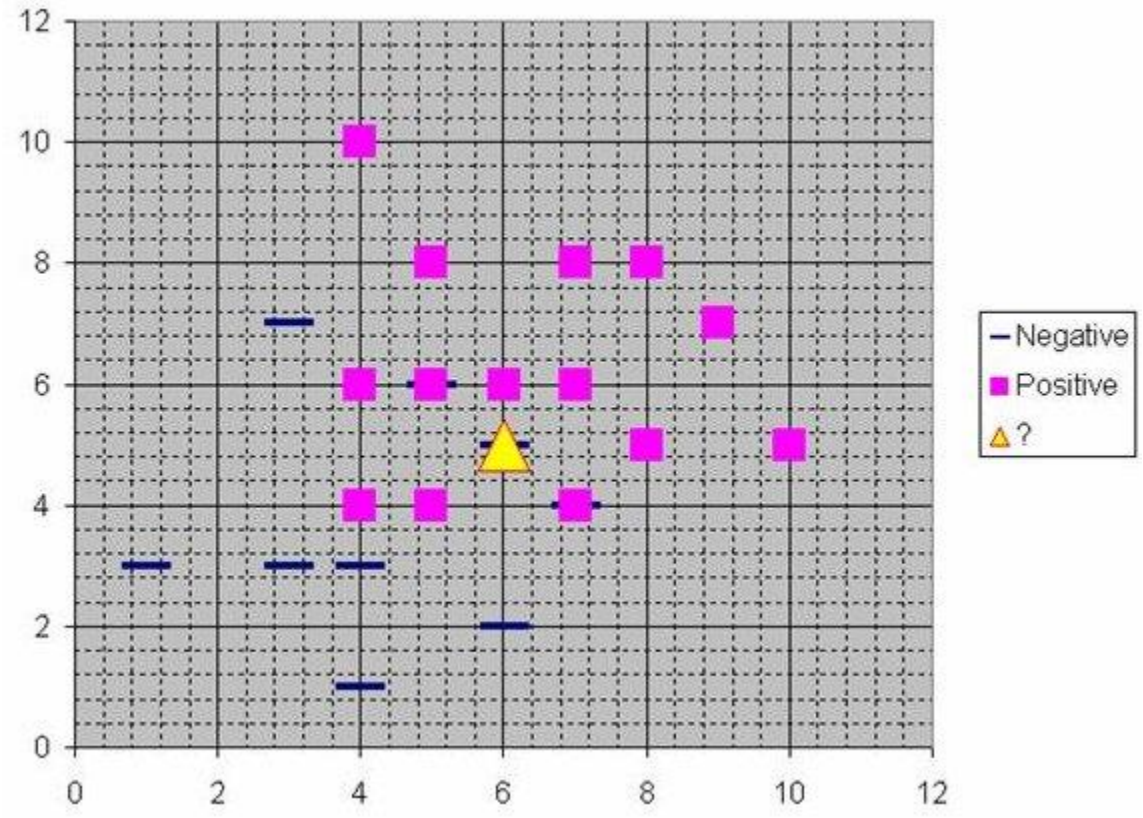
PNN and GRNN networks have advantages and disadvantages compared to multilayer perceptron (MLP) networks:

- It is usually faster to train a PNN/GRNN network than a MLP network.
- PNN/GRNN networks often are more accurate than MLP networks.
- PNN/GRNN networks are relatively insensitive to outliers (wild points).
- PNN networks generate accurate predicted target probability scores.
- PNN networks approach Bayes optimal classification.
- PNN/GRNN networks are slower than MLP networks at classifying new cases.

- PNN/GRNN networks require more memory space to store the model.
- PNN/GRNN networks are very similar to RBF networks with a large number of nodes. See page 261 for information about RBF networks.

How PNN/GRNN networks work

Although the implementation is very different, probabilistic neural networks are conceptually similar to *K-Nearest Neighbor* (k-NN) models. The basic idea is that the predicted target value of an item is likely to be about the same as other items that have close values (i.e., close proximity in multi-dimensional space) of the training data predictor variables. Consider this figure:



Assume that each case in the training set has two predictor variables, *x* and *y*. The cases are plotted using their *x,y* coordinates as shown in the figure. Also assume that the target variable has two categories, *positive* which is denoted by a square and *negative* which is denoted by a dash. Now, suppose we are trying to predict the value of a new case represented by the triangle with predictor values *x*=6, *y*=5.1. Should we predict the target as positive or negative?

Notice that the triangle is positioned almost exactly on top of a dash representing a negative value. But that dash is in a fairly unusual position compared to the other dashes

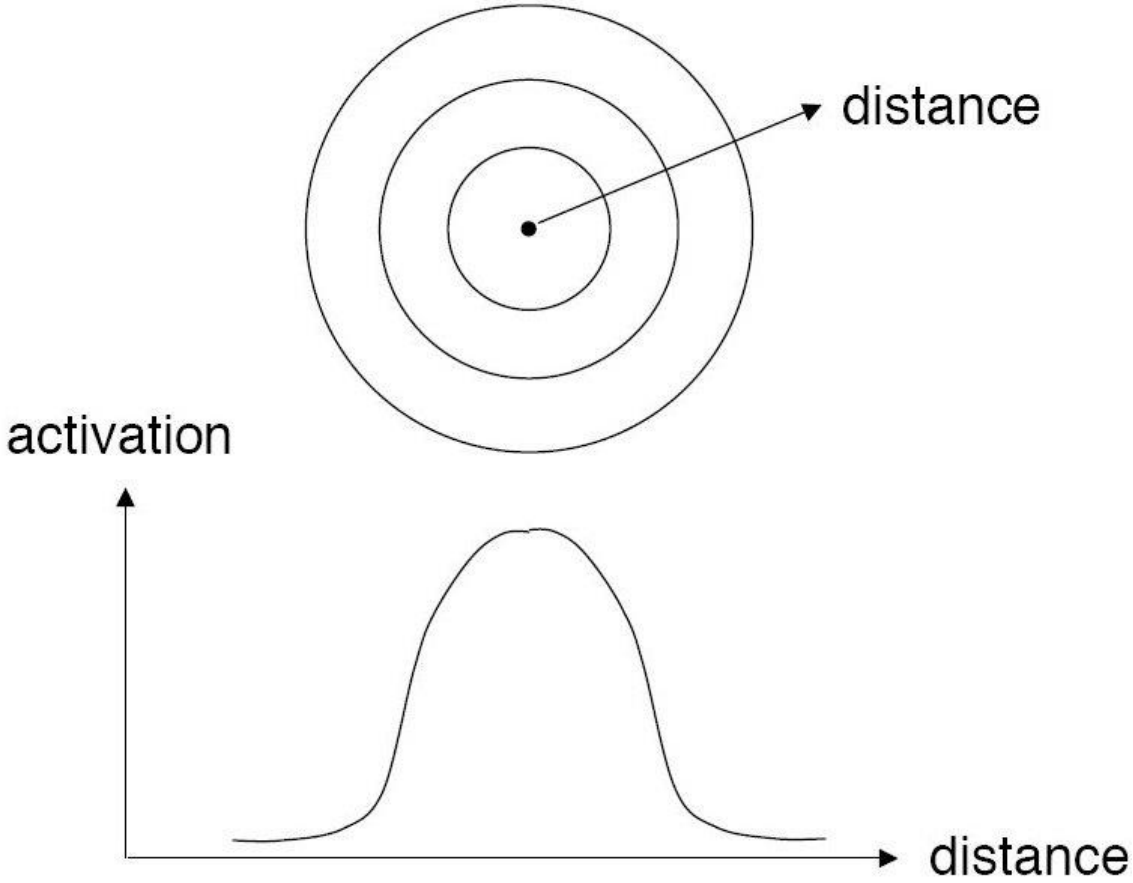
which are clustered below the squares and left of center. So it could be that the underlying negative value is an odd case.

The nearest neighbor classification performed for this example depends on how many neighboring points are considered. If 1-NN is used and only the closest point is considered, then clearly the new point should be classified as negative since it is on top of a known negative point. On the other hand, if 9-NN classification is used and the closest 9 points are considered, then the effect of the surrounding 8 positive points may overbalance the close negative point.

A probabilistic neural network builds on this foundation and generalizes it to consider *all* of the other training points. The distance is computed from the point being evaluated to each of the other points, and a *radial basis function* (RBF) (also called a *kernel function*) is applied to the distance to compute the weight (influence) for each point. The radial basis function is so named because the radius distance is the argument to the function.

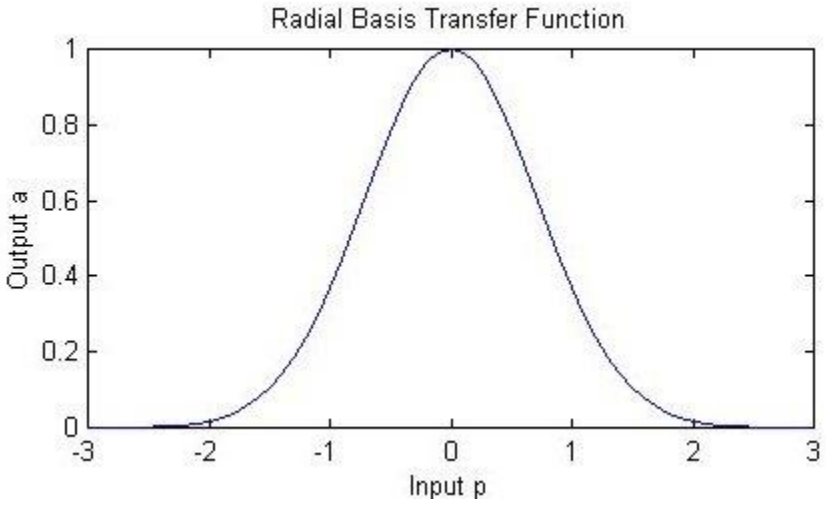
Weight = RBF(*distance*)

The further some other point is from the new point, the less influence it has.

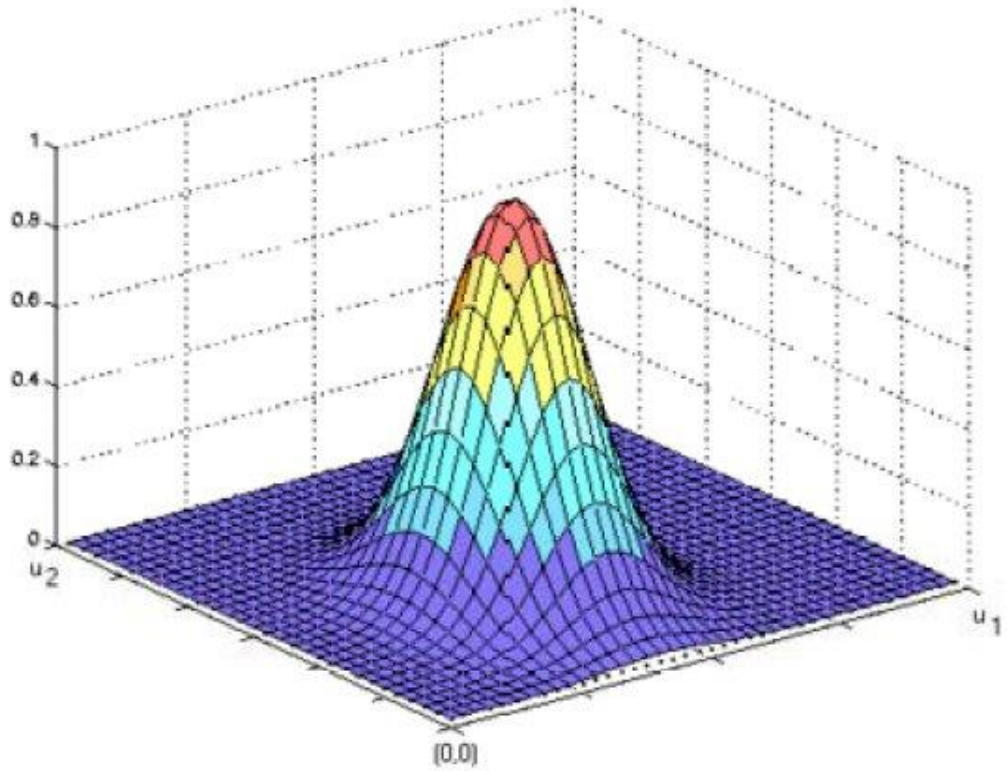


Radial Basis Function

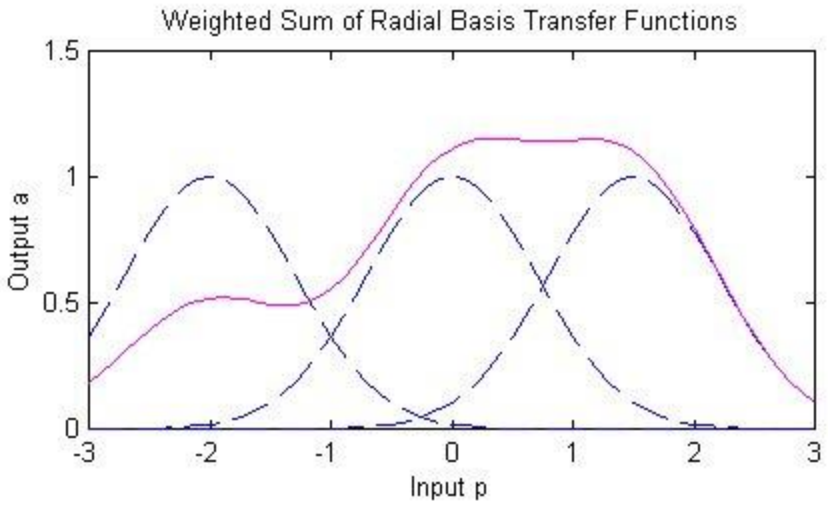
Different types of radial basis functions could be used, but the most common is the Gaussian function:



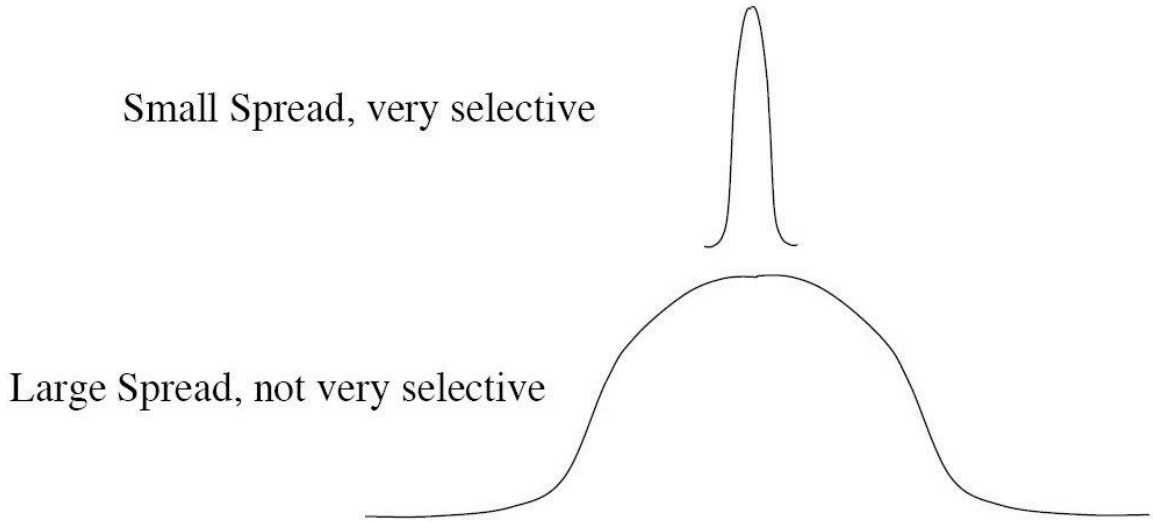
If there is more than one predictor variable, then the RBF function has as many dimensions as there are variables. Here is a RBF function for two variables:



The best predicted value for the new point is found by summing the values of the other points weighted by the RBF function.



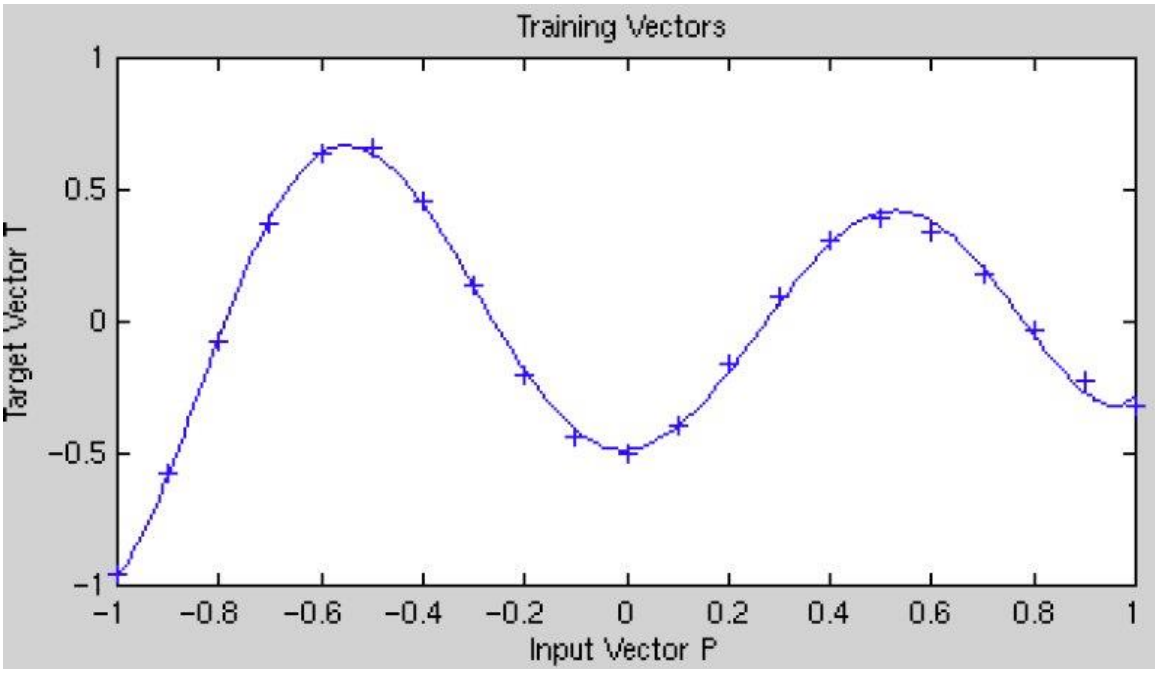
The peak of the radial basis function is always centered on the point it is weighting. The sigma value (σ) of the function determines the spread of the RBF function; that is, how quickly the function declines as the distance increased from the point.



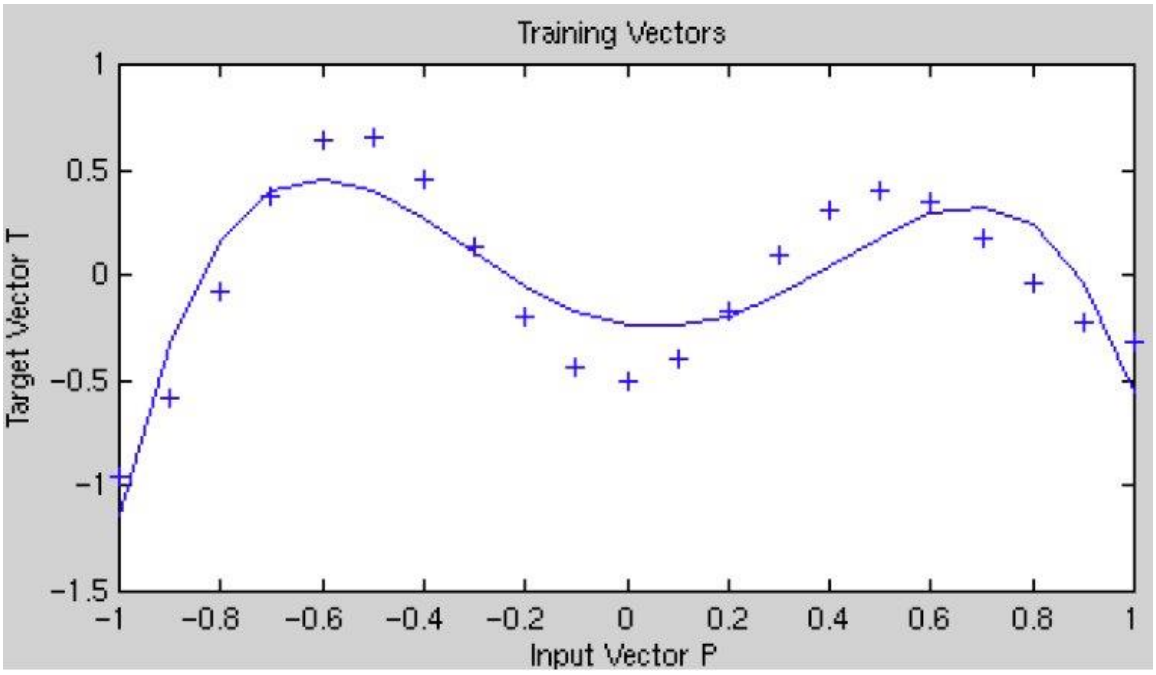
With larger sigma values and more spread, distant points have a greater influence.

The primary work of training a PNN or GRNN network is selecting the optimal sigma values to control the spread of the RBF functions. DTREG uses the conjugate gradient algorithm to compute the optimal sigma values. See page 80 for information about setting the parameters for the conjugate gradient optimization.

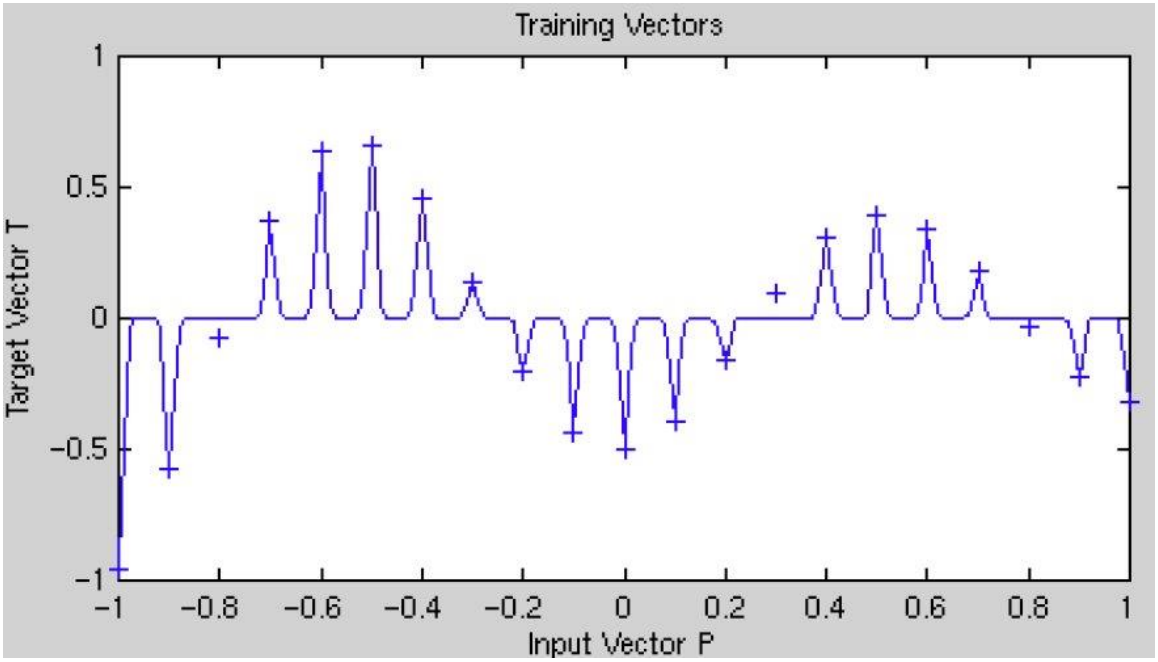
Suppose our goal is to fit the following function:



If the sigma values are too large, then the model will not be able to closely fit the function, and you will end up with a fit like this:



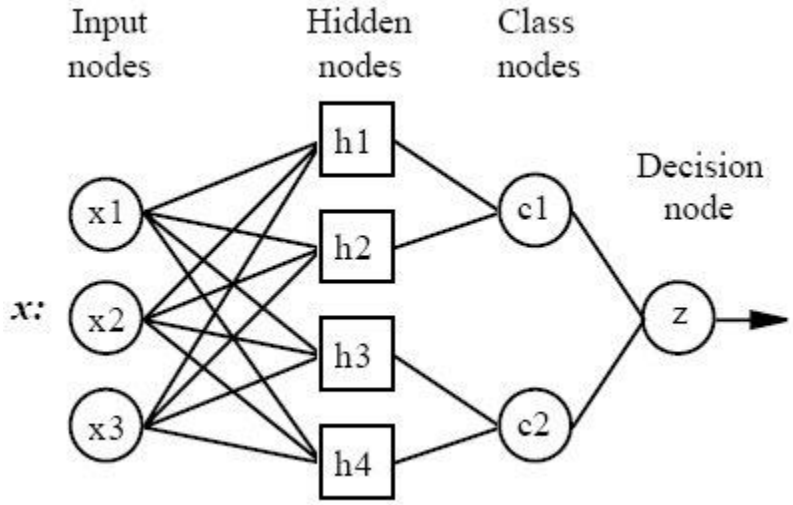
If the sigma values are too small, the model will over fit the data because each training point will have too much influence:



DTREG allows you to select whether a single sigma value should be used for the entire model, or a separate sigma for each predictor variable, or a separate sigma for each predictor variable and target category. DTREG uses the Leave-One-Out (LOO) method of evaluating sigma values during the optimization process. This measures the error by building the model with all training rows except for one and then evaluating the error with the excluded row. This is repeated for all rows, and the error is averaged.

Architecture of a PNN/GRNN Network

In 1990, Donald F. Specht proposed a method to formulate the weighted-neighbor method described above in the form of a neural network. He called this a "Probabilistic Neural Network". Here is a diagram of a PNN/GRNN network:



All PNN/GRNN networks have four layers:

Input layer – There is one neuron in the input layer for each predictor variable. In the case of categorical variables, *N*-1 neurons are used where *N* is the number of categories. The input neurons (or processing before the input layer) standardizes the range of the values by subtracting the median and dividing by the interquartile range. The input neurons then feed the values to each of the neurons in the hidden layer.

Hidden layer – This layer has one neuron for each case in the training data set. The neuron stores the values of the predictor variables for the case along with the target value. When presented with the ***x*** vector of input values from the input layer, a hidden neuron computes the Euclidean distance of the test case from the neuron's center point and then applies the RBF kernel function using the sigma value(s). The resulting value is passed to the neurons in the pattern layer.

Pattern layer / Summation layer – The next layer in the network is different for PNN networks and for GRNN networks. For PNN networks there is one pattern neuron for each category of the target variable. The actual target category of each training case is stored with each hidden neuron; the weighted value coming out of a hidden neuron is fed only to the pattern neuron that corresponds to the hidden neuron's category. The pattern neurons add the values for the class they represent (hence, it is a weighted vote for that category).

For GRNN networks, there are only two neurons in the pattern layer. One neuron is the **denominator summation unit** the other is the **numerator summation unit**. The

denominator summation unit adds up the weight values coming from each of the hidden neurons. The numerator summation unit adds up the weight values multiplied by the actual target value for each hidden neuron.

Decision layer – The decision layer is different for PNN and GRNN networks. For PNN networks, the decision layer compares the weighted votes for each target category accumulated in the pattern layer and uses the largest vote to predict the target category.

For GRNN networks, the decision layer divides the value accumulated in the numerator summation unit by the value in the denominator summation unit and uses the result as the predicted target value.

Removing unnecessary neurons

One of the disadvantages of PNN/GRNN models compared to multi-level feed forward networks is that PNN/GRNN models are large due to the fact that there is one neuron for each training row. This causes the model to run slower than multilayer perceptron networks when using scoring to predict values for new rows.

DTREG provides an option to cause it remove unnecessary neurons from the model after the model has been constructed (see the parameter settings beginning on page 80).

Removing unnecessary neurons has three benefits:

1. The size of the stored model is reduced.
2. The time required to apply the model during scoring is reduced.
3. Removing neurons often improves the accuracy of the model.

The process of removing unnecessary neurons is a slow (order N^2), iterative process. Leave-one-out validation is used to measure the error of the model with each neuron removed. The neuron that causes the least increase in error (or possibly the largest reduction in error) is then removed from the model. The process is repeated with the remaining neurons until the stopping criterion is reached. For models with more than 1000 training rows, the neuron removal process may become impractically slow. If you have a multi-CPU computer, you can speed up the process by allowing DTREG to use multiple CPU's for the process. See page 16 for information about how to do this.

When unnecessary neurons are removed, the "Model Size" section of the analysis report shows how the error changes with different numbers of neurons. You can see a graphical chart of this by clicking Chart/Model size.

There are three criteria that can be selected to guide the removal of neurons:

- **Minimize error** – If this option is selected, then DTREG removes neurons as long as the leave-one-out error remains constant or decreases. It stops when it finds a neuron whose removal would cause the error to increase above the minimum found.
- **Minimize neurons** – If this option is selected, DTREG removes neurons until the leave-one-out error would exceed the error for the model with all neurons.
- **# of neurons** – If this option is selected, DTREG reduces the least significant neurons until only the specified number of neurons remain.

Support Vector Machines (SVM)

It's not enough to help the feeble up, but to support him after.
– William Shakespeare

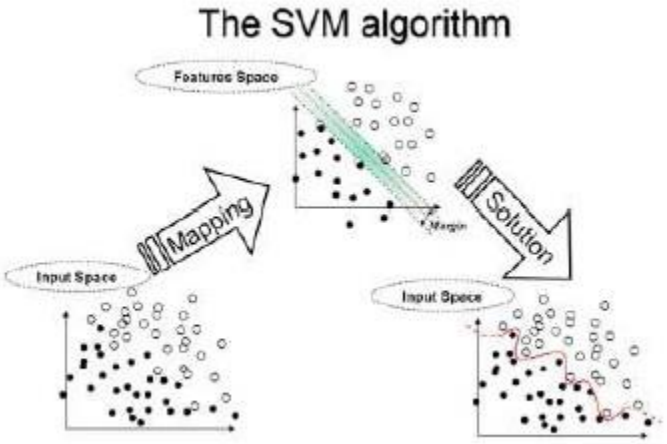
Introduction to Support Vector Machine (SVM) Models

A Support Vector Machine (SVM) performs classification by constructing an *N*-dimensional hyperplane that optimally separates the data into two categories. SVM models are closely related to *neural networks*. In fact, a SVM model using a sigmoid kernel function is equivalent to a two-layer, feed-forward neural network.

Support Vector Machine (SVM) models are a close cousin to classical neural networks. Using a kernel function, SVM's are an alternative training method for polynomial, radial basis function and multi-layer perceptron classifiers in which the weights of the network are found by solving a quadratic programming problem with linear constraints, rather than by solving a non-convex, unconstrained minimization problem as in standard neural network training.

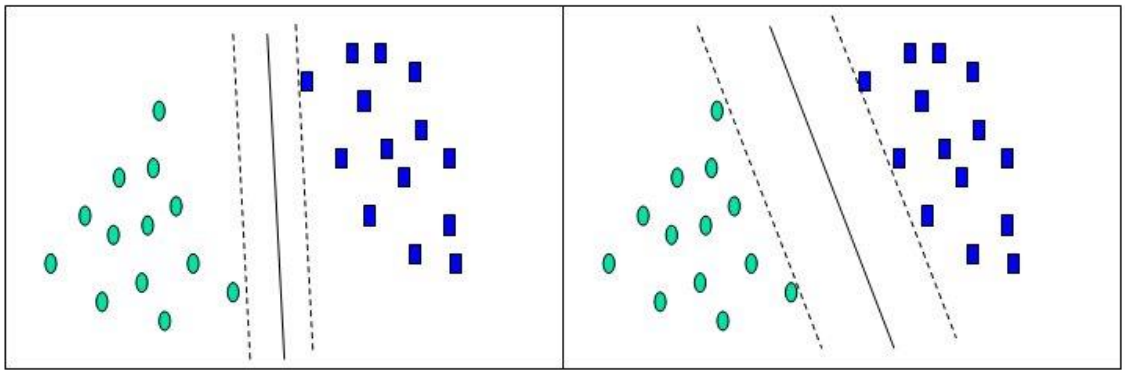
In the parlance of SVM literature, a predictor variable is called an *attribute*, and a transformed attribute that is used to define the hyperplane is called a *feature*. The task of choosing the most suitable representation is known as *feature selection*. A set of features that describes one case (i.e., a row of predictor values) is called a *vector*. So the goal of SVM modeling is to find the optimal hyperplane that separates clusters of vector in such a way that cases with one category of the target variable are on one side of the plane and cases with the other category are on the other size of the plane. The vectors near the hyperplane are the *support vectors*.

The figure below presents an overview of the SVM process.



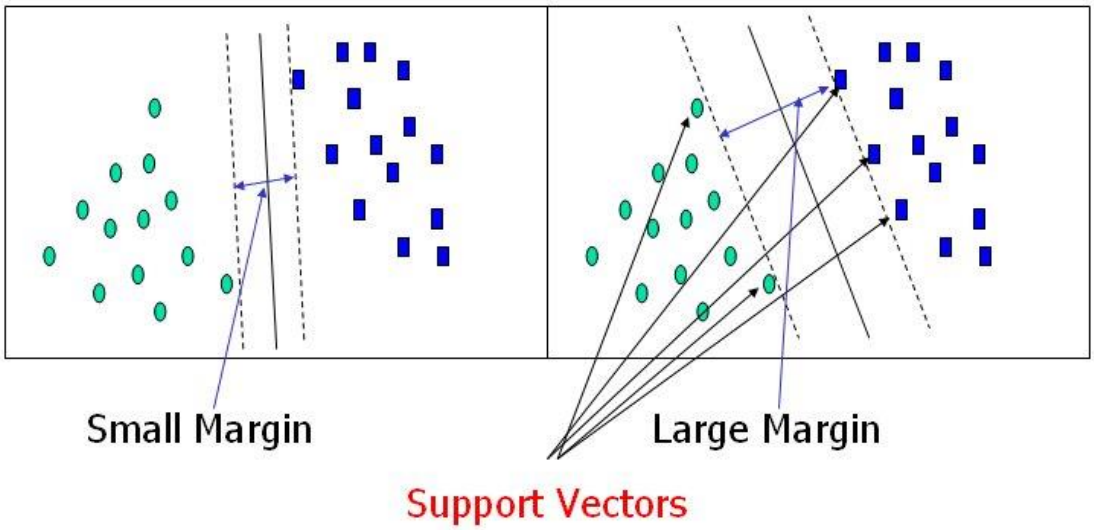
A Two-Dimensional Example

Before considering *N*-dimensional hyperplanes, let's look at a simple 2-dimensional example. Assume we wish to perform a classification, and our data has a categorical target variable with two categories. Also assume that there are two predictor variables with continuous values. If we plot the data points using the value of one predictor on the X axis and the other on the Y axis we might end up with an image such as shown below. One category of the target variable is represented by rectangles while the other category is represented by ovals.



In this idealized example, the cases with one category are in the lower left corner and the cases with the other category are in the upper right corner; the cases are completely separated. The SVM analysis attempts to find a 1-dimensional hyperplane (i.e. a line) that separates the cases based on their target categories. There are an infinite number of possible lines; two candidate lines are shown above. The question is which line is better, and how do we define the optimal line.

The dashed lines drawn parallel to the separating line mark the distance between the dividing line and the closest vectors to the line. The distance between the dashed lines is called the *margin*. The vectors (points) that constrain the width of the margin are the *support vectors*. The following figure which is used with the kind permission of Jaiwei Han (Han, Jiawei and Micheline Kamber) illustrates this.

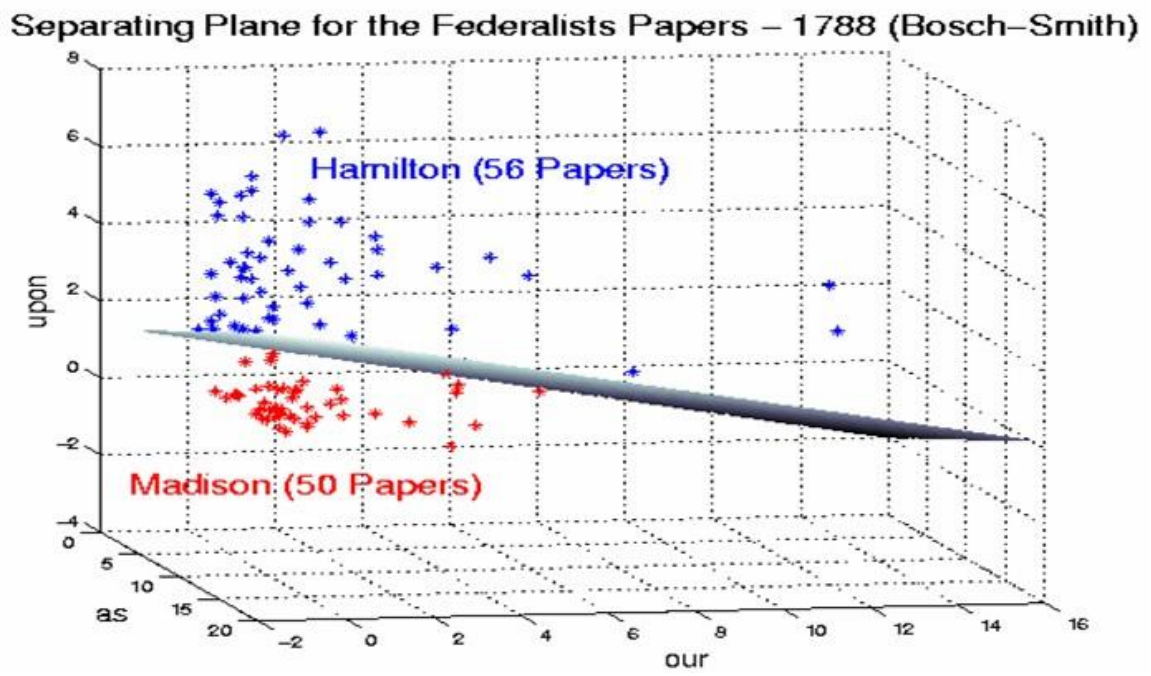


An SVM analysis finds the line (or, in general, hyperplane) that is oriented so that the margin between the support vectors is maximized. In the figure above, the line in the right panel is superior to the line in the left panel.

If all analyses consisted of two-category target variables with two predictor variables, and the cluster of points could be divided by a straight line, life would be easy. Unfortunately, this is not generally the case, so SVM must deal with (a) more than two predictor variables, (b) separating the points with non-linear curves, (c) handling the cases where clusters cannot be completely separated, and (d) handling classifications with more than two categories.

Flying High on Hyperplanes

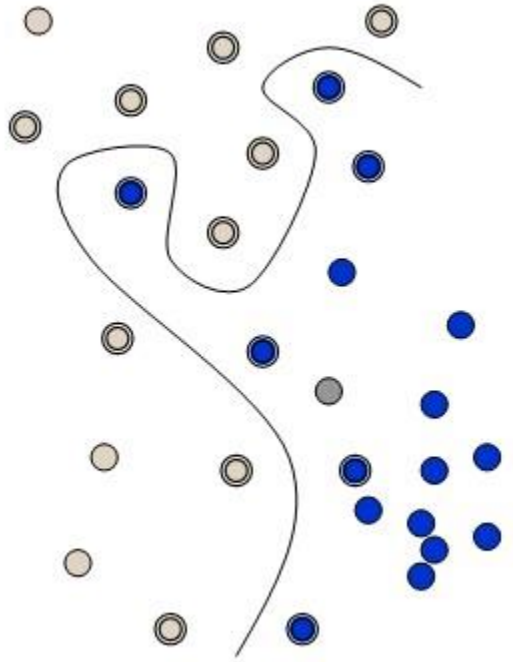
In the previous example, we had only two predictor variables, and we were able to plot the points on a 2-dimensional plane. If we add a third predictor variable, then we can use its value for a third dimension and plot the points in a 3-dimensional cube. Points on a 2-dimensional plane can be separated by a 1-dimensional line. Similarly, points in a 3-dimensional cube can be separated by a 2-dimensional plane. See the figure below from Fung, 1998.



As we add additional predictor variables (attributes), the data points can be represented in *N*-dimensional space, and a (*N*-1)-dimensional hyperplane can separate them.

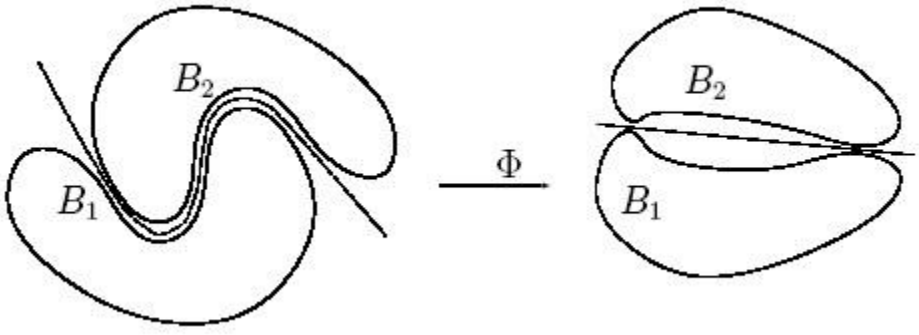
When Straight Lines Go Crooked

The simplest way to divide two groups is with a straight line, flat plane or an *N*-dimensional hyperplane. But what if the points are separated by a nonlinear region such as shown below?

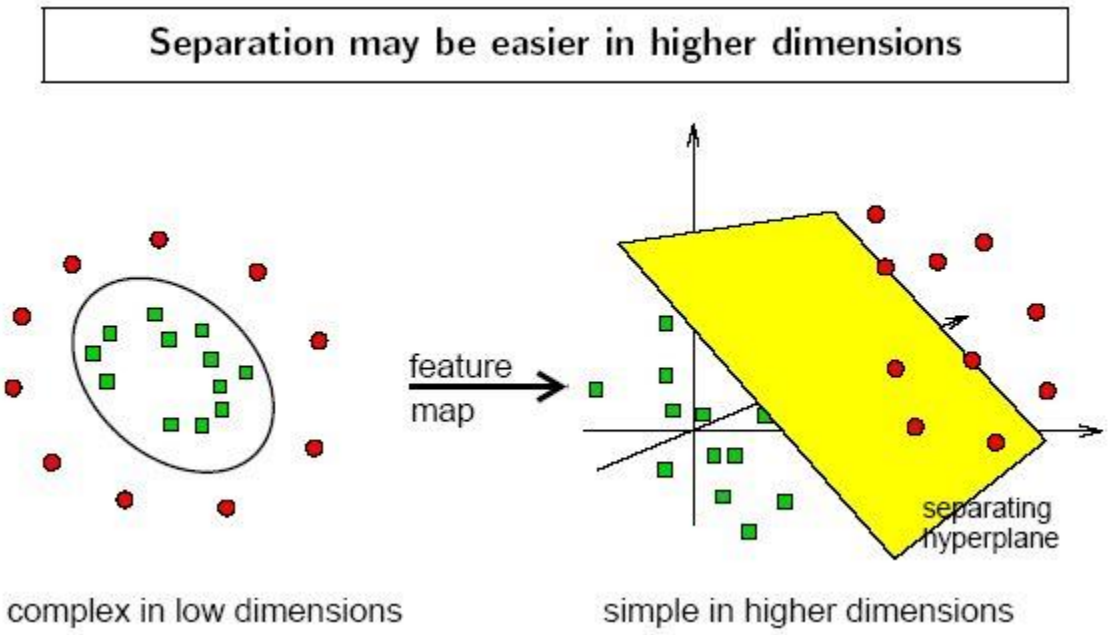


In this case we need a nonlinear dividing line.

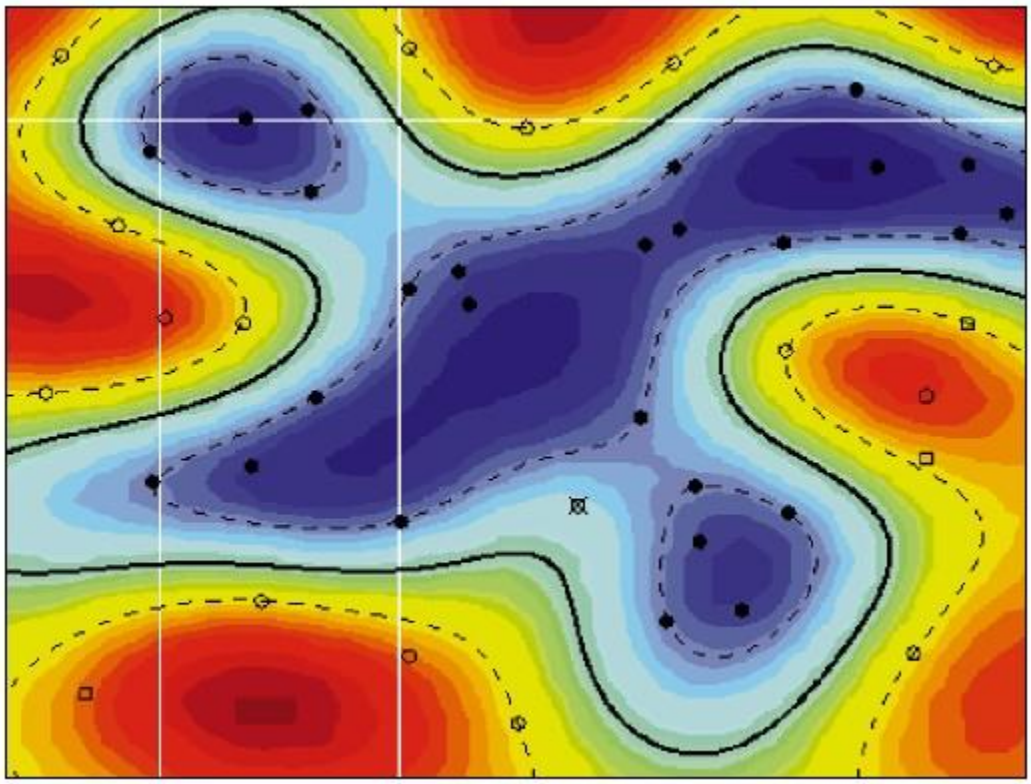
Rather than fitting nonlinear curves to the data, SVM handles this by using a *kernel function* to map the data into a different space where a hyperplane can be used to do the separation.



The kernel function may transform the data into a higher dimensional space to make it possible to perform the separation. The following figure by Florian Markowetz illustrates this:



The concept of a kernel mapping function is very powerful. It allows SVM models to perform separations even with very complex boundaries such as shown below.

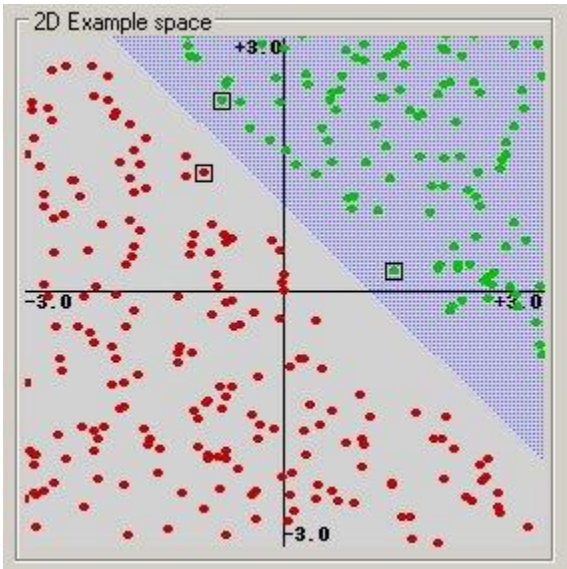


The Kernel Trick

Many kernel mapping functions can be used – probably an infinite number. But a few kernel functions have been found to work well in for a wide variety of applications. The default and recommended kernel function is the Radial Basis Function (RBF).

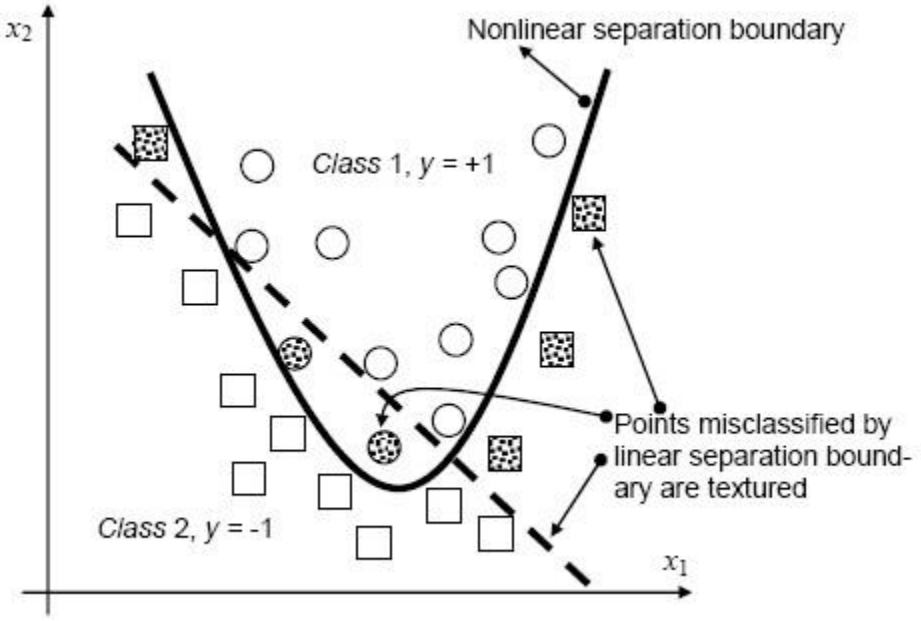
Kernel functions supported by DTREG:

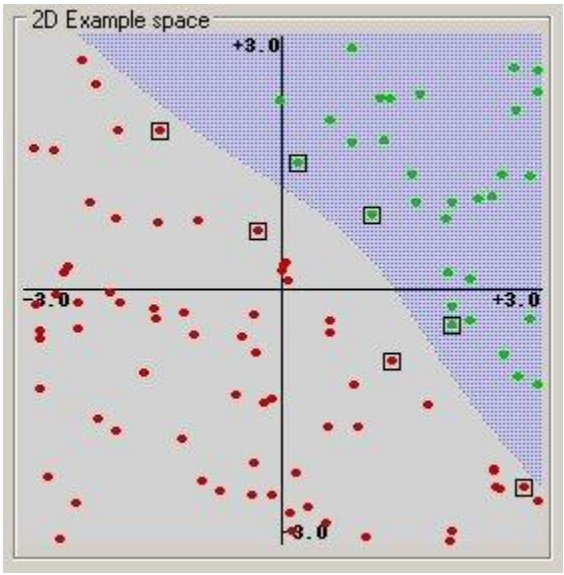
Linear: u'*v



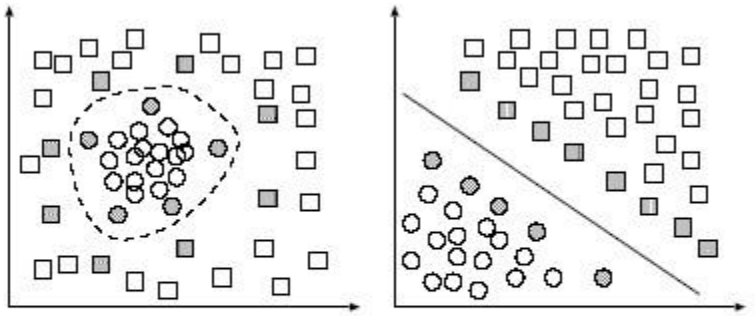
(This example was generated by pcSVMdemo:
http://www.procoders.net/en/Procoders/open_source/pcSVMdemo)

Polynomial: (gamma*u'*v + coef0)^degree
See the following figure from Kecman, 2004.



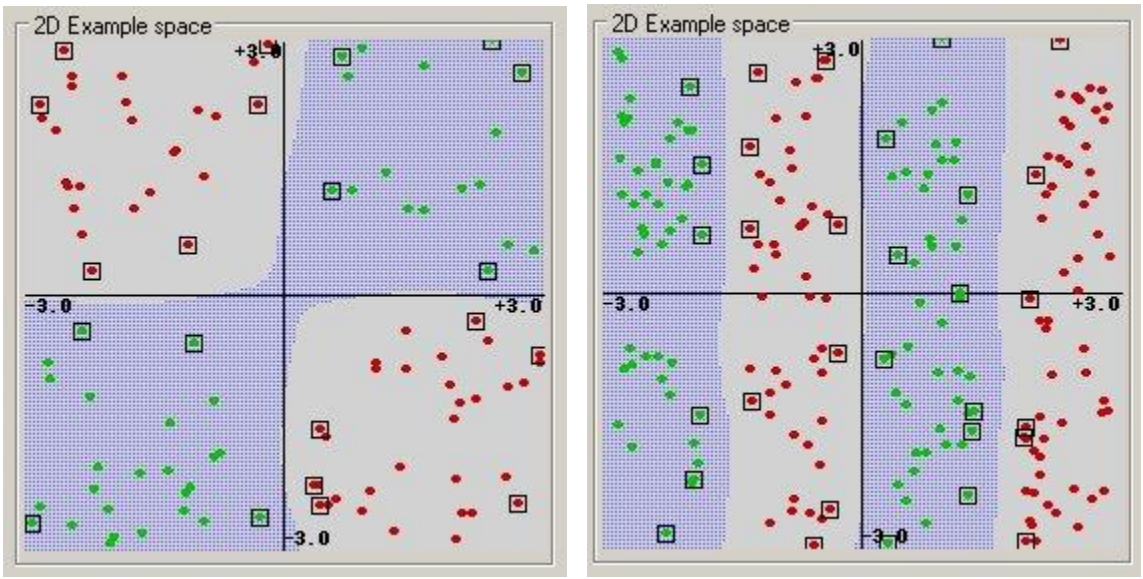


Radial basis function: exp(-gamma*|u-v|^2)
A Radial Basis Function (RBF) is the default and recommended kernel function. The RBF kernel non-linearly maps samples into a higher dimensional space, so it can handle nonlinear relationships between target categories and predictor attributes; a linear basis function cannot do this. Furthermore, the linear kernel is a special case of the RBF. A sigmoid kernel behaves the same as a RBF kernel for certain parameters. The RBF function has fewer parameters to tune than a polynomial kernel, and the RBF kernel has less numerical difficulties. The following figure from Yang, 2003 illustrates RBF mapping.

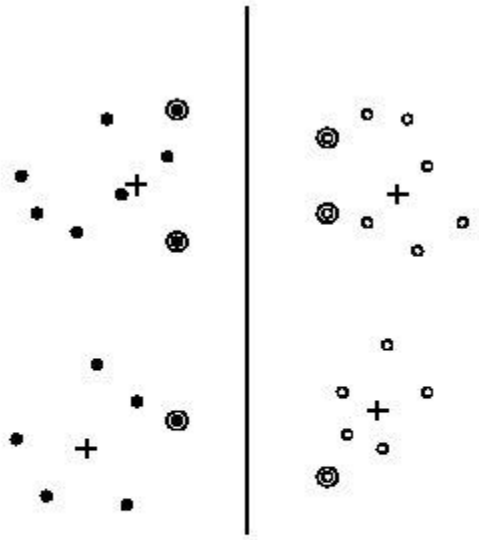


(a) Radial Basis Function (b) RBF mapping

Separable classification with Radial Basis kernel functions in different space. Left: original space. Right: feature space.

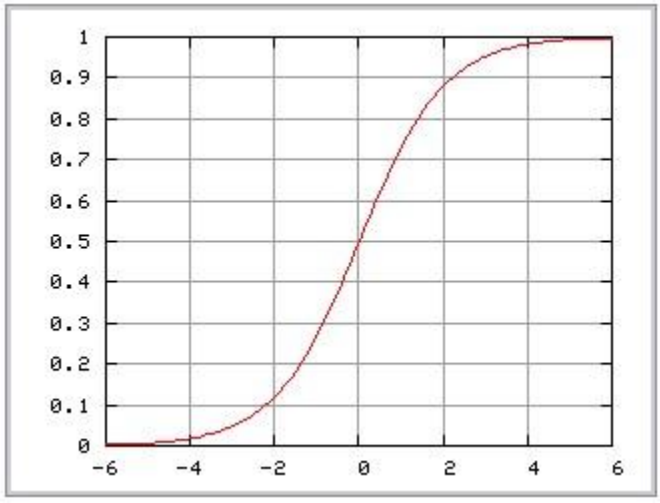


An SVM model using a radial basis function kernel has the architecture of an RBF network. However, the method for determining the number of nodes and their centers is different from standard RBF networks with the centers of the RBF notes on the support vectors (see the figure below from C. Campbell).



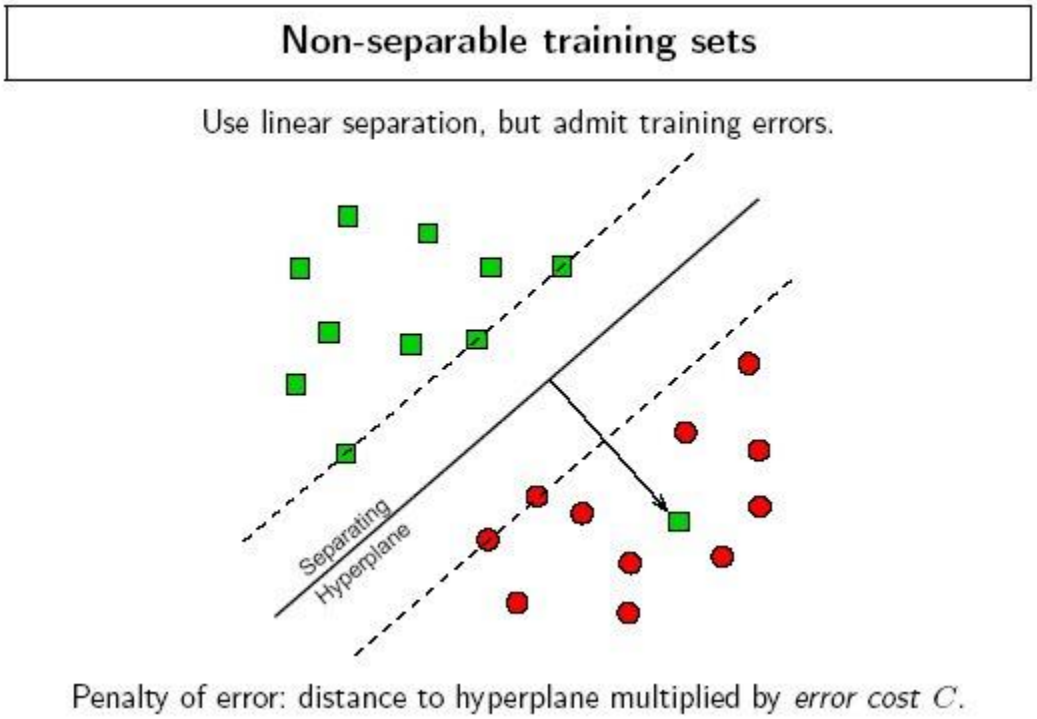
A classical RBF network finds the centers of RBF nodes by k-means clustering (marked by crosses). In contrast an SVM with RBF kernels uses RBF nodes centered on the support vectors (circled), i.e., the datapoints closest to the separating hyperplane (the vertical line illustrated).

Sigmoid: tanh(gamma*u'*v + coef0)



Parting Is Such Sweet Sorrow

Ideally an SVM analysis should produce a hyperplane that completely separates the feature vectors into two non-overlapping groups. However, perfect separation may not be possible, or it may result in a model with so many feature vector dimensions that the model does not generalize well to other data; this is known as *over fitting*. The following figure from a slide by Florian Markowetz of Max Planck Institute for Molecular Genetics illustrates a non-separable training set.



To allow some flexibility in separating the categories, SVM models have a cost parameter, *C*, that controls the trade off between allowing training errors and forcing rigid margins. It creates a *soft margin* that permits some misclassifications. The penalty associated with a misclassified point is the distance from the point to the hyperplane multiplied by the cost factor *C*. Increasing the value of *C* increases the cost of misclassifying points[7] and forces the creation of a more accurate model that may not generalize well. DTREG provides grid and pattern search facilities that can be used to find the optimal value of *C*.

[7] Technically, *C* is the cost of the sum of the distances of wrong-size points from the margins.

Classification with More Than Two Categories

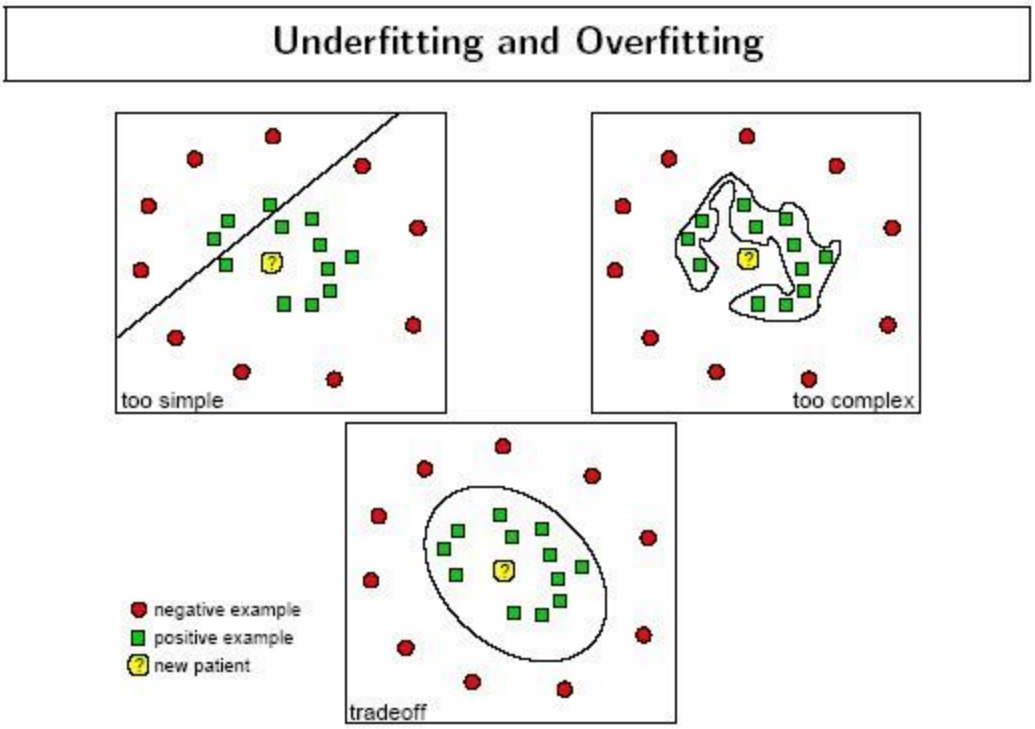
The idea of using a hyperplane to separate the feature vectors into two groups works well when there are only two target categories, but how does SVM handle the case where the target variable has more than two categories? Several approaches have been suggested, but two are the most popular: (1) "one against many" where each category is split out and all of the other categories are merged; and, (2) "one against one" where $k(k-1)/2$ models are constructed where k is the number of categories. DTREG uses the more accurate (but more computationally expensive) technique of "one against one". For a discussion of why this method is used and comparisons with other approaches see Hsu and Lin, 2002.

Optimal Fitting Without Over fitting

The accuracy of an SVM model is largely dependent on the selection of the kernel parameters such as *C*, *Gamma*, *P*, etc. DTREG provides two methods for finding optimal parameter values, a **grid search** and a **pattern search**. A grid search tries values of each parameter across the specified search range using geometric steps. A pattern search (also known as a "compass search" or a "line search") starts at the center of the search range and makes trial steps in each direction for each parameter. If the fit of the model improves, the search center moves to the new point and the process is repeated. If no improvement is found, the step size is reduced and the search is tried again. The pattern search stops when the search step size is reduced to a specified tolerance.

To avoid over fitting, cross-validation is used to evaluate the fitting provided by each parameter value set tried during the grid or pattern search process.

The following figure by Florian Markowetz illustrates how different parameter values may cause under or over fitting:



Standing On The Shoulders of Giants

The SVM implementation used by DTREG is partially based on the outstanding LIBSVM project by Chih-Chung Chang and Chih-Jen Lin (Chang and Lin, 2005). They have made both theoretical and practical contributions to the development of support vector machines, and their work on LIBSVM is acknowledged with gratitude. Parts of LIBSVM are used under the following terms:

“This software(LIBSVM) is provided by the copyright holders and contributors ‘as is’ and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the regents or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.”

Gene Expression Programming

If evolution really works, how come mothers only have two hands?
– Ed Dussault

Introduction to Gene Expression Programming

Gene Expression Programming is a procedure that mimics biological evolution to create a computer program to model some phenomenon. Gene expression programming can be used to create many different types of models including decision trees, neural networks and polynomial constructs. The type of gene expression programming implemented in DTREG is **Symbolic Regression** – so named because it creates a symbolic mathematical or logical function.

DTREG provides a full implementation of the Gene Expression Programming algorithm developed by Cândida Ferreira (Ferreira 2006). Here are some of the features of DTREG's implementation:

- Continuous and categorical target variables
- Automatic handling of categorical predictor variables
- A large library of functions that you can select for inclusion in the model
- Mathematical and logical (AND, OR, NOT, etc.) function generation
- Choice of many fitness functions
- Both static linking functions and evolving homeotic genes
- Fixed and random constants
- Nonlinear regression to optimize constants
- Parsimony pressure to optimize the size of functions
- Automatic algebraic simplification of the combined function
- Several forms of validation including cross-validation and hold-out
- Computation of the relative importance of predictor variables
- Automatic generation of C or C++ source code for the functions
- Multi-CPU execution for multiple target categories and cross-validation

Introduction to Symbolic Regression

In ordinary mathematical regression, the procedure is given the form of the function to be fitted to the data. This could be a linear function for linear regression or a general

mathematical function for nonlinear regression. The regression procedure computes the optimal values of parameters for the function to make the function fit a data set as well as possible, but the regression procedure does not alter the form of the function. For example, a linear regression problem with two variables has the form:

$$y = a + bx$$

Where *x* is the independent variable, *y* is the dependent variable, and *a* and *b* are parameters whose values are to be computed by the regression algorithm. This type of procedure is classified as *parametric regression* because the goal is to estimate parameters for a function whose form is known (or assumed).

With *nonparametric regression* the form of the function is not known in advance, and it is the goal of the procedure to find a function that will fit the data. So we are looking for $f(\cdot)$ that will best fit

$$y = f(x_1, x_2, \dots, x_n)$$

Where y is the dependent variable and there are *n* independent *x* variables.

There are many possible forms of nonparametric functions – neural networks and decision trees are types of nonparametric functions. Symbolic regression is a subset of nonparametric regression that restricts the functions to be mathematical or logical expressions.

Symbolic Regression Example – Kepler's Third Law

Around 1605, the German mathematician and astronomer Johannes Kepler discovered three astronomical laws that describe the orbits of planets around the Sun. Kepler's work was based on the precise astronomical observations recorded by Danish astronomer Tycho Brahe. Kepler's third law states "The squares of the orbital periods of planets are directly proportional to the cubes of the semi-major axis of the orbits." Mathematically, this is:

$$Period^2 = constant \cdot Distance^3$$

Let's see if symbolic regression can figure this out without the help of a genius astronomer. We will use the following data as input to the procedure:

Planet	Distance	Period
Venus	0.72	0.61
Earth	1.00	1.00
Mars	1.52	1.84
Jupiter	5.20	11.90
Saturn	9.53	29.40
Uranus	19.10	83.50

Gene expression programming was used to model this data. Two genes were used per chromosome, and there were 7 symbols in the head section of each gene. After four generations, DTREG found a perfect fit to the data. The expression generated and displayed by DTREG is:

```
Period = sqrt(Distance)*Distance
```

Simplifying this we find:

$$Period = \sqrt{Distance} \cdot Distance$$
$$Period = Distance^{\frac{3}{2}}$$
$$Period^2 = Distance^3$$

This is exactly Kepler's third law. The DTREG analysis for this problem can be found in the GepKepler.dtr program file in the Examples folder.

Odd Parity Example

In this example, symbolic regression will be used to find a logical expression to compute the parity for a 3-input binary circuit. The output parity value should be 1 if there are an odd number of inputs with the value 1, and the output should be 0 if there are an even number of inputs with the value 1. Here is the data for the analysis:

In1	In2	In3	Parity
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

For this problem we will allow DTREG to use only three functions in the expression: AND, OR, NOT. We will use 3 genes per chromosome, and we will use the AND function to link the genes. After 418 generations to train the model and an additional 397 generations to simplify it, DTREG generated the following function which perfectly fits the data:

```
Parity = (In3|(!(In1&In2)))&((!(In1|In2))|(In1&In2)|!In3)&In2|(In1|In3)
```

Where '|' is the OR operator, '&' is AND, and '!' is NOT. The project file for this example is named GepParity3.dtr; it can be found in the Examples folder.

Genetic Algorithms

Genetic algorithms (GA) have been in widespread use since the 1980's, but the first experiments with computer simulated evolution go back to 1954.

Genetic algorithms are basically a smart search procedure. The goal is to find a solution in a multi-dimensional space where there is no known exact algorithm. Genetic algorithms are often thousands or even millions of times faster than exhaustive search procedures. Exhaustive search is impractical for high dimension problems. The use of random mutations allows genetic algorithms to avoid being trapped in locally-optimal regions which is a serious problem for hill-climbing algorithms typically used for iterative/convergence procedures. Genetic algorithms have been used to solve otherwise intractable problems such as the Traveling Salesperson Problem.

Genetic algorithms mimic biological evolution, and the terms used for genetic algorithms are based on biological features.

In biological DNA systems, the basic units are the adenine (A), thymine (T), guanine (G) and cytosine (C) nucleotides that join the helical strands. In genetic algorithms, the basic unit is called a *symbol***.** The nature of symbols depends on the particular genetic algorithm. In gene expression programming, the symbols consist of functions, variables and constants. Symbols for variables and constants are called *terminals*, because they have no arguments.

An ordered set of symbols form a *gene***,** and an ordered set of genes form a *chromosome*. In GEP programs, genes typically have 4 to 20 symbols, and chromosomes are typically built from 2 to 10 genes; chromosomes may consist of only a single gene. The DNA strand for a mammal typically contains about 5×10^9 nucleotides.

Genetic Algorithms for Symbolic Regression

Many efforts have been made to use genetic algorithms to solve symbolic regression problems – that is, to generate symbolic functions to model data. One of the problems that plagues most of the efforts is finding a way to efficiently mutate and cross-breed symbolic expressions so that the resulting expressions have a valid mathematical syntax.

For example, if you mutate $(2 * x + 3)$ into $(x\ 2 + 3 *)$ it isn't any good, because it isn't syntactically correct.

One approach to this problem is to perform a mutation, check the result and then try a different random mutation until a syntactically valid expression is generated. Obviously, this can be a time consuming process for complex expressions.

A second approach is to limit what type of mutations can be performed – for example, only exchanging complete sub-expressions. The problem with this approach is that if limited mutations are used, the evolution process is hindered, and it may take a large number of generations to find a solution, or it may be completely unable to find the optimal solution.

Gene Expression Programming

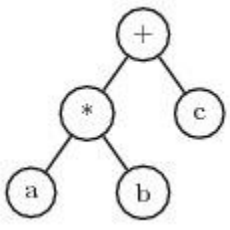
An elegant solution to the expression-mutation problem was discovered in 1999 by Cândida Ferreira (Ferreira 1996). Ferreira devised a system for encoding expressions that allows fast application of a wide variety of mutation and cross-breeding techniques while guaranteeing that the resulting expression will always be syntactically valid. This approach is called Gene Expression Programming (GEP). Experiments have shown that GEP is 100 to 60,000 times faster than older genetic algorithms.

Expression Trees and Karva

The key to GEP's ability to quickly mutate valid expressions is the way it encodes symbols in genes. This notation is called the *Karva Language* (Ferreira 1996). Expressions encoded using Karva are called *K-expressions*. Consider the simple mathematical expression

$$a * b + c$$

This can be encoded as an *expression tree* of the form



An expression tree is an excellent way to represent an expression in a computer, because the tree can be arbitrarily complex, and expression trees can be evaluated quickly.

To convert an expression tree to the Karva notation, start at the left-most symbol in the top line of the tree and scan symbols left-to-right and top-to-bottom. Each time a symbol

is encountered, add it to the K-expression in left-to-right order. When there are no more symbols on a line, advance to the left end of the following line. Using this method, the tree shown above is converted to the K-expression:

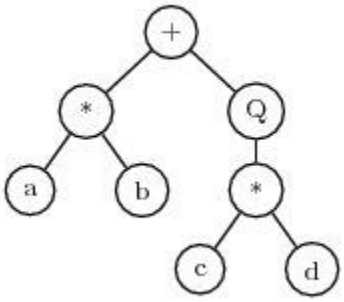
```
+*cab
```

Note that + is the first symbol found on the first line, at the end of that line scanning begins on the second line and finds `*` followed by `c`. It then starts with the third line and finds `a` and `b`.

As a second example, consider the expression

$$a * b + \sqrt{c * d}$$

The corresponding expression tree is



Where '`Q`' represents square root. This can be translated to the K-expression

```
+*Qab*cd
```

The process of converting an expression tree to a K-expression can be carried out quickly by a computer. A reverse process can quickly convert a K-expression back to an expression tree.

Genes

A gene consists of a fixed number of symbols encoded in the Karva language. A gene has two sections, the *head* and the *tail*. The head is used to encode functions for the expression. The tail is a reservoir of extra terminal symbols that can be used if there aren't enough terminals in the head to provide arguments for the functions. Thus, the head can contain functions, variables and constants, but the tail can contain only variables and constants (i.e. terminals). The number of symbols in the head of a gene is specified as a parameter for the analysis (see page 95). The number of symbols in the tail is determined by the equation

$$t = h \cdot (MaxArg - 1) + 1$$

Where *t* is the number of symbols in the tail, *h* is the number of symbols in the head, and *MaxArg* is the maximum number of arguments required by any function that is allowed to be used in the expression. For example, if the head length is 6 and the allowable set of functions consists of binary operators (+, -, *, /), then the tail length is:

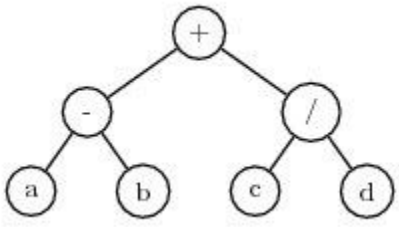
$$t = 6 \cdot (2 - 1) + 1 = 7$$

The purpose of the tail is to provide a reservoir of terminal symbols (variables and constants) that can be used as arguments for functions in the head if there aren't enough terminals in the head.

Consider a gene with three symbols in the head and which uses binary arithmetic operators. The tail will then have $3 \cdot (2 - 1) + 1 = 4$ terminal symbols. Here is an example of such a gene. The head is in front of the comma, and the tail follows the comma:

```
+-/,abcd
```

Ignoring the distinction between the head and the tail, this K-expression can be converted to this expression tree:

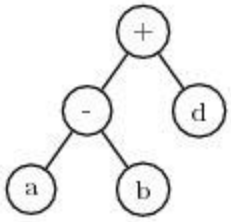


Note that the head of the gene consisted only of functions, but the tail provided enough terminals to fill in the arguments for the functions.

During mutation, symbols in the head can be replaced by either function or terminal symbols. Symbols in the tail can be replaced only by terminals. Using the same example K-expression shown above, assume mutation replaces the '/' symbol with `d`. Then the K-expression is:

```
+-d,abcd
```

And the expression tree becomes



Note that this expression tree has fewer nodes than the previous one. This illustrates an important point: by allowing mutation to replace functions with terminals and terminals with functions, the size of the expression can change was well as its content. As a further example, assume the next mutation changes the first symbol in the K-expression from '+' to `c`. The K-expression becomes:

```
c-d,abcd
```

The expression tree for this is:

The "tree" consists of a single node which is the variable `c`. Note that the number of symbols in the gene did not change, but some symbols are not used. The symbols that are not used are called the *noncoding region* of the gene. Because the functional length of a gene may be less than the number of symbols it holds, it is called an *open reading frame* (ORF). Biological genes also have noncoding regions.

If you experiment with K-expressions you will find that any possible mutation will result in a valid expression as long as the following rules are adhered to:

1. Symbols in the head can be replaced with functions, variables and constants.
2. Symbols in the tail can be replaced only with variables and constants (terminals).
3. The tail is of sufficient length to provide terminals for all possible functions that can occur in the head. (See the formula for tail length above.)

This is the key to the efficiency of gene expression programming. It is easy for a computer program to follow these three rules while performing mutations, and it never has to check whether the resulting expression has valid syntax. By allowing a broad range of mutations, the process can efficiently explore a high dimensional space, and the expressions can change in size as functions are replaced by terminals and terminals by functions.

Chromosomes and Linking Functions

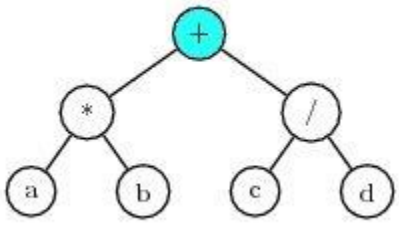
A chromosome consists of one or more genes. The number of genes in a chromosome is a parameter for the analysis (see page 95). If there is more than one gene in a

chromosome, then a *linking function* is used to join the genes in the final function. The linking function can be static or evolving (see page 106).

For example, consider a chromosome with two genes having the K-expressions:

Gene 1: `*ab`
Gene2: `/cd`

If '+' is used as the static linking function, then the combined expression is:



Which is equivalent to $(a * b + c/d)$.

Homeotic Genes

In addition to specifying a static linking function, you can allow the linking functions to be selected dynamically by evolution. This is done using *homeotic genes*.

In biology, homeotic genes control macro organization such as determining that arms should be attached to shoulders and legs to hips. Mutations in homeotic genes produce bizarre creatures. An example is the Antennapedia mutant of the fruit fly Drosophila, where legs are found sprouting where the antennae would normally be. Often, mutations in homeotic genes produce nonviable organisms.

In gene expression programming, a homeotic gene is used to link together the regular genes in a chromosome. There is never more than one homeotic gene in a chromosome, and there is no homeotic gene if a static linking function is used.

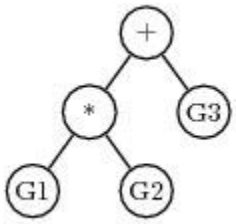
Homeotic genes have the same structure as regular genes: They have a head section with a length specified as a parameter (see page 107), a tail section, and a set of symbols. The symbols in homeotic genes consist of references to ordinary genes and linking functions.

Homeotic genes undergo mutation, inversion, transposition and crossover just as regular genes do during evolution. Separate parameters are available to set the mutation rates for homeotic genes (see page 106). Symbols and functions are never exchanged between regular genes and homeotic genes.

For example, if a chromosome has 3 regular genes, G1, G2 and G3, and a homeotic gene, then the homeotic gene might have a K-expression of

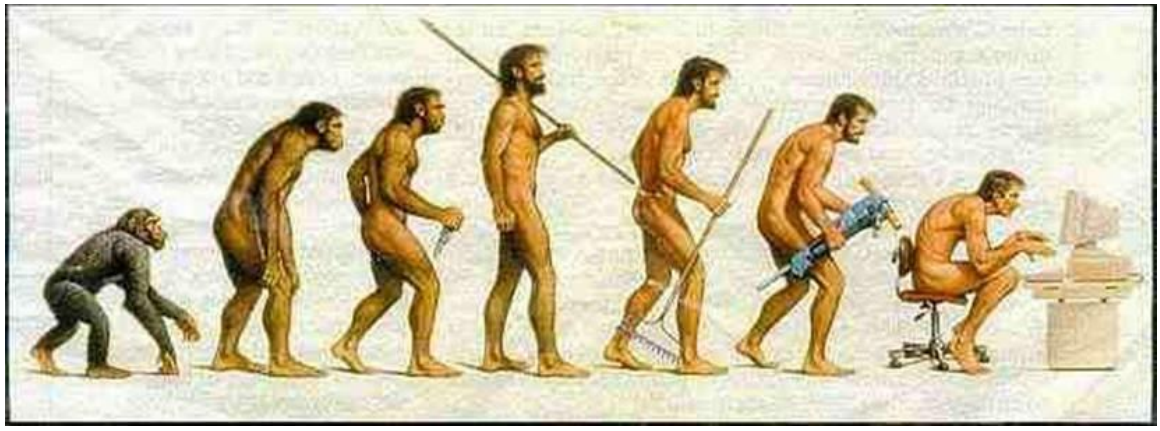
```
+*G3G1G2
```

And the expression tree would be



Where G1, G2 and G3 are the expression trees for the regular genes.

Mathematical Evolution



Evolution is the engine of gene expression programming. An initial population of candidate functions is created, then mutation, breeding and natural selection are used to evolve functions that more closely model the data.

The main steps in the training and evolution of a gene expression program are:

1. Create an initial population of viable individuals (chromosomes).
2. Use evolution to attempt to create individuals that fit the data well.
3. Use evolution to try to find a simpler, more parsimonious function.
4. Use nonlinear regression to find optimal values of constants.

Initial Population Creation

Gene expression programming and other genetic algorithms work by evolving sets of individuals (chromosomes). But before the evolution process begins, an initial, *founder population* of individuals must be constructed that can mutate, breed and be selected for

subsequent generations. The number of individuals in the population is a parameter for the analysis (see page 95).

The Karva language used to represent expressions in gene expression programming guarantees that all expressions will have valid mathematical syntax. But Karva does not guarantee that the expressions will produce meaningful values when they are evaluated. For example, the K-expression

```
/a0
```

is syntactically valid, but it generates the expression $(a/0)$, which, of course, is infinite. There are many other cases where the results cannot be evaluated such as taking the square root of a negative number, finding the log of a negative number or overflowing the range of numbers by raising a large value to a huge power. Expressions that cannot be evaluated to generate meaning values are called *unviable* and receive a fitness score of zero. Expressions are also classified as unviable if they are unable to correctly classify any members of the population. Some fitness functions place additional conditions on viability: For example, the *Hits with penalty* fitness function only classifies an expression as viable if both the true positive (TP) and true negative (TN) hit counts are greater than zero (see page 97).

The creation of the initial population is done by randomly selecting functions and terminals for the genes. Some of the resulting individuals may be viable, and some may be unviable. If the population has no viable individuals, another population is randomly created. This process is repeated up to several thousand times until an initial population is found with at least one viable individual. If it is impossible to create an initial population with a viable individual, then the analysis cannot be performed.

On a philosophical note, it is difficult to imagine how an initial population could have been created for biological evolution. Gene expression programming starts with the machinery for evolution in existence and ready to run – mutation, inversion, transposition, cross-replication and selection. It also starts the process with viable individuals having a structure suitable for evolution – symbols, genes and chromosomes.

In the natural world, the starting point would be simple elements and molecules with no pre-existing organization of genes, chromosomes, DNA or RNA. The machinery for evolution and passing on genetic material from generation to generation would not exist. So evolution – at least as it is currently understood – cannot be used to explain how unorganized chemicals organized themselves into DNA and RNA which are essential for evolution. This is one of the stronger arguments for Intelligent Design.

The Process of Evolution

Once the initial population has been created, the process of evolution can be used to find individuals that model the data well. Here is an outline of the evolution process:

1. Convert K-expressions in chromosomes to expression trees.
2. Compute the fitness score for each individual by comparing the predicted target value with the actual target value for all training cases.
3. If the fitness score is sufficiently good, or if the maximum number of generations has been evolved, or if the maximum execution time has been reached, stop the evolution.
4. Transfer the best (most fit) individual to the next generation without modification.
5. Use roulette-wheel sampling to select individuals for the next generation.
6. Perform mutations.
7. Perform inversions.
8. Perform transposition.
9. Perform recombination to combine genetic material from pairs of individuals.
10. Return to step 1 for the next evolution cycle.

Natural Selection and Fitness

The principle of natural selection is that healthy, fit individuals should breed and produce offspring at a faster rate than sick, unfit individuals. Through this selection process, each generation becomes healthier and more fit. In order for this to take place, there must be some characteristics of individuals that determine fitness for the environment, and there must be a selection mechanism that favors the breeding of individuals with greater fitness.

In gene expression programming, fitness is based on how well an individual models the data. If the target variable has continuous values, the fitness can be based on the difference between predicted values and actual values. For classification problems with a categorical target variable, fitness can be measured by the number of correct predictions. DTREG provides a variety of fitness functions that you can choose from for an analysis (see page 95).

Evolution stops when the fitness of the best individual in the population reaches some limit that is specified for the analysis or when a specified number of generations have been created or a maximum execution time limit is reached.

All of the fitness functions produce fitness scores in the range 0.0 to 1.0 with 1.0 being ideal fitness – that is, the individual exactly fits the data. If a function is unviable – for example it takes the square root of a negative number or divides by zero – then its fitness score is 0.0.

Once the fitness has been calculated for the individuals in the population, roulette-wheel sampling is used to select which individuals move on to the next generation. Each individual is assigned a slot of a roulette wheel, and the size of the slot is proportional to

the fitness of the individual. Unviable individuals whose fitness is 0.0 have slots that can never be selected, so they are not propagated to the next generation. Roulette-wheel sampling causes individuals to be selected with a probability proportional to their fitness, and it eliminates unviable individuals. Since individuals are not removed from the population once they are selected, individuals may be selected more than once for the next generation.

Gene expression programming makes one exception to the roulette-wheel sampling procedure: The most fit individual in each generation is unconditionally replicated unchanged into the next generation. This is known as *elitism*. The reason this is done is to guarantee that there will never be a loss of the best individual from one generation to the next.

Mutation, Inversion, Transposition and Recombination

In order for a population to improve from generation to generation innovations must occur that cause some individuals to have qualities never before seen. These innovations come about from mutation. In gene expression programming there are several types of mutation, some are simple random changes in the symbols of genes, others are more complex involving reversing the order of symbols or transposing symbols or genes within the chromosome.

Mutation is not necessarily beneficial; often the change results in a less fit individual or in an unviable individual who cannot survive. But there is a possibility that a mutation may produce an individual with extraordinary qualities – a "genius" individual. The operation of evolution depends on mutations producing some individuals with greater fitness. Through natural selection, their offspring improve the overall quality of the population. As described above, elitism guarantees that a genius never dies unless a better genius is found to take its place. If elitism applied to people, Isaac Newton might have lived until Albert Einstein was born, and Einstein might still be alive today.

Several types of mutation are used by gene expression programming. See the section beginning on page 104 for detailed information about each method.

Mutation – Simple mutation just replaces symbols in genes with replacement symbols. Symbols in the heads of genes can be replaced by functions or terminals (variables and constants). Symbols in the tail sections can be replaced only by terminals.

Inversion – Inversion reverses the order of symbols in a section of a gene.

Transposition – Transposition selects a group of symbols and moves the symbols to a different position within the same gene. Gene transposition moves entire genes around in the chromosome.

Recombination – During recombination, two chromosomes are randomly selected, and genetic material is exchanged between them to produce two new chromosomes. It is

analogous to the process that occurs when two individuals are bred, and the offspring share a mixture of genetic material from both parents.

Parsimony Pressure and Expression Simplification

If two expressions do an equally good job of fitting a data set, the simpler expression is usually preferred. For symbolic regression, complexity is measured by the number of symbols and functions in the expression. Gene expression programming has two techniques for selecting simpler expressions over more complex ones.

The first approach is to adjust the fitness scores of individuals so that fitness is reduced by an amount proportional to the complexity of the expression. This penalty for complexity is called *parsimony pressure*. See page 95 for information about how to adjust how much parsimony pressure is applied.

While parsimony pressure is effective at guiding evolution toward simpler expressions, experiments have shown that parsimony pressure may hinder the process of evolving toward greater fitness. It is not uncommon for more complex expressions to do a better job of fitting than less complex ones, so pushing evolution to favor simpler expressions may increase the number of generations required to find a solution, or it may make it impossible to find a good solution. If parsimony pressure is used, you also should build a model with it turned off, and verify that the simpler solution does not lose significant accuracy.

The second approach to finding parsimonious solutions is to divide the task into two phases: (1) primary training without parsimony pressure, and (2) secondary training which uses parsimony pressure. Since the primary training is done without parsimony pressure, evolution can focus on finding the most accurate model as quickly as possible. Once primary training is finished, a second round of training begins using the final population from primary training as the starting population for the secondary training.

During secondary training, parsimony pressure is used to try to find a simpler expression that is at least as good as the best one found during primary training. While secondary training is being performed, the primary goal is still to improve accuracy, and the secondary goal is to find simpler expressions. So a simpler expression will be selected only if its accuracy meets or exceeds the best accuracy previously found. If a more accurate expression is found, it is used even if the result is an increase in complexity. So it is possible that during the secondary training complexity could actually increase in order to improve accuracy. But experiments have shown that this rarely happens, and secondary training usually results in simpler expressions. Since there is never any risk of losing accuracy with this approach, and it may result in a simpler and possibly more accurate expression, it is recommended.

Algebraic Simplification

DTREG includes a sophisticated procedure for performing algebraic simplification on expressions after gene expression programming has evolved the best expressions. This simplification does not alter the mathematical meaning of expressions; it just does simplifications such as grouping common terms and simplifying identities. Here are some examples of simplifications that it can perform:

$$(a + b + a + a + b + a + a) \Rightarrow 5a + 2b$$

$$\frac{a+b}{b+a} \Longrightarrow 1$$

$$a \text{ AND NOT } a \Longrightarrow 0$$

$$\left(\sqrt{a}\right)^2 \Longrightarrow a$$

$$\frac{\sin(x)}{\cos(x)} \Longrightarrow \tan(x)$$

See page 100 for more information about algebraic simplification.

Optimization of Random Constants

In addition to functions and variables, expressions can contain constants. You can specify a set of explicit constants, and you can allow DTREG to generate and evolve random constants. While evolution can do a good job of finding an expression that fits data well, it is difficult for evolution to come up with exact values for real constants.

DTREG provides an optional final step to the GEP process to refine the values of random constants. If this option is enabled, DTREG uses a sophisticated nonlinear regression algorithm to refine the values of the random constants. This optimization is performed after evolution has developed the functional form and linking and simplification have been performed. DTREG uses a model/trust-region technique along with an adaptive choice of the model Hessian. The algorithm is essentially a combination of Gauss-Newton and Levenberg-Marquardt methods; however, the adaptive algorithm often works much better than either of these methods alone.

If nonlinear regression does not improve the accuracy of the model, the original model is used. So there is no risk of losing accuracy by using this option.

K-Means Clustering

Developed between 1975 and 1977 by J. A. Hartigan and M. A. Wong (Hartigan and Wong, 1979), K-Means clustering is one of the older predictive modeling methods. K-Means Clustering is a relatively fast modeling method, but it is also among the least accurate models that DTREG offers.

The basic idea of K-Means clustering is that clusters of items with the same target category are identified, and predictions for new data items are made by assuming they are of the same type as the nearest cluster center.

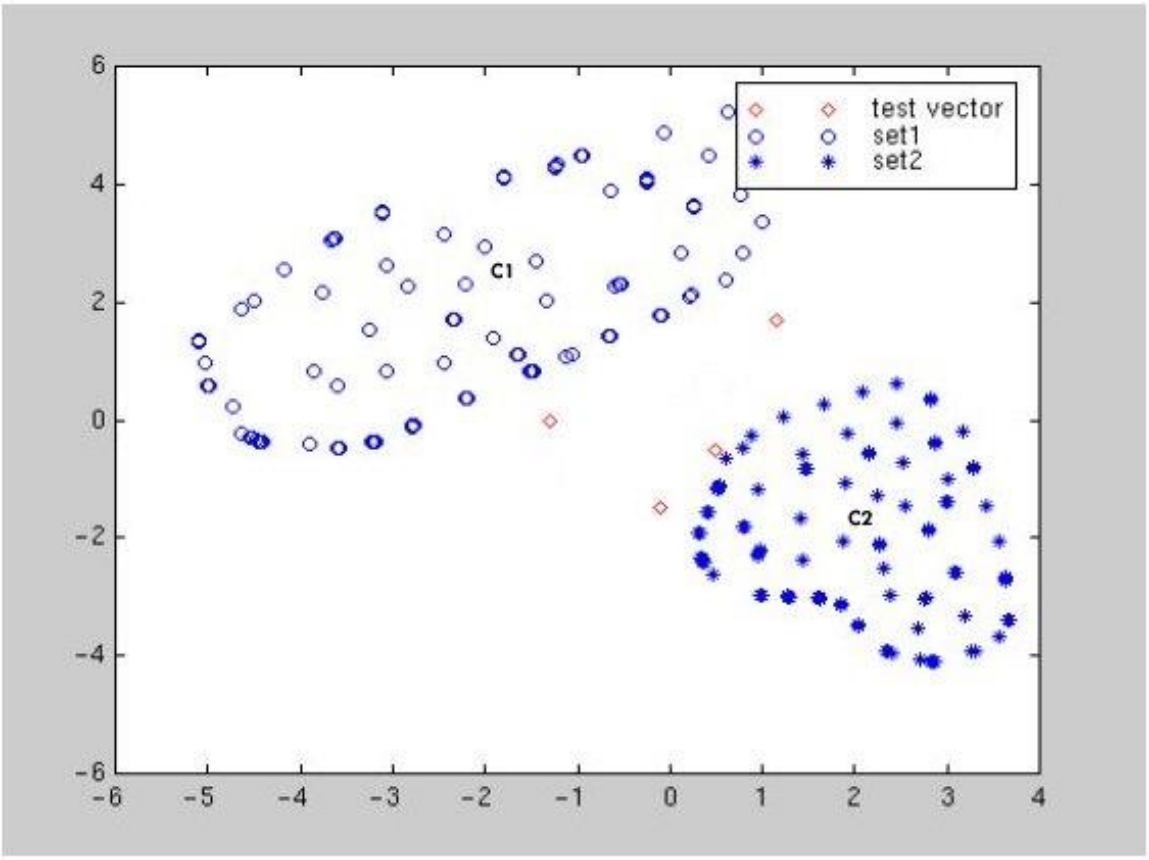
K-Means clustering is similar to two other more modern methods:

- **Radial Basis Function neural networks** (see page 258). An RBF network also identifies the centers of clusters, but RBF networks make predictions by considering the Gaussian-weighted distance to all other cluster centers rather than just the closest one.
- **Probabilistic Neural Networks** (see page 279). Each data point is treated as a separate cluster, and a prediction is made by computed the Gaussian-weighted distance to each point.

Usually, both RBF networks and PNN networks are more accurate than K-Means clustering models. PNN networks are among the most accurate of all methods, but they become impractically slow when there are more than about 10000 rows in the training data file. K-Means clustering is faster than RBF or PNN networks, and it can handle large training files.

K-Means clustering can be used only for classification (i.e., with a categorical target variable), not for regression. The target variable may have two or more categories.

To understand K-Means clustering, consider a classification involving two target categories and two predictor variables. The following figure (Balakrishnama and Ganapathiraju) shows a plot of two categories of items. Category 1 points are marked by circles, and category 2 points are marked by asterisks. The approximate center of the category 1 point cluster is marked "C1", and the center of category 2 points is marked "C2".



Four points with unknown categories are shown by diamonds. K-Means clustering predicts the categories for the unknown points by assigning them the category of the closest cluster center (C1 or C2).

There are two issues in creating a K-Means clustering model:

1. Determine the optimal number of clusters to create.
2. Determine the center of each cluster.

Most K-Means clustering programs don't provide any systematic way to find out the optimal number of clusters, and it usually isn't as obvious as shown in the figure above. So the person trying to create a model must experiment and try guesses to see what works best. DTREG provides an automatic search function that creates models using a varying number of clusters, tests each one and reports which is best. The model performance tests can be performed using cross-validation or holdout sampling. You can turn off the automatic search and specify a fixed number of clusters if you prefer.

Given the number of clusters, the second part of the problem is determining where to place the center of each cluster. Often, points are scattered and don't fall into easily recognizable groupings. Cluster center determination is done in two steps:

A. Determine starting positions for the clusters. This is performed in two steps:
    1. Assign the first center to a random point.

2. Find the point furthest from any existing center and assign the next center to it. Repeat this until the specified number of cluster centers have been found.

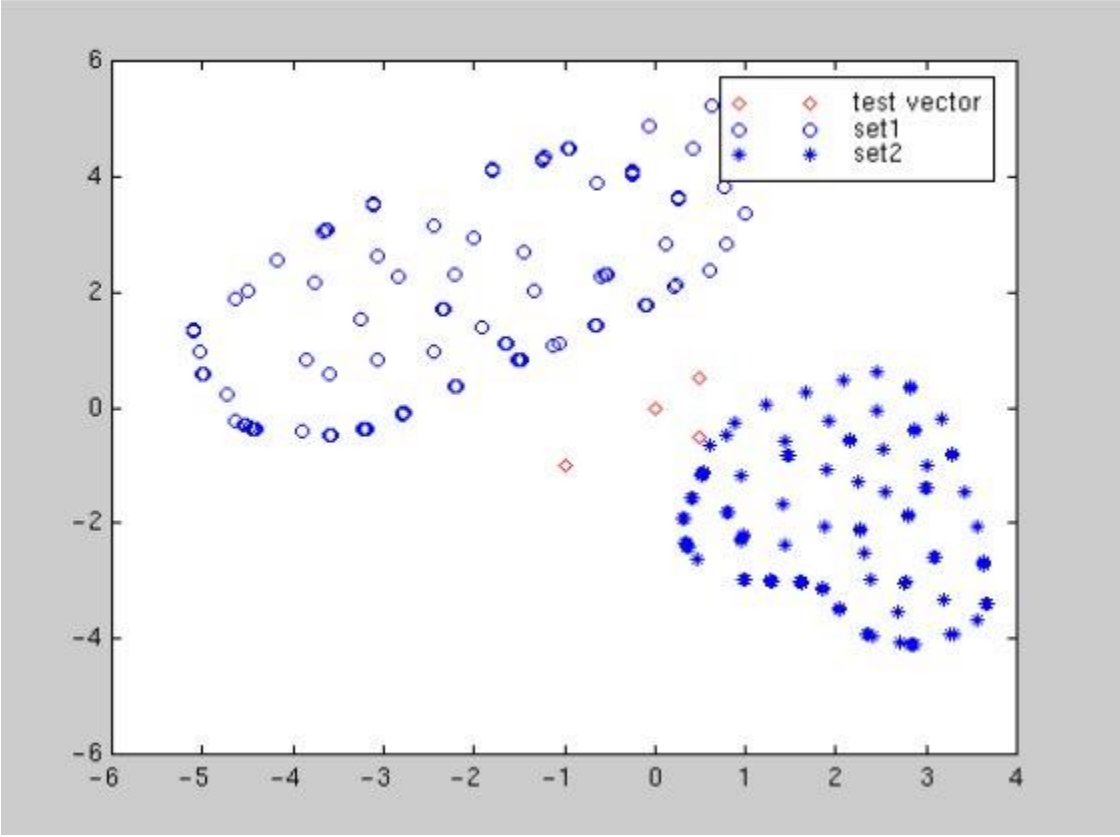
B. Adjust the center positions until they are optimized. DTREG does this using a modified version of the Hartigan-Wong algorithm that is much more efficient than the original algorithm.

Discriminant Analysis

Originally developed in 1936 by R.A. Fisher (Fisher, 1936), Discriminant Analysis is a classic method of classification that has stood the test of time. Discriminant analysis often produces models whose accuracy approaches (and occasionally exceeds) more complex modern methods.

Discriminant analysis can be used only for classification (i.e., with a categorical target variable), not for regression. The target variable may have two or more categories.

To explain discriminant analysis, let's consider a classification involving two target categories and two predictor variables. The following figure (Balakrishnama and Ganapathiraju) shows a plot of the two categories with the two predictors on orthogonal axes:

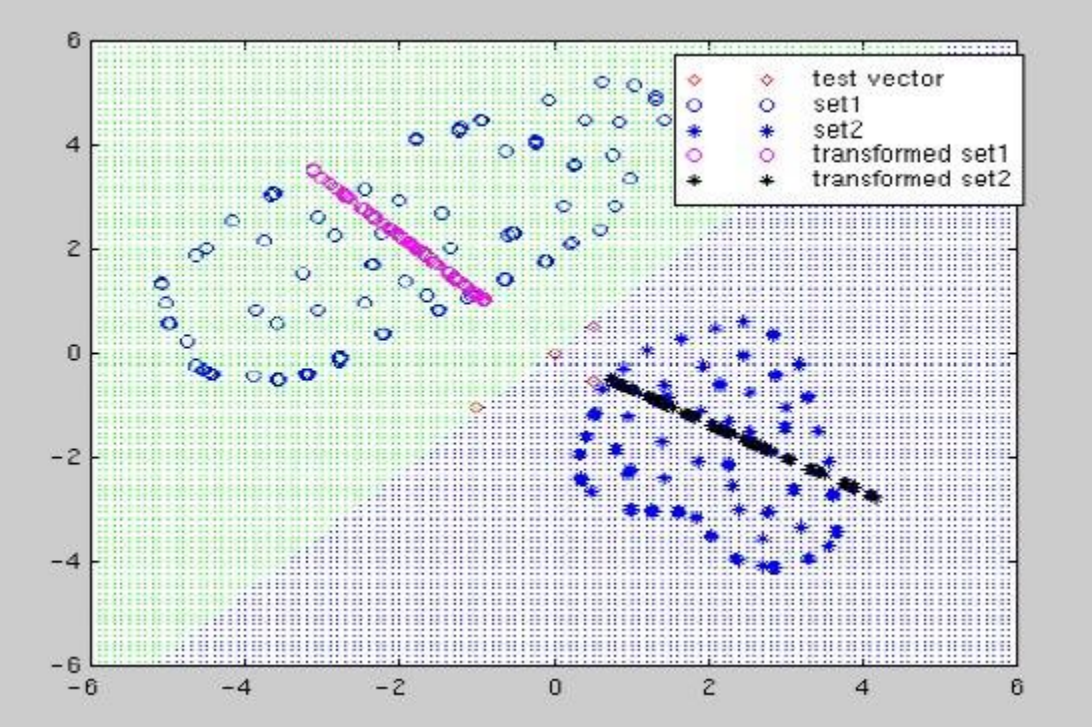


A visual inspection shows that category 1 objects (open circles) tend to have larger values of the predictor on the Y axis and smaller values on the X axis. However, there is overlap between the target categories on both axes, so we can't perform an accurate classification using only one of the predictors.

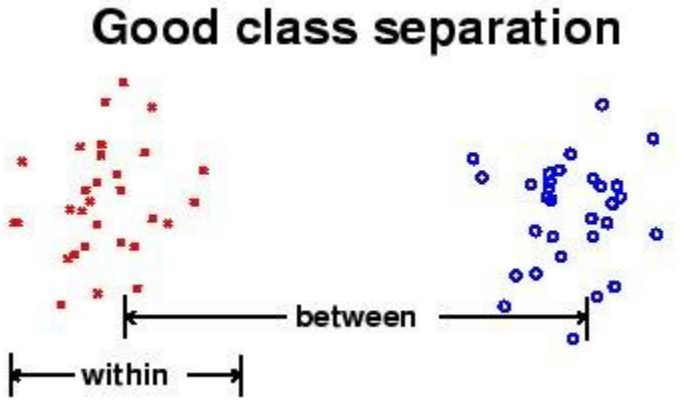
Linear discriminant analysis finds a linear transformation ("discriminant function") of the two predictors, X and Y, that yields a new set of transformed values that provides a more accurate discrimination than either predictor alone:

TransformedTarget = C1*X + C2*Y

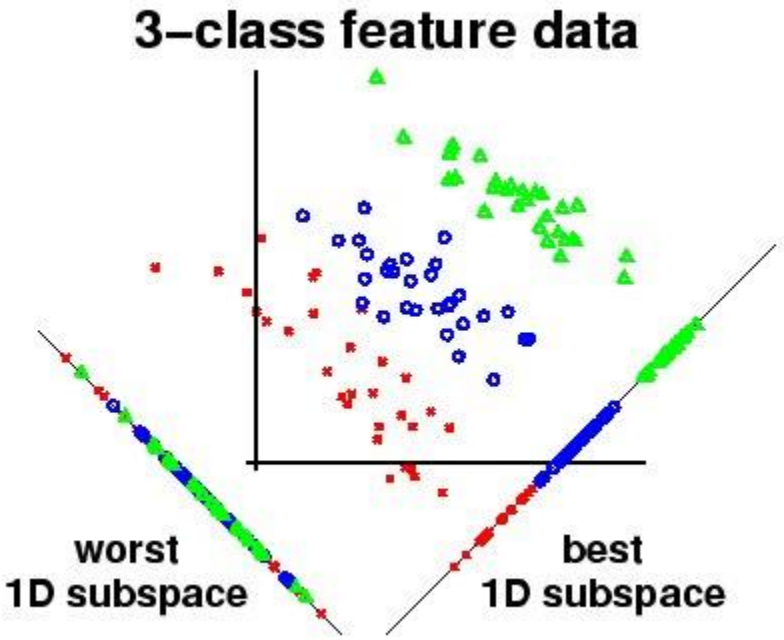
The following figure (also from Balakrishnama and Ganapathiraju) shows the partitioning done using the transformation function:



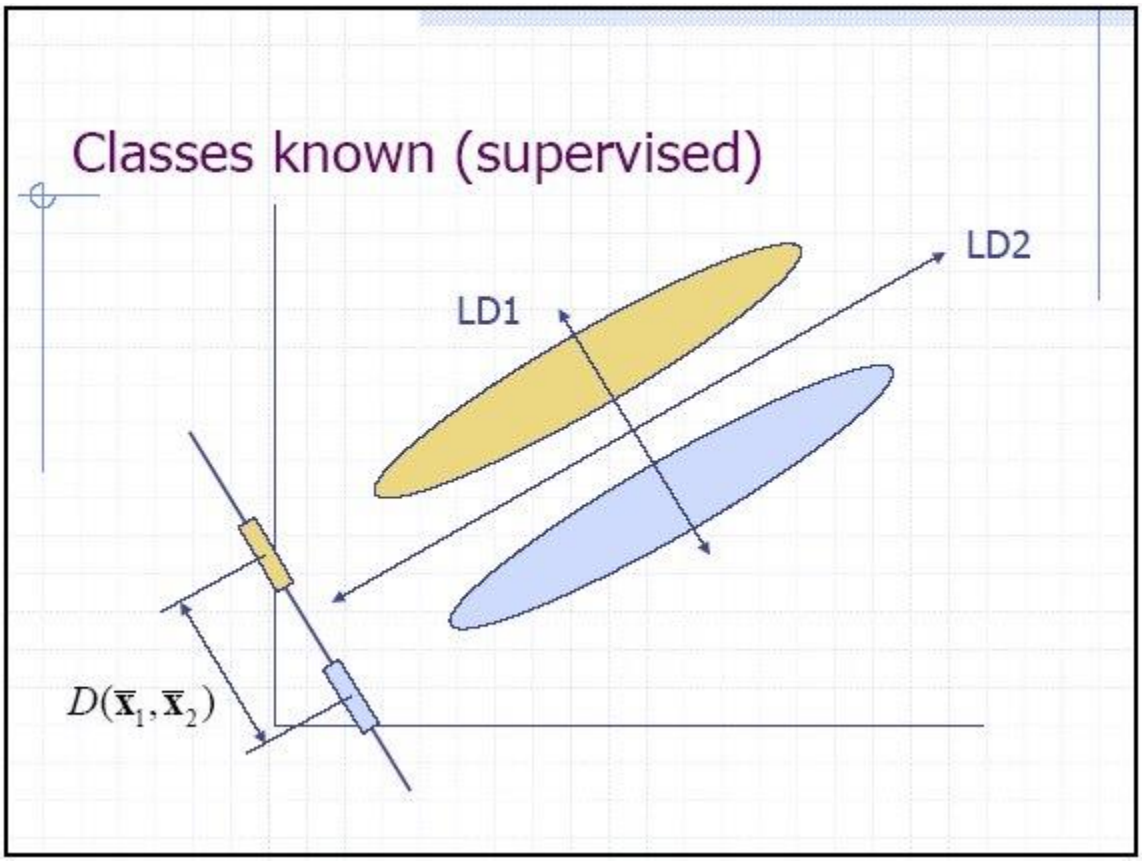
A transformation function is found that maximizes the ratio of between-class variance to within-class variance as illustrated by this figure produced by Ludwig Schwardt and Johan du Preez (Schwardt and Preez, 2005):



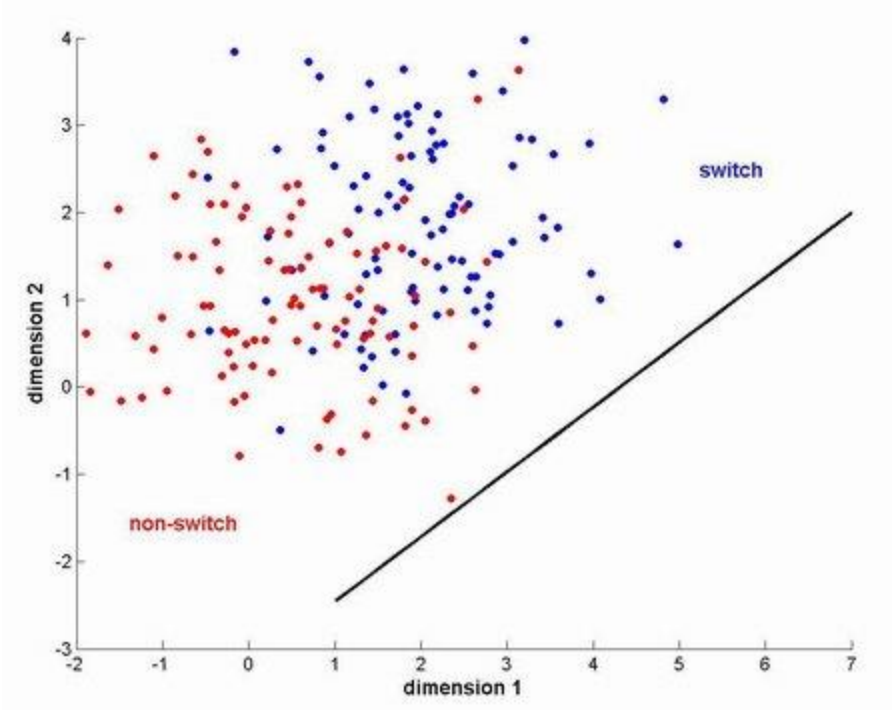
The transformation seeks to rotate the axes so that when the categories are projected on the new axes, the differences between the groups are maximized. The following figure (also by Schwardt and du Preez) shows two rotates axes. Projection to the lower right axis achieves the maximum separation between the categories; projection to the lower left axis yields the worst separation.



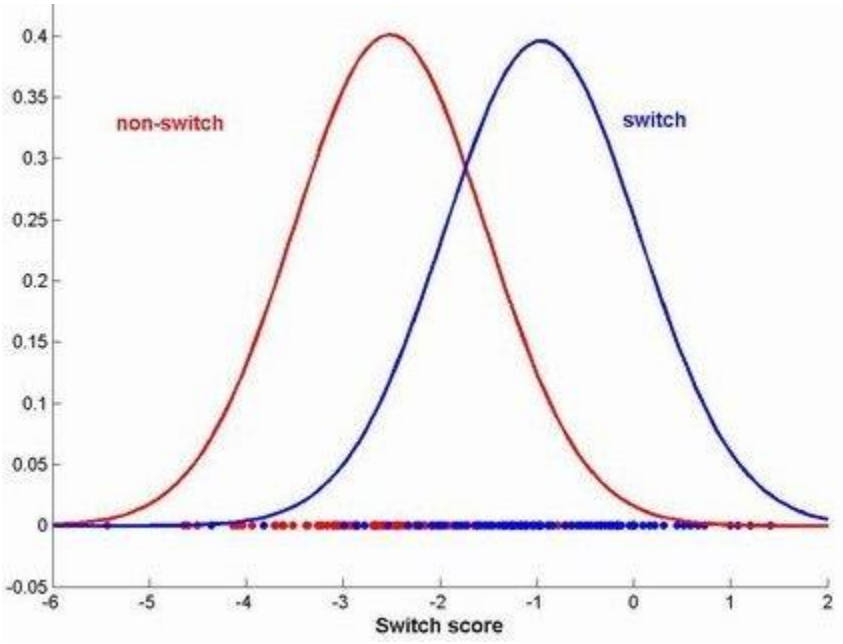
The following figure by Randy Julian (Julian, Lilly Labs) illustrates a distribution projected on the transformed axis labeled "D". Note that the projected values produce complete separation on the transformed axis, whereas there is overlap on both the original X and Y axes.



In the ideal case, a projection can be found that completely separates the categories (such as shown above). However, in most cases there is no transformation that provides complete separation, so the goal is to find the transformation that minimizes the overlap of the transformed distributions. The following figure by Alex Park and Christine Fry illustrates a distribution of two categories ("switch" in blue and "non-switch" in red). The black line shows the optimal axis found by linear discriminant analysis that maximizes the separation between the groups when they are projected on the line.



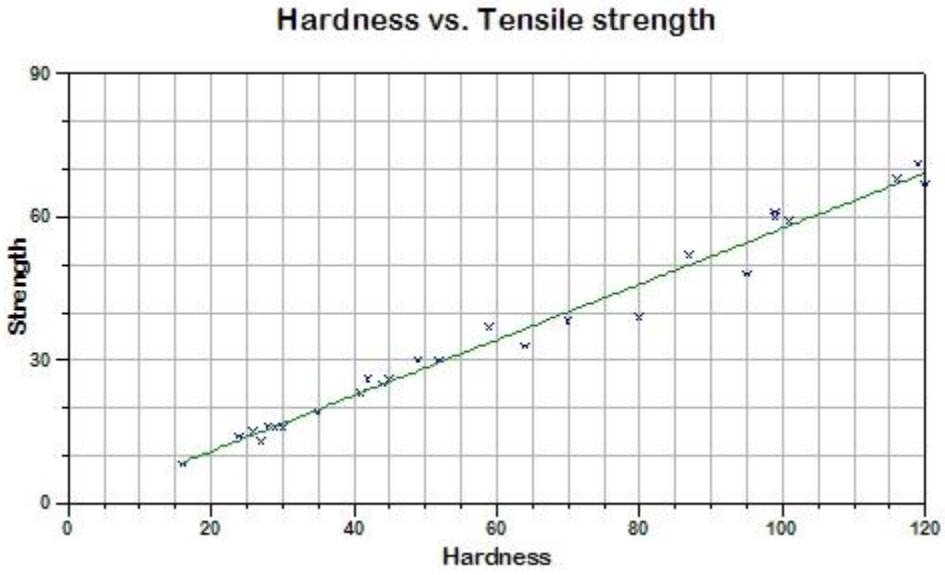
The following figure (also by Alex Park and Christine Fry) shows the distribution of the switch and non-switch categories as projected on the transformed axis (i.e., the black line shown in the figure above):



Note that even after the transformation there is overlap between the categories, but setting a cutoff point around -1.7 on the transformed axis yields a reasonable classification of the categories.

Linear Regression

I guide you in the way of wisdom and lead you along straight paths.
– Proverbs 4:11 (NIV)



Introduction to Linear Regression

Linear regression is the oldest and most widely used predictive model. The method of minimizing the sum of the squared errors to fit a straight line to a set of data points was published by Legendre in 1805 and by Gauss in 1809. The term "least squares" is from Legendre's term, *moindres carrés*. However, Gauss claimed that he had known the method since 1795. The term "regression" was coined in the nineteenth century to describe a biological phenomenon, namely that the progeny of exceptional individuals tend on average to be less exceptional than their parents and more like their more distant ancestors [from Wikipedia].

A linear regression model fits a linear function to a set of data points. The form of the function is:

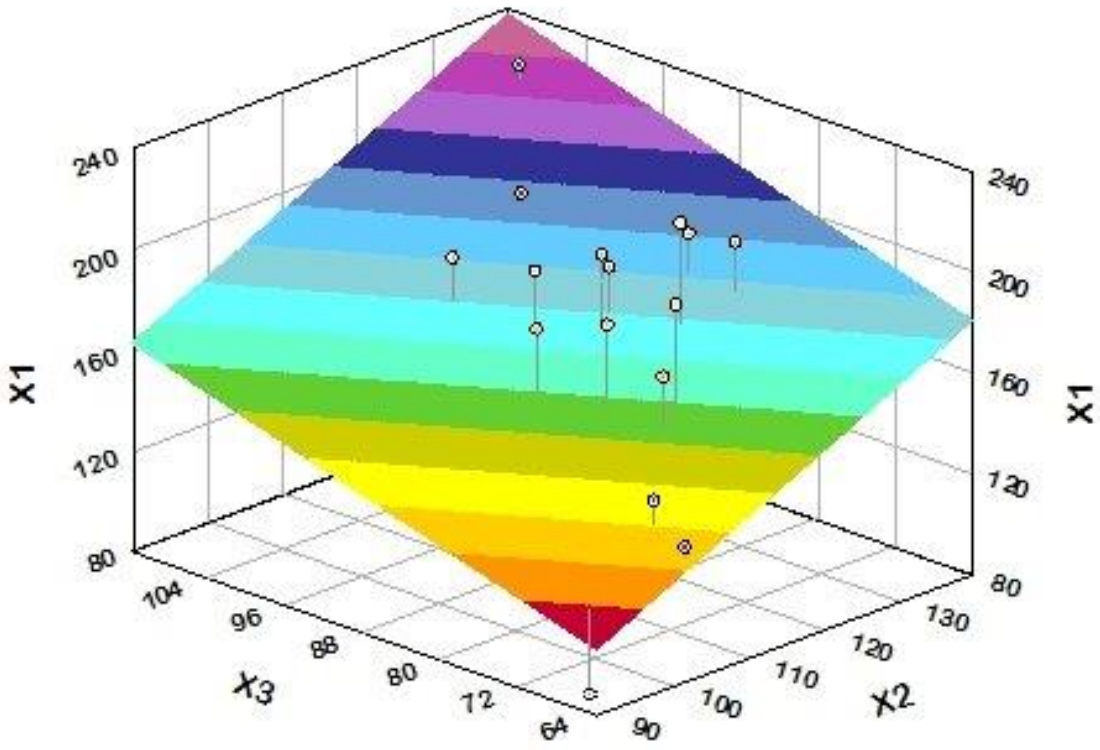
$$Y = \beta_0 + \beta_1 \cdot X_1 + \beta_2 \cdot X_2 + \cdots + \beta_n \cdot X_n$$

Where *Y* is the target variable, X_1, X_2,… *Xn* are the predictor variables, and β_1, …, β_n are coefficients that multiply the predictor variables. β_0 is a constant.

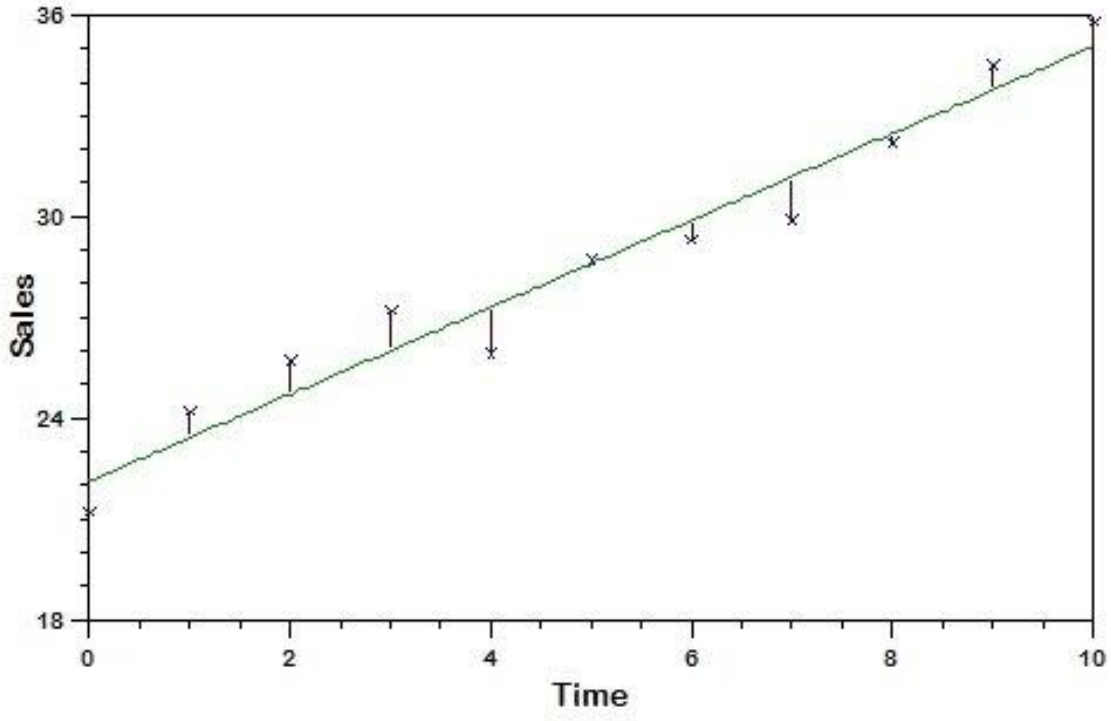
For example, the function shown above relating the strength of a material to hardness has the fitted equation:

Strength = -0.8453 + 0.58388*Hardness

If there is a single predictor variable (X_1), then the function describes a straight line. If there are two predictor variables, then the function describes a plane. If there are *n* predictor variables, then the function describes an *n*-dimensional hyperplane. Here is a plot of a fitted plane with two predictor variables:



If a perfect fit existed between the function and the actual data, the actual value of the target value for each record in the data file would exactly equal the predicted value. Typically, however, this is not the case, and the difference between the actual value of the target variable and its predicted value for a particular observation is the error of the estimate which is known as the "*deviation*" or "*residual*". The following plot depicts the residuals as red vertical lines connecting the data points and the fitted line.



The goal of regression analysis is to determine the values of the β parameters that minimize the sum of the squared residual values for the set of observations. This is known as a "least squares" regression fit. It is also sometimes referred to as "ordinary least squares" (OLS) regression.

Since linear regression is restricted to fitting linear (straight line/plane) functions to data, it rarely works as well on real-world data as more general techniques such as neural networks which can model non-linear functions. However, linear regression has a number of strengths:

- Linear regression is the most widely used method, and it is well understood.
- Training a linear regression model is usually much faster than methods such as neural networks.
- Linear regression models are simple and require minimum memory to implement, so they work well on embedded controllers that have limited memory space.
- By examining the magnitude and sign of the regression coefficients (β) you can infer how predictor variables affect the target outcome.

It is possible to use linear regression to fit functions with non-linear variables. To do this, use DTL (see page 153) or an external program to generate transformed values of variables, and then use the transformed variables as predictor variables for the function. For example, if you generate a new variable, X_2 using the transformation:

$$X_2 = X \cdot X$$

and include both X and X_2 as predictor variables, then the fitted function will be:

$$Y = \beta_0 + \beta_1 \cdot X_1 + \beta_2 \cdot X_2$$

Which is equivalent to

$$Y = \beta_0 + \beta_1 \cdot X + \beta_2 \cdot X^2$$

Linear regression is best suited for analyses with a continuous target variable, but DTREG also can create linear regression models to perform classification with a categorical target variable. When the target variable has two categories, a function is created to predict 1 for one of the categories and 0 for the other. If the target variable has more than two categories, DTREG creates a separate linear regression function for each category. A category function is trained to generate 1 if the category it is modeling is true and 0 for any other category.

If there are categorical predictor variables, DTREG generates a separate predictor variable for each category. A created predictor-category variable has the value 1 if the predictor variable has the category it represents and 0 if the predictor variable has any other category. If a categorical predictor variable has *n* categories, then (*n*-1) dummy variables are generated. Each generated variable has the value 1 if the variable's value matches its associated category. All generated variables have the value 0 if the value of the predictor variable matches the remaining category. For example, if predictor variable TicketClass has three categories, FirstClass, Tourist and SuperSaver, then DTREG will arbitrarily select two of the categories for generated variables; let's assume it selects FirstClass and Tourist. Then if the value of TicketClass is FirstClass, the generated variables would have the values: FirstClass=1, Tourist=0. If TicketClass was Tourist, then the generated variables would have the values: FirstClass=0, Tourist=1. And if TicketClass was SuperSaver, then the generated variables would have the values: FirstClass=0, Tourist=0.

Several computational algorithms can be used to perform linear regression. DTREG uses Singular Value Decomposition (SVD) which is robust and less sensitive to predictor variables that are nearly collinear.

Output Generated for Linear Regression

In addition to statistics measuring how well the function fits the data (see page 189), DTREG generates a table showing the computed β coefficient values.

Computed Coefficient (Beta) Values

Variable	Coefficient	Std. Error	t	Prob(t)	95% Confidence Interval	
Hardness	0.583884	0.016	36.40	< 0.00001	0.5508	0.6169
Constant	-0.845341	1.106	-0.76	0.45203	-3.124	1.434

A line is displayed showing the computed β coefficient for each predictor variable. If a constant (β_0) is included in the equation, the last line shows the value of "Constant".

Using the information in this table, we conclude that the function is:

Strength = -0.845341 + 0.583884*Hardness

In addition to the coefficient value, the standard error of the coefficient is displayed along with several other statistics:

t Statistic

The "t" statistic is computed by dividing the estimated value of the β coefficient by its standard error. This statistic is a measure of the likelihood that the actual value of the parameter is *not* zero. The larger the absolute value of t, the less likely that the actual value of the parameter could be zero. The t statistic probability is computed using a two-sided test.

Prob(t)

The "Prob(t)" value is the probability of obtaining the estimated value of the coefficient if the actual coefficient value is zero. The smaller the value of Prob(t), the more significant the coeficient and the less likely that the actual value is zero. For example, assume the estimated value of a parameter is 1.0 and its standard error is 0.7. Then the t value would be 1.43 (1.0/0.7). If the computed Prob(t) value was 0.05 then this indicates that there is only a 0.05 (5%) chance that the actual value of the parameter could be zero. If Prob(t) was 0.001 this indicates there is only 1 chance in 1000 that the parameter could be zero. If Prob(t) was 0.92 this indicates that there is a 92% probability that the actual value of the parameter could be zero; this implies that the term of the regression equation containing the parameter can be eliminated without significantly affecting the accuracy of the regression. One thing that can cause Prob(t) to be 1.00 (or near 1.00) is having redundant parameters. If at the end of an analysis several parameters have Prob(t) values of 1.00, check the function carefully to see if one or more of the parameters can be removed.

ANOVA Table

```
--  ANOVA and F Statistics  --

  Source      DF    Sum of Squares    Mean Square     F value    Prob(F)
----------  ------  --------------  --------------  ----------  ---------
Regression       2        16510.39        8255.195      10.885   0.000432
Error           24        18201.91        758.4128
Total           26         34712.3
```

An "Analysis of Variance" table provides statistics about the overall significance of the model being fitted.

F Value, and Prob(F)

The "F value" and "Prob(F)" statistics test the overall significance of the regression model. Specifically, they test the null hypothesis that all of the regression coefficients are equal to zero. This tests the full model against a model with no variables and with the estimate of the dependent variable being the mean of the values of the dependent variable. The F value is the ratio of the mean regression sum of squares divided by the mean error sum of squares. Its value will range from zero to an arbitrarily large number.

The value of Prob(F) is the probability that the null hypothesis for the full model is true (i.e., that all of the regression coefficients are zero). For example, if Prob(F) has a value of 0.010 then there is 1 chance in 100 that all of the regression parameters are zero. This low a value would imply that at least some of the regression parameters are nonzero and that the regression equation does have some validity in fitting the data (i.e., the independent variables are not purely random with respect to the dependent variable).

Confidence interval

The confidence interval shows the range of values for the computed coefficient that covers the actual coefficient value with the specified confidence. For example, the results above show a 95% confidence interval of 0.5508 to 0.6169 for the Hardness coefficient. This means that we are 95% confident that the true coefficient of Hardness falls in this range. You can set the percentage for the confidence interval on the Linear Regression Property Page (see page 115).

Coefficients for categorical predictor variables

If some of the predictor variables have categorical values, then the table of computed coefficients has a line for each variable generated for categories. Here is an example:

```
  --------------  Coefficient (Beta) Values for Survived = 1  (Yes)  --------------

Variable    Coefficient    Std. Error       t       Prob(t)     95% Confidence Interval
---------  -------------  ------------  ---------  ---------  ------------  ------------
Class
     Crew       0.131181       0.02164       6.06  < 0.00001       0.08875        0.1736
    First       0.306734       0.02771      11.07  < 0.00001        0.2524        0.3611
   Second       0.120654       0.02852       4.23    0.00002       0.06473        0.1766
Age
    Adult      -0.181296       0.04097      -4.43    0.00001       -0.2616        -0.101
Sex
     Male       -0.49068       0.02301     -21.33  < 0.00001       -0.5358       -0.4456
Constant        0.767591       0.04186      18.34  < 0.00001        0.6855        0.8497
```

Note that variables were generated for Crew, First and Second categories of Class. The variable generated for Age is 1 if Age=Adult and 0 otherwise. Similarly, there is a generated value for Sex that has the value 1 if Sex=Male and 0 otherwise.

Logistic Regression

Introduction to Logistic Regression

Logistic Regression is a type of predictive model that can be used when the target variable is a categorical variable with two categories – for example live/die, has disease/doesn't have disease, purchases product/doesn't purchase, wins race/doesn't win, etc. A logistic regression model does not involve decision trees and is more akin to nonlinear regression such as fitting a polynomial to a set of data values.

Logistic regression can be used only with two types of target variables:

1. A categorical target variable that has exactly two categories (i.e., a *binary* or *dichotomous* variable).
2. A continuous target variable that has values in the range 0.0 to 1.0 representing probability values or proportions.

As an example of logistic regression, consider a study whose goal is to model the response to a drug as a function of the dose of the drug administered. The target (dependent) variable, Response, has a value 1 if the patient is successfully treated by the drug and 0 if the treatment is not successful. Thus the general form of the model is:

Response = f(dose)

The input data for Response will have the value 1 if the drug is effective and 0 if the drug is not effective. The value of Response predicted by the model represents the probability of achieving an effective outcome, P(Response=1|Dose). As with all probability values, it is in the range 0.0 to 1.0.

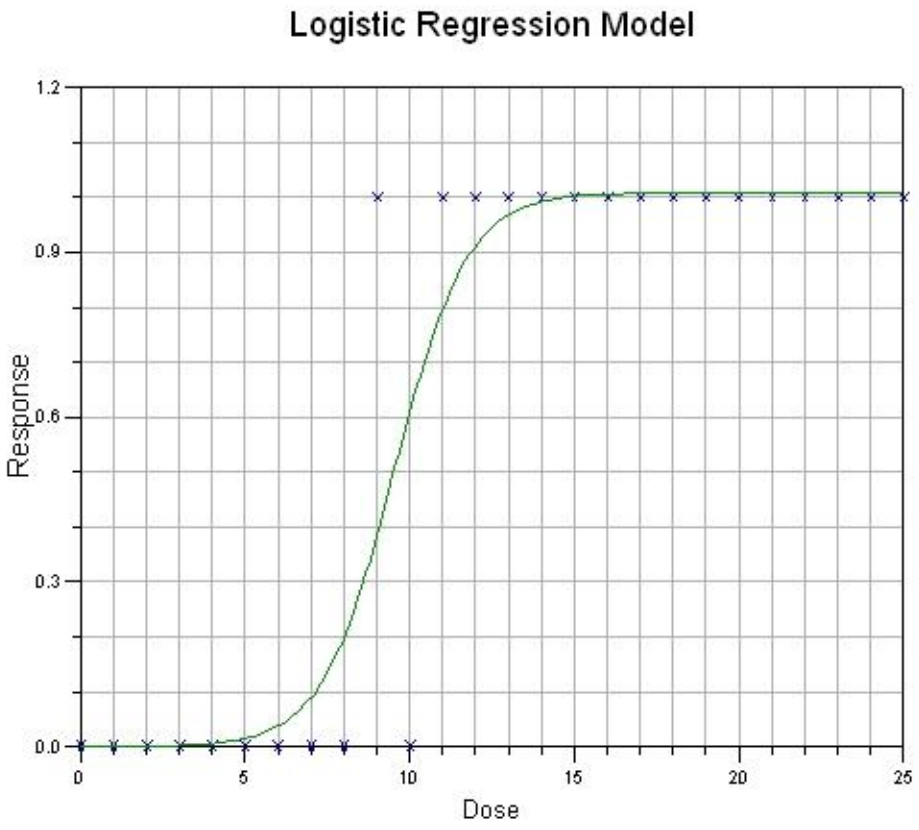
One obvious question is "Why not simply use linear regression?" In fact, many studies have done just that, but there are two significant problems:

1. There are no limits on the values predicted by a linear regression, so the predicted response might be less than 0 or greater than 1 – clearly nonsensical as a response probability.

2. The response usually is *not* a linear function of the dosage. If a minute amount of the drug is administered, no patients will respond. Doubling the dose to a larger but still minute amount will not yield any positive response. But as the dosage is increases a threshold will be reached where the drug begins to become effective. Incremental increases in the dosage above the threshold usually will elicit an increasingly positive effect. However, eventually a saturation level is reached, and beyond that point increasing the dosage does not increase the response.

The Dose-Response Curve

The logistic regression dose-response curve has an S (sigmoidal) shape such as shown here:



Notice that all of the Response values are 0 or 1. The Dose varies from 0 to 25. Below a dose of 9 all of the Response values are 0. Above a dose of 10 all of the response values are 1.

The Logistic Model Formula

The logistic model formula computes the probability of the selected response as a function of the values of the predictor variables.

If a predictor variable is categorical variable with two values, then one of the values is assigned the value 1 and the other is assigned the value 0. Note that DTREG allows you to use any value for categorical variables such as "Male" and "Female", and it converts these symbolic names into 0/1 values. So you don't have to be concerned with recoding categorical values.

If a predictor variable is a categorical variable with more than two categories, then a separate dummy variable is generated to represent each of the categories except for one which is excluded. The value of the dummy variable is 1 if the variable has that category, and the value is 0 if the variable has any other category; hence, no more than one dummy variable will be 1. If the variable has the value of the excluded category, then all of the dummy variables generated for the variable are 0. DTREG automatically generates the dummy variables for categorical predictor variables; all you have to do is designate variables as being categorical.

In summary, the logistic formula has each continuous predictor variable, each dichotomous predictor variable with a value of 0 or 1, and a dummy variable for every category of predictor variables with more than two categories less one category.

The form of the logistic model formula is:

$$P = 1/(1 + \exp(-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_k X_k)))$$

Where β_0 is a constant and β_i are coefficients of the predictor variables (or dummy variables in the case of multi-category predictor variables). The computed value, *P*, is a probability in the range 0 to 1. The exp() function is e raised to a power. You can exclude the β_0 constant by turning off the option "Include constant (intercept) term" on the logistic regression model property page.

Output Generated for a Logistic Regression Analysis

Summary statistics for the model

```
============  Logistic Regression Parameters  ============

Predict: DeathPenalty = 1  (Yes)

Number of parameters calculated = 4
Number of data rows used = 147

Wald confidence intervals are computed for 95% probability.

Log likelihood of model = -88.142490
Deviance (-2 * Log likelihood) = 176.284981
Akaike's Information Criterion (AIC) = 184.284981
Bayesian Information Criterion (BIC) = 196.246711
```

The summary statistics begin by showing the name of the target variable and the category of the target whose probability is being predicted by the model. You can select the category on the logistic regression property page for the analysis.

The **log likelihood** of the model is the value that is maximized by the process that computes the maximum likelihood value for the β parameters. Technically, it is the value of the likelihood function,

$$\log(L) = \sum_i \beta \mathbf{x}_i y_i - \sum_i \log(1 + e^{\beta x_i})$$

The **Deviance** is equal to -2*log-likelihood.

Akaike's Information Criterion (AIC) is -2*log-likelihood+2*k where k is the number of estimated parameters.

The **Bayesian Information Criterion (BIC)** is -2*log-likelihood + k*log(n) where k is the number of estimated parameters and n is the sample size. The Bayesian Information Criterion is also known as the **Schwartz criterion**.

Computed Beta Parameters

```
  -----------------  Computed Parameter (Beta) Values  -----------------

   Variable      Parameter   Std. Error   Pr. Chi Sq.   Lower C.I.    Upper C.I.
--------------  ----------  ------------  -----------  ------------  ------------
BlackDefendant      0.5952         0.394       0.1308        -0.177         1.367
WhiteVictim         0.2565         0.400       0.5216        -0.528         1.041
Serious             0.1871         0.061       0.0022         0.067         0.307
Constant           -2.6516         0.675     < 0.0001        -3.974        -1.329
```

The computed beta parameters are the maximum likelihood values of the β parameters in the logistic regression model formula (see above). By using them in an equation with the corresponding values of the predictor (X) variables, you can compute the expected probability, P, for an observation.

In addition to the maximum likelihood value, the standard error for the estimate is displayed along with the Chi squared probability that the true value of the parameter is not zero. The last two columns display the Wald upper and lower confidence intervals. You can select the confidence interval percentage range on the Logistic Regression property page.

The odds ratios corresponding to the parameter values are displayed in the next table. The odds ratios are computed by raising e (base of natural logs) to the power of the parameter value.

Odds Ratios

Variable	Odds Ratio	Lower C.I.	Upper C.I.
BlackDefendant	1.8134	0.8378	3.9247
WhiteVictim	1.2924	0.5898	2.8316
Serious	1.2057	1.0694	1.3594

If a predictor variable is categorical, then a dummy variable is generated for each category except for one. In this case, there is a β parameter for each dummy variable, and the categories are shown indented under the names of the variables like this:

Computed Parameter (Beta) Values

Variable	Parameter	Std. Error	Pr. Chi Sq.	Lower C.I.	Upper C.I.
Class					
Crew	0.8845	0.1643	< 0.0001	0.5624	1.2065
First	1.7733	0.1896	< 0.0001	1.4016	2.1450
Second	0.7742	0.1921	< 0.0001	0.3977	1.1507
Age					
Adult	-1.0225	0.2726	0.0002	-1.5568	-0.4881
Sex					
Male	-2.2831	0.1534	< 0.0001	-2.5838	-1.9825
Constant	1.1915	0.2765	< 0.0001	0.6495	1.7334

Likelihood Ratio Statistics

```
  ------  Likelihood Ratio Statistics  ------

   Variable      L. Ratio    DF   Pr. Chi Sq.
--------------  ----------  ----  -----------
BlackDefendant       2.321    1       0.12763
WhiteVictim          0.413    1       0.52020
Serious             10.234    1       0.00138
Constant            18.609    1       0.00002
```

If you enable the option "Compute likelihood ratio significance tests" on the logistic regression property page, then a table similar to the one shown above will be printed. The likelihood ratio significance tests are computed by performing a logistic regression with each parameter omitted from the model and comparing the log likelihood ratio for the model with and without the parameter. These significance tests are considered to be more reliable than the Wald significance test. However, since the logistic regression must be recomputed with each predictor omitted, the computation time increases in proportion to the number of predictor variables. If a predictor variable is a categorical variable with multiple categories, the significance test is performed with all of the categories included and all of them excluded.

Computational Issues for Logistic Regression

Failure to Converge

An iterative Newton-Raphson algorithm is used to calculate the maximum likelihood values of the parameters. This procedure uses the partial second derivatives of the parameters in the Hessian matrix to guide incremental parameter changes in an effort to maximize the log likelihood value for the likelihood function. The algorithm iterates until the absolute value of the largest parameter change is less than the value specified for "Tolerance" on the logistic regression property page.

Most logistic regression analyses converge to a solution in a dozen or so iterations, but you may occasionally run into one that does not converge. If this happens, try enabling the option "Use Firth's procedure" on the logistic regression property page. Firth's procedure slows down the calculations, but it usually results in achieving convergence. Note: if Firth's procedure is enabled, unbiased parameter values are calculated which may be somewhat different than what you would get with Firth's procedure turned off.

Singular Hessian Matrix

The Hessian matrix with the partial second derivatives of the parameter values is used to guide the convergence process. If the Hessian matrix is singular, the logistic regression procedure will be unsuccessful and a warning message will be displayed.

Complete and Quasi-Complete Separation of Values

Complete separation is a condition where one predictor or a linear combination of predictors perfectly predicts the target value. For example, consider a situation where every value of the Response target variable is 0 if Dose is less than 10 and every value is 1 if Dose is greater than 10. Then the value of Response can be perfectly predicted by checking if Dose is less than or greater than 10. In this case it is impossible to compute the maximum likelihood values for the β parameters because the slope of the logistic function would be infinite.

At the beginning of each logistic regression analysis, a check is made for complete separation on each predictor variable. If complete separation is detected, a report will be generated similar to this:

```
-----------  Report On Separation of Variables  -----------

Warning: Complete separation of target values occurs on Age
```

The example above indicates that values of the target variable are completely determined by the Age predictor variable. If separation occurs for a particular category of a multi-category predictor variable, the category will be shown in brackets after the variable name, for example "Race[2]".

Quasi-complete separation occurs when values of the target variable overlap or are tied at a single or only a few values of a predictor variable. The analysis does not check for quasi-complete separation, but the symptoms are extremely large calculated values for the β parameters or large standard errors. The analysis also may fail to converge.

If complete or quasi-complete separation is detected, the predictor variable(s) showing separation should be removed from the analysis.

Correlation, Factor Analysis, Principal Components

Correlation, Factor Analysis, and Principal Components Analysis are different than the other procedures in DTREG, because they do not generate predictive models. Instead, these procedures are used for exploratory analysis where you are trying to understand the nature and relationship between variables.

See page 119 for information about setting parameters to control these procedures.

Introduction to Correlation

Correlation is a measure of the association between two variables. That is, it indicates if the value of one variable changes reliably in response to changes in the value of the other variable. The **correlation coefficient** can range from -1.0 to +1.0. A correlation of -1.0 indicates that the value of one variable decreases as the value of the other variable increases. A correlation of +1.0 indicates that when the value of one variable increases, the other variable increases. Positive correlation coefficients less than 1.0 mean that an increasing value of one variable tends to be related to increasing values of the other variable, but the increase is not regular – that is, there may be some cases where an increased value of one variable results in a decreased value of the other variable (or no change). A correlation coefficient of 0.0 means that there is no association between the variables: a positive increase in one variable is not associated with a positive or negative change in the other.

Types of Correlation Coefficients

When used without qualification, "correlation" refers to the linear correlation between two continuous variables, and it is computed using the **Pearson Product Moment** function. A Pearson correlation coefficient of 1.0 occurs when an increase in value of one variable results in an increase in value of the other variable in a linear fashion. That is, doubling the value of one variable doubles the value of the other variable.

If two variables have an association but the relationship is not linear, then the Pearson correlation coefficient will be less than 1.0 even if there is a perfectly reliable change in one variable as the other changes. The **Spearman rank-order correlation** coefficient is the most popular method for handling non-linear correlation. Spearman correlation sorts the values being correlated and replaces the values by their order (rank) in the sorted list. So the smallest value is replaced by 1, the second smallest by 2, etc. Correlation is then computed using the rank-orders rather than the original data values. The Spearman correlation coefficient will be 1.0 if a positive change in one variable produces a positive change in the other variable even if the response is not linear.

Most correlation programs can compute correlations only between two continuous variables. Since DTREG allows categorical variables, it must also compute correlations between categorical variables. It's not too hard to grasp the idea of correlating two categorical variables with dichotomous values such as correlating Sex (male/female) with Outcome (live/die), but it is harder to imagine correlating categorical variables with multiple categories such as Marital Status with State of Residence. However, there are established correlation procedures for handling these cases, and DTREG implements procedures for handling all combinations of correlations between continuous, dichotomous, and general categorical variables. The table below shows the method used for each case.

	Continuous	**Dichotomous**	**Multi-Category**
Continuous	Pearson or Spearman	Point biserial	Tau squared
Dichotomous	Point biserial	Phi coefficient	Cramer's V or Entropy
Multi-Category	Tau squared	Cramer's V or Entropy	Cramer's V or Entropy

The correlation between two multi-category variables is essentially an ANOVA to determine if there is a significant difference between the number of cases that fall in the cells of an *n* by *m* array where *n* and *m* are the number of categories of the two variables. These correlations can vary only from 0.0 to 1.0; they cannot be negative.

The Correlation Matrix

If you compute the correlation between *n* variables, then these correlations can be presented in the form of an *n* by *n* matrix such as shown here:

	V1	V2	V3	V4	V5	V6
V1	1.0000	0.4944	0.7134	-0.1041	0.1141	0.0762
V2	0.4944	1.0000	0.3882	0.0535	-0.0597	0.1423
V3	0.7134	0.3882	1.0000	-0.0247	0.2038	0.0583
V4	-0.1041	0.0535	-0.0247	1.0000	0.6201	0.6353
V5	0.1141	-0.0597	0.2038	0.6201	1.0000	0.4551
V6	0.0762	0.1423	0.0583	0.6353	0.4551	1.0000

Introduction to Factor Analysis and Principal Components Analysis

When you find a set of variables that are highly correlated with each other, it is reasonable to wonder if this mutual association may be due to some common underlying cause. For example, suppose values for the following variables are collected for an incoming college freshman class: High school GPA, IQ, SAT Verbal, SAT Math,

Height, Weight, Waist size, and Chest size. A correlation matrix for these variables is likely to show large positive correlations between High school GPA, IQ, and SAT scores. Similarly, Height, Weight, Waist and Chest measurements will probably be positively correlated. So, the question is whether High school GPA, IQ, and SAT scores are related because of some underlying, common factor. The answer, of course, is yes, because they are all measures of intelligence. Similarly, Height, Weight, Waist, and Chest measurements are all related to physical size. So the conclusion is that there are only two underlying factors that are being measured by the eight variables, and these factors are intelligence and physical size. These common factors are sometimes called *latent variables*. Since "intelligence" is an abstract concept, it cannot be measured directly: instead, measures such as GPA, IQ, etc. are used to estimate the intelligence of an individual.

In the simple example presented above, it's not too difficult to isolate the pattern of correlations that link the variables in the two groups; but when you have hundreds of variables and there are multiple underlying factors, it is much more difficult to identify the factors and the variables associated with each factor.

The purpose of Factor Analysis is to identify a set of underlying factors that explain the relationships between correlated variables. Generally, there will be fewer underlying factors than variables, so the factor analysis result is simpler than the original set of variables.

Principal Component Analysis is very similar to Factor Analysis, and the two procedures are sometimes confused. Both procedures are built on the same mathematical techniques. **Factor Analysis** assumes that the relationship (correlation) between variables is due to a set of underlying factors (latent variables) that are being measured by the variables.

Principal Components Analysis is not based on the idea that there are underlying factors that are being measured. It is simply a technique for finding a linear combination of the original variables that produce orthogonal (uncorrelated) variables that explain the maximum amount of variance in the original variables. It is often used to reduce the number of variables while retaining most of the predictive power.

The goal of PCA is to rigidly rotate the axes of an *n*-dimensional space (where *n* is the number of variables) to a new orientation that has the following properties:

1. The first axis corresponds to the direction with the most variance among the variables, and subsequent axes have progressively less variance in their direction.
2. The correlation between each pair of rotated axes is zero. This is a result of the axes being orthogonal to each other (i.e., they are uncorrelated).

PCA is performed by finding the eigenvalues and eigenvectors of the covariance or correlation matrix. The eigenvectors represent a linear transformation from the original variable coordinates to rotated coordinates that satisfy the criteria listed above. For example, if you have variables X_1 through X_n. Then the eigenvector components would be:

$$EVC_1 = a_{11}X_1 + a_{12}X_2 + \ldots + a_{1n}X_n$$

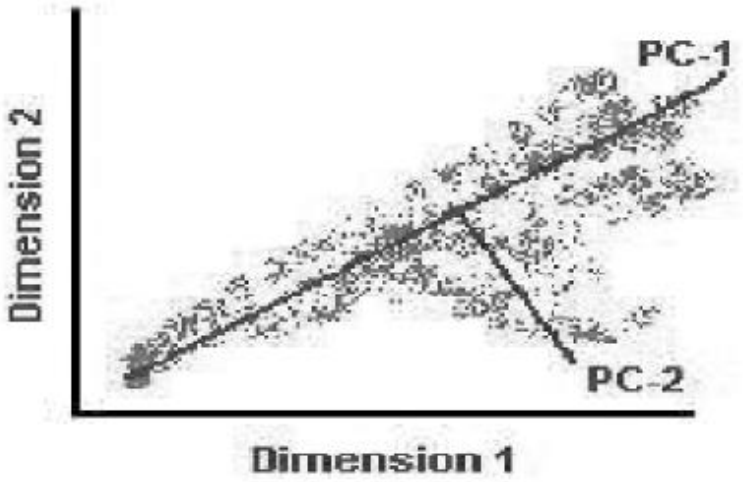
Through

$$EVC_m = a_{m1}X_1 + a_{m2}X_2 + \ldots + a_{mn}X_n$$

Where a_{mn} is the eigenvector value of the *m*th eigenvector component and the *n*th variable. Note that the principal components are just linear combinations of the variables. There is an option on the PCA properties page where you can specify a file to which the coefficients of the PCA function will be written (see page 119).

DTREG can compute the variable values after being transformed by eigenvectors and write them to a file. See the PCA option screen on page 119 for information about this option.

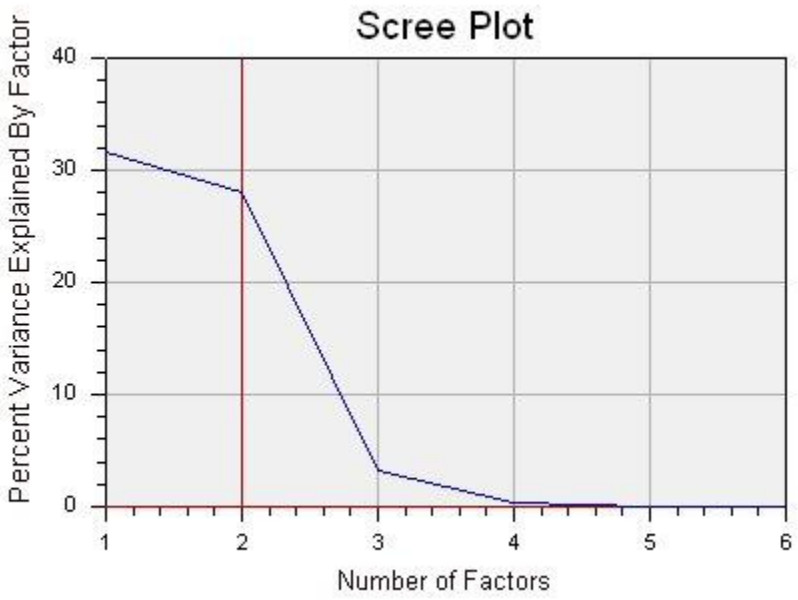
Here is an example showing two principal components fitted to two variables. Note that PC_1 is oriented along a line that has the maximum variance (dispersion) of values, and PC_2 is orthogonal (perpendicular) to PC_1.



Determining the Number of Factors to Use

In the example at the beginning of this chapter, we concluded that the eight variables were related to two underlying factors, intelligence and physical size. However, the choice of two factors was arbitrary. It is likely that IQ and SAT scores will have higher correlation with each other than with GPA, because GPA is largely affected by motivation and effort. Similarly, weight, waist, and chest size may be measures of heft while height may be something different. So perhaps we should use four factors: (1) IQ, SAT Verbal, SAT Math; (2) GPA; (3) Weight, Waist, and Chest size; (5) Height.

Determining the number of factors to use has been an issue since the beginning of factor analysis. There is no perfect way to determine how many factors to use: there are a number of suggested guidelines, but ultimately it is a judgment call. One of the most useful measures is a chart called a Scree Plot that shows what percentage of the variance is accounted for by each factor. A small scree plot is shown in the analysis report; a larger and prettier one can be seen by clicking Charts/Model Size after finishing an analysis. Here is an example of a Scree Plot:



The horizontal axis shows the number of factors. The vertical axis shows the percent of the overall variance explained by each factor. Notice that there is a sharp drop off after 2 factors. So in this case, it is reasonable to retain two factors.

DTREG includes a number of methods for controlling how many factors are used. See the Factor Analysis property page described on page 119 for details.

Output Generated by Factor Analysis

Factor Importance (Eigenvalue) Table

The Factor Importance table shows the relative and cumulative amount of variance explained by each factor. Here is an example of such a table:

```
  ==============  Factor Importance  ==============

Factor  Eigenvalue  Variance %  Cumulative %       Scree Plot
------  ----------  ----------  ------------  --------------------
    1      1.90099    31.683       31.683     ********************
    2      1.68129    28.022       59.705     *****************
    3      0.18959     3.160       62.865     **
    4      0.02137     0.356       63.221
    5     -0.01090      .            .
    6     -0.20007      .            .

  Maximum allowed number of factors = 2
  Stop when cumulative explained variance = 80%
  Minimum allowed eigenvalue = 0.50000
  Number of factors retained = 2
```

This chart lists each factor, its associated eigenvalue, the percent of total variance explained by the factor, and the total cumulative variance explained by all factors up to and including this one. A small scree plot is show on the right.

One popular rule of thumb in determining how many factors to use is to only use factors whose eigenvalues are at least 1.0. However, experience has show that this may exclude useful factors, so a smaller eigenvalue cutoff is recommended.

Table of Communalities

```
  ==============  Communalities  ==============

    Initial   Final   Common Var. %  Unique Var. %
    -------  -------  -------------  -------------
V1   0.5908   0.9285      92.848          7.152
V2   0.3250   0.2554      25.540         74.460
V3   0.5332   0.5693      56.927         43.073
V4   0.5938   0.9234      92.342          7.658
V5   0.4861   0.4449      44.488         55.512
V6   0.4326   0.4608      46.083         53.917
```

A *communality* is the percent of variance in a variable that is accounted for by the retained factors. For example, in the table above, about 93% of the variance in V1 is accounted for by the factors, while only 44% of the variance of V5 is accounted for.

Factor Loading Matrix

```
  ==============  Un-rotated Factor Matrix  ==============

           Fac1       Fac2
        ---------  ---------
    V1   0.6167 *   0.7404 *
    V2   0.3619     0.3527
    V3   0.5359     0.5311
    V4   0.6727 *  -0.6862 *
    V5   0.5724    -0.3423
    V6   0.5677    -0.3722
------  ---------  ---------
  Var.    1.901      1.681
```

The factor loading matrix shows the correlation between each variable and each factor. For example, V1 has a 0.6167 correlation with Factor 1 and a 0.7404 correlation with Factor 2. From the factor matrix shown above, we see that Factor 1 is related most closely to V4 followed by V1. V5 and V6 are also moderately significant variables on Factor 1. Factor 2 is related to V1 and V4. So when trying to interpret the meaning of Factor 2, you should try to figure out the common connection between V1 and V4.

Factor Rotation

```
==============  Rotated Factor Matrix  ==============

  Rotation method: Varimax

           Fac1       Fac2
        ---------  ---------
    V1  -0.0056     0.9636 *
    V2   0.0494     0.5030
    V3   0.0674     0.7515 *
    V4   0.9566 *  -0.0911
    V5   0.6583 *   0.1072
    V6   0.6740 *   0.0813
------  ---------  ---------
  Var.    1.901      1.681
```

There are several methods of rotating the factor matrix that make the relationship between the variables and the factors easier to understand. The factor matrix presented above is the result of rotating the factor matrix presented in the previous section. In this case Varimax rotation was used. After a Varimax rotation, some of the factor loadings will be large, and the rest will be close to zero making it easy to see which variables correlate strongly with the factor. Varimax is the most popular rotation method. After performing the Varimax rotation, it is easy to see that Factor 1 is related to variables V4, V5, and V6 whereas Factor 2 is related to variables V1, V2, and V3.

A Varimax rotation is an orthogonal transformation. That means the factor axes remain orthogonal to each other, and the factors are uncorrelated. A Promax rotation relaxes that restriction and allows the rotated axes to be oblique and correlated with each other.

When a Promax rotation is done, DTREG displays a table showing the correlations between the rotated factors:

```
====  Correlation Between Rotated Factor Axes  ====

          Fac1     Fac2
        -------  -------
  Fac1   1.0000  -0.1400
  Fac2  -0.1400   1.0000
```

Using Principal Components transformations

As discussed above, principal components are weighted, linear combinations of the variables, and the principal components are ordered in decreasing order of explained variance. It is possible to generate new variables whose values are computed using the eigenvectors. For example, a new variable, PC1, could be computed for each set of variable values using the formula:

$$PC1 = a_{11}X_1 + a_{12}X_2 + \ldots + a_{1n}X_n$$

Then this computed variable can be used in a predictive model instead of the original variables. Since the principal components (and eigenvectors) are ordered in decreasing order of explained variance, it is often possible to use fewer principal component variables than original variables. For example, the following table taken from a DTREG report shows the percent of total variance explained by each principal component and the cumulative amount explained:

```
Factor  Eigenvalue  Variance %  Cumulative %       Scree Plot
------  ----------  ----------  ------------  --------------------
    1     6.12685    47.130       47.130     ********************
    2     1.43328    11.025       58.155     ****
    3     1.24262     9.559       67.713     ****
    4     0.85758     6.597       74.310     **
    5     0.83482     6.422       80.732     **
    6     0.65741     5.057       85.789     **
    7     0.53536     4.118       89.907     *
    8     0.39610     3.047       92.954     *
    9     0.27694     2.130       95.084     *
   10     0.22024     1.694       96.778
   11     0.18601     1.431       98.209
   12     0.16930     1.302       99.511
   13     0.06351     0.489      100.000
```

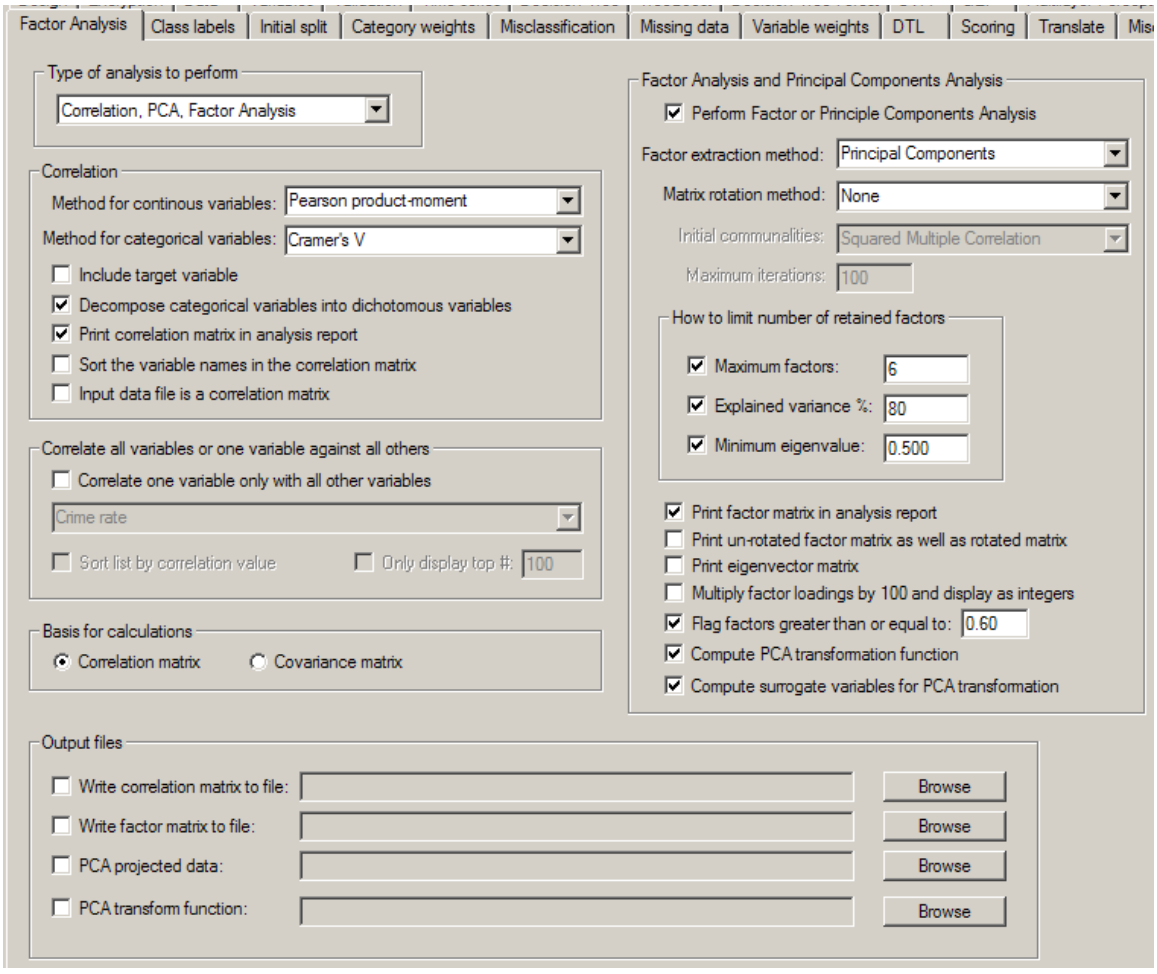
There were 13 original variables, but the cumulative effect of using only the first five principal components accounts for 80.732% of the variance.

One word of caution: principal components are formed from a linear combination of the variables. If the variables are related in a nonlinear manner, the principal components will not correctly reflect the relationship.

The Enterprise Version of DTREG contains features to (1) compute principal component transformations, (2) use the PCA transformations to convert the input data to PCA transformed values, and (3) use PCA transformation functions computed in one model to automatically generate new PCA variables in a subsequent model.

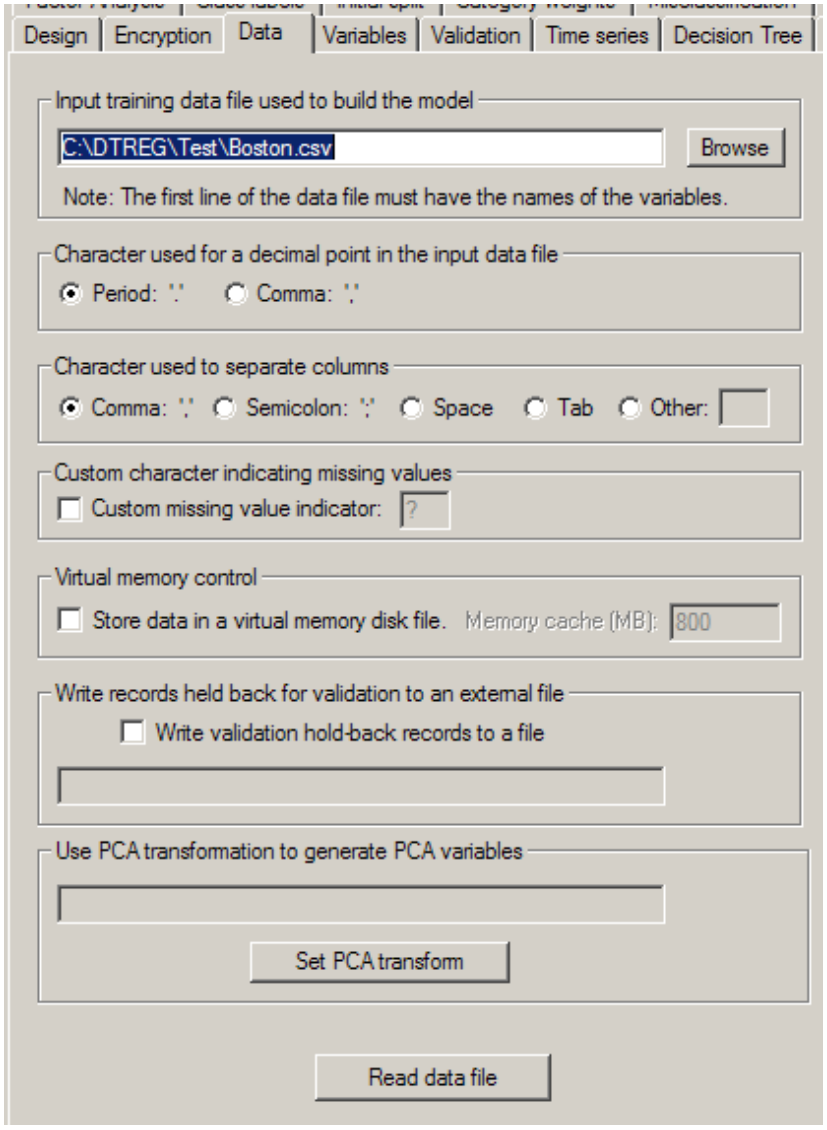
Here are the steps in computing PCA transform functions and then using them to generate PCA variables in a subsequent model.

1. Perform a PCA analysis, select the criteria to determine how many principal components will be stored, and check the option "Compute PCA transformation function" on the PCA properties page.

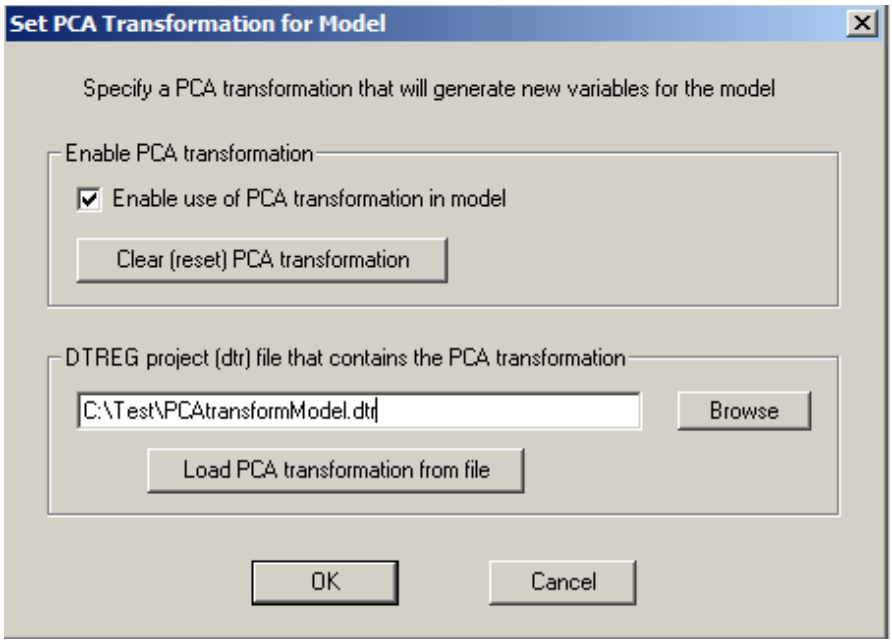


2. After the PCA analysis has been performed, save the generated model to a DTREG project file (.dtr file).
3. Open or create a new project in which you want to use the PCA transformation.

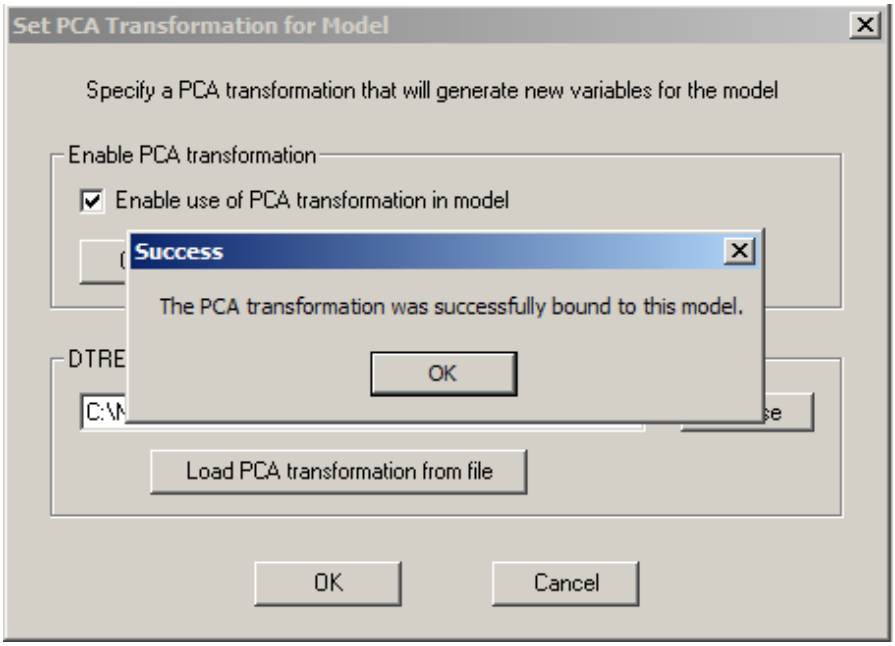
4. On the Data property page for the new model, click the button “Set PCA transform”.



5. A popup screen will appear looking like this:



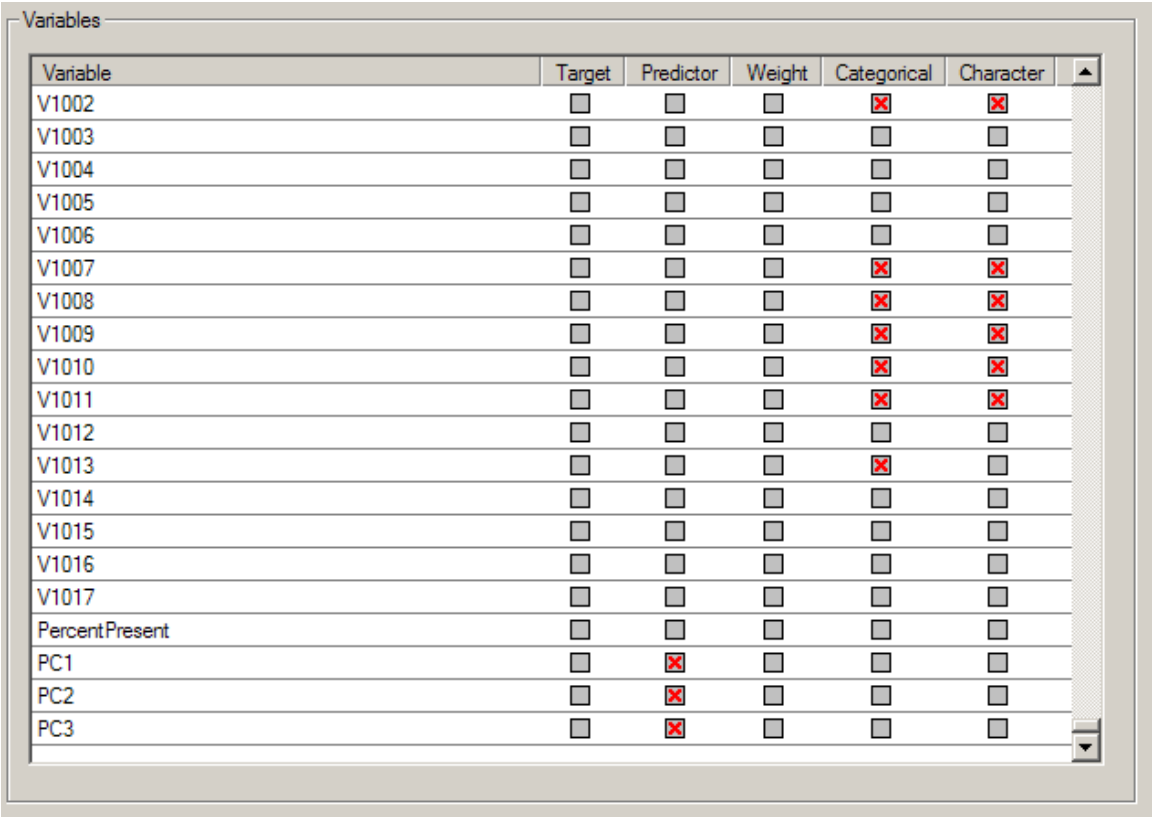
6. Check the box "Enable use of PCA transformation in model", specify the name of the DTREG project file contain the previously-computed PCA transformation, then click the "Load PCA transformation from file" button. DTREG will read the project file containing the PCA transformation function and attach the PCA transformation function to this project. DTREG will report if the PCA transformation was found in the auxiliary project and successfully attached to this project:



7. Once the transformation has been read from the auxiliary project file and bound to this model, the auxiliary project file is no longer needed. The PCA transformation function becomes part of the new project, and it will be stored with

the new project file. If surrogate variables were computed with the PCA transformation, they also will become part of the new model, and they will be used to handle missing values going into the PCA transformation.

8. After binding a PCA transformation function to the model, new variables will appear in the list of variables on the Variables Property Page with names PC*n* where *n* is the principal component number.



9. You can then use these variables as predictors in the new model. The PCA variables are also available for predicting values using the Score Function (see page 163). If you use the DTREG COM DLL component, the PCA transformations will be applied to the input data for computing predictions. If you use DTL with PCA transformations, variables created by DTL may be used as inputs to the PCA transformation function, but the PCA variables created by the transformation are not available to the DTL program.

Handling Missing Data Values

Missing data values are an unfortunate but frequent occurrence in many predictive modeling situations. For example, demographic information obtained for marketing analysis may have hundreds of variables, but not all of the information will be available (or even relevant) to some of the people. In medical studies some tests may be performed for some patients but not others.

Specifying missing values in input data

There are three ways to denote a missing value in an input data record:

1. Leave the column blank.
2. Put a single period ('.') in the column without any numbers around it.
3. Put a question mark ('?') in the column.

Types of missing variables

DTREG recognizes three types of variables: target, predictor, and weight. If the target or weight variables for a data record have missing values, the data record is unconditionally excluded from the analysis. Also, if *all* of the predictor variables have missing values, the data record is excluded. But if some predictor variables are available but others have missing values, DTREG provides four methods for handling the data records with missing values:

Exclude the data row

The simplest way to deal with records having some missing predictor variable values is to exclude those rows from the analysis. If there are many data rows available and the percentage of rows with missing values is small, then this may be the best method. Excluding rows is fast, and it prevents any error from being introduced due to the missing values.

Replace missing values with median/mode values

The second approach is to replace missing predictor values by the median value of the variable. For categorical predictors, the mode (most frequent category) is used for the replacement. Using the median/mode introduces some error into the model for that variable, but it allows the non-missing values of the other predictors to contribute to the model.

Surrogate Variables

The most sophisticated method is to use *surrogate variables* to impute the predictor values that are missing. A surrogate variable is another predictor variable that is associated (correlated) with the primary predictor variable. DTREG fits a linear or polynomial function to estimate the missing variable value based on the available value of the surrogate variable.

Before the model building process starts, DTREG examines each potential surrogate variable for each primary predictor variable and computes the association between the variables. Continuous and categorical predictor variables with two categories may have surrogates and be used as surrogates. Categorical variables with more than two categories cannot have surrogates nor can they be used as surrogates. The mode is used as the replacement value for categorical variables with more than two categories.

If there are *n* eligible variables, then $n * (n - 1)$ potential matches must be evaluated. For each potential variable pair, the *association* is calculated. The association measures how closely the variables are related. Association values range from 0 (no association) to 100 (perfect association). The surrogates with the highest association are connected to the primary predictor. So each predictor has a different set of surrogate variable functions.

The method used to compute the association depends on the type of the predictor:

Continuous predictors – Linear regression is used to fit a function:

$$predictor = f(surrogate)$$

The association is then computed as 100 times the proportion of the variance of the predictor explained by the function. So if the function output exactly matches the predictor, the association is 100.

Categorical predictors – A slightly different method is used to compute the association for categorical predictors with two categories. If the potential surrogate is also categorical, the values of the predictor and the surrogate are compared and the proportion of the values that match (have the same category) is computed; call this *MatchProportion*. Then association is computed using the formula:

$$Association = 200 * |MatchProportion - 0.5|$$

If the proportion of matching rows is 0.5, then the association is 0.0, because there is a 50/50 chance of a match. If the proportion matching is either 1.0 or -1.0 then the association is 100. A negative match proportion means that the variables are associated in the opposite direction. A match proportion of (-1.0) means that the category values are exactly opposite; hence, the predictor value can be imputed by reversing the category value of the surrogate. If the primary predictor is categorical and the surrogate is

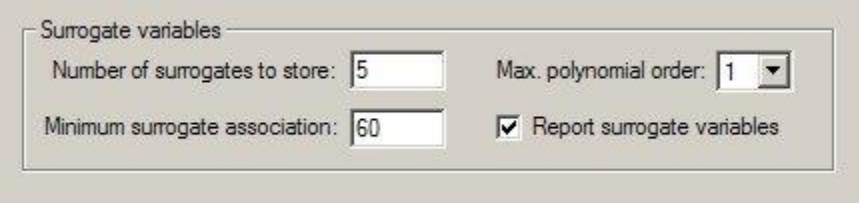
continuous, a function is fitted to the 0/1 predictor values and a threshold of 0.5 is used to convert the value computed by the function to the predictor category value.

When a predictor variable is encountered with a missing value, DTREG examines each of the associated surrogate variables looking for one that has a non-missing value on that data row. The surrogates are examined in the order of decreasing association values. When a surrogate variable is found with a non-missing value, the surrogate function is used to compute the replacement value for the variable. If all surrogates have missing values, the median/mode is used replace the missing value.

Surrogate variables are almost always the best method for handling missing values. However, there are situations where surrogate variables may improve the accuracy of the model on the training data but produce inferior results on the validation results compared with using median/mode values. So it is recommended that you build models using both surrogate variables and median/mode values and compare the validation results.

Surrogate variables are used (1) during the model building process, (2) when using the Score function (page 163) to predict values for a data file, and (3) when the DTREG COM library is used to predict values (page 375). If the Translate procedure (page 169) is used to generate source code for a model, surrogate variable calculations are included in the generated source code.

Parameters related to selecting surrogate variables are specified on the Variables property page which is described on page 41. Here is the surrogate variable portion from that page:



The following parameters can be specified:

Number of surrogates to store – This is the maximum number of surrogate variables that DTREG will store for each predictor variable. Fewer surrogates may be stored if no significant associations are found.

Minimum surrogate association – The association computed for each potential surrogate is compared to this value. If the association is smaller than this, then the surrogate is excluded.

Maximum polynomial order – This controls whether linear, quadratic, or cubic functions are used for surrogate associations. If a polynomial order greater than 1 is specified, DTREG computes the association for all polynomials up to that order, and it

only uses the higher order polynomials if they provide superior fit (greater association) than lower-order polynomials.

Report surrogate variables – If this option is checked, then DTREG adds a table to the analysis report showing which surrogate variables were stored for each predictor along with the polynomial coefficients and the association. See page 182 for an example of the surrogate variable report.

Surrogate Splitters

A *surrogate splitter* is similar to a surrogate variable, but it is specialized for decision tree based models – single trees, TreeBoost, and decision tree forests.

When a decision tree is created, each predictor variable is evaluated at each split point to determine how well it can partition the values. After the best predictor has been determined, other candidate predictors are examined and the splits generated by them are compared with the primary split. The association is computed by comparing the split generated by the predictor with the primary predictor. The best surrogate splitters are stored along with the primary splitter. If the primary splitter value is missing, surrogate splits are examined looking for a non-missing value on a surrogate predictor.

One of the key differences between surrogate variables and surrogate splitters is that a different set of surrogate splitters is stored for *each split*. So the same predictor may have different surrogate splitter variables at different spit points in the decision tree. In contrast, surrogate variables are computed once before the model building process begins, and the same set of surrogate variables is always used for a particular predictor variable.

How Trees are Built and Pruned

Train up a tree in the way it should go, and when you are old sit under the shade of it.
– Charles Dickens

The process DTREG uses to build and prune a tree is complex and computationally intensive. Here is an outline of the steps:

1) Build the tree
    a) Examine each node and find the best possible split
        i) Examine each predictor variable
            (1) Examine each possible split on each predictor
    b) Create two child nodes
    c) Determine which child node each row goes into. This may involve using surrogate splitters.
    d) Continue the process until a stopping criterion (e.g., minimum node size) is reached.
2) Prune the tree
    a) Build a set of cross-validation trees
    b) Compute the cross validated misclassification cost for each possible tree size
    c) Prune the primary tree to the optimal size

Building Trees

The process used to split a node is the same whether the node is the root node with all of the rows or a child node many levels deep in the tree. The only difference is the set of rows in the node being split.

Splitting Nodes

DTREG tries each predictor variable to see how well it can divide the node into two groups.

If the predictor is continuous, a trial split is made between each discrete value (category) of the variable. For example, if the predictor being evaluated is Age and there are 80 values of Age ranging from 10 to 79, then DTREG makes a trial split putting the rows with a value of 10 for Age in the left node and the rows with values from 11 to 79 in the right node. The improvement gained from the potential split is remembered, and then the next trial split is done putting rows with Age values of 10 and 11 in the left group and values from 12 to 79 in the right group. The number of splits evaluated is equal to the number of discrete values of the predictor variable less one.

You can control the maximum number of discrete values used for continuous variables by setting the value of "Max. categories for predictor variables" on the Design property screen (see page 33). If there are more actual discrete values than this parameter setting, values are grouped together into value ranges.

This process is repeated by moving the split point across all possible division points. The best improvement found from any split point is saved as the best possible split for that predictor variable in this node. The process is then repeated for each other predictor variable. The best split found for any predictor variable is used to perform the actual split on the node. The next best five splits are saved as "competitor splits" for the node.

When examining the possible splits for a categorical predictor variable, the calculations are more complex and potentially much more time consuming.

If the predictor variable is categorical and the target variable is continuous, the categories of the predictor variable are sorted so that the mean value of the target variable for the rows having each category of the predictor are increasing. For example, if the target variable is "Income" and the predictor variable has three categories, *single*, *married* and *divorced*, the categories are ordered so that the mean value of Income for the people in each predictor category is increasing. The splitting process then tries each split point between each category of the predictor. This is very similar to the process used for continuous predictor variables except the categories are arranged by values of the target variable rather than by values of the predictor variable. The number of splits evaluated is equal to the number of categories of the predictor variable less one.

If both the target variable and the predictor variable are categorical, the process gets more complex. In this case, to perform an exhaustive search DTREG must evaluate a potential split for every possible combination of categories of the predictor variable. The number of splits is equal to $2^{(k-1)}-1$ where k is the number of categories of the predictor variable. For example, if there are 5 categories, 15 splits are tried; if there are 10 categories, 511 splits are tried; if there are 16 categories, 32,767 splits are tried; if there are 32 categories, 2,147,483,647 splits are tried. Because of this exponential growth, the computation time to do an exhaustive search becomes prohibitive when there are more than about 12 predictor categories. In this case, DTREG uses the clustering technique described below to group the target categories.

There is one case where classification trees are efficient to build using exhaustive search even with categorical predictors having a large number of categories. That is the case where the target variable has only two possible categories. Fortunately, this situation occurs fairly often – the target categories might be live/die, bought-product/did-not-buy, malignant/benign, etc. For this situation, DTREG has to evaluate only many splits as the number of categories for the predictor variable less one.

In order to make it feasible to construct classification trees with target variables that have more than two categories and predictor variables that have a large number of categories, DTREG switches from using an exhaustive search to a cluster analysis method when the

number of predictor categories exceeds a threshold that you can specify on the Model Design property page (see page 33). This technique uses cluster analysis to group the categories of the target variable into two groups. DTREG is then able to try only (*k*-1) splits, where *k* is the number of predictor categories.

Once DTREG has evaluated each possible split for each possible predictor variable, a node is split using the best split found. The runner-up splits are remembered and displayed as "Competitor Splits" in the report.

Evaluating Splits

The ideal split would divide a group into two child groups in such a way so that all of the rows in the left child have the same value on the target variable and all of the rows in the right group have the same target value – but different from the left group. If such a split can be found, then you can exactly and perfectly classify all of the rows by using just that split, and no further splits are necessary or useful. Such a perfect split is possible only if the rows in the node being split have only two possible values on the target variable.

Unfortunately, perfect splits do not occur often, so it is necessary to evaluate and compare the quality of imperfect splits. Various criteria have been proposed for evaluating splits, but they all have the same basic goal which is to favor homogeneity within each child node and heterogeneity between the child nodes. The heterogeneity – or dispersion – of target categories within a node is called the "node impurity". The goal of splitting is to produce child nodes with minimum impurity.

The impurity of every node is calculated by examining the distribution of categories of the target variable for the rows in the group. A "pure" node, where all rows have the same value of the target variable, has an impurity value of 0 (zero). When a potential split is evaluated, the probability-weighted average of the impurities of the two child nodes is subtracted from the impurity of the parent node. This reduction in impurity is called the *improvement* of the split. The split with the greatest improvement is the one used. Improvement values for splits are shown in the node information that is part of the report generated by DTREG.

DTREG provides two methods for evaluating the quality of splits when building classification trees, (1) Gini and (2) Entropy,. Only one method is provided when building regression trees, and that is minimum variance within nodes. The minimum variance/least squares criteria is essential the same criteria used by traditional, numeric regression analysis (i.e., line and function fitting).

Experience has shown that the splitting criterion is not very important, and Gini and Entropy yield trees that are very similar. Gini is considered slightly better than Entropy, so it is the default criteria used for classification trees. See Breiman, Friedman, Olshen and Stone *Classification And Regression Trees* (1984) for a technical description of the Gini and Entropy criteria.

Assigning Categories to Nodes

When a decision tree is used to predict values of the target variable, rows are run through the tree down to the point where they reach a terminal node. The category assigned to the terminal node is the predicted value for the row being evaluated. So a natural question is how categories are assigned to nodes.

For regression trees built with a continuous target variable, the value assigned to a node is simply the average value of the target variable for all rows that end up in the node weighted by the row weights.

For classification trees built with a categorical target variable, the determination of what category to assign to a node is more complex: it is the category that minimizes the *misclassification cost* for the rows in the node. The calculation of the misclassification cost is somewhat complex. The formula involves the distribution of target categories in the node compared with the distribution in the total (learning) sample. The category weights and the misclassification costs also affect the assigned category. In the simplest case, every row that is misclassified has a cost of 1 and every row that is correctly classified has a cost of 0, so the category with the most rows in the node is assigned to the node. The misclassification cost for every node is displayed in the report generated by DTREG. A misclassification summary table is included near the end of the report.

If you wish, you can specify specific costs for misclassifying one target category as another target category. For example, you might want to assign a greater cost to classifying a heart attack as indigestion than classifying indigestion as a heart attack. These misclassification costs are implemented by generating *altered prior* (category weight) values that are used in the calculation. See Breiman, Friedman, et al (1984) for a detailed description of how misclassification costs are used.

Missing Values and Surrogate Splitters

Ideally, every row would have values for every variable. Unfortunately, in the real world, missing values are encountered often: People being surveyed refuse or forget to answer questions, some questions may not apply to all people, some medical tests may not be performed on all patients, etc.

Some simple programs discard rows that have any missing values. But this is a waste of valuable information that may be available on other variables.

DTREG uses a sophisticated technique involving *surrogate splitters* to estimate the values of predictor variables with missing values.

Surrogate splitters are predictor variables that are not as good at splitting a group as the primary splitter but which yield similar splitting results; they mimic the splits produced by the primary splitter.

DTREG compares which rows are sent to the left and right child groups by the primary splitter with the rows sent to the corresponding child groups by every other predictor variable. The *association* between the primary splitter and each alternate predictor is computed as a function of how closely the alternate predictor matches the primary splitter. (This roughly corresponds to a count of how many rows each predictor sends left and right, but the actual calculation is more complex.) The alternate predictor variables are then ranked in decreasing order of association.

The largest possible association value is 1.0 which means the surrogate sends exactly the same set of rows to the left and right groups as the primary splitter. An association value of 0.0 means that the surrogate does no better at assigning rows than simply putting them in the most probable group.

Surrogate splitters are similar to competitor splitters in the sense that they both yield splits of benefit but are not as good as the primary splitter. Often, the same variable will be listed as both a competitor and a surrogate. However, there is a significant difference between the way variables are ranked as competitors and as surrogates. Competitor splits are runners-up to the primary split: they are judged the same way the primary splitter is judged by how much improvement they make in reducing node impurity. Surrogate splitters are not ranked by the amount of improvement they produce but rather by how closely they mimic the split selected for the primary splitter. The optimal split point for a surrogate maximizes the association between the surrogate and the primary splitter; it does not necessarily maximize the improvement. If you compare entries for the same variable in the competitor and surrogate lists, you may see different split points selected and different values for the improvement from the splits.

Surrogate splitters are used to classify rows that have missing values in the primary splitter. They function both when the tree is being built and later when the tree is used to score additional datasets.

When a row is encountered that has a missing value on the primary splitter, DTREG searches the list of surrogate splitters and uses the one with the highest association to the primary splitter that has a non-missing value for the row.

Surrogate splitters provide the most accurate classification of rows with missing values. This is the default and recommended method for handling missing predictor values.

In addition to their function in classifying rows with missing predictor values, the association between the primary splitter and surrogate splitters is used in the calculation of the overall importance of variables. To understand why this is done, consider two variables that are very similar and highly correlated, for example height and weight. At some split point, weight may be selected as the primary splitter because it is slightly better than height. If this preference for weight prevails at many split points, weight would appear to be extremely important and height as unimportant. However, if you removed weight as a predictor variable and reran the analysis, an identical tree very well might be built using height as the splitting variable wherever weight was used before.

Hence, height is nearly as important as weight. When one variable hides the importance of another variable, it is known as *masking*. By considering not only which variables are used as primary splitters but also the association of the surrogates, DTREG is able to provide a more accurate evaluation of variable importance.

Stopping Criteria

If no limits were placed on the size of a tree, DTREG theoretically might build a tree so large that every row of the learning dataset ended up in its own terminal node. But doing this would be computationally expensive, and the tree would be so large that it would be difficult or impossible to interpret and display.

Several criteria are used to limit how large a tree DTREG constructs. Once a tree is built, the pruning method described in a following section is used to reduce its size to the optimal number of nodes.

The following criteria are used to limit the size of a tree as it is build:

- **Minimum size node to split**. On the Design property page, you can specify that nodes containing fewer than a specified number of rows are not to be split.
- **Maximum tree depth**. On the Design property page, you can specify the maximum number of levels in the tree that are to be constructed.

Pruning Trees

Every branch of mine that bears no fruit, he takes away, and every branch that does bear fruit he prunes, that it may bear more fruit.
– Jesus (John 15:2)

One of the classic problems in building decision trees is the question of how large a tree to build. Early programs such as AID (Automatic Interaction Detection) used stopping criteria such as those described in a preceding section along with other criteria such as the improvement from splits to decide when to stop. This is known as *forward pruning*. But analysis of trees generated by these programs showed that they often were not of the optimal size.

DTREG does not use its stopping criteria as the primary means for deciding how large a tree should be. Instead, it uses relaxed stopping criteria and builds an overly-large tree. It then analyzes the tree and prunes it back to the optimal size. This is known as *backward pruning*. Backward pruning requires significantly more calculations than forward pruning, but the optimal tree sizes are much more accurately calculated. See page 209 for information about displaying a chart showing error rate versus model size.

Why Tree Size Is Important

There are two reasons why it is desirable to generate trees of the optimal size.

First, if a situation can be described and explained equally well by two descriptions, the description that is simpler and more concise is generally preferred. The same is true with decision trees: if two trees provide equivalent predictive accuracy, the simpler tree is preferred because it is easier to understand and faster to use for making predictions.

Second, and more importantly, ***smaller trees may provide greater predictive accuracy for unseen data than larger trees***. This is a non-intuitive fact that warrants explanation.

When creating a decision tree, a *learning dataset* is used. This dataset contains a set of rows that are a representative sample of the overall population. The process used to build the decision tree selects optimal splits to fit the tree to the learning dataset. Once the tree has been built, the records in the learning dataset can be run through the tree to see how well the tree fits the data. The rate of classification errors measured when running the learning dataset through a tree built using that dataset is known as the "*resubstitution cost*" for the tree. (It is called *re*substitution because the same data is *re*run through the tree.)

For the *learning dataset*, the accuracy of the fit always improves (resubstitution cost decreases) as the tree is grown larger. It is always possible to grow a sufficiently large tree to provide 100% accuracy in predicting the learning dataset. In an extreme case, the tree might be grown so large that every row of the learning dataset ended up in its own terminal node. Obviously, with such a tree, an exactly correct value of the target value for every row could be predicted.

However, it is desirable that a decision tree not only accurately model the learning dataset from which it was built, but also that it be able to predict the values of other cases that are presented to it later after it has been constructed. The ability to predict values for independent datasets is known as *generalization*.

While a large tree may fit the learning dataset with extreme accuracy, its size may reduce its generalization accuracy. As an analogy, consider fitting a suit of clothes. Manufactured clothes sold in stores are made to fit various sizes, but they are designed so that there is some slack and leeway around a specified size. In contrast, a custom tailored suit is made precisely to fit a specific individual. While the custom tailored suit will fit one person extremely well, it will not fit other people in the same size range as well as a generic suit. In the same way, adding extra nodes to a tree to "custom tailor" it to the learning dataset may introduce misclassifications when it is later applied to a different dataset.

Another way to understand why large trees can be inferior to smaller trees is that the large trees fit and model minor "noise" in the data, whereas smaller trees model only the significant data factors.

See page 209 for information about generating a chart showing misclassification error rate versus model size.

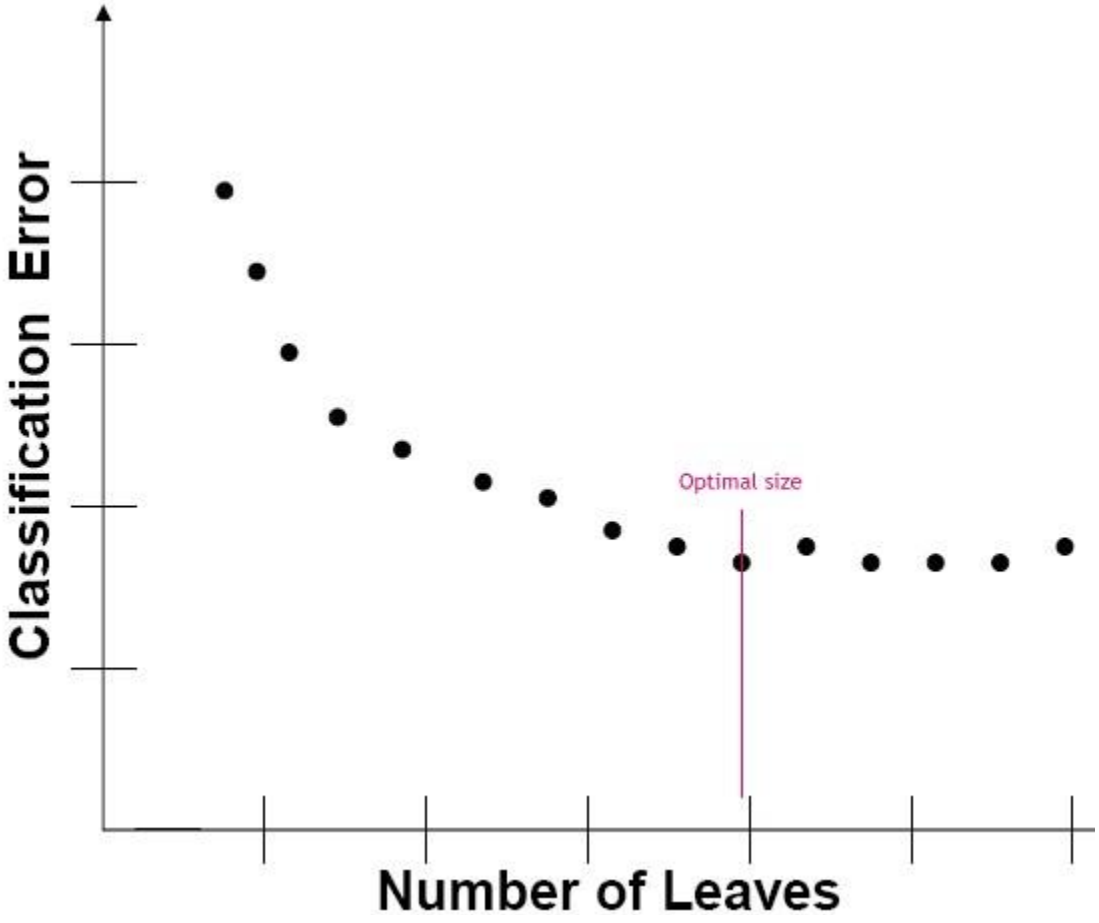
The primary goal of the pruning process is to generate the optimal size tree that can be generalized to other data beyond the learning dataset.

V-Fold Cross Validation

You're dealing with the demon of external validation. You can't beat external validation. You want to know why? Because it feels soooo good!
– Barbara Hall, *Northern Exposure*

The method used by DTREG to determine the optimal tree size is *V-fold cross validation*. Research has shown that this method is highly accurate, and it has the advantage of not requiring a separate, independent dataset for assessing the accuracy and size of the tree.

If a tree is built using a specific learning dataset, and then independent test datasets are run through the tree, the classification error rate for the test data will decrease as the tree increases in size until it reaches a minimum at some specific size. It the tree is grown beyond that point, the classification errors will either remain constant or increase. A graph showing how classification errors typically vary with tree size is shown below:



In order to perform tests to measure classification error as a function of tree size, it is necessary to have test data samples independent of the learning dataset that was used to build the tree. However, independent test data frequently is difficult or expensive to obtain, and it is undesirable to hold back data from the learning dataset to use for a separate test because that weakens the learning dataset. *V-fold cross validation* is a

technique for performing independent tree size tests without requiring separate test datasets and without reducing the data used to build the tree.

Cross validation would seem to be paradoxical: we need independent data that was not used to build the tree to measure the generalized classification error, but we want to use all data to build the tree. Here is how cross validation avoids this paradox.

All of the rows in the learning dataset are used to build the tree. This tree is intentionally allowed to grow larger than is likely to be optimal. This is called the *reference*, unpruned tree. The reference tree is the best tree that fits the learning dataset.

Next, the learning dataset is partitioned into some number of groups called "folds". The partitioning is done using stratification methods so that the distribution of categories of the target variable are approximately the same in the partitioned groups. The number of groups that the rows are partitioned into is the 'V' in "V-fold cross classification". Research has shown that little is gained by using more than 10 partitions, so 10 is the recommended and default number of partitions in DTREG.

For the point of discussion, let's assume 10 partitions are created. DTREG then collects the rows in 9 of the partitions into a new pseudo-learning dataset. A test tree is built using this pseudo-learning dataset. The quality of the test tree for fitting the full learning dataset will, in general, be inferior to the reference tree because only 90% of the data was used to build it. Since the 10% (1 out of 10 partitions) of the data that was held back from the test tree build is independent of the test tree, it can be used as an independent test sample for the test tree.

The 10% of the data that was held back when the test tree was built is run through the test tree and the classification error for that data is computed. This error rate is stored as the independent test error rate for the first test tree.

A different set of 9 partitions is now collected into a new pseudo-learning dataset. The partition being held back this time is selected so that it is different than the partition held back for the first test tree. A second test tree is built and its classification error is computed using the data that was held back when it was built.

This process is repeated 10 times, building 10 separate test trees. In each case, 90% of the data is used to build a test tree and 10% is held back for independent testing. A different 10% is held back for each test tree.

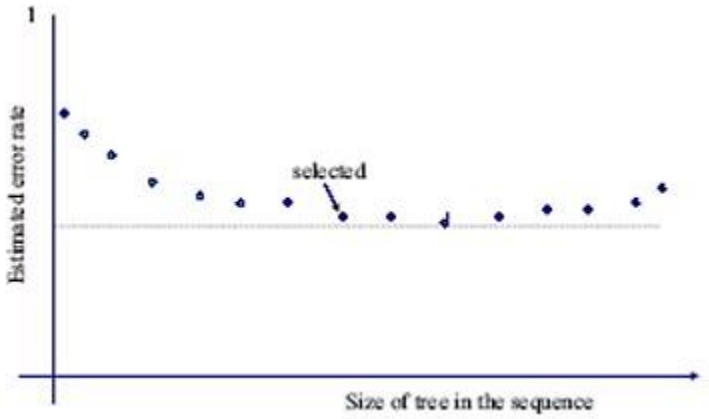
Once the 10 test trees have been built, their classification error rate as a function of tree size is averaged. This averaged error rate for a particular tree size is known as the "*Cross Validation cost*" (or "*CV cost"*). The cross validation cost for each size of the test trees is computed. The tree size that produces the minimum cross validation cost is found. This size is labeled as "*Minimum CV*" in the tree size report DTREG generates. See page 183 for an example of a tree size report with cross validation statistics.

The reference tree is then pruned to the number of nodes matching the size that produces the minimum cross validation cost. The pruning is done in a stepwise fashion, removing the least important nodes during each pruning cycle. The decision as to which node is the "least important" is based on the cost complexity measure as described in *Classification And Regression Trees* by Breiman, Friedman, Olshen and Stone (1984).

It is important to note that the test trees built during the cross-validation process are used only to find the optimal tree size. Their structure (which may be different in each test tree) has no bearing on the structure of the reference tree which is constructed using the full learning dataset. The reference tree pruned back to the optimal size determined by cross validation is the best tree to use for scoring future datasets.

Adjusting the Optimal Tree Size

If you plot the cross-validation error cost for a tree versus tree size, the error cost will drop to a minimum point at some tree size, then it will rise as the tree size is increased beyond that point. Often, the error cost will bounce up and down in the vicinity of the minimum point, and there will be a range of tree sizes that produce approximately the same low error cost. A graph illustrating this is shown below:



Note that the absolutely smallest misclassification cost is only slightly smaller than the misclassification cost for a tree that is several nodes smaller. Since smaller and simpler trees are preferred over larger trees that have the same predictive accuracy, you may prefer to prune back to the smaller tree if the increase in misclassification cost is minimal. The cross validation cost for each possible tree size is displayed in the Tree Size report that DTREG generates. See page 183 for an example.

On the "Validation" property page for the model, DTREG provides several options for controlling the size that is used for pruning:

- **Prune to the minimum cross-validated error** – If you select this option, DTREG will prune the tree to the size the produces the absolutely minimum cross-validated classification error.
- **Allow 1 standard error from minimum** – Many researchers believe that it is acceptable to prune to a smaller tree as long as the increase in misclassification cost does not exceed one standard error of the variance in the cross validation misclassification cost. The standard error for the cross validation cost values is displayed in the Tree Size report. See page 183 for an example.
- **Allow this many S.E. from the minimum** – Using this option, you can specify an exact number of standard errors from the minimum misclassification cost you will allow.

Example Analyses

The DTREG installation program installs a set of example projects in a folder named "Examples" under the DTREG installation directory. Normally, this is C:\Program files\DTREG\Examples. A good way to get started using DTREG is to browse the examples in that directory and run some of them.

Most of the example analyses came from the UCI Repository of Machine Learning Databases (http://www.ics.uci.edu/~mlearn/MLRepository.html). Irvine, CA: University of California, Department of Information and Computer Science. This repository has greatly benefited the development of many decision tree and machine learning programs.

Summary information about some of the examples is presented below. Other information can be found in the "Notes" section displayed on the Design property page within DTREG.

TITANIC.DTR – The sinking of the Titanic is a famous event, and new books are still being published about it. Many well-known facts - from the proportions of first-class passengers to the "women and children first" policy, and the fact that that policy was not entirely successful in saving the women and children in the third class - are reflected in the survival rates for various classes of passenger. These data were originally collected by the British Board of Trade in their investigation of the sinking. For each person on board the fatal maiden voyage of the ocean liner Titanic, this dataset records sex, age (adult/child), booking class (first/second/third class, or crew) and whether or not that person survived.

IRIS.DTR – This is a classification problem dating back to 1936. Its originator, R. A. Fisher, developed the problem to test clustering analysis and other types of classification programs prior to the development of computerized decision tree generation programs. The dataset is small consisting of 150 records. The target variable is categorical specifying the species of iris. The predictor variables are measurements of plant dimensions.

BOSTON.DTR – This is a regression tree example to predict the value of houses in various areas around Boston based on characteristics of the locale such as proximity to the Charles River and major highways, socioeconomic status, air pollution and other factors.

LIVERDISORDER.DTR – This is a dataset from England that generates a classification tree to predict liver disorders. The target variable is liver condition (healthy or abnormal). The predictor variables are various blood chemical measurements along with the number of alcoholic drinks consumed per day.

HOUSEVOTES.DTR – This is a classification problem that attempts to predict the political party affiliation of U.S. House members based on how they voted on various

bills in 1984. The target variable is political party (Republican/Democrat). The predictor variables are Yes/No votes cast on various bills.

LANDINGCONTROL.DTR – This is a classification problem to decide whether it is better to use manual or automatic (autopilot) control when landing the space shuttle. The target variable has two categories, Automatic and Manual. The predictor variables include wind direction, velocity and visibility.

BRIDGES.DTR – This is a classification problem that attempts to classify the type of various bridges around Pittsburg based on predictors such as their length, type of material and date of construction.

HORSECOLIC.DTR – This is a classification problem to decide if horses suffering from colic need to be treated surgically. The target variable categories are surgical or non-surgical. The predictor variables describe the horse's condition such as age, temperature, degree of discomfort, etc.

CLEVELANDHEART14.DTR – This is a classification problem that attempts to predict heart disease due to vessel narrowing. The target variable, 'num', is the number of vessels showing narrowing. The focus is on predicting a value of 0 (no disease) versus non-disease which indicates narrowing in some vessels.

DTREG .NET Class Library

The optional DTREG .NET class library makes it easy for production applications to call DTREG as an "engine" to compute the predicted value for data records using a predictive model created by DTREG.

Any type of model (Single Tree, TreeBoost or Decision Tree Forest, SVM, etc.) can be used with the DTREG COM library to generate predicted values

Because of the standardization of the .NET interface, it is easy to call DTREG functions from programs written in C#, VB.NET and other .NET languages.

Example C# program

Here is an example of a complete C# program:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DTREGclassLibrary;

namespace TestDTREGclassLibrary
{
    /*-----------------------------------------------------------------------------------------
     *  Class to build a DTREG model.
     */
    class BuildModel
    {
        /*-----------------------------------
         * Routine that trains a DTREG model.
         */
        public int BuildTheModel()
        {

            int intStatus;
            /*
             *  Establish a reference to a DTREGclass object.
             */
            DTREGclass objDtreg = new DTREGclass();
            m_objDtreg = objDtreg;
            /*
             *  Enter our registration information.
             */
            intStatus = objDtreg.SetRegistration("registered name", "registration key");
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Initialize for a new model.
             */
            intStatus = objDtreg.BeginTraining("Test model training");
            if (CheckStatus(intStatus)) return(intStatus);
            /*
             *  Set the type of model to build.
             *  1 = Single decision tree.
             *  2 = TreeBoost.
             *  3 = Decision tree forest.
             *  4 = Logistic regression.
             *  5 = SVM
             *  7 = LDA
             *  9 = Neural network
             *  10 = PNN/GRNN
             *  11 = RBF
             *  12 = Cascase correlation
             *  13 = GEP
             *  14 = Linear regression
             *  15 = K-Means
```

```
             *  16 = GMDH
             *  17 = Correlation, factor analysis.
             */
            intStatus = objDtreg.SetModelType(2);
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Define 5 variables.  The first variable is the categorical target variable.
             *  The other 4 variables are continuous predictor variables.
             */
            intStatus = objDtreg.BeginVariableDefinitions();
            if (CheckStatus(intStatus)) return (intStatus);
            intStatus = objDtreg.DefineVariable("Species", 2, true);
            if (CheckStatus(intStatus)) return (intStatus);
            intStatus = objDtreg.DefineVariable("Sepal length", 1, false);
            if (CheckStatus(intStatus)) return (intStatus);
            intStatus = objDtreg.DefineVariable("Sepal width", 1, false);
            if (CheckStatus(intStatus)) return (intStatus);
            intStatus = objDtreg.DefineVariable("Petal length", 1, false);
            if (CheckStatus(intStatus)) return (intStatus);
            intStatus = objDtreg.DefineVariable("Petal width", 1, false);
            if (CheckStatus(intStatus)) return (intStatus);
            intStatus = objDtreg.EndVariableDefinitions();
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Feed in data records.
             *  The column delimiter character is ","
             */
            intStatus = objDtreg.BeginStoringData(",", 0);
            if (CheckStatus(intStatus)) return (intStatus);
            foreach (var item in DataRow)
            {
                intStatus = objDtreg.StoreDataRow(item);
                if (CheckStatus(intStatus)) return (intStatus);
            }
            intStatus = objDtreg.EndOfData();
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Train the model.  The type of the model was set by SetModelType().
             */
            intStatus = objDtreg.GenerateModel();
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Get analysis reports from the model, and write them to files.
             *  --- Change the folder for the files ---
             */
            String txtReport = objDtreg.GetAnalysisReport();
            String txtXMLReport = objDtreg.GetAnalysisReportXML();
            System.IO.File.WriteAllText(@"C:\Test\AnalysisReport.txt", txtReport);
            System.IO.File.WriteAllText(@"C:\Test\AnalysisReport.xml", txtXMLReport);
            /*
             *  Write the project to a file.
             */
            intStatus = objDtreg.SaveProject(@"C:\Test\Model.dtr");
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Finished
             */

            return(0);
        }

        /*-------------------------------------------------------------------------------
         *  Check a status code and display a message box if there is an error.
         *  Return true if there is an error or false for success.
         */
        bool CheckStatus(int intStatus)
        {
            if (intStatus != 0) {
                MessageBox.Show(m_objDtreg.StatusMessage(intStatus), "Error");
                return(true);
            } else {
                return(false);
            }
        }
/*
 *  Data rows for model training.
 */
string []DataRow = {
  "Setosa,5.1,3.5,1.4,0.2",
  [… More data rows …]
```

```
  };
}

}
```

Licensing and Use of DTREG

Use and Distribution of DTREG

There are two versions of the DTREG program: demonstration and registered. You are welcome to make copies of the demonstration version of DTREG and pass them on to friends or post this program on bulletin boards or distribute it via disk catalog services, CD ROMS, or other means provided the entire DTREG distribution is included in its original, unmodified form. A distribution fee may be charged for the cost of the diskette, shipping and handling. Vendors are encouraged to contact the author to get the most recent version of DTREG.

As a demonstration product, you are granted a no-cost, trial period of 30 days during which you may evaluate DTREG. If you find DTREG to be useful, educational, and/or entertaining, and continue to use it beyond the 30 day trial period, you are required to compensate the author by purchasing it.

In return for purchasing DTREG, you will be authorized to continue using DTREG beyond the trial period on a single computer. Contact the author for information about multi-system licenses.

The registered version of DTREG may *not* be redistributed or used on more than one computer system.

Copyright Notice

Web page

Up-to-date information about DTREG can be found on the web page: http://www.dtreg.com

Contacting the author

Phil Sherrod, the author of DTREG, can be contacted at PhilSherrod@comcast.net

Disclaimer

This software and documentation are provided on an "as is" basis. This program may contain "bugs" and inaccuracies, and its results should not be assumed to be correct unless they are verified by independent means. Phillip H. Sherrod disclaims all warranties

relating to this software, whether expressed or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose. Neither Phillip H. Sherrod nor anyone else who has been involved in the creation, production, or delivery of this software shall be liable for any indirect, consequential, or incidental damages arising out of the use or inability to use such software, even if Phillip H. Sherrod has been advised of the possibility of such damages or claims. The person using the software bears all risk as to the quality and performance of the software.

This agreement shall be governed by the laws of the State of Tennessee and shall inure to the benefit of Phillip H. Sherrod and any successors, administrators, heirs and assigns. Any action or proceeding brought by either party against the other arising out of or related to this agreement shall be brought only in a state or federal court of competent jurisdiction located in Williamson County, Tennessee. The parties hereby consent to in personam jurisdiction of said courts.

References

Agresti, Alan. *Categorical Data Analysis, Second Edition*. Wiley series in probability and statistics, 2002.

Aldenderfer, Mark S. and Roger K. Blashfield. *Cluster Analysis*. Sage Publications, 1984.

Allison, Paul D. *Logistic Regression Using The SAS System: Theory and Application*. SAS Institute Inc., Cary, NC, 1999.

Balakrishnama, S. and A. Ganapathiraju, *Linear Discriminant Analysis – A Brief Tutoria*l, Institute for Signal and Information Processing, Mississippi State University.

Berk, Richard A. (2003) "An Introduction to Ensemble Methods for Data Analysis" *UCLA Department of Statistics Technical Report*.

Bishop, Christopher M. (2005) *Neural Networks for Pattern Recognition*. Oxford University Press.

Blake, C.L. & Merz, C.J. (1998). *UCI Repository of Machine Learning Databases* [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

Blok, Hendrick J. *On the nature of the stock market: Simulations and experiments*. PhD thesis, University of British Columbia, 2000. (http://www.zoology.ubc.ca/~rikblok/lib/blok00b.html)

Breiman, Leo, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Pacific Grove: Wadsworth, 1984.

Breiman, Leo (1996) "Bagging Predictors." *Machine Learning* 26:123-140.

Breiman, Leo (2001). "Decision Tree Forests." *Machine Learning* 45 (1):5-32, October 2001.

Campbell, C. *An Introduction to Kernel Methods*.

Caruana, Rich and Alexandru Niculescu-Mizil. *An Empirical Comparison of Supervised Learning Algorithms Using Different Performance Metrics*. Computer Science, Cornell University, Ithaca NY 14850.

Cattell, Raymond B. *Factor Analysis – An introduction and Manual for the Psychologist and Social Scientist*. Harper & Brothers, 1952.

Chang, Chih-Chung and Chih-Jen Lin. *LIBSVM – A Library for Support Vector Machines*. April, 2005. http://www.csie.ntu.edu.tw/~cjlin/libsvm/

Chen, C.H. (1996) *Fuzzy Logic and Neural Network Handbook*. McGraw-Hill.

Chen, Chao, Andy Liaw, Leo Breiman, *Using Random Forest to Learn Imbalanced Data*.

Chen, Sheng, Xia Hong and Chris J. Harris, "Orthogonal Forward Selection for Constructing the Radial Basis Function Network with Tunable Nodes", 2005.

Chen, Sheng, Xunxian Weng and Chris J. Harris: "Experiments with Repeating Weighted Boosting Search for Optimization in Signal Processing Applications". IEEE *Transactions on Systems, Man and Cybernetics – Part, B Cybernetics*, Vol. 35, No. 4, August 2005.

Cristianini, Nello and John Shawe-Taylor: *An Introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

Efron, Bradley and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1998.

Fahlman, Scott E. and Christian Libiere (1990) *The Cascade-Correlation Learning Architecture*. Carnegie Mellon University. CMU-CS-90-100

Fawcett, Tom. *ROC Graphs: Notes and Practical Considerations for Data Mining Researchers*. March 16, 2004.

Ferreira, Cândida. *Gene Expression Programming. Mathematical Modeling by an Artificial Intelligence, 2nd Edition*. Springer-Verlag--Studies in computational intelligence 21, 2006.

Fisher, R.A (1936). *The use of multiple measures in taxonomic problems*, Ann. Eugenics, 7:179—188, 1936.

Fung, Glenn. *CS 525 Project*. Fall, 1998.

Freund, Y. (1995). Boosting a weak learning algorithm by majority, *Information and Computation* 121(2): 256-285.

Freund, Y. and Schapire, R. (1996a). Experiments with a new boosting algorithm, *Machine Learning: Proceedings of the Thirteenth International Conference*, Morgan Kauffman, San Francisco, pp. 148-156.

Friedman, Jerome H., Trevor Hastie and Robert Tibshirani (1998) "Addative Logistic Regression: A Statistical View of Boosting." Stanford University, Dept. of Statistics, *Technical Report*.

Friedman, Jerome H. (1999a). Greedy Function Approximation: A Gradient Boosting Machine. *Technical report*, Dept. of Statistics, Stanford University.

Friedman, Jerome H. (1999b). Stochastic Gradient Boosting. *Technical report*, Dept. of Statistics, Stanford University.

Friedman, Jerome H. and Bogdan E. Popescu (2003) *Importance Sampled Learning Ensembles*.

Fung, Glenn. Siemens Medical Solutions. The Disputed Federalist Papers: SVM Feature Selection via Concave Minimization.

Gorsuch, Richard L. *Factor Analysis*. W. B. Saunders Co. 1974

Han, Jiawei and Micheline Kamber *Data Mining: Concepts and Techniques. Slides for Textbook Chapter 6*. http://www-courses.cs.uiuc.edu/~cs498han/slides/06.ppt#1095

Hand, David, Heikki Mannila, Padhraic Smyth. *Principles of Data Mining*. The MIT Press, 2001.

Harman, Harry H. *Modern Factor Analysis*. The University of Chicago Press. 1967.

Hartigan, J.A. and Wong, M.A. 1979. *A K-Means Clustering Algorithm*. Applied Statistics 28.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning; Data Mining, Inference, and Prediction*. Springer, 2001.

Hastie, T.J. and R.J. Tibshirani. *Generalized Additive Models*. Chapman & Hall/CRC, 1999.

Heinze, G. and Schemper, M. (2002). *A solution to the problem of separation in logistic regression.* Statistics in Medicine, 21, 2409 - 2419.

Heinze, G. and Ploner, M. (2003). *Fixing the nonconvergence bug in logistic regression with SPLUS and SAS*. Computer Methods and Programs in Biomedicine, 71, 181-187.

Heinze, G. (1999). *Technical Report 10/1999: The application of Firth's procedure to Cox and logistic regression*. Section of Clinical Biometrics, Department of Medical Computer Sciences, University of Vienna, Vienna, Austria.

Hosmer, David W., Stanley Lemeshow. *Applied Logistic Regression, Second Edition*. Wiley Series in Probability and Statistics, 2000.

Huber, P. (1964). Robust estimation of a location parameter, *Annals of Math. Stat.* 53: 73-101.

Hsu, C.-W and C.-J. Lin. A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415-425, 2002.

Huberty, Carl J. *Applied Discriminant Analysis*. John Wiley & Sons, 1994.

Ivakhnenko G.A. *Self-Organisation of Neuronet with Active Neurons for Effects of Nuclear Tests Explosions Forecasting*. System Analysis Modeling Simulation

Julian, Randy. *Using LDA*. Lilly Research Laboratories (http://miner.chem.purdue.edu/Lectures/Lecture10.pdf).

Klecka, William R. *Discriminant Analysis*. Sage Publications, 1980

Kecman, Vojislav. Support Vector Machines Basics. *School of Engineering Report 616*. The University of Auckland, School of Engineering. April, 2004.

Kleinbaum, David G., Mitchel Klein. *Logistic Regression, A Self Learning Text, Second Edition*. Springer, 1992.

Kordík, Pavel, Pavel Náplava, Miroslav Šnorek, Marko Genyk-Berezovskyj. "The Modified GMDH Method Applied to Model Complex Systems" *Department of Computer Science and Engineering, CTU, FEE Karlovo nám. 13, Prague, Czech Republic*

Kubat, Miroslav and Stan Matwin. *Addressing the Curse of Imbalanced Training Sets: One-Sided Selection.*

Loh, W.Y. and Shih, Y.S. (1997). *Split selection methods for classification trees*. Statistica Sinica 7: 815-840.

Maindonald, John and John Braun. *Data Analysis and Graphics Using R, An Example-based Approach*. Cambridge University Press, 2003.

Markowetz, Florian. "Classification by Support Vector Machines. Practical DNA Microsarray Analysis 2003." Max Planck Institute for Molecular Genetics, Computational Molecular Biology, Berlin.
https://phssec1.fhcrc.org/secureplone/www.bioconductor.org/workshops/2003/NGFN03/svm.pdf

Masters, Timothy (1993) *Practical Neural Network Recipes in C++*. Morgan Kaufmann.

Masters, Timothy (1995) *Advanced Algorithms for Neural Networks, A C++ Sourcebook*. John Wiley & Sons, Inc.

Meyer, David, Friedrich Leisch and Kurt Hornik (Nov., 2002). "Benchmarking Support Vector Machines", *Report No. 78*, Vienna University of Economics and Business Administration.

Meyer, David (Jan. 23, 2004). *Results of a benchmark study with focus on SVM's and resample/combine methods.*
http://www.imbe.med.uni-erlangen.de/links/EnsembleWS/talks/Meyer.pdf

Minsky, Marvin and Seymour Papert. *Perceptrons*. MIT Press, 1969.

Moller, Martin Fodslette (1993) *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*. Pergamon Press.

Momma, Michinari and Kristin P. Bennett. *A Pattern Search Method for Model Selection of Support Vector Regression*. SIAM Conference on Data Mining, 2002.

Morgan, J. N. and Messenger. "*THAID -- A sequential analysis program for the analysis of nominal scale dependent variables"*, Survey Research Center, U of Michigan. (1973)

Morgan, J. N. and J. A. Sonquist. [AID – Automatic Interaction Detection] "*Problems in the analysis of survey data and a proposal*", JASA, 58, 415-434. (1963)

Murphy, Patrick M and Michael J. Pazzani (1994). Exploring the Decision Forest: An Empirical Investigation of Occam's Razor in Decision Tree Induction. *Journal of Artificial Intelligence Research*, 1, (pp. 257-275).

Nguyen, Derrick and Bernard Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of adaptive weights", in *Proc. IJCNN*, vol. 3, pp. 21-26, July 1990.

Orr, Mark J.L. (1966): *Introduction to Radial Basis Function Networks*, Centre for Cognitive Science, University of Edinburgh, Scotland.

Park, Alex and Christine Fry. *Statistical modeling of user switching behavior based on reward histories*. (http://web.mit.edu/9.29/www/brett/ca_model.html)

Price, Kenneth V., Rainer M. Storn, Jouni A. Lampien (2005) *Differential Evolution, A Practical Approach to Global Optimization*. Springer-Verlag.
Qian, Bo and Khaled Rasheed (2004) "Hurst Exponent and Financial Market Predictability". Department of Computer Science, University of Georgia, Athens, GA USA.

Quinlan, J. Ross. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

Reyment, Richard and K.G. Koreskog. *Applied Factor Analysis in the Natural Sciences*. Cambridge University Press, 1993.

Rosenblatt, Frank. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review*. 1958

Rumelhart, David and James McClelland. *Parallel Distributed Processing*. MIT Press, 1986.

Rummell, R.J. *Applied Factor Analysis*, Northwestern University Press, 1970.

Schwardt, Ludwig and Johan du Preez. *Manipulating Feature Space*, PR414/PR813. The University of Stellenbosch. Feb. 15, 2005.

Segal, Mark R (2003). "Machine Learning Benchmarks and Decision Tree Forest Regression". Division of Biostatistics, University of California, San Francisco.

Shawe-Taylor, John and Nello Cristianni: *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

Sherrod, Phillip H. *NLREG Nonlinear Regression Analysis Program*. Phillip H. Sherrod, 2003. (http://www.nlreg.com)

Specht, Donald F. "Probabilistic Neural Networks" *Neural Networks*, 3: (1990).

Specht, Donald F. "Enhancements to Probabilistic Neural Networks," *Proceedings of the International Joint Conference on Neural Networks* (IJCNN '92). 1992.

Steinberg, Dan and Phillip Colla. *CART: Tree-Structured Non-Parametric Data Analysis*. San Diego, CA: Salford Systems, 1995.

Venables, W.N and B.D Ripley. *Modern Applied Statistics with S, Forth Edition*. Springer Science+Business Media, Inc., 2002.

Wilson, D. Randall and Tony R. Martinez. "Improved Center Point Selection for Probabilistic Neural Networks." *Proceedings of the International Conference on Artifical Neural Networks and Genetic Algorithms*, 9ICANNGA 1997), pp. 514-517, 1997.

Witten, Ian H, Eibe Frank. *Data Mining; Practical Machine Learning Tools and Technique with JAVA Implementations*. Academic Press, 2000.

Yang, Haiqin. *Margin Variations in Support Vector Regression for the Stock Market* Prediction. Masters thesis, The Chinese University of Hong Kong, June, 2003.

Zhang, Heping and Burton Singer. *Recursive Partitioning in the Health Sciences*. Springer, 1999.

Index

D

E

F

G

H

I

J

K

L

M

N

S

T

U

V

W

X

Y