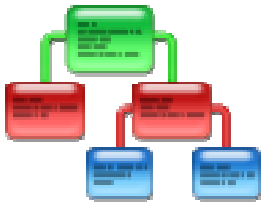
DTREG

Classification and Regression Trees And Support Vector Machines
For Predictive Modeling and Forecasting

Phillip H. Sherrod

Copyright © 2003-2006
All rights reserved

www.dtreg.com



DTREG builds classification and regression decision trees, support vector machine (SVM), discriminant analysis and logistic regression models that describe data relationships and can be used to predict values for future observations

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 Support Vector Machine (SVM), Discriminant Analysis 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 9
Data Mining 9
Data Modeling 10
Decision Trees 10

Introduction to Decision Trees 13
Building a Decision Tree Model 14
Overview of the Tree Building Process 14
Overview of Using Decision Trees 15
Using a Decision Tree to Predict Target Variable Values (Scoring) 16
Classes of Variables 17
Types of Variables 17
Regression and Classification Models 18

Using DTREG 21
Installing DTREG 21
DTREG's Main Screen 21
Creating a New Project 22
New Project Example 24
Opening an Existing Project 27
Example Projects Installed With DTREG 28
Running an Analysis to Build a Decision Tree 29
Viewing the Generated Decision Tree 30

Specifying Properties for a Model 31
Design Property Page 33
Title of project 33
Write a report of the analysis to a *project*_Log.txt disk file 33
Type of model to build 33
Tree fitting algorithm 34
How to categorize continuous variables 34
Cluster analysis control 34
Data Property Page 35
Data File Format 35
Continuing Data Lines 37
Specifying Missing Values in Data Files 37
Variables Property Page 38
Single Tree Model Property Page 41
Type of model to build 41
Minimum size node to split 41
Maximum tree levels 41
Method for validating and pruning the tree 42
Tree Pruning Control 43
TreeBoost Property Page 43
Decision Tree Forest Property Page 50

Forest size controls 51
Random Predictor Control 51
How to Handle Missing Values....52
How to Compute Variable Importance 52
Support Vector Machine (SVM) Property Page 53
Discriminant Analysis Property page....61
Logistic Regression Property Page 63
Class Labels Property Page....65
Designating a Focus Category 67
Initial Split Property Page 67
Category Weights Property Page....69
Misclassification Cost Property Page....71
Missing Data Property Page....74
Variable Weights Property Page 77
Miscellaneous Property Page 78
Random Number Starting Seeds....78

DTL: Data Transformation Language....81
DTL Property Page....81
The main() function....82
Global Variables....83
Implicit Global Variables 83
Explicit Global Variables 84
Static Global Variables....87
Using the StoreData() function to generate data records 87
The StartRun() and EndRun() Functions 88

Scoring Data Values....91
Input and output scoring files....92
Variables written to the output scoring file 92
Start scoring the data....93
Using scoring for validation with a test dataset 94
How missing values are handled during scoring....95

Translation: Generating Code for Scoring....97
Translate Property Page 98
How to call the scoring function – C and C++ programs....100
Generated header file 100
Generated Source File....102
How to call the scoring function – SAS® programs....103
Data types of variables....104
Generated header file 105
Generated Model Execution Source File 106

The Output Report Generated by DTREG....107
Project Parameters....108
Input Data....108
Summary of Variables 109
Summary of Categories....109
Tree Size and Validation Statistics 109

Node Splits 111
Node Summary 111
Target Category Distribution 112
Node Split Information 113
Competitor Predictor Variables 114
Surrogate Splitters 114
Analysis of Variance 115
Misclassification Summary Table 116
Confusion Matrix 117
Focus Category Report 117
Probability Threshold Report 119
Lift and Gain Table 122
How Lift and Gain Values are calculated 125
Terminal Node Table 128
Variable Importance Table 129

Viewing a Decision Tree 130
What's in a node – Classification tree 130
What's in a node – Regression tree 131

Charts and Graphs 133
Model Size Chart 133
Focus Category Impurity Chart 134
Focus Category Loss Chart 135
Lift and Gain Chart 136
Gain Chart 137
Lift Chart 138
Cumulative Lift Chart 138
ROC Chart 139
Probability Threshold Chart 141
Threshold Balance Chart 143
Variable Importance Chart 144
Actual versus Predicted (Residual) Chart 145

TreeBoost – Stochastic Gradient Boosting 147
Features of TreeBoost Models 148
How TreeBoost Models Are Created 149

Decision Tree Forests 151
Features of Decision Tree Forest Models 151
How Decision Tree Forests Are Created 152
No Overfitting or Pruning 153
Internal Measure of Test Set (Generalization) Error 153

Support Vector Machines (SVM) 155
Introduction to Support Vector Machine (SVM) Models 155
A Two-Dimensional Example 156
Flying High on Hyperplanes 158
When Straight Lines Go Crooked 159

The Kernel Trick 162
Parting Is Such Sweet Sorrow 166
Classification With More Than Two Categories 167
Optimal Fitting Without Over fitting 167
Standing On The Shoulders of Giants 168

Discriminant Analysis 171
Introduction to Discriminant Analysis 171

Logistic Regression 177
Introduction to Logistic Regression 177
The Dose-Response Curve 178
The Logistic Model Formula 179
Output Generated for a Logistic Regression Analysis 180
Summary statistics for the model 180
Computed Beta Parameters 180
Likelihood Ratio Statistics 182
Computational Issues for Logistic Regression 182
Failure to Converge 182
Singular Hessian Matrix 183
Complete and Quasi-Complete Separation of Values 183

How Trees are Built and Pruned 185
Building Trees 185
Splitting Nodes 185
Evaluating Splits 187
Assigning Categories to Nodes 188
Missing Values and Surrogate Splitters 188
Stopping Criteria 190
Pruning Trees 190
Why Tree Size Is Important 191
V-Fold Cross Validation 193
Adjusting the Optimal Tree Size 195

Decision Trees Compared To Other Modeling Methods 197
Supervised and Unsupervised Machine Learning 197
Linear, Nonlinear and Logistic Regression 197
Neural Networks 198
The History of Decision Tree Analysis 199

Example Analyses 201

DTREG COM Library 203

Licensing and Use of DTREG 207
Use and Distribution of DTREG 207
Copyright Notice 207
Disclaimer 207

References **209**

Index **215**

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. Experience over the last 40 years has shown decision trees to be a highly effective method for analyzing and modeling many types of data. DTREG builds classification and regression decision trees that describe data relationships and predict values for future observations.

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 122.

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 single tree, TreeBoost, decision tree forests, support vector machines (SVM), 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.

Decision Trees

One of the most flexible modeling methods is decision trees and "ensemble" tree method such as TreeBoost and Decision Tree Forests.

The DTREG program analyzes ("mines") a set of data values and generates a decision tree that can be used to predict the value of a "**target variable**" based on the values of a set of "**predictor variables**". (For information about target and predictor variables, see page 17.) Like a real tree, a decision tree has a "root", "branches" and "leaves". A prediction is made by entering the tree at the root and following the branches left or right based on values of the predictor variables until a leaf is reached. Each leaf shows the most likely value for the target variable given the set of predictor values that led to the leaf. (See the example tree on page 13.)

The concept of decision trees is ancient – it is rooted (no pun intended) in the basic idea of deductive reasoning. But the ability to analyze a large set of data records with many variables requires so much computational power that it was impractical until modern, high-speed computers were developed.

Decision trees have a number of advantages over competing procedures:

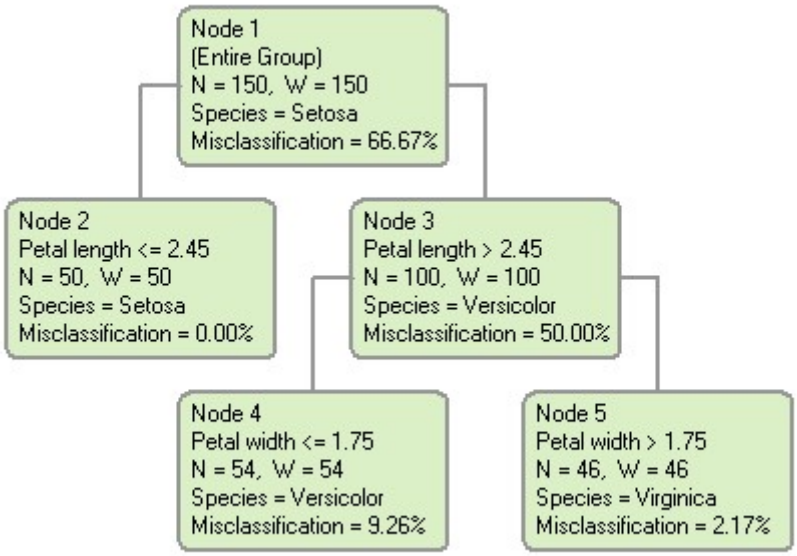
- Decision trees are easy to build. Just feed a dataset into DTREG, and it will do all the work of building a decision tree and pruning it to the optimal size.
- Decision trees are easy to understand. Unlike nonlinear regression models, or, worse, neural networks, decision trees provide a clear, logical representation of the data model. They can be understood and used by people who do not have to be mathematicians and statisticians.

- Decision trees handle both continuous (ordinal, interval) and categorical (nominal) variables. Categorical variables such as gender, race, religion, marital status and geographic region are difficult to model using numerically-oriented techniques such as regression and neural networks. In contrast, categorical variables are handled easily by decision trees. DTREG uses advanced techniques that enable it to build classification trees even when there are a large number of predictor categories.
- Decision trees can perform classification as well as regression. The predicted value from a decision tree is not simply a numerical value but can be a predicted category such as male/female, malignant/benign, frequent buyer/occasional buyer, etc.
- DTREG accepts text data as well as numeric data. If you have categorical variables with data values such as "Male", "Female", "Married", "Tennessee", "Protestant", etc., there is no need to code them as numeric values.
- Decision trees automatically handle interactions between variables. For example, men in the North may have different characteristics than men in the South; likewise, there may be differences between southern women and northern women. So it is necessary to consider both gender and region when making a prediction. Decision trees automatically deal with these interactions by partitioning the cases and then analyzing each group.
- Decision trees identify important variables. By examining which variables are used to split nodes near the top of the tree, you can quickly determine the most important variables. DTREG carries this further by analyzing all the splits generated by each variable and the selection of surrogate splitters. A table ranking overall variable importance is included in the analysis report (see page 129). The determination of variable importance is particularly useful in studies where many variables are available (for example, demographic data).
- Decision trees handle missing data values well. DTREG uses a sophisticated technique involving "surrogate splitters" (see page 188) to handle cases with missing values. This allows cases with some available values and some missing values to be utilized to the maximum extent when building models. It also enables DTREG to predict values for cases with missing data.

- Decision trees do not require the specification of the form of a function to be fitted to the data as is required by nonlinear regression.
- DTREG can generate both single-tree models and also TreeBoost and Decision Tree Forest models consisting of a ensembles of trees. See the chapter beginning on page 147 for information about TreeBoost models; see the chapter beginning on page 151 for information about decision tree forests.
- DTREG can generate Support Vector Machine (SVM) models which are similar to neural networks. SVM models are particularly good at pattern recognition such as facial recognition and identifying printed or hand-written characters.
- DTREG includes a full implementation of Linear Discriminant Analysis that is fast and works well for many types of data analyses.
- DTREG can perform Logistic Regression to model data that has a categorical target variable with two categories. See the chapter beginning on page 177 for information about logistic regression.
- DTREG includes a full Data Transformation Language (DTL) for transforming variables, creating new variables and selecting which cases are to be analyzed. See the chapter starting on page 81 for information about DTL.

Introduction to 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 130.

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 185.

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 13, 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 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 35 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 virtually an unlimited number of records; it is quite feasible to analyze datasets with millions of records, although the computation time may be lengthy.

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 185 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:

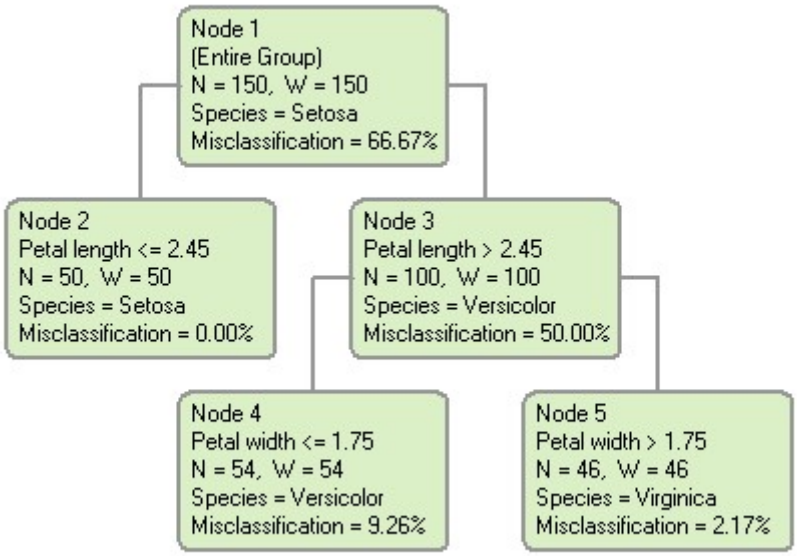
- You can use the tree to make inferences that help you understand the "big picture" of the model. One of the great advantages of decision trees is that they are easy to interpret even by non-technical people. For example, if the decision tree models product sales, a quick glance might tell you that 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 91.

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.

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.

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 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.

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

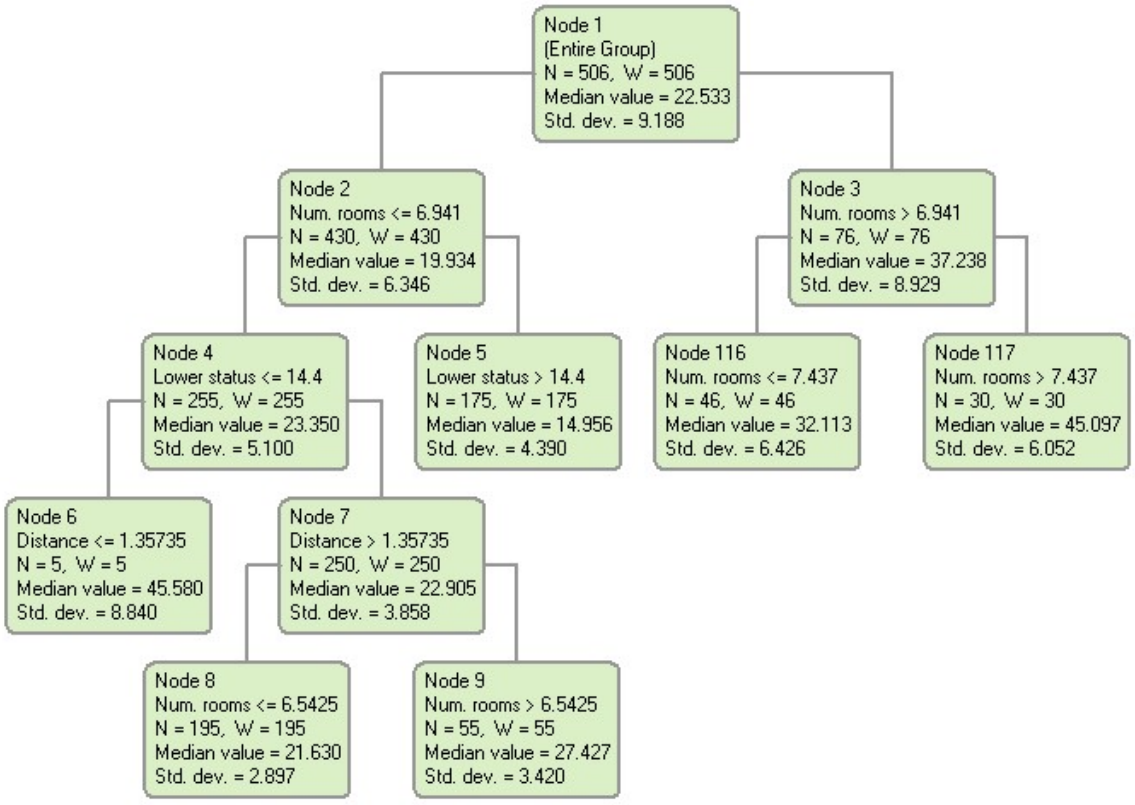
categorical gender variable. Because categorical values are stored and compared as string values, a categorical value of 001 is different than a value of 1. In contrast, values of 001 and 1 would be equal for continuous variables.

Regression and Classification Models

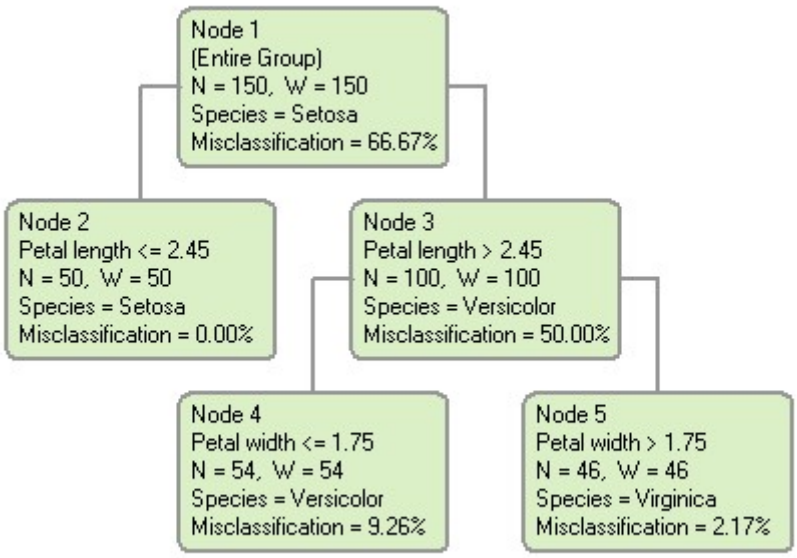
DTREG will generate a regression model or a classification model depending on whether the target variable is continuous or categorical.

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.



Using DTREG

Once you understand the concept of decision trees, it is very easy to use DTREG to analyze data and build decision trees to model data.

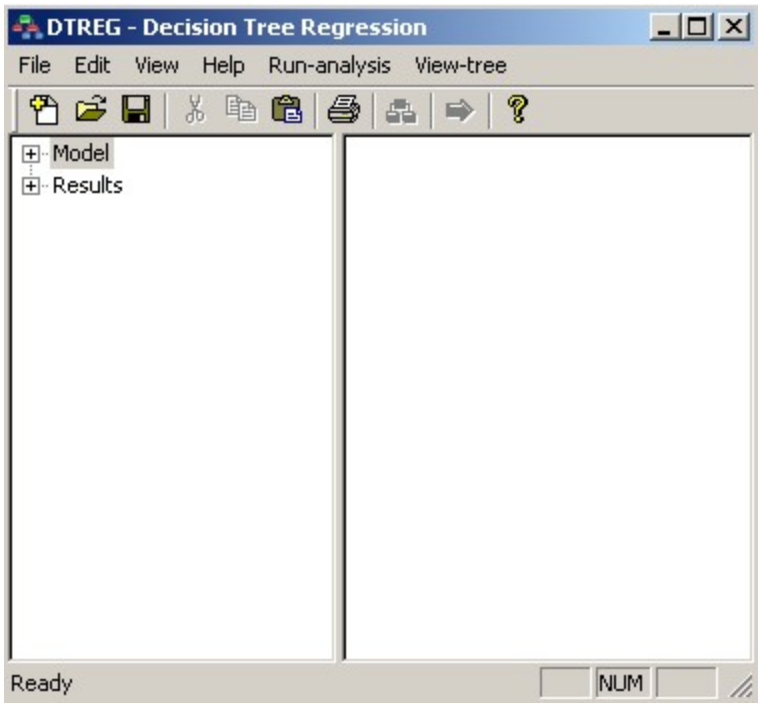
Installing DTREG

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:

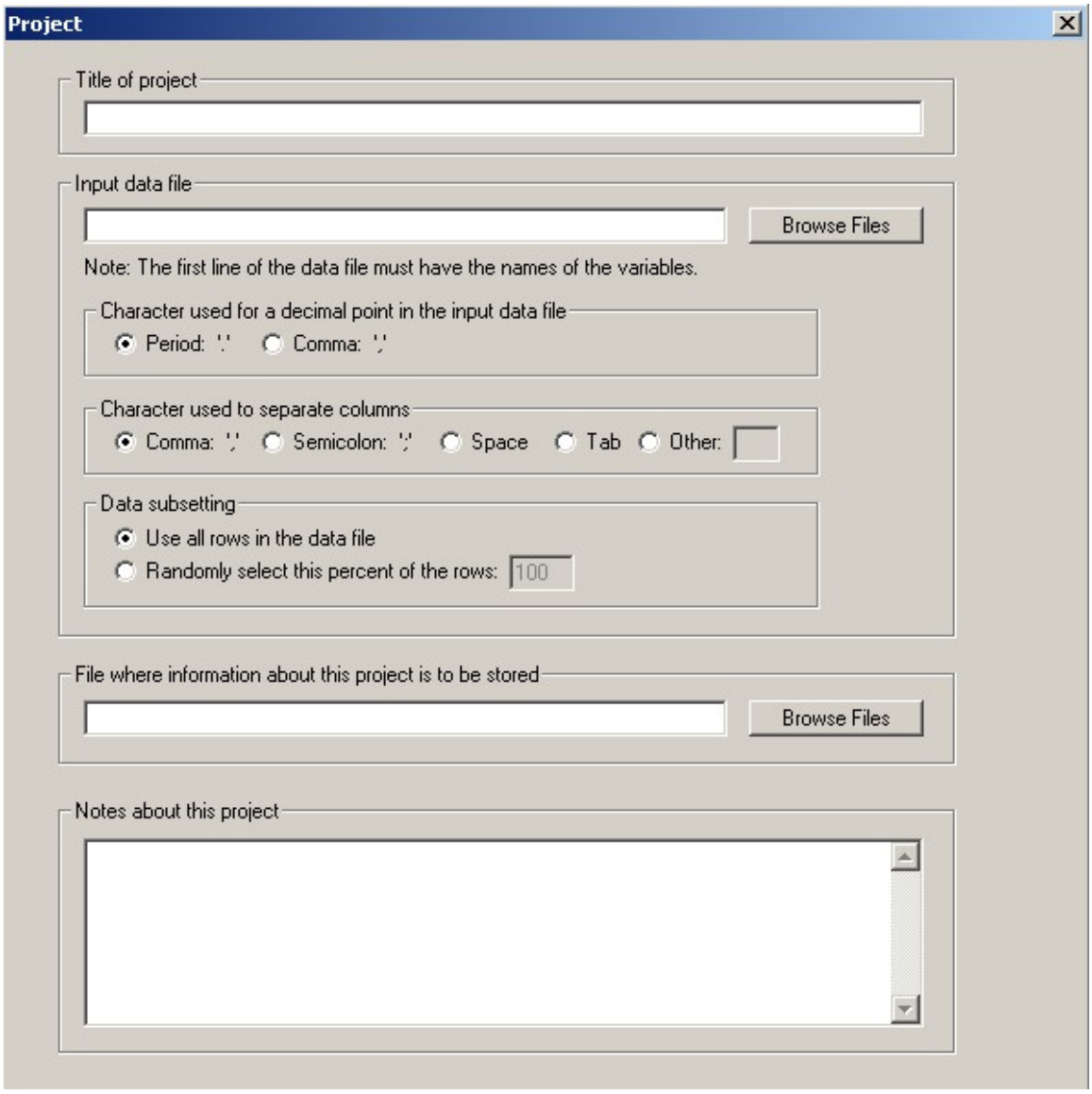


From this screen, you can

- Create a new project to build a decision tree by clicking 
- Open an existing decision tree project by clicking 

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 35 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:

- Variables (see page 38)
- Class labels (see page 65)
- Design (see page 33)
- Initial split (see page 67)
- Category weights (see page 69)
- Misclassification Costs (see page 71)

New Project Example

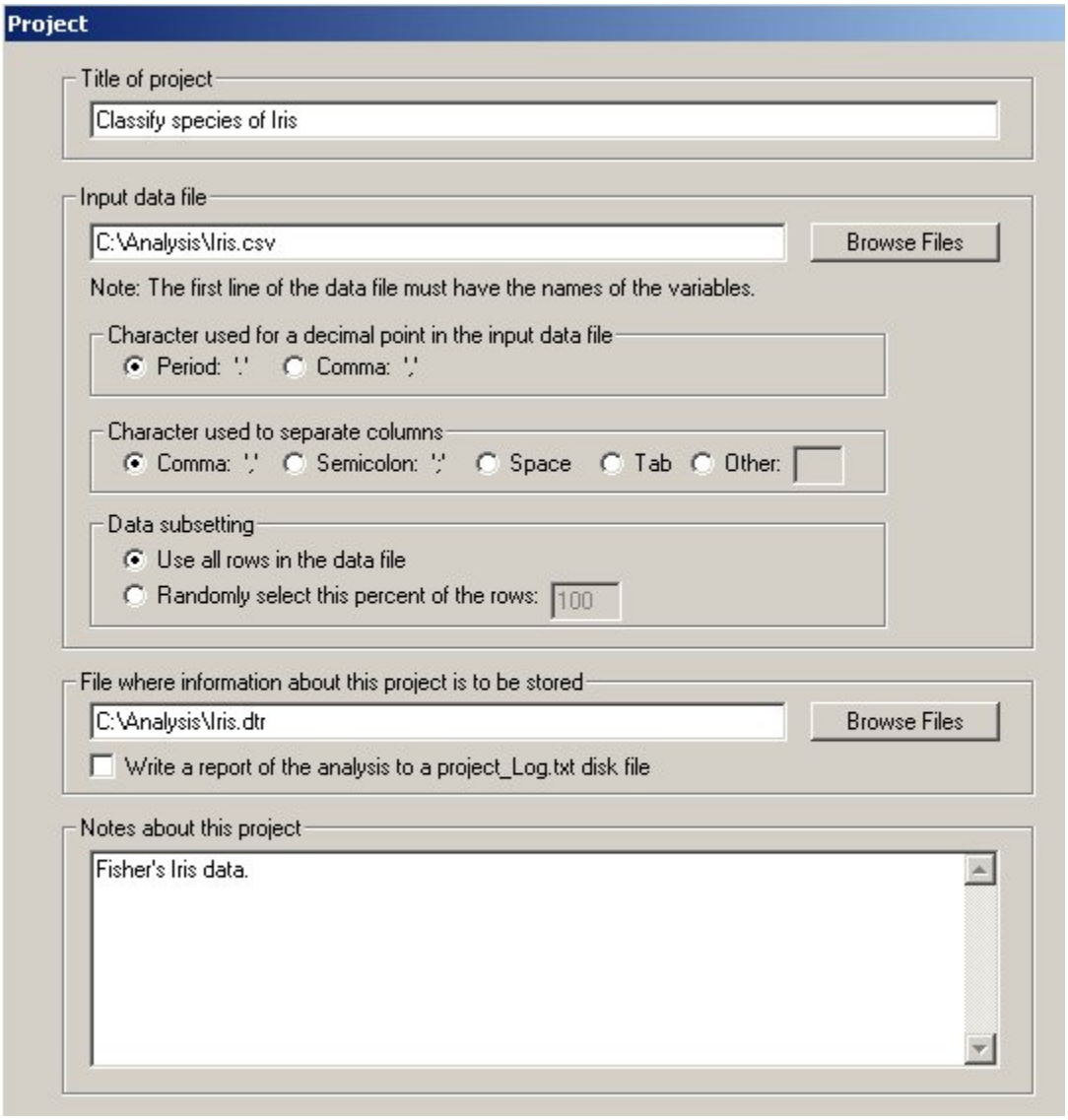
To illustrate the process of creating a new project, let's consider a concrete case. We will look at the steps involved in setting up a DTREG project to classify species of irises based on measurements of the plants. The data we will use is from the classic study devised by R. A. Fisher in 1936 (Fischer, 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
```

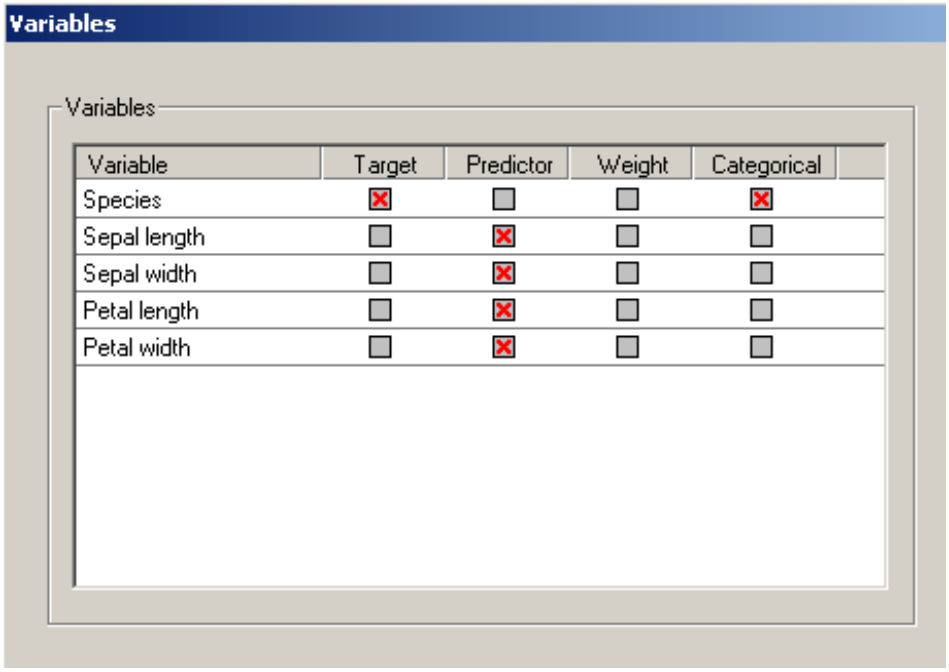
The first line of the file has the names of the variables separated by whatever character you selected as the column delimiter (by default it is a comma). In this case, there are 5 variables: Species, Sepal length, Sepal width, Petal length and Petal width. Variable names and values that contain spaces 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 35 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:



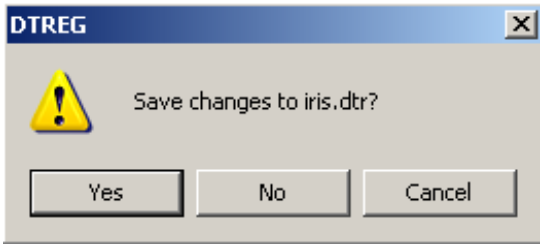
On the second screen, specify information about the variables:



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

After setting information about variables, click the "Next" button to advance through the setup screens.

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

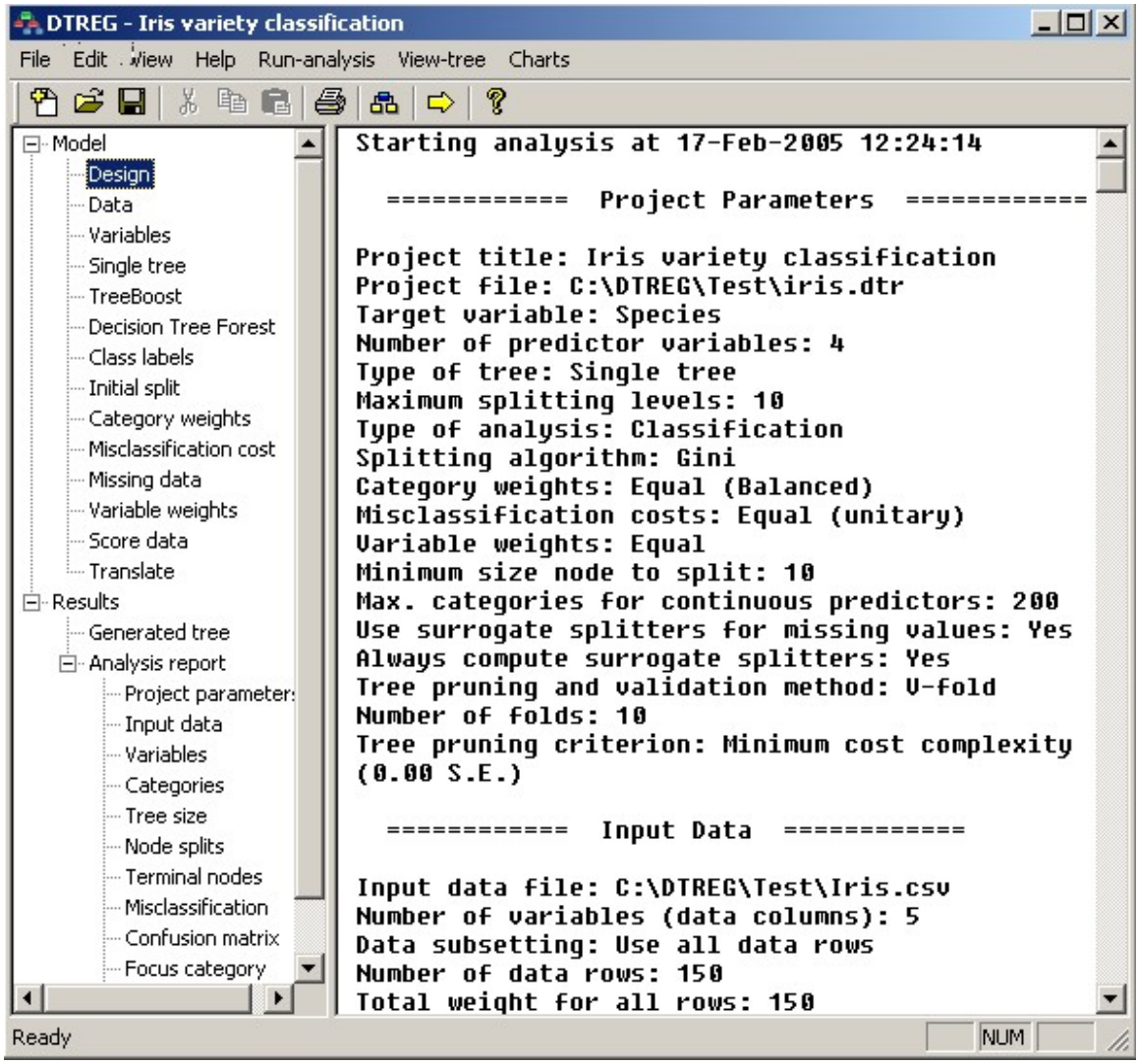
All of the information about a DTREG project is stored in a project database. This includes parameters that control the analysis, information about variables, the name of the data input file, the generated report, and information required to construct and display the generated decision tree. These project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



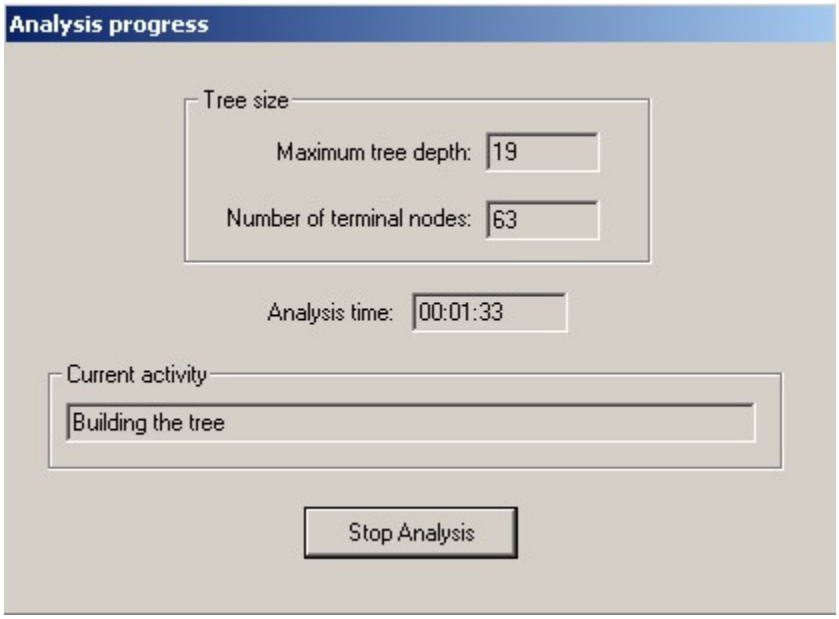
Example Projects Installed With DTREG

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 201 for additional information about example analyses.

Running an Analysis to Build a Decision Tree

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

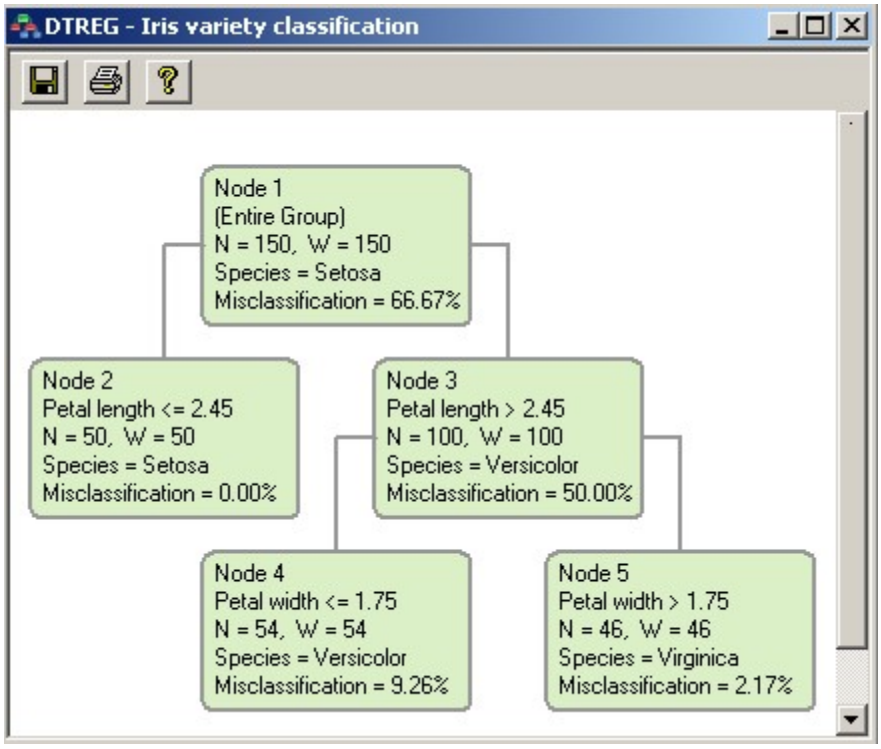
While an analysis is running, a progress screen similar to this will be displayed:



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

Viewing the Generated Decision Tree

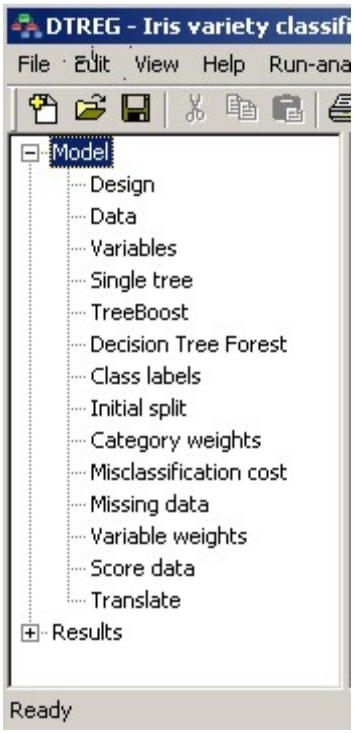
Once an analysis has been completed, you can view the generated decision tree by clicking the  toolbar icon or by clicking "View-tree" on the main menu. To save the decision tree in a jpg, png or bmp disk file, click the disk icon. To print the decision tree, click the printer icon.



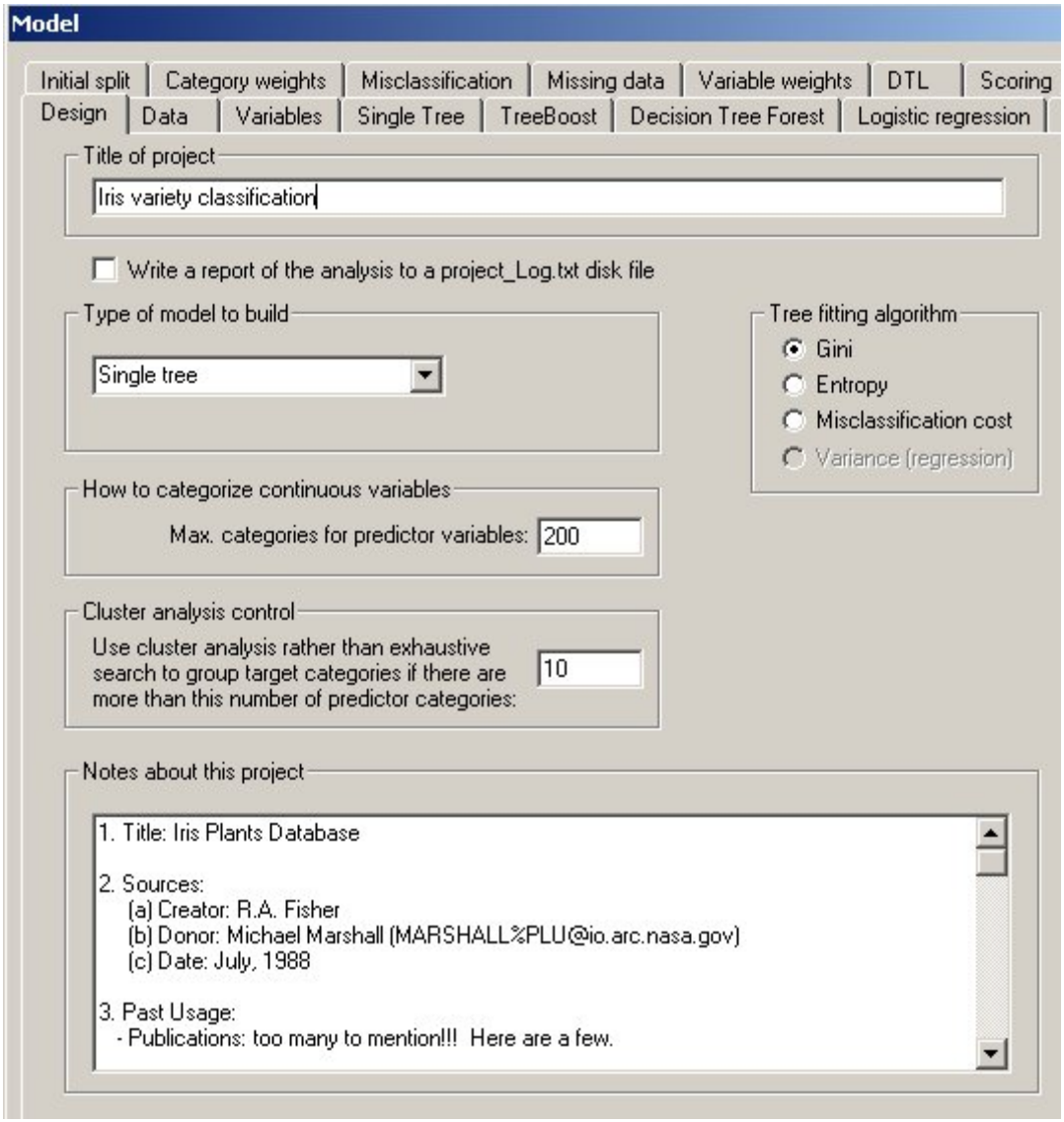
Specifying Properties for a Model

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:



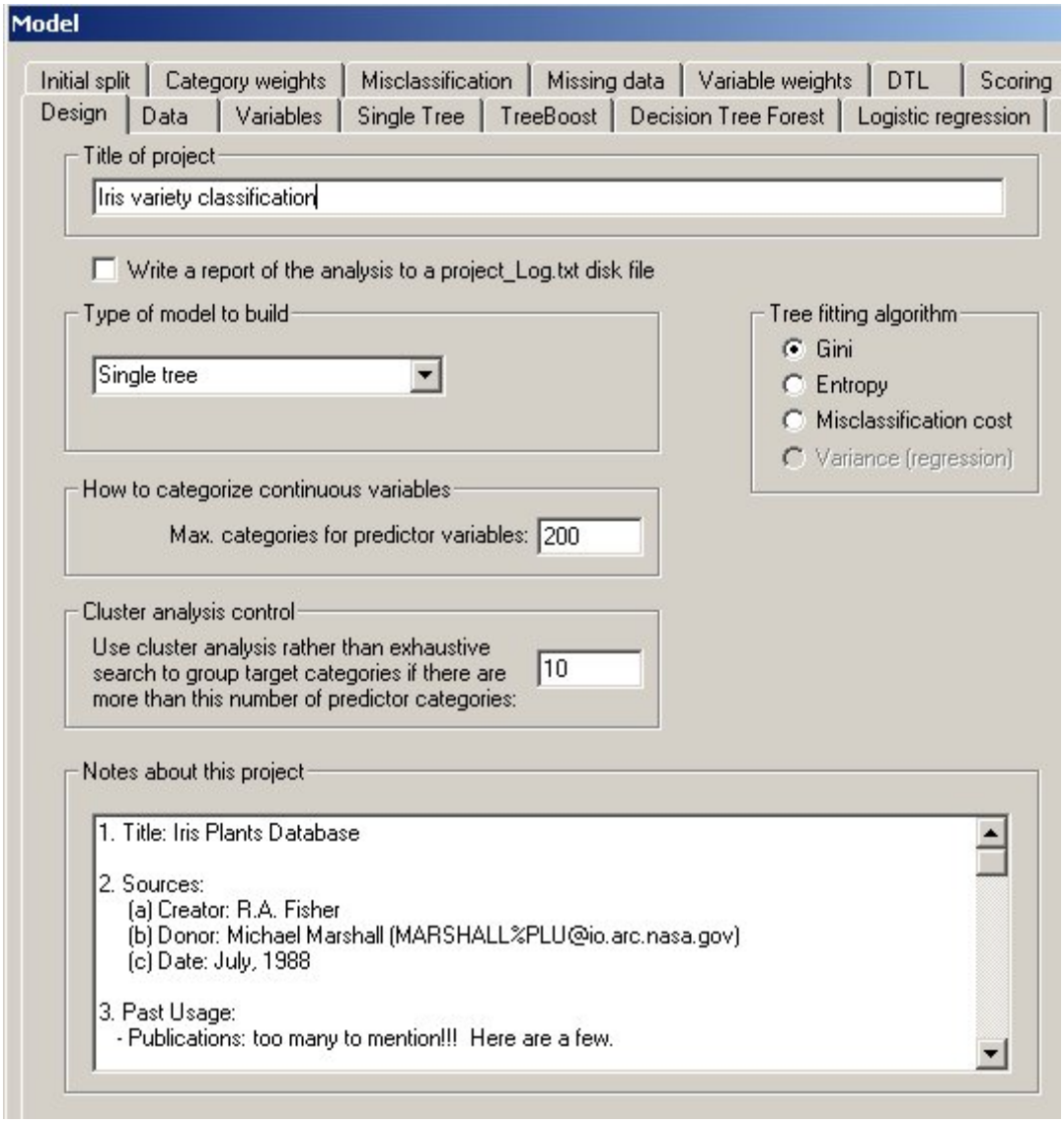
The Model screen displays with tabs for each property, similar to the one shown below:



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.

Tree fitting algorithm – 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.

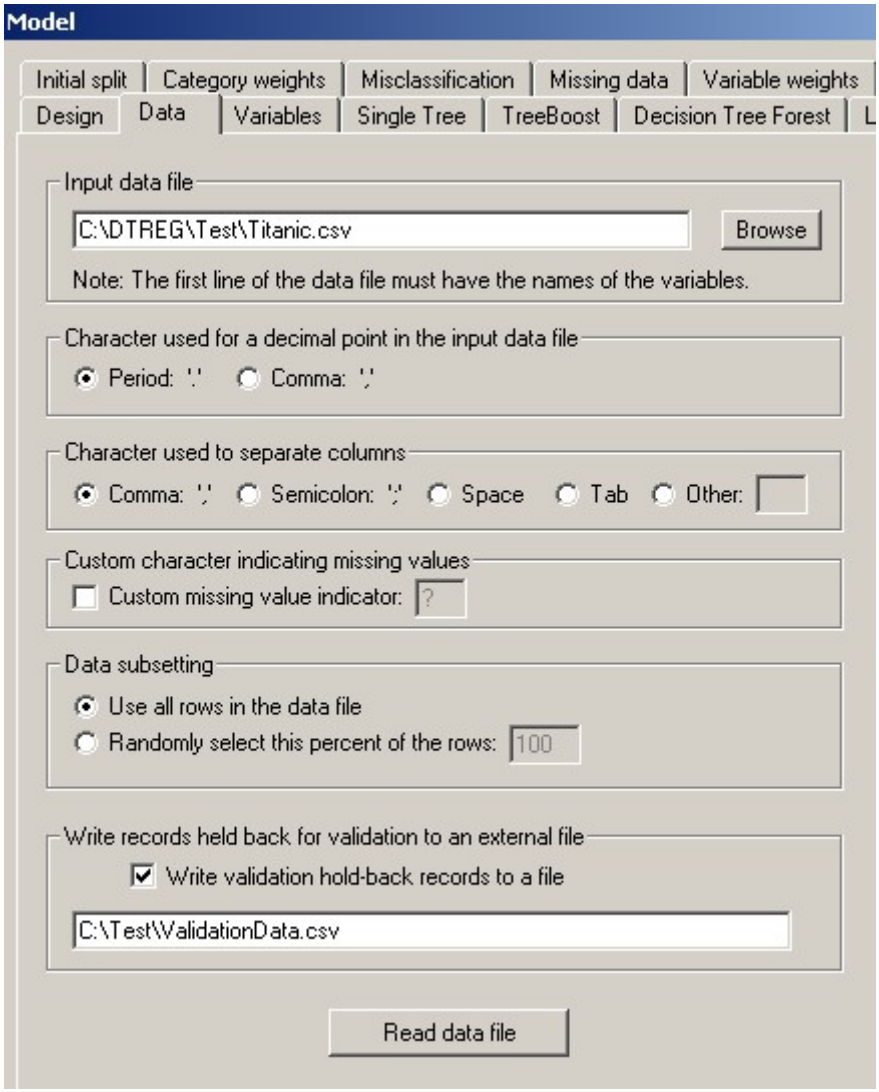
- **Variance** -- The variance splitting method is always used for regression trees. It causes DTREG to use the split that minimizes the sum of variance (i.e. sum of squared errors) in the child nodes.

How to categorize continuous variables – The values of continuous predictor variables are grouped into categories before they are used to build the 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.

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.

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.

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.

There are three 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.

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
```

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:

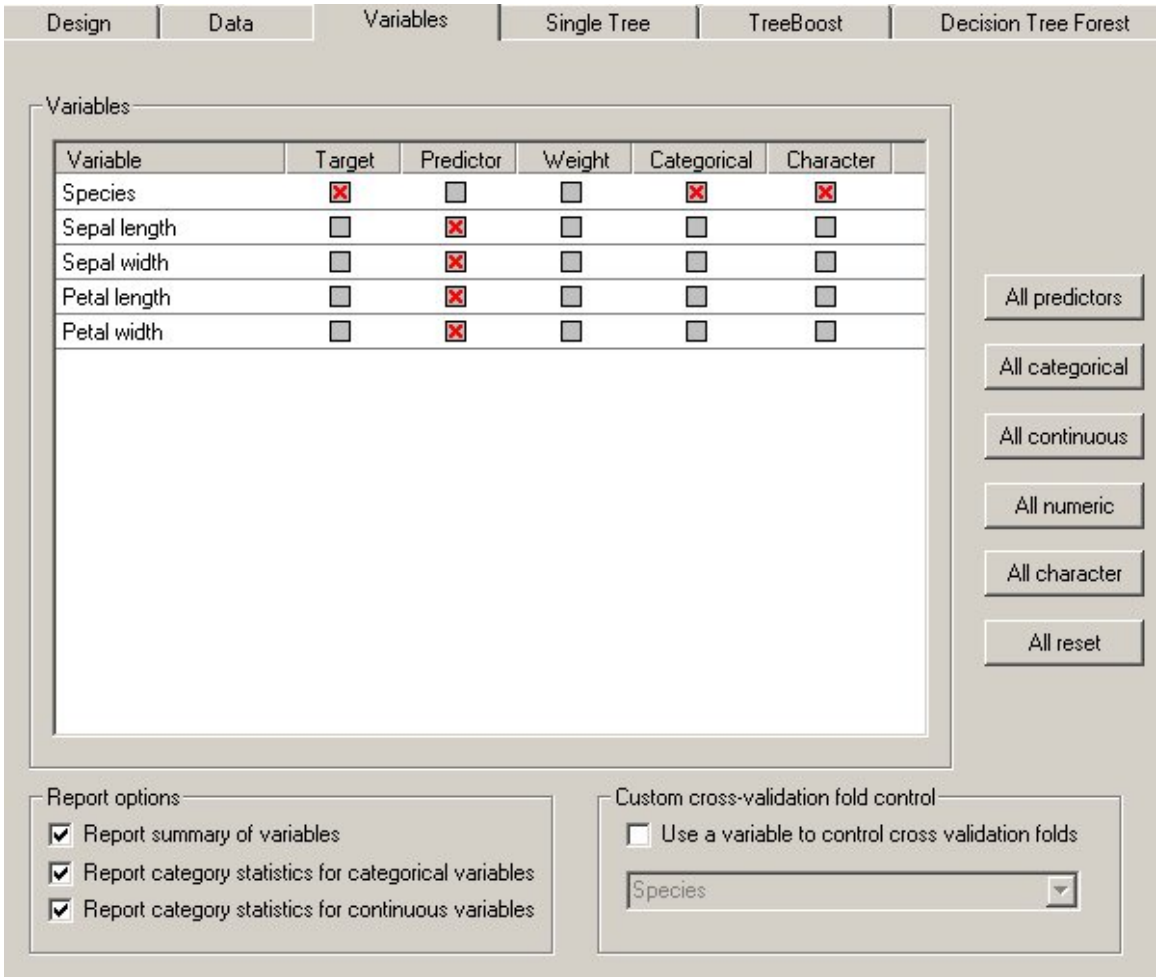
- A field that is entirely empty (nothing between the commas).
- The question mark character ('?')
- A single period ('.').
- A character you specify using the "Custom missing value indicator" specification.

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 35).

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 decision tree. 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 97). 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:

- **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.

- **All categorical** – Click this button to set the categorical checkboxes for all variables.

- **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.

There are two 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 109.
- **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.

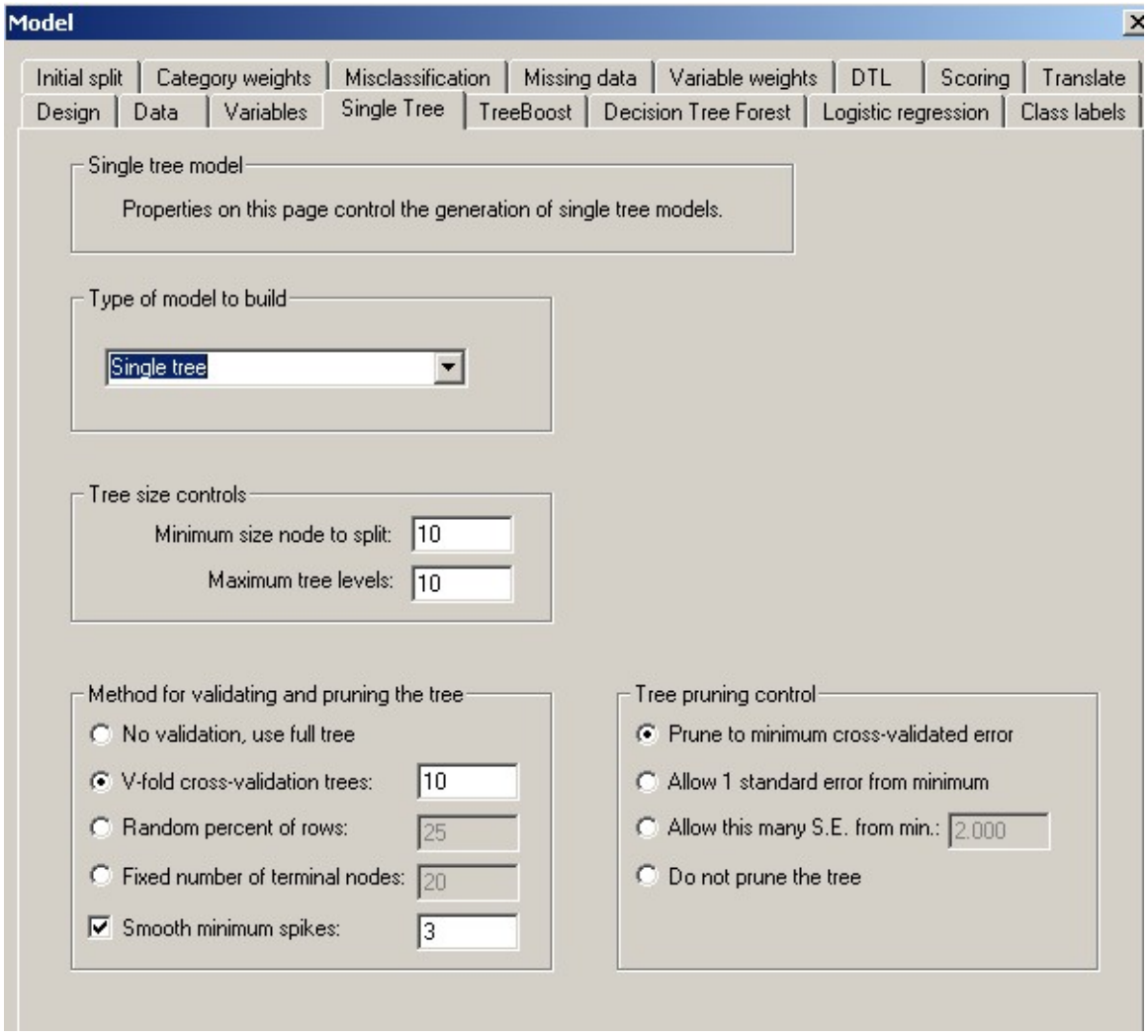
Cross-validation Control Variable

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.

Single Tree Model Property Page

The Single Tree property page is used to specify parameters for single tree models. Use the TreeBoost and Decision Tree Forest pages to specify parameters for those types of 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 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 190 for information about how pruning is done.

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 193.

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.

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 190.

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 195.

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".

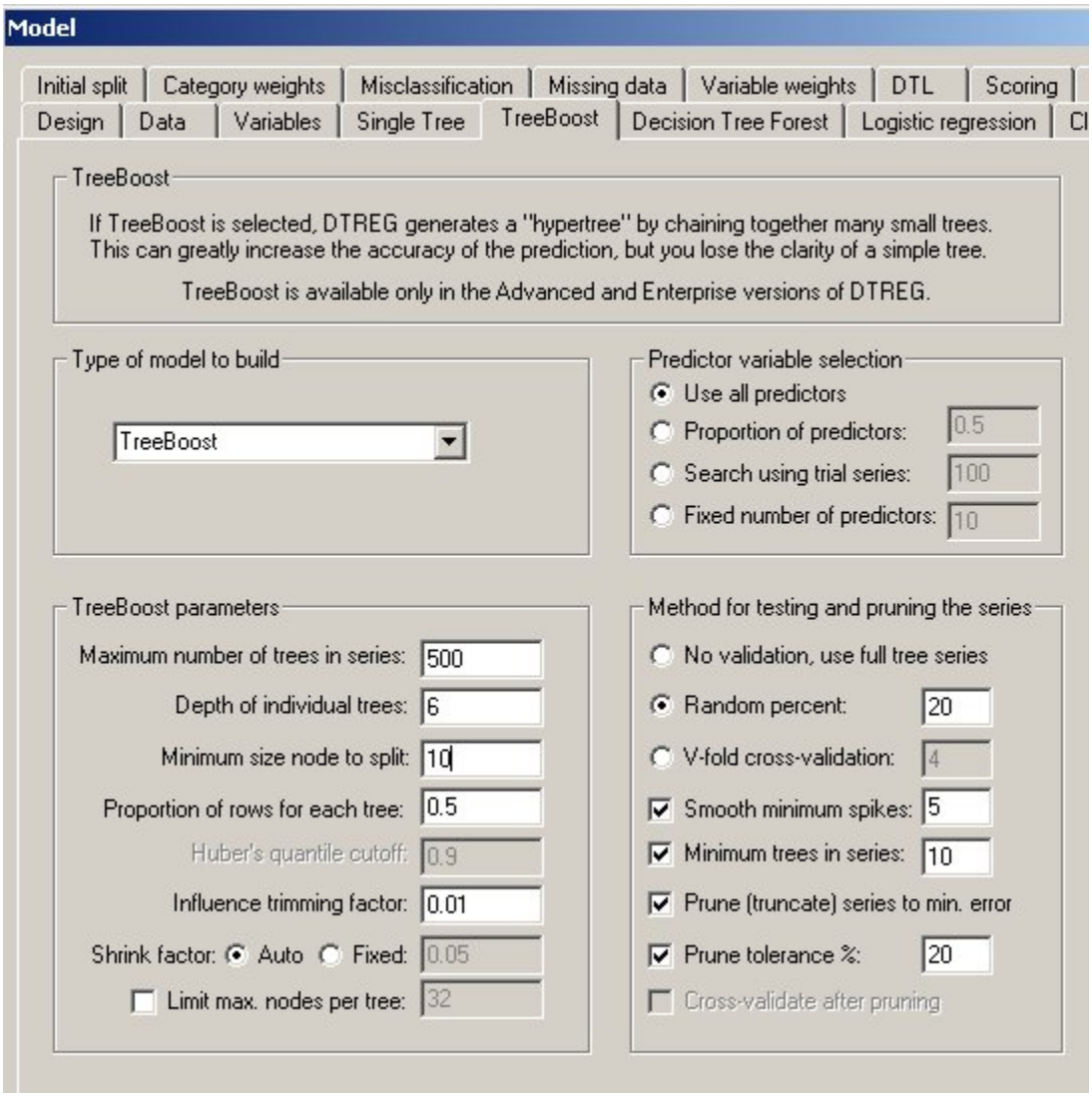
Do not prune the tree – Select this option if you want DTREG to perform cross-validation but not prune the tree. You will get the cross-validation statistics, but the full, unpruned tree will be generated.

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 147.

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.

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 193.

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.

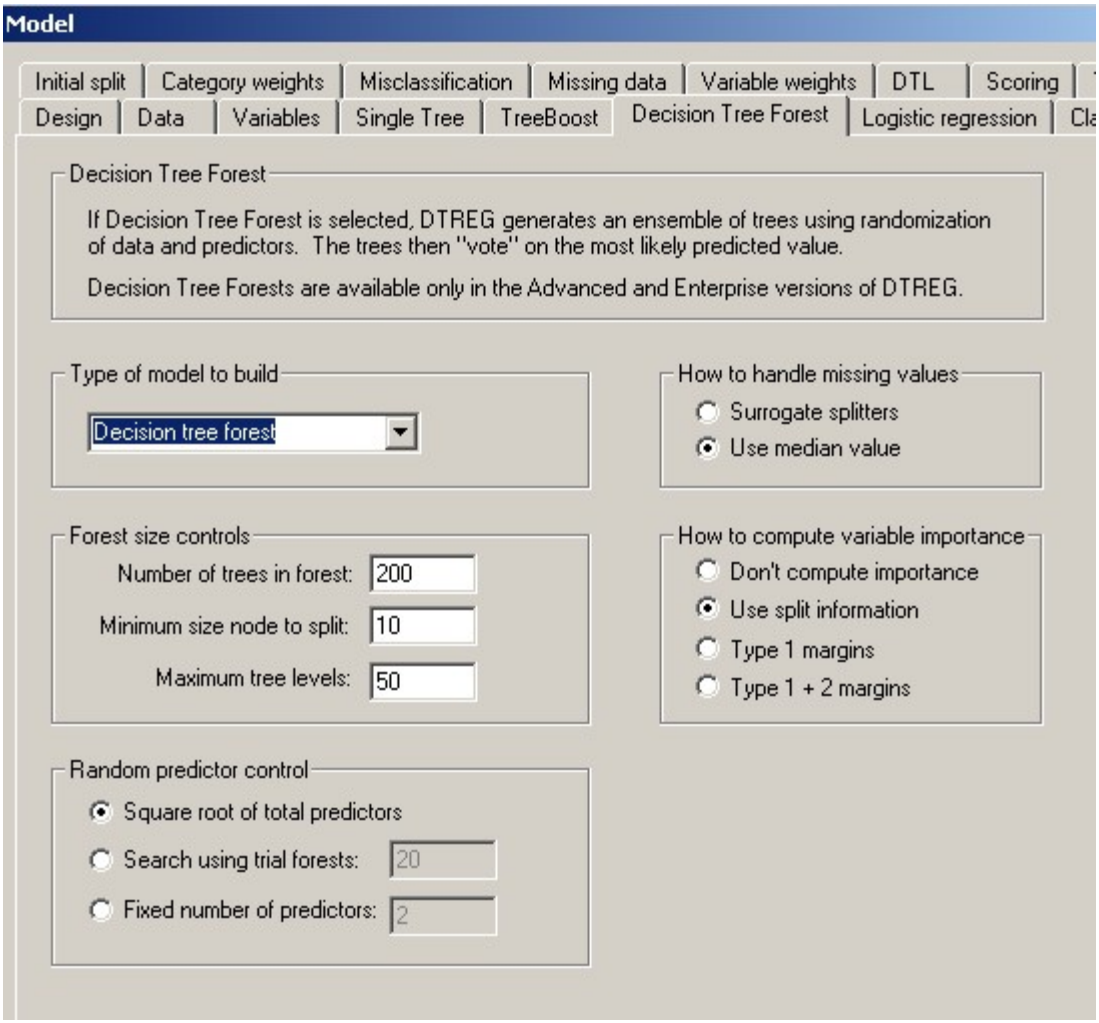
Predictor variable selection: In some cases, it may be possible to improve the quality of a TreeBoost model by considering only a random subset of the predictors for each split rather than all predictors. This is somewhat similar to the predictor selection method used by Decision Tree Forest Models. 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 151.

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 74) to control whether DTREG always computes surrogate splitters or only computes them when there are missing values in a node.

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.

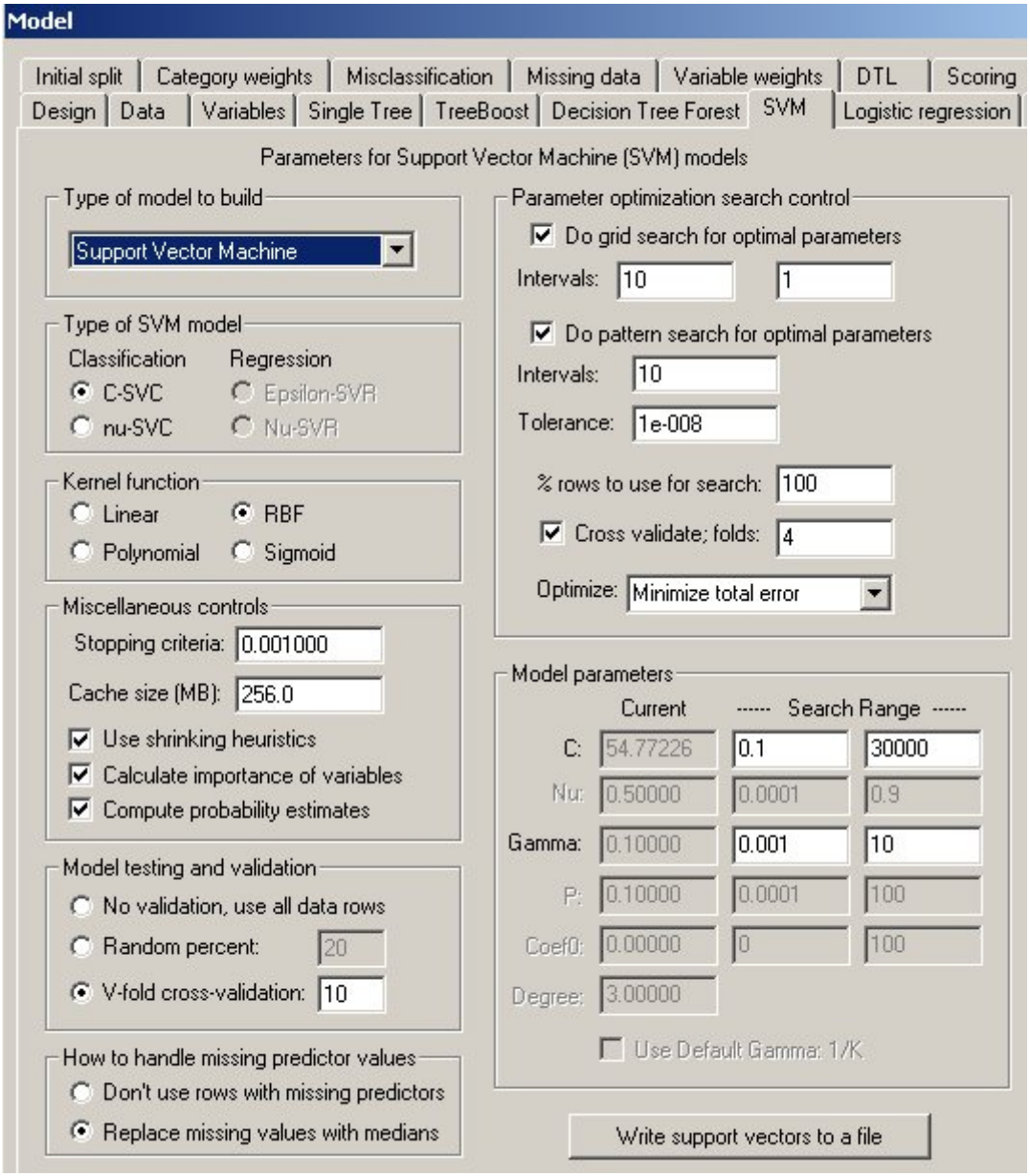
Type 1 + 2 margins – DTREG first calculates the importance using type 1 margins as described above. It then examines each data row and determines how many trees in the forest correctly voted for the row with the original data minus the number of trees that correctly voted for the row using the permuted 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.

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 155.

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

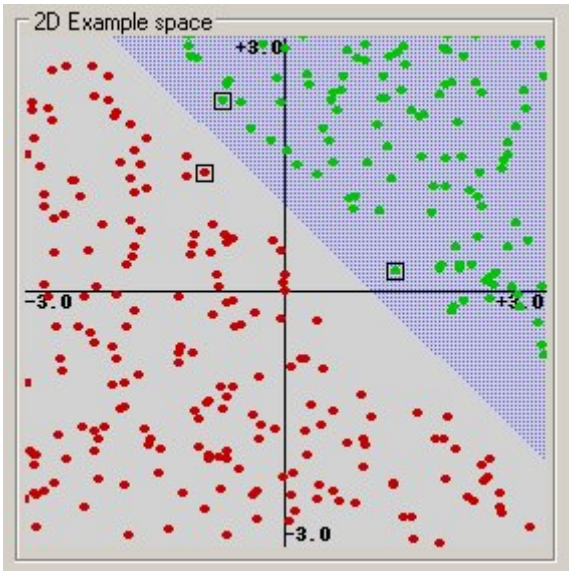


Type of model to build: Select the type of model you want DTREG to build. If you select a model type other than support vector machine, all of the other controls on this screen will be disabled.

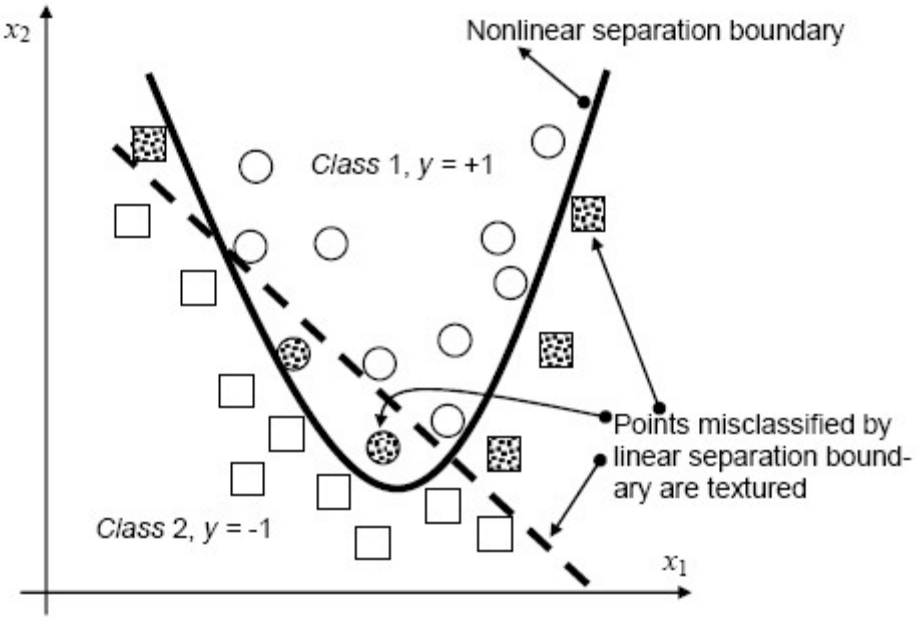
Type of SVM model – DTREG offers several types of SVM models. For classification models with a categorical target variable, you can select either C-SVC or ν-SVC models. For regression models with a continuous target variable, you can select either ε-SVR or ν-SVR models. For most applications, the results 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.

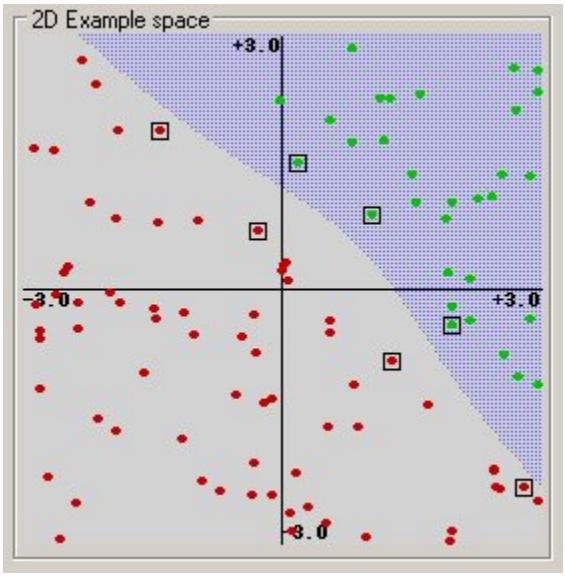
Kernel function – SVM models are built around a *kernel function* that transforms the input data into an *n*-dimensional space where a hyperplane can be constructed to partition the data. DTREG provides four kernel functions, Linear, Polynomial, Radial Basis Function (RBF) and Sigmoid (S-shaped). There 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

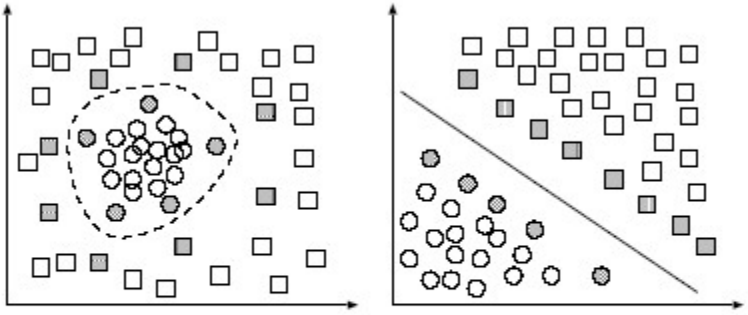


Polynomial: (gamma*u'*v + coef0)^degree
See the following figure from Kecman, 2004.



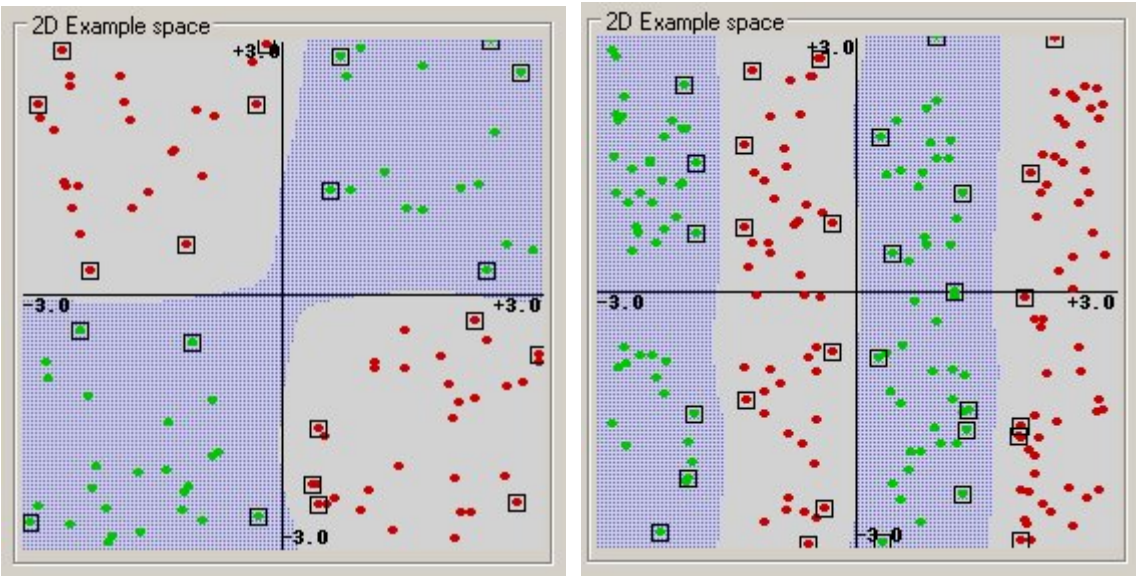


Radial basis function: exp(-gamma*|u-v|^2)
A Radial Basis Function (RBF) is the default and recommended kernel function. The RBF kernel non-linearly maps samples into a higher dimensional space, so it can handle nonlinear relationships between target categories and predictor attributes; a linear basis 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.

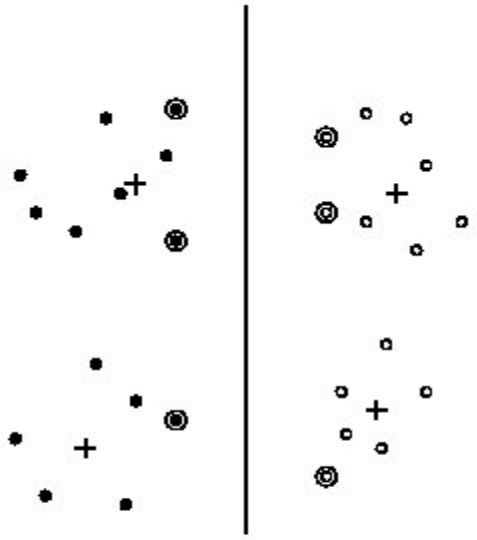


(a) Radial Basis Function (b) RBF mapping

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

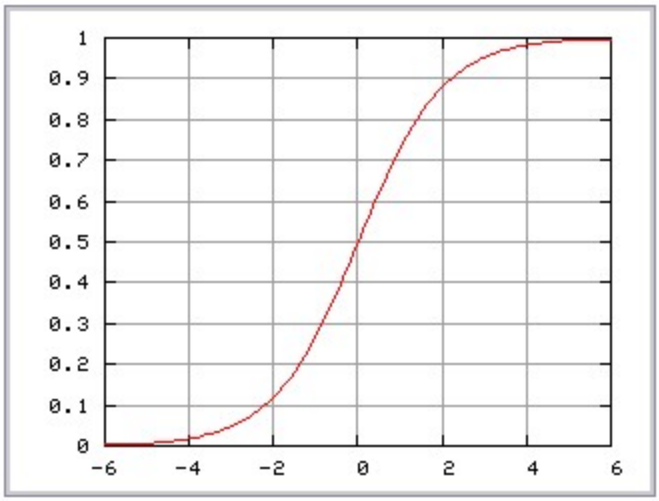


An SVM model using a radial basis function kernel has the architecture of an RBF network. However, the method for determining the number of nodes and their centers is different from standard RBF networks with the centers of the RBF notes on the support vectors (see the figure below from C. Campbell).



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 (feed-forward neural network): tanh(gamma*u'*v + coef0)



Stopping criteria – 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.

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.

V-fold cross validation – If this option is selected, V SVM models will be constructed with $(V\text{-}1)/V$ 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 two choices for dealing with predictor variables that have missing values. You can either exclude those rows from the analysis, or you can replace the missing values with the median values for the variable. In the case of categorical variables, the most common category is used as the replacement. Rows are always excluded if the value of the target variable is missing.

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.

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.

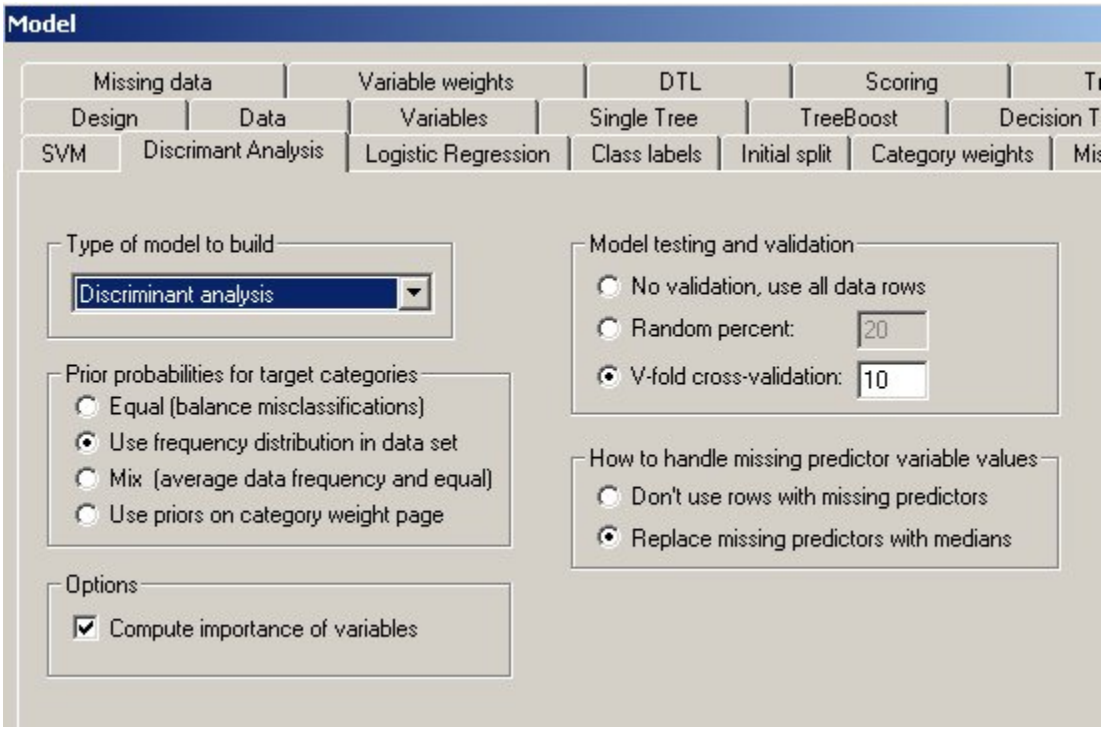
Write Support Vectors to a File – Click this button to open a dialog box where you can specify a file where the support vectors for the generated model should be written. This button is enabled only if a model has been built and support vectors have been found.

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 171.

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 69).

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.

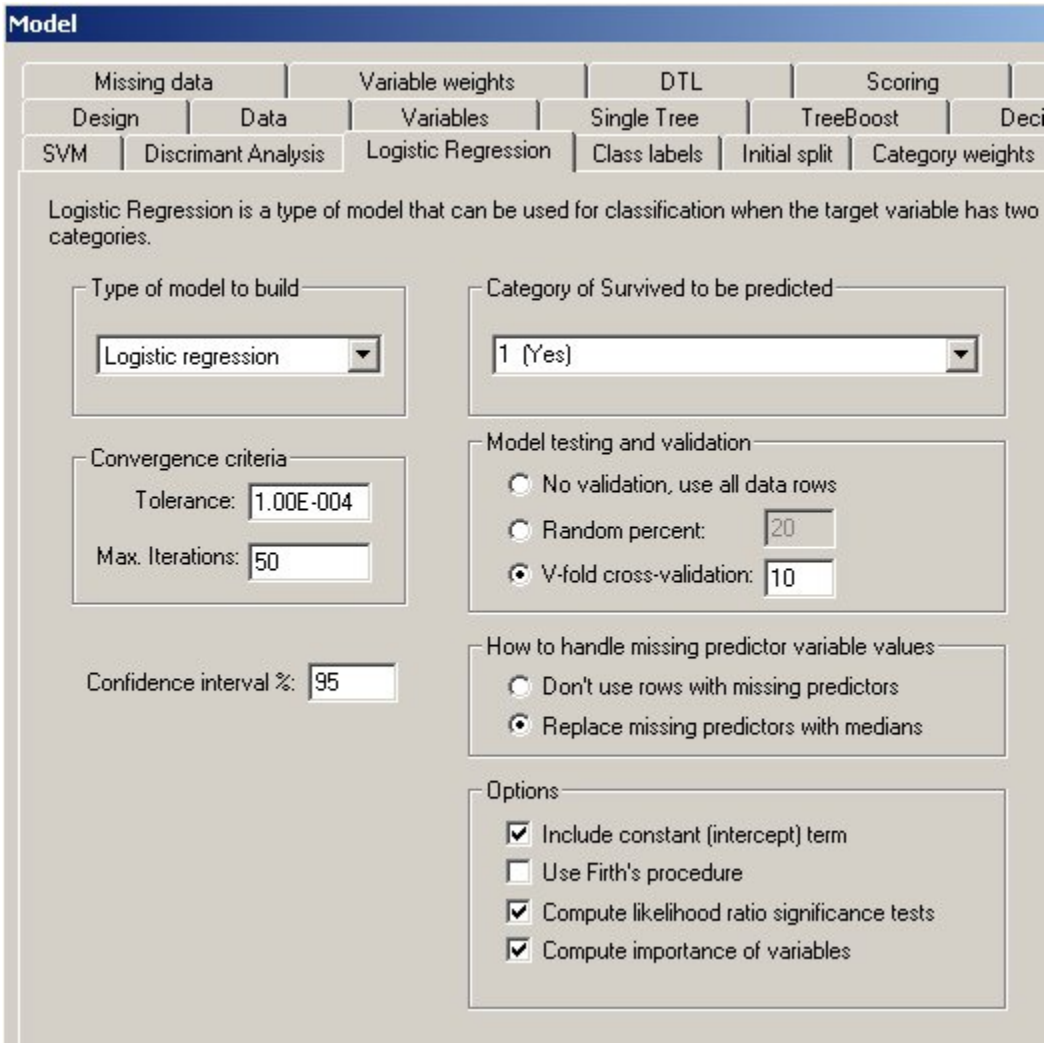
How to handle missing predictor variable values: Select whether you want DTREG to replace missing values of predictor variables with the median value or whether you want it to reject data rows that have any missing predictor values. The mode (most common category) is used rather than the median for categorical predictors. Note: if all of the predictor values are missing or if the target or weight variable values are missing, the record is always rejected.

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 177.

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.

Category to be predicted – Select which category of the target variable is to be predicted by the model. If the target variable is continuous and contains probability values, then this field 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 – Two methods are available for handling missing values on predictor variables. If you select the option "Don't use rows with missing values", then if a data row has a missing value on any predictor variable, the entire row is excluded from the analysis. If you select "Replace missing predictors with medians" then any missing predictor values will be replaced by the median value of the predictor. In the case of a categorical predictor, the most commonly occurring category of the predictor will be used. If the target or weight variables have missing values, rows are always excluded. If all of the predictor values for a row are missing, the row is excluded.

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.

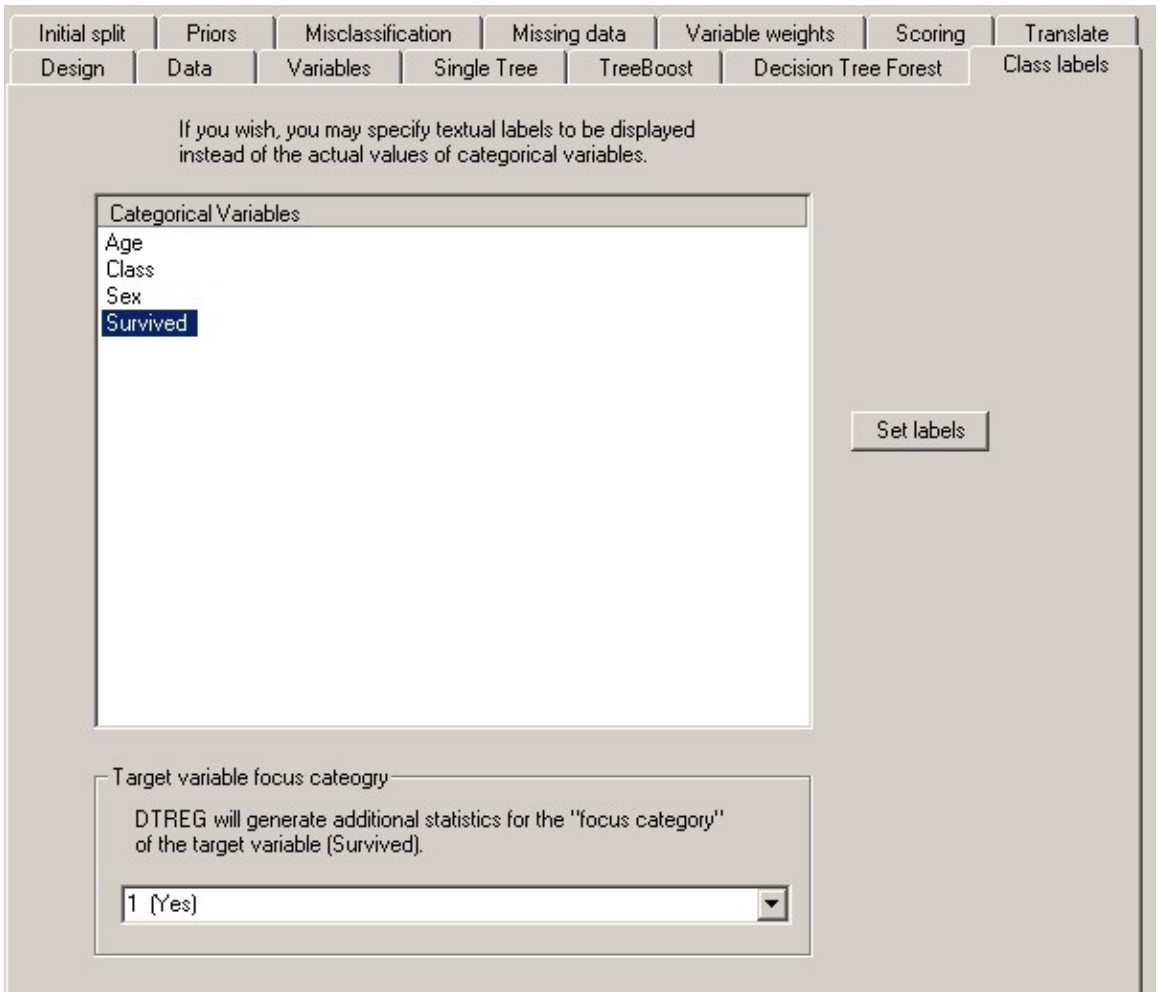
Compute likelihood ratio significance tests – Check this box to request that likelihood ratio significance tests be computed for the parameters. Likelihood ratio significance tests are a more accurate method of accessing which parameters are significant in the model than the usual Wald significance tests. 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.

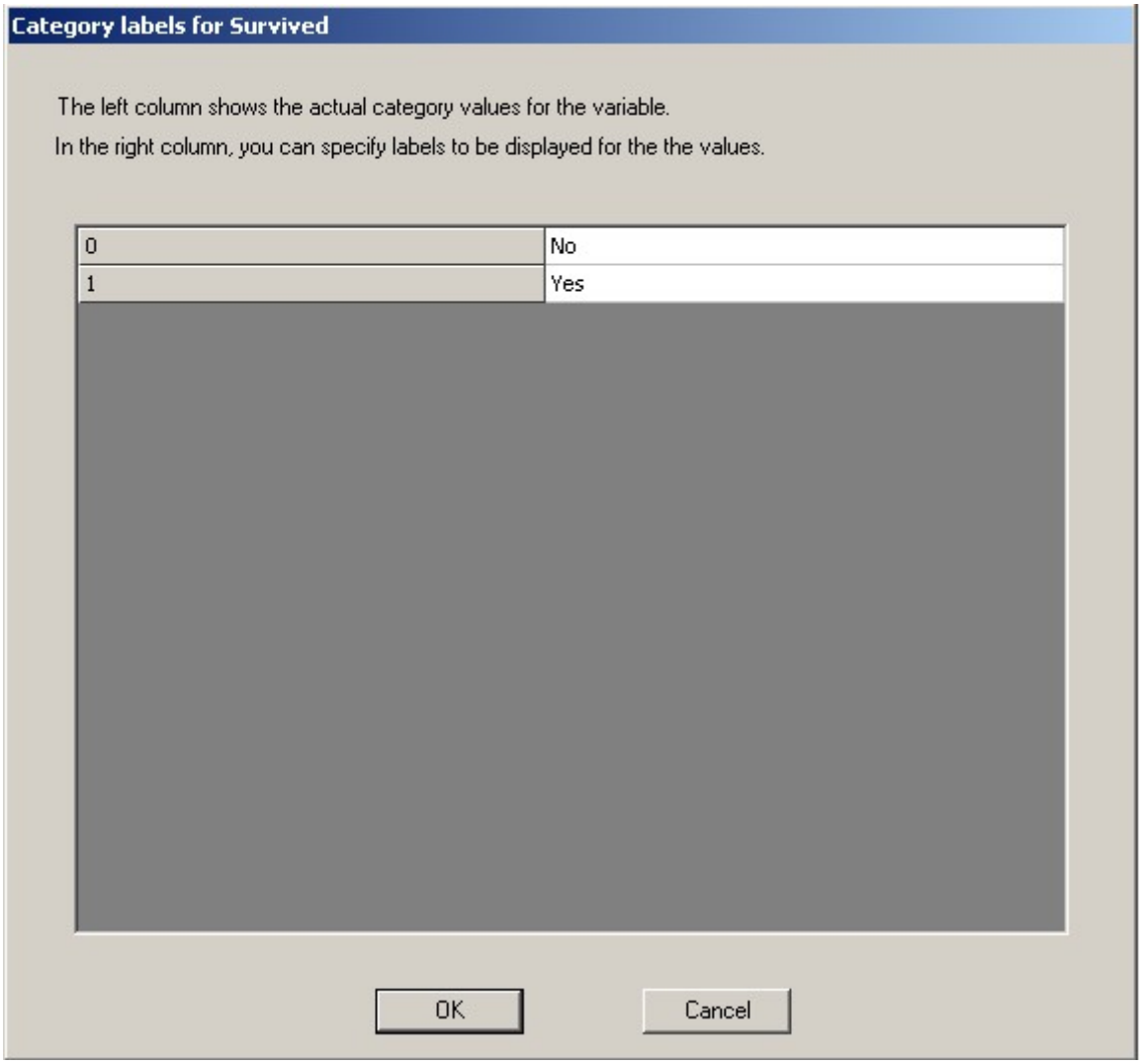
Compute importance of variables – If you select this option, DTREG will compute the relative importance of each predictor variable and display it in the analysis report.

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".

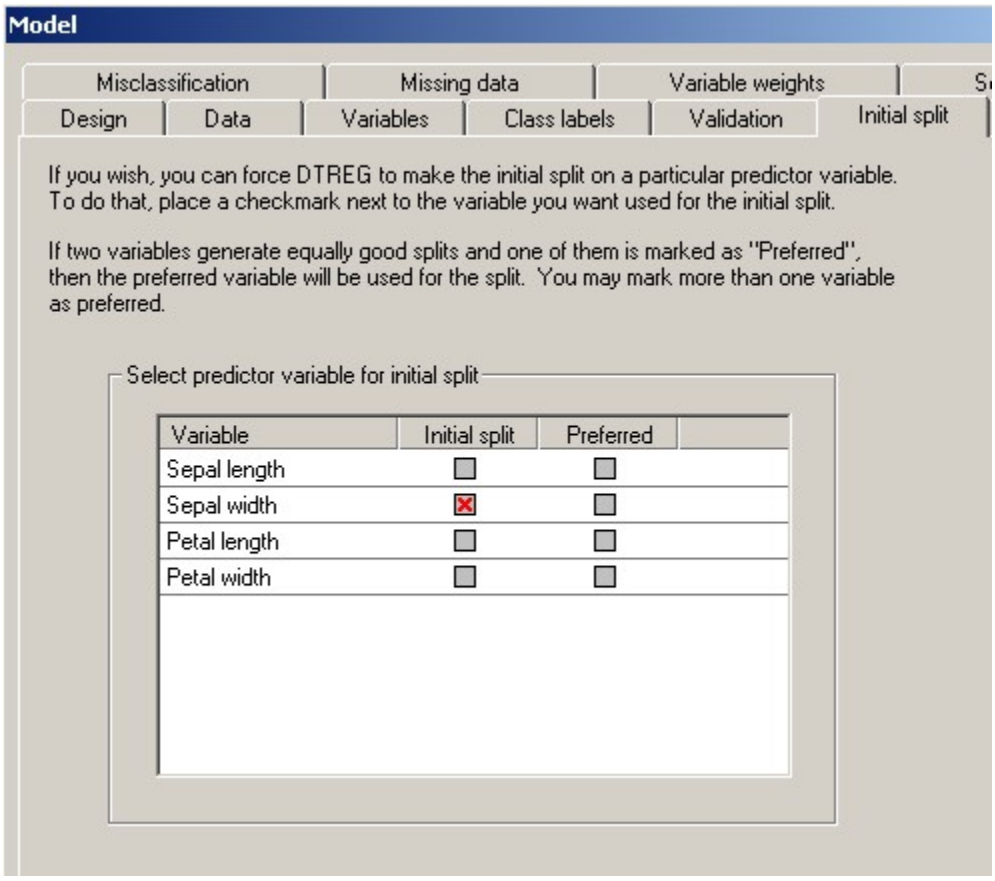
Assigned label strings are used for esthetic purposes only, and they have no effect on the generation of the decision tree, and class labels are *not* written to the output file when data is scored.

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.



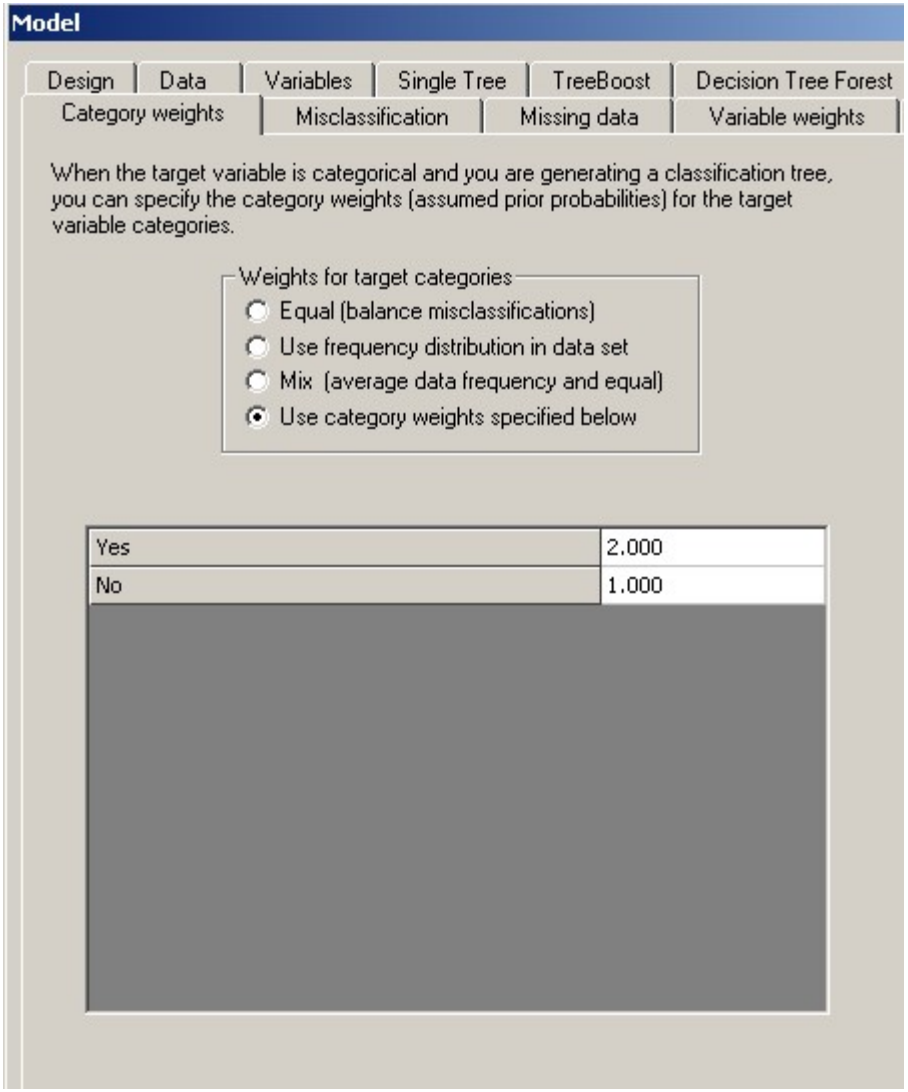
The name of each predictor variable will be shown in the list. Next to the variable names are two columns:

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.

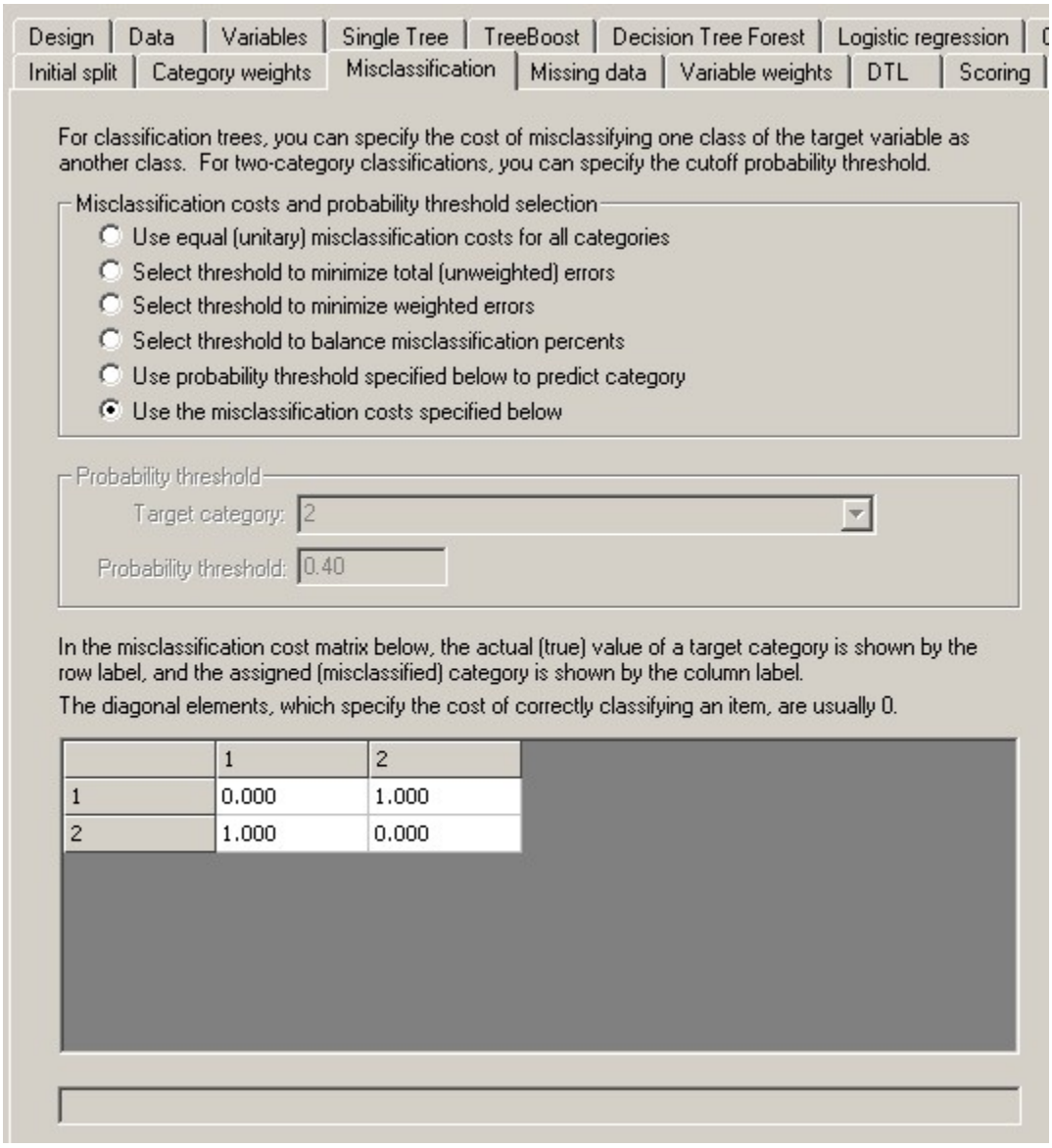
Use frequency distribution in data set – If you select this option, DTREG will compute the distribution of the categories of the target variable in the training dataset and use those proportions as the category weights. If the training sample was drawn at random from the whole population, and the category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

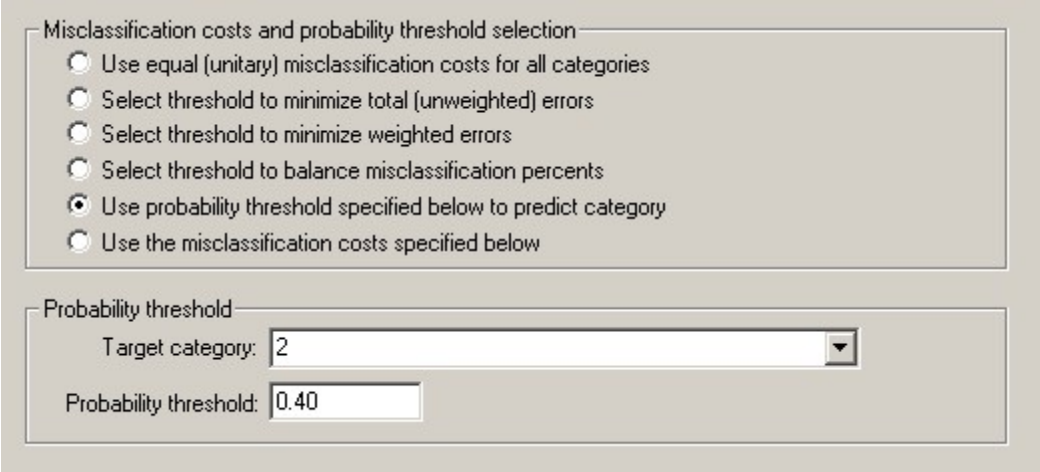
Use equal (unitary) misclassification costs for all categories – If you select this option, DTREG will use the same misclassification costs (1.00) for all categories.

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 TreeBoost, Decision Tree Forest, Discriminant Analysis or Logistic Regression 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 TreeBoost, Decision Tree Forest, Discriminant Analysis or Logistic Regression 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 TreeBoost, Decision Tree Forest, Discriminant Analysis or Logistic Regression 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 performing a TreeBoost, Decision Tree Forest, Discriminant Analysis or Logistic Regression analysis. 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

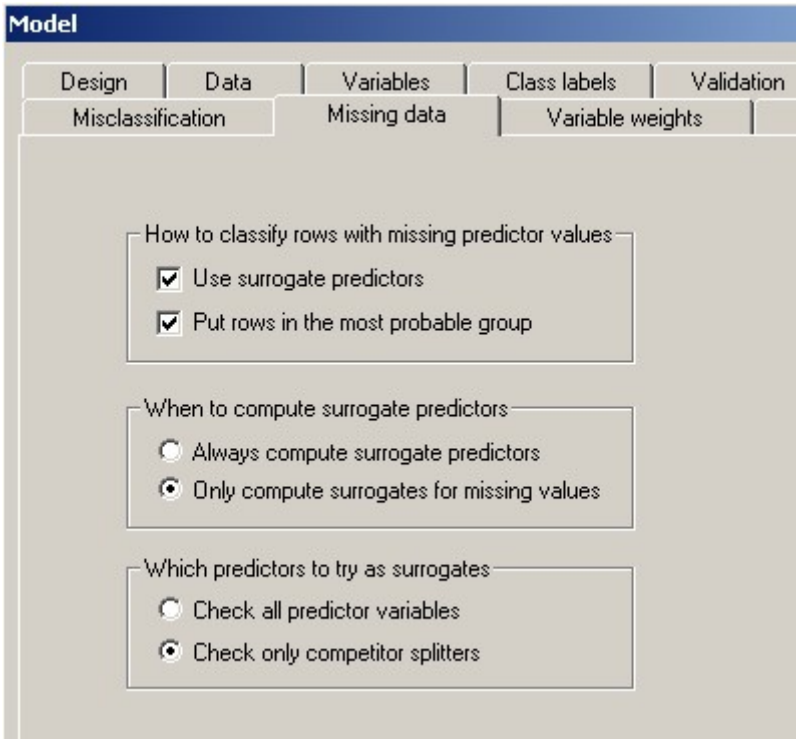
The Probability Threshold Chart described on page 141 and the Probability Threshold Report described on page 119 can be used to determine how a probability threshold will affect the predictions.

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 188.

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.

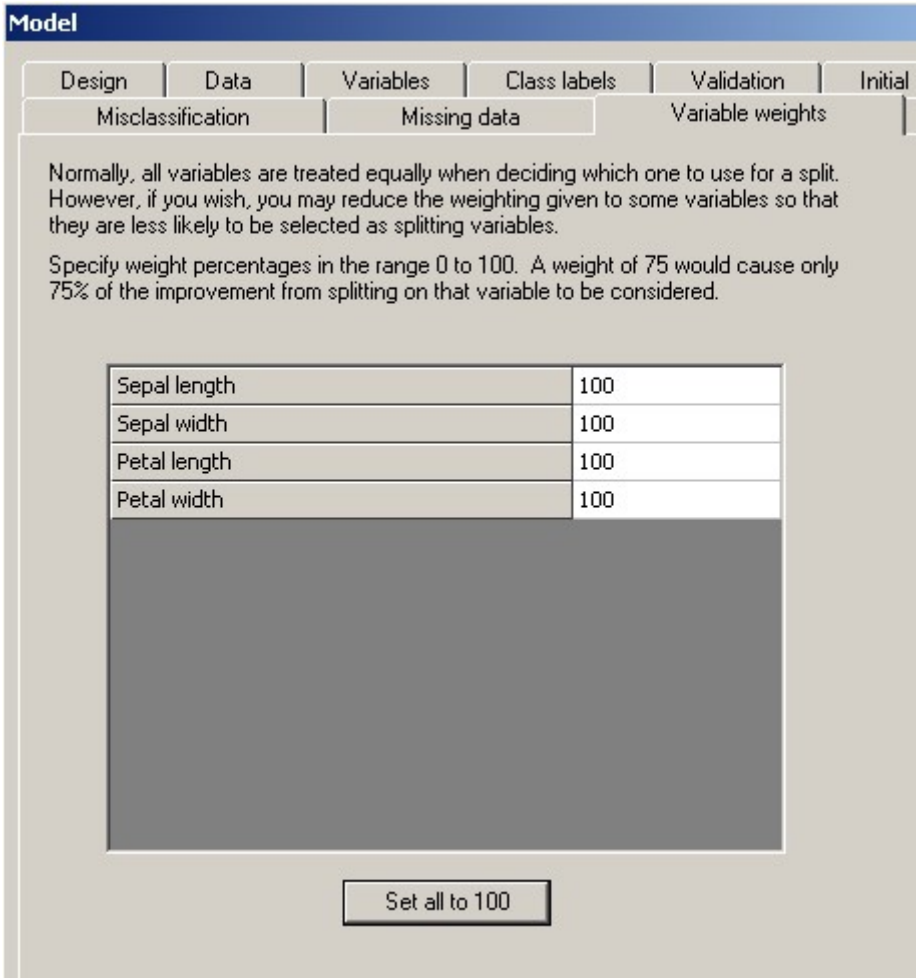
2. The association values assigned to surrogate predictors are used as a component in calculating the overall importance of variables. So if surrogate splitters are not calculated, the overall importance scores will be less accurate.

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

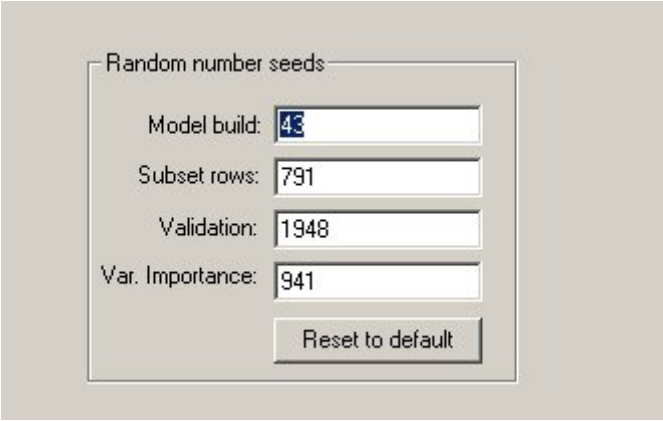
There are several reasons why you might want to use weighting to reduce the likelihood of splitting on a variable:

1. The variable may be difficult or expensive to obtain, so you don't want to have it enter the decision tree 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.

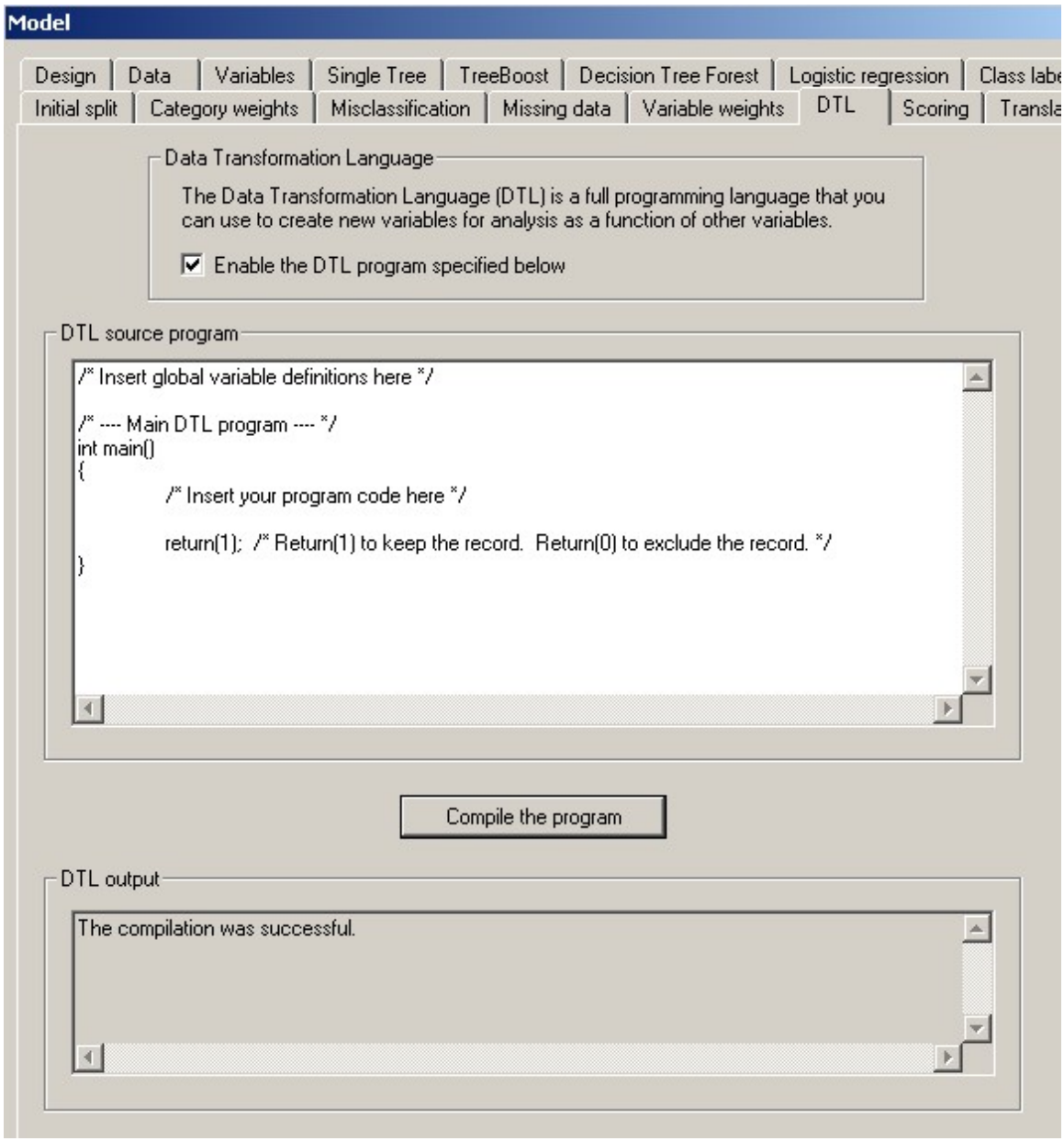
Validation – This random number generator is used to select the rows that go into cross validation folds.

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

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.

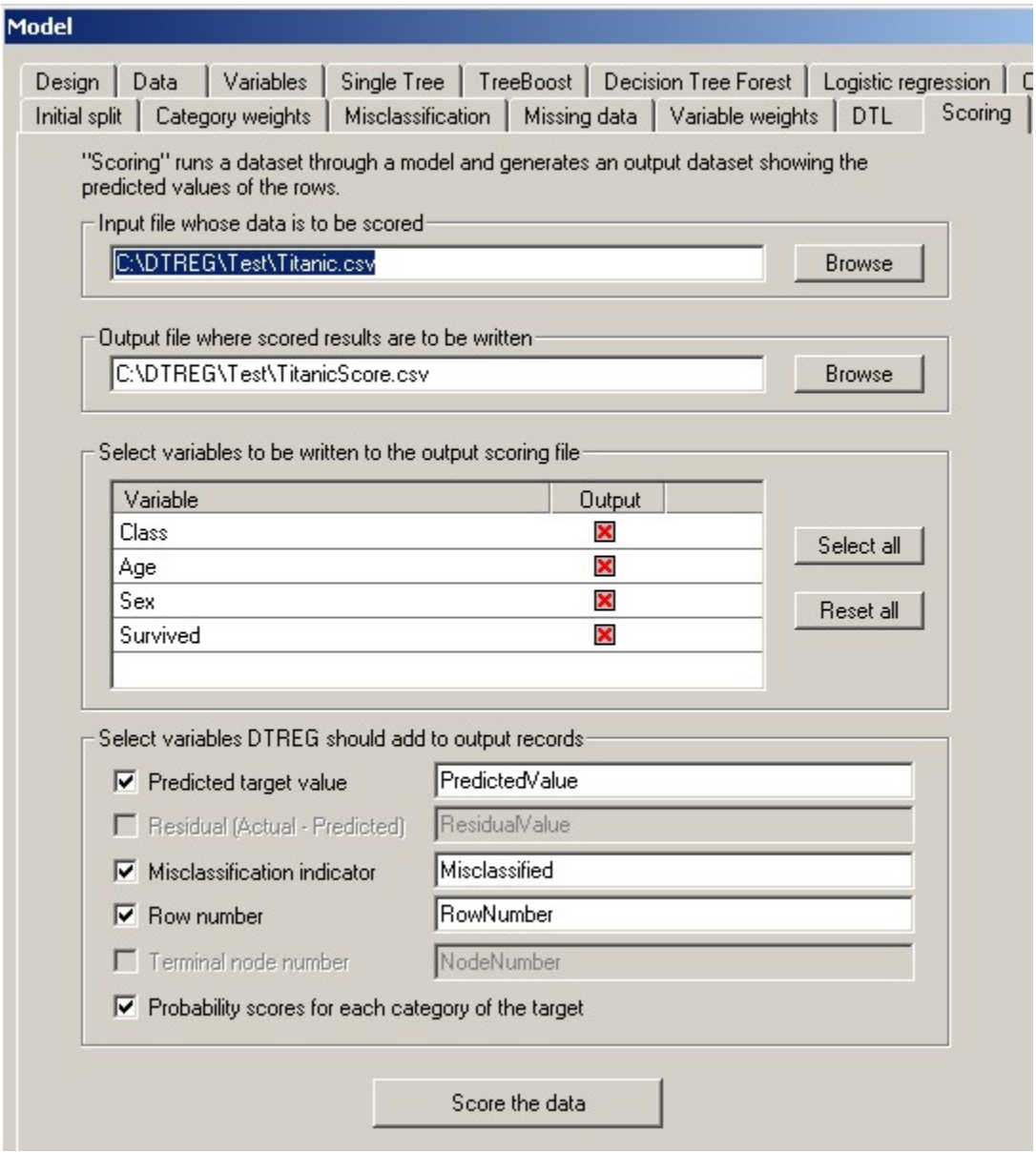
In the following example, the DTL program opens an output file in the StartRun() function, writes information about each data record in the main() function and closes the file in the EndRun() function. Note the use of a static global variable to store the file handle number between iterations.

```
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 decision tree 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 decision tree. This could be the same data file that was used to generate the tree, or it could be some other file for which you wish to use a decision tree to predict values.

The input data file must have the same format as an input file used to build a decision tree:

- 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 35).
- 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 35).
- 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** – When a TreeBoost, SVM model with probability estimates enabled, Discriminant Analysis or Logistic Regression classification model is created, DTREG computes a 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. When you build a decision tree forest model, probability scores are not computed. Instead, DTREG uses the proportion of votes of the trees in the forest as a pseudo-probability value for each target category.

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 188.

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 74.) 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 203 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.

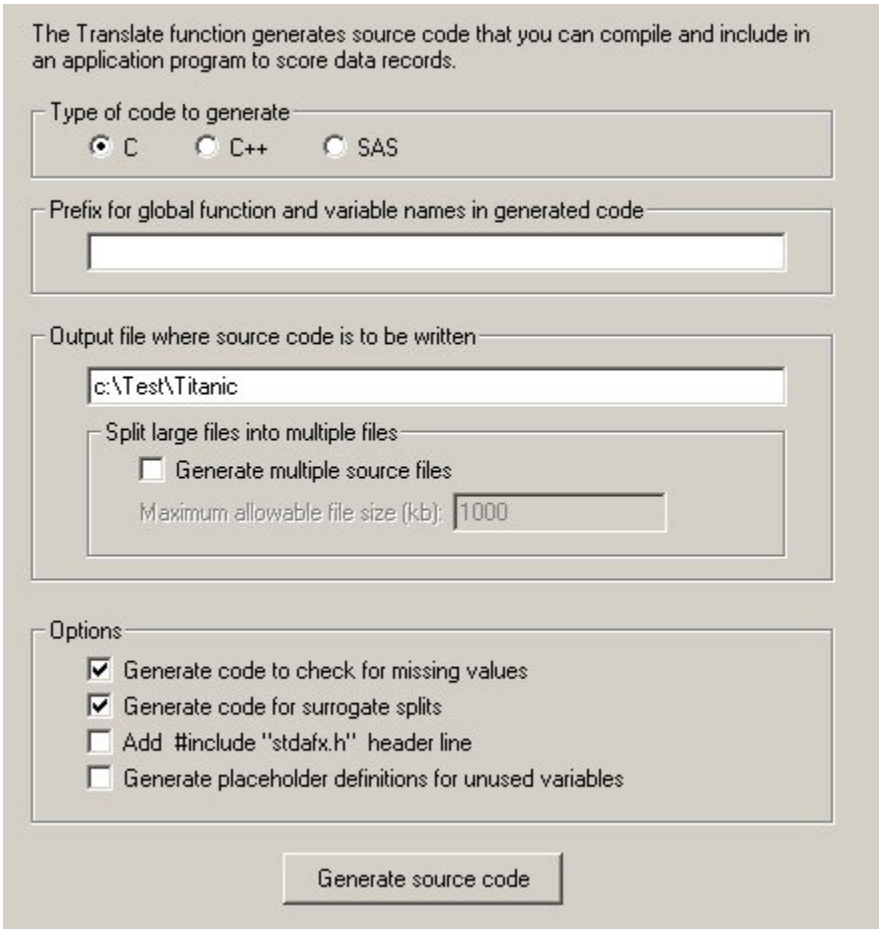
The Translate function is available only in the Enterprise Version of DTREG.

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 decision tree 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 188 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
If scoring code is generated for a categorical, TreeBoost, SVM, Discriminant Analysis or Logistic Regression model, there will be references to external 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. If a decision tree forest model is generated, the Prob_*category* variables will have the proportion of the votes for each category.

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 38) 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. The values are 1=Single Tree, 2=TreeBoost, 3=Decision Tree Forest, 4=Logistic Regression, 5=Discriminant Analysis.

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, TreeBoost, SVM, Discriminant Analysis or Logistic Regression 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:

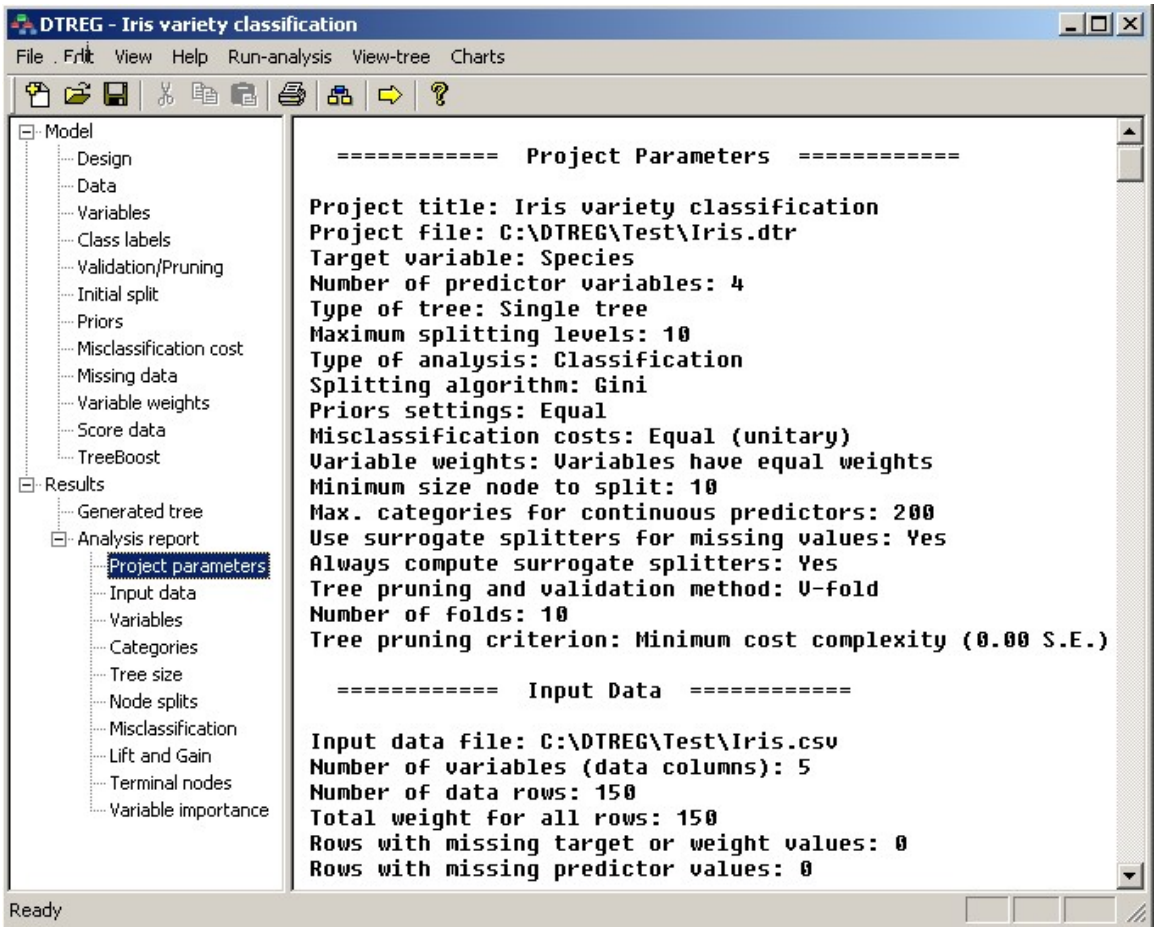
```
LINK ScoreRecord;
```

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  ============

Species
     50   33.33%  Setosa
     50   33.33%  Versicolor
     50   33.33%  Virginica
Sepal length
      1    0.67%  4.3
      3    2.00%  4.4
```

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 38).

Three columns 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.

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 190.

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 193 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 195 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.

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 188.

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%)
```

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%).

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 193 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.

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 65). 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 134 and 135).

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.

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 and a TreeBoost, decision tree forest, SVM, discriminant analysis or logistic regression model is generated. A graphical depiction of the probability threshold response is available in the Probability Threshold Chart described on page 141.

Classification methods such as TreeBoost, Decision Tree Forest, 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. In the case of Decision Tree Forest models where an ensemble of trees "vote" on the category, the proportion of votes for each category can be used as an approximate likelihood measure (although it is not a true probability). Support Vector Machine (SVM) models also 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 71.

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 chart – This is the area under the Receive Operating Characteristic (ROC) curve for the model. Sometimes this statistic is known as "AUC". 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.

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 and Gain  ============

  ---  Lift and Gain for training data  ---

Lift/Gain for Survived = Yes

 Bin   Class %     Cum %      Cum %     Cum      % of       % of
Index  of bin   Population  of class   Gain   Population   Class   Lift
-----  -------  ----------  --------  ------  ----------  ------  ------
   1     96.38     10.04      29.96     2.98     10.04     29.96    2.98
   2     59.28     20.08      48.38     2.41     10.04     18.42    1.83
   3     33.03     30.12      58.65     1.95     10.04     10.27    1.02
   4     17.65     40.16      64.14     1.60     10.04      5.49    0.55
   5     26.24     50.20      72.29     1.44     10.04      8.16    0.81
   6     20.81     60.25      78.76     1.31     10.04      6.47    0.64
   7     28.05     70.29      87.48     1.24     10.04      8.72    0.87
   8      7.69     80.33      89.87     1.12     10.04      2.39    0.24
   9     23.53     90.37      97.19     1.08     10.04      7.31    0.73
  10      9.43    100.00     100.00     1.00      9.63      2.81    0.29

   Average gain = 1.612
   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.

The basic idea of lift and gain is to sort the predicted target values in decreasing order of purity on some target category (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.

The **Lift** value (last column) is calculated by dividing the percent of rows in a bin with the specified target category (column 2) by the overall percent of cases. In the table above, the lift for the first row is calculated as 2.98 = 96.38/32.30.

The **Cumulative Gain** (column 5) is calculated by dividing the cumulative percent of rows with the target category in all bins up to the row (column 4) by the cumulative

percentage of all rows in all bins up to the row (column 3). In the table above, the cumulative gain for the second bin is calculated as 2.41 = 48.38/20.08.

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 2.41 which is calculated by dividing 48.38 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. For single-tree lift/gain charts, it will be the percentage of the cases that end up in the terminal node.

% 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, 29.96% of all the cases with category Yes ended up in the first bin.

Lift – This is computed by dividing the percent of the cases with the specified category ("% of class") in the bin by the percent of the total cases in the bin ("% of population").

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 for TreeBoost and Decision Tree Forest models. In the case of a single-tree model, there is a bin for each terminal node in the tree. 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 trees or the largest numerical values for regression trees.

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. In the case of a single-tree model, the number of bins matches the number of terminal nodes.

1. For a single-tree model, sort the terminal nodes in descending order based on **PredictedTarget**. For a TreeBoost or Decision Tree Forest model, sort the data rows in descending order of **PredictedTarget**.

2. For a TreeBoost or Decision Tree Forest model, 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 136 for information about generating lift and gain charts.

Terminal Node Table

The terminal node table displays summary statistics about each terminal node in the tree.

```
  ============  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 74).

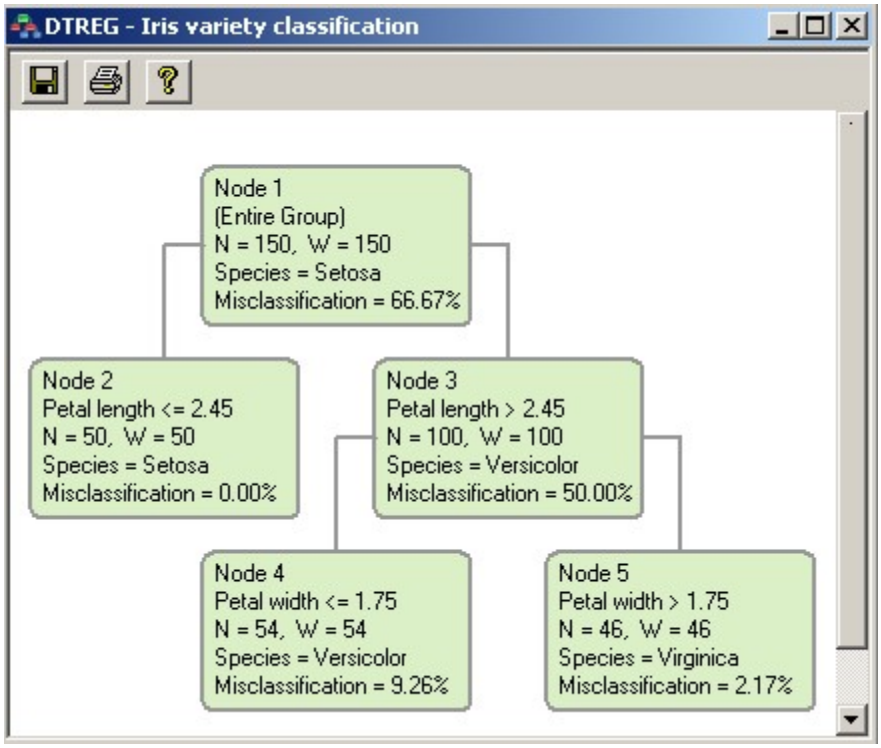
The importance score for the most important predictor is scaled to a value of 100.00. Other predictors will have lower scores. Only predictors with scores greater than zero are shown in the table.

See page 144 for information about displaying a chart of variable importance.

Viewing a Decision Tree

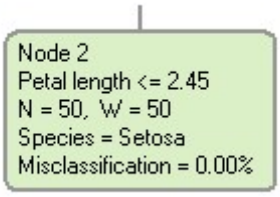
I think that I shall never see a poem lovely as a tree.
– Joyce Kilmer

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:

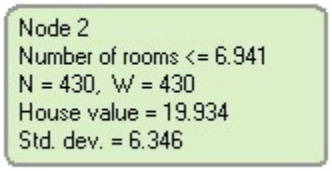


Five lines of information are presented in this node:

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 188 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:



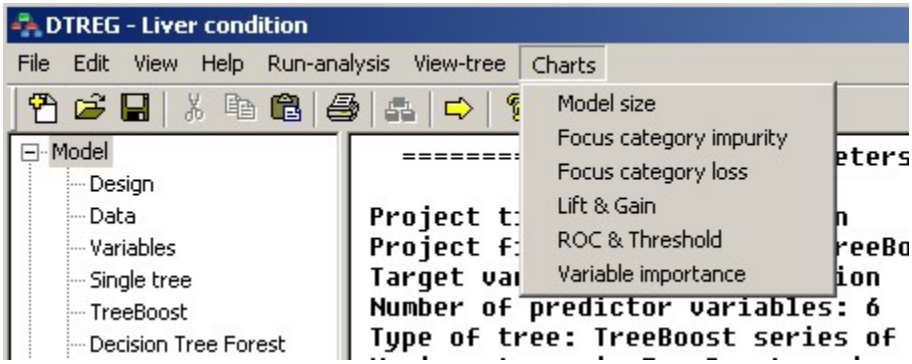
In his example, this node was produced by splitting its parent node on the predictor variable "Number of rooms". There were 430 rows with values of "Number of rooms" less than or equal to 6.941 that were assigned to this node.

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.

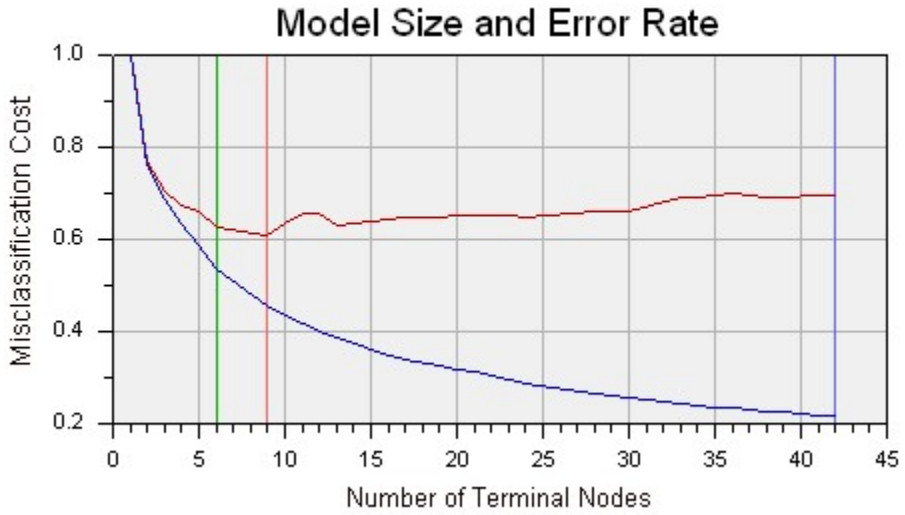
Charts and Graphs

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.

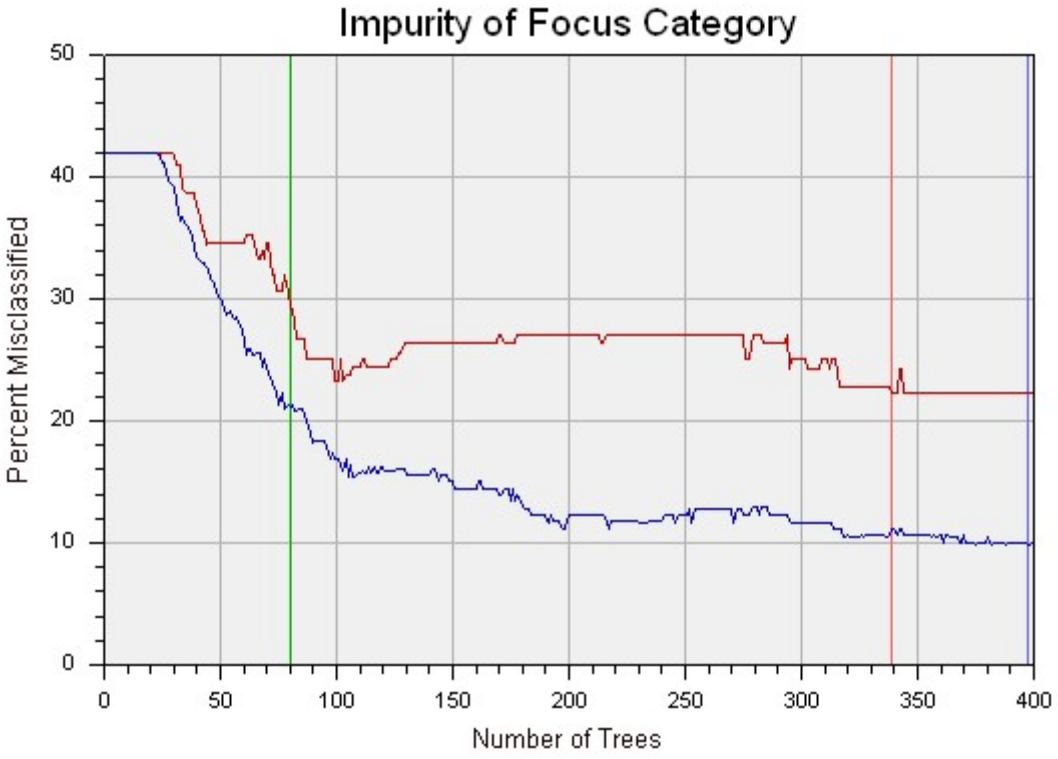
Model Size Chart



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.

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



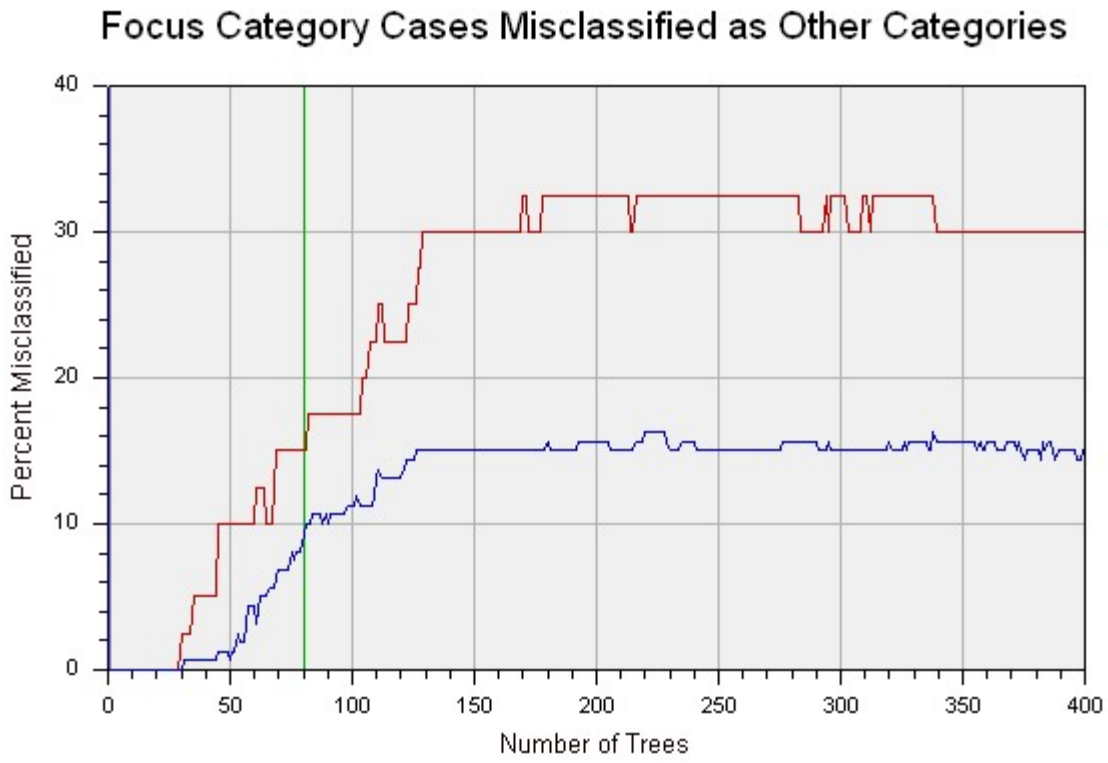
The Focus Category Impurity Chart shows the impurity of the designated focus category of the target variable as a function of the size of the model. For a single-tree model, the model size is the number of terminal nodes in the tree. For a TreeBoost model, the model size is the number of trees in the 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 65).

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.

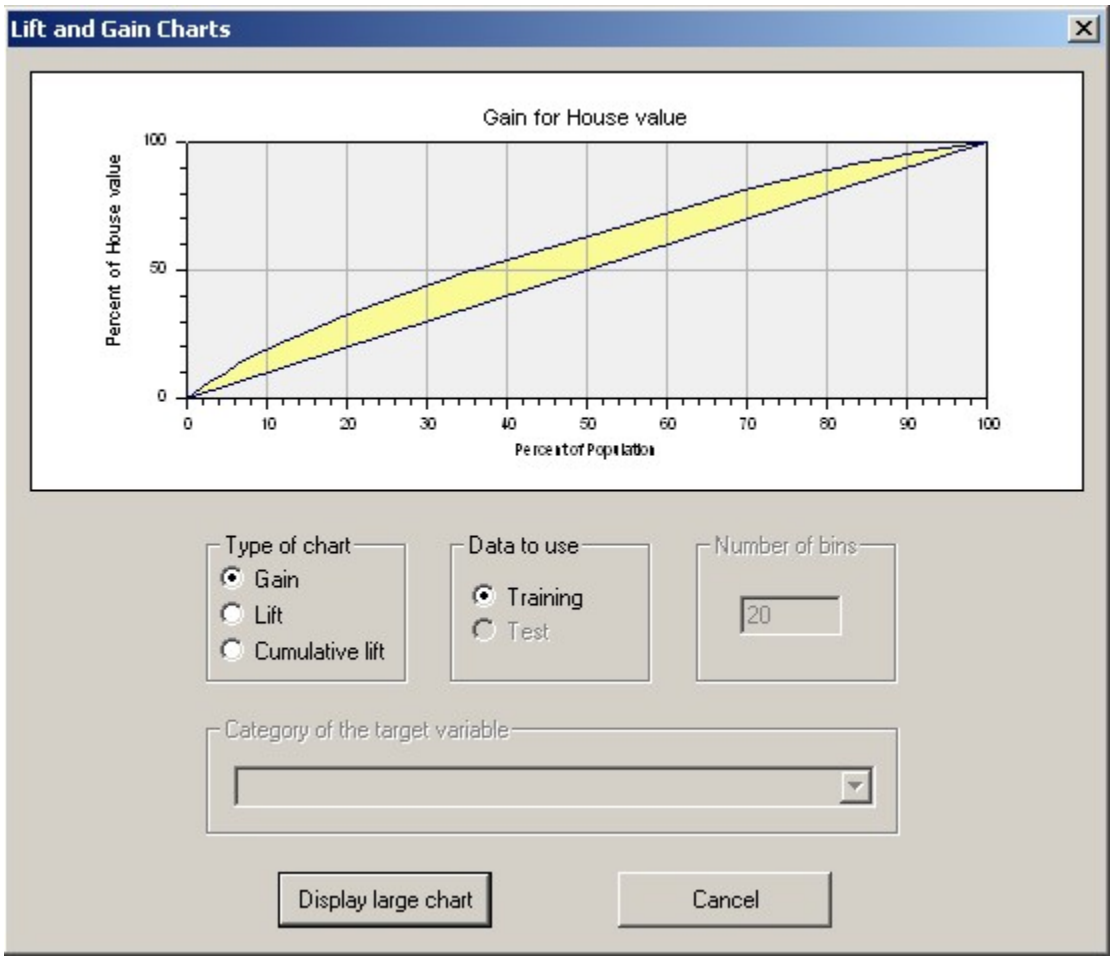
The blue line on the chart represents the loss for the training data. The red line shows the loss for the validation (test) data. A blue vertical line shows the size with the minimum loss on the training data line; a red vertical line shows the size with the minimum loss for the validation data. A green 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 65).

Lift and Gain Chart

When you select the "Lift & Gain" chart item, DTREG displays a screen with options related to these charts. See page 125 for information about how Lift and Gain values are calculated.

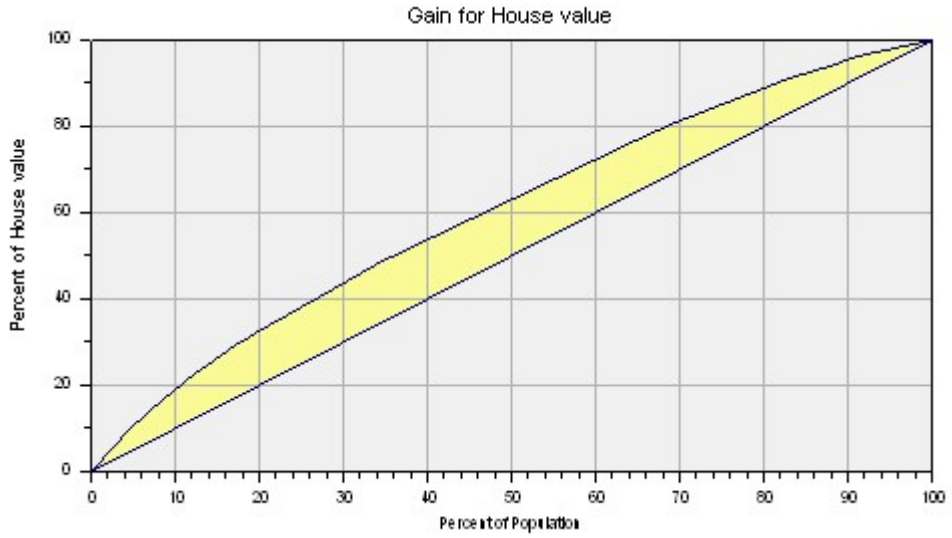


Select the type of chart you want to view (Gain, Lift or Cumulative lift) and the data to be used for the chart (Training or Test). For a TreeBoost or Decision Tree Forest model, you also can select the number of bins to divide the data into. For single-tree models, there is always one bin for each terminal node. For classification models, select which

category of the target variable the lift/gain is to be calculated for. See page 122 for information about how lift and gain values are computed and used.

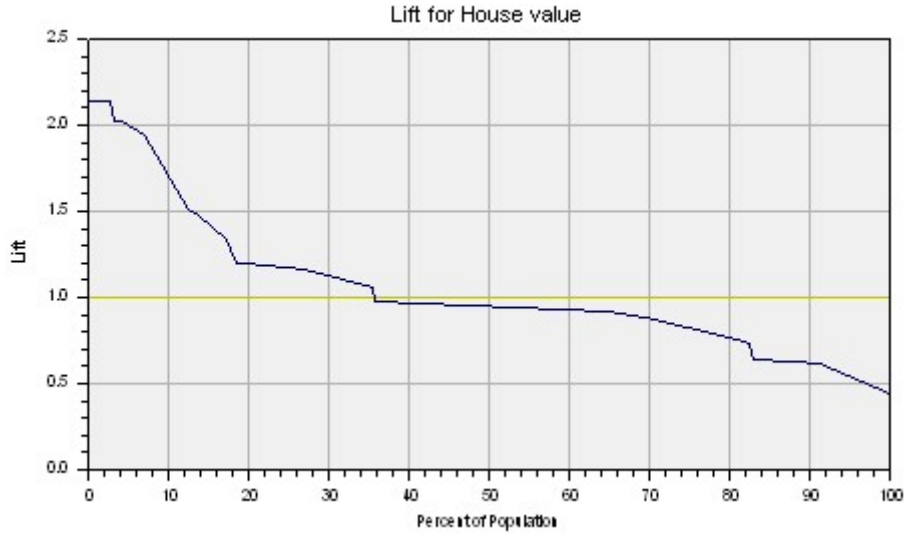
Gain Chart

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.



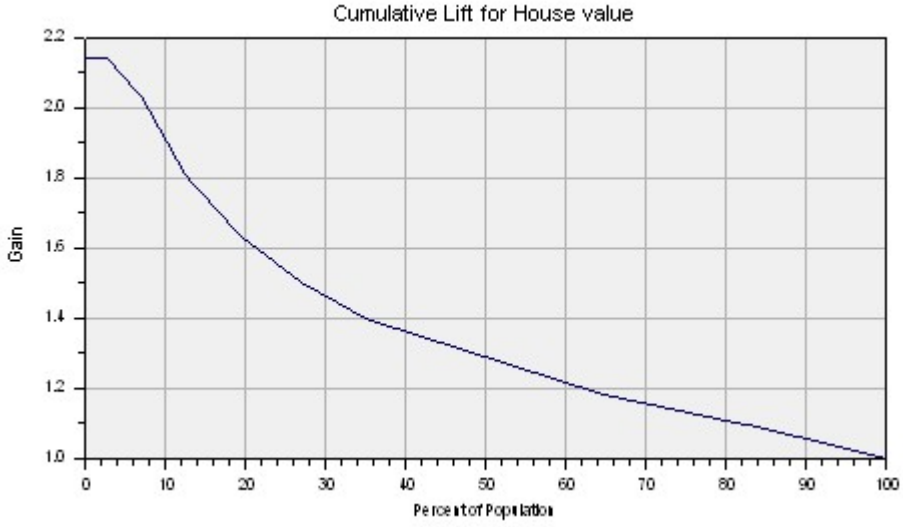
Lift Chart

A lift chart displays the lift for each bin on the vertical axis and the cumulative population on the horizontal axis.



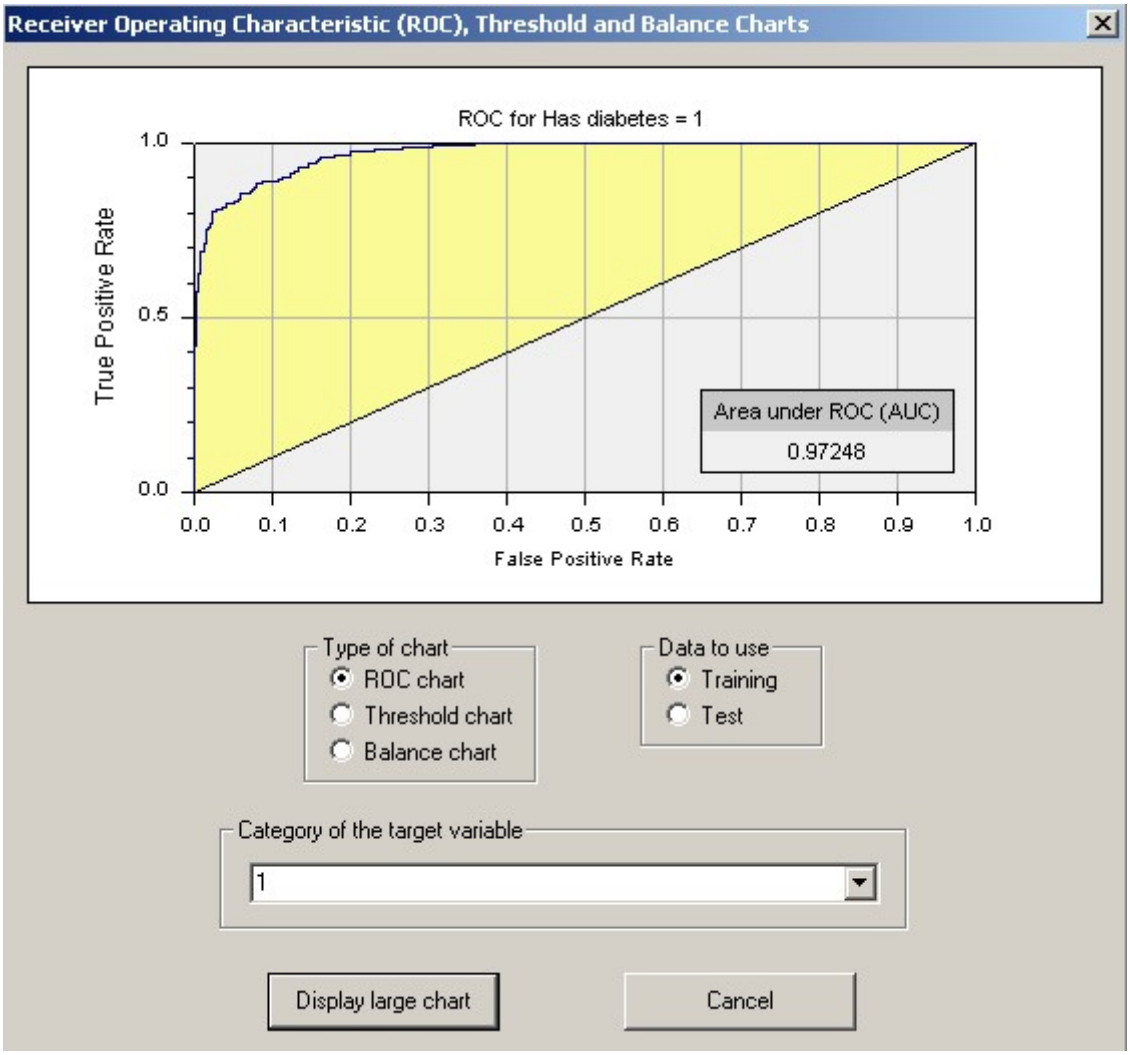
Cumulative Lift Chart

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 run using TreeBoost, Decision Tree Forest, SVM (with the probability option turned on), Discriminant Analysis or Logistic Regression models. ROC charts are not available for regression analyses or for single-tree models.



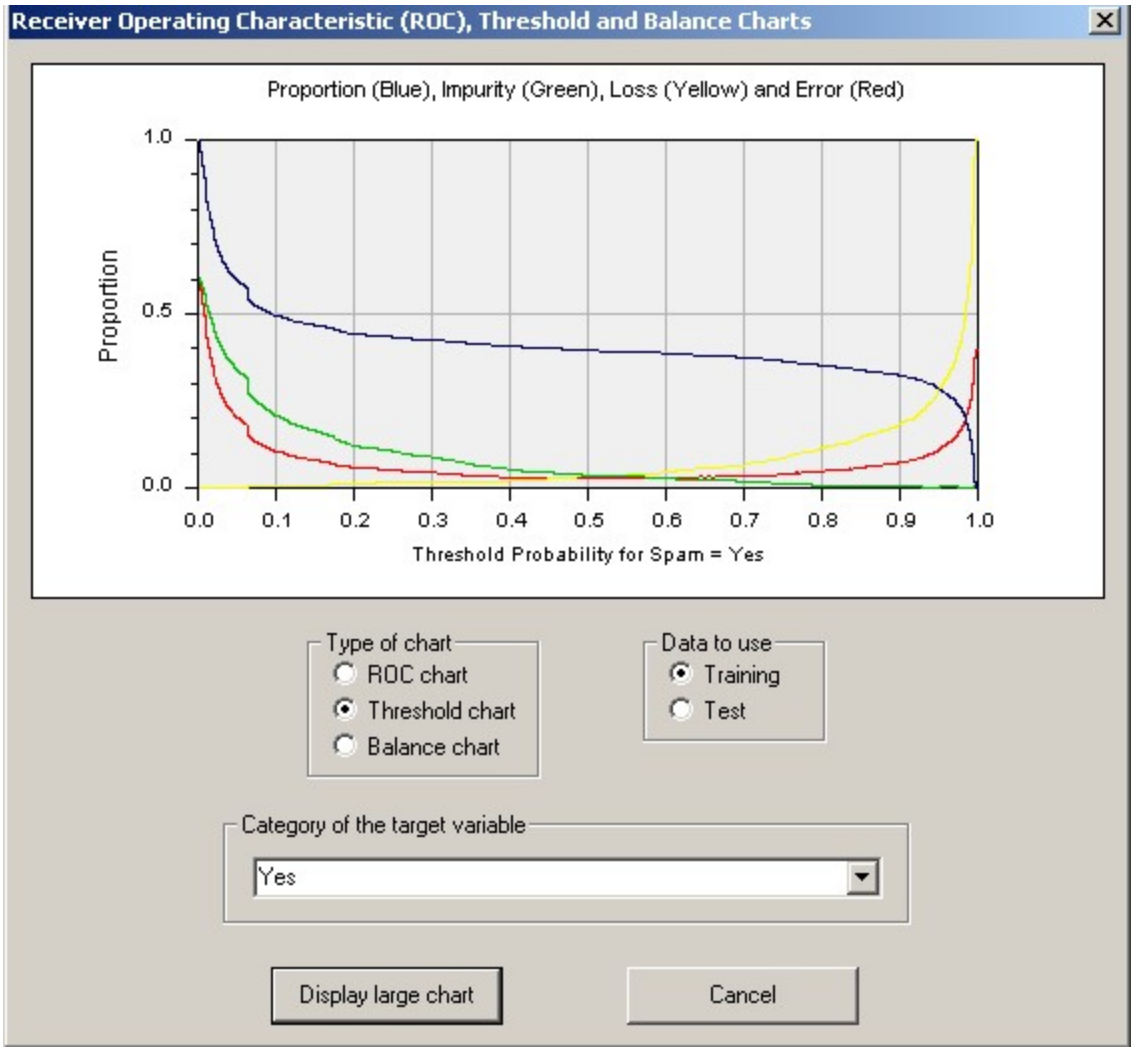
Classification methods such as TreeBoost, SVM, Discriminant Analysis and Logistic Regression not only predict a specific category for each case but also generate probability scores that indicate the relative likelihood for each possible category. Usually the category with the highest probability is selected as the predicted category. In the case of Decision Tree Forest models where an ensemble of trees "vote" on the category, the proportion of votes for each category can be used as an approximate likelihood measure (although it is not a true probability).

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.

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.

Probability Threshold Chart

A Probability Threshold Chart is available when a classification analysis has been run using TreeBoost, Decision Tree Forest, SVM, Discriminant Analysis or Logistic Regression models. Threshold charts are not available for regression analyses or for single-tree models. A table showing the probability threshold response is generated in the analysis report. See page 119 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. In the case of Decision Tree Forest models where an ensemble of trees "vote" on the category, the proportion of votes for each category can be used as an approximate likelihood measure (although it is not a true probability). 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.

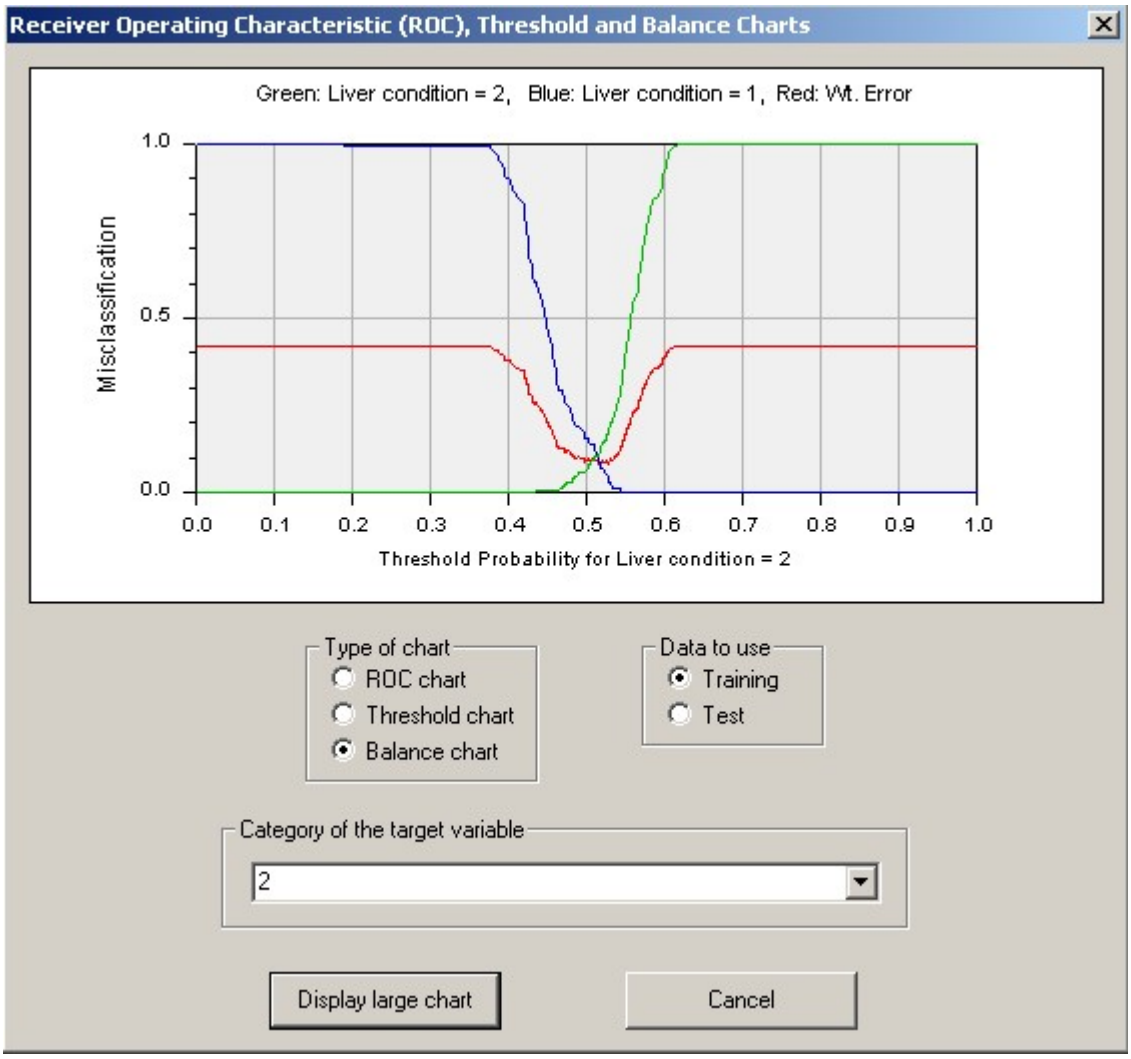
Green line, impurity – The "impurity" is the proportion of cases whose actual (true) category is different than the selected category but which are misclassified as having the target category. In other words, it is the proportion of cases that are given the selected target category that actually belong 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 71.

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 run using TreeBoost, Decision Tree Forest, SVM, Discriminant Analysis or Logistic Regression models. Threshold balance charts are not available for regression analyses or for single-tree models. A table showing the probability threshold response is generated in the analysis report. See page 119 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:

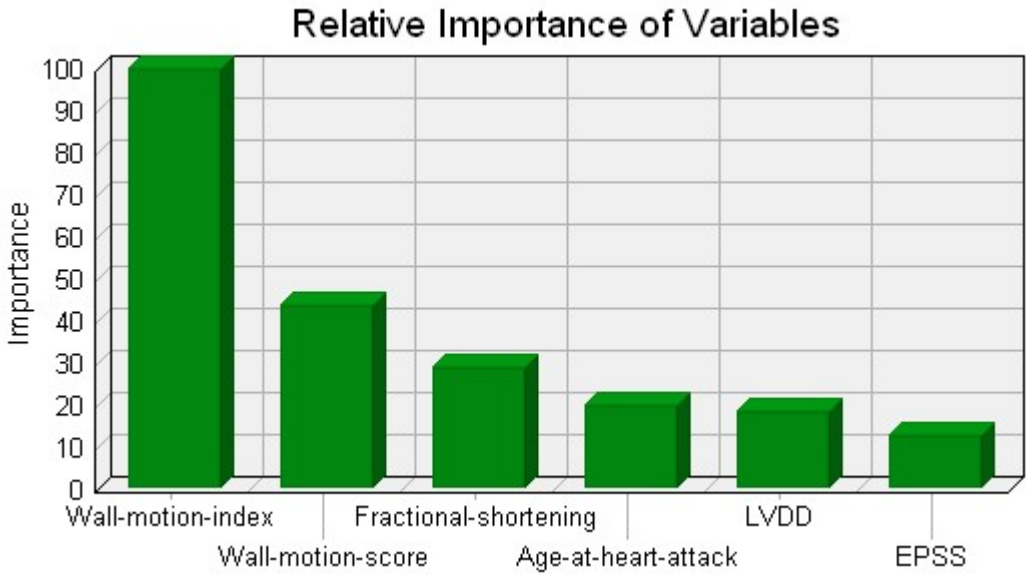
Green line – Proportion of cases misclassified for one of the target categories.

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.

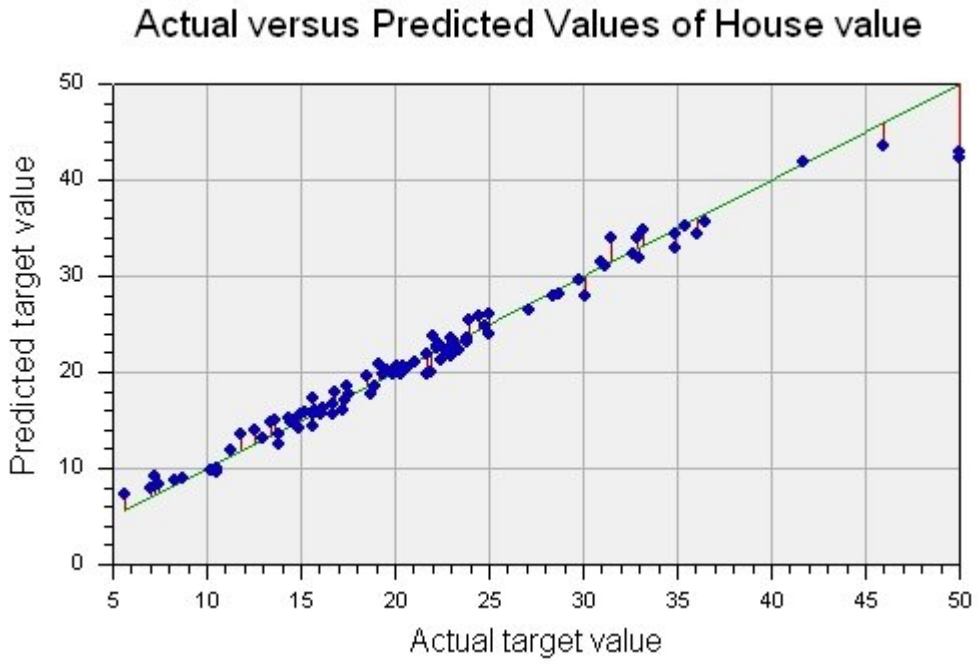
Variable Importance Chart

The Variable Importance chart is a bar chart showing the relative importance for the 10 most important variables.



Actual versus Predicted (Residual) 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.



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 43 for the TreeBoost property page where you select TreeBoost models and set parameters.

One of the original boosting algorithms, called "AdaBoost.M1," was developed in 1997 by Freund and Schapire. This algorithm has been studied extensively, and it has shown promise when applied to neural networks.

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.
- TreeBoost procedures are invariant under all (strictly) monotone transformations of the predictor variables. So transformations such as ($a*x+b$), log(x) or exp(x) do not affect the model. Hence, there is no need for input transformations.
- 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 43).

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).[1]

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.

[1] "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 Overfitting or Pruning

"Overfitting" 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 overfitting, 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 50 for the Decision Tree Forest property page where you select Decision Tree Forest models and set parameters.

Support Vector Machines (SVM)

It's not enough to help the feeble up, but to support him after.
– William Shakespeare

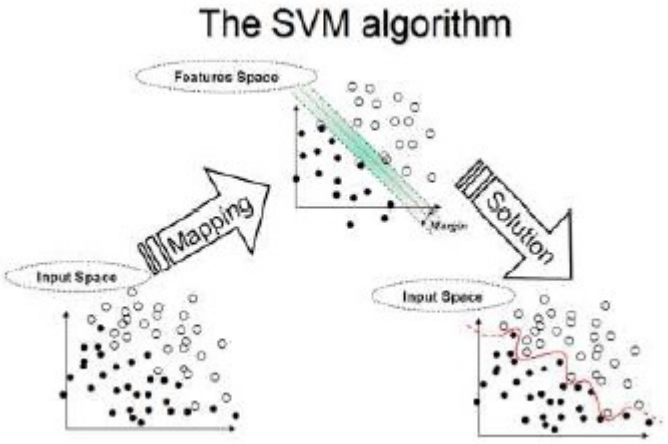
Introduction to Support Vector Machine (SVM) Models

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.

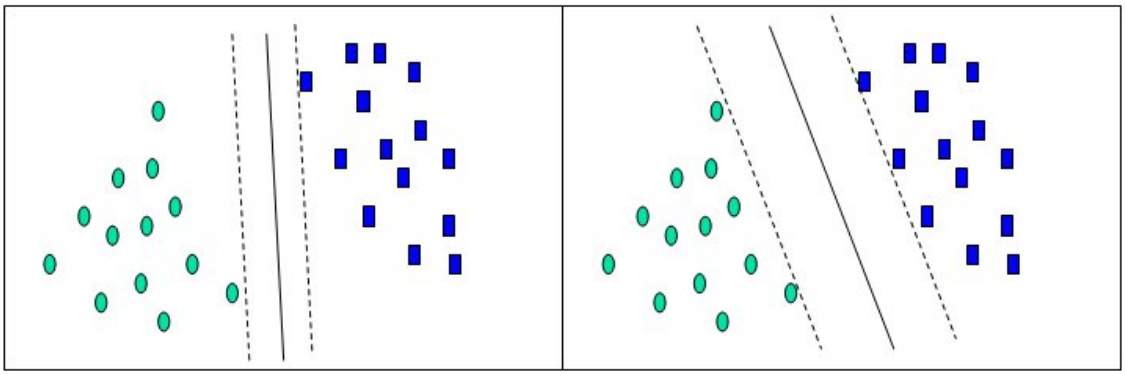
In the parlance of SVM literature, a predictor variable is called an *attribute,* and a transformed attribute that is used to define the hyperplane is called a *feature*. The task of choosing the most suitable representation is known as *feature selection*. A set of features that describes one case (i.e., 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.



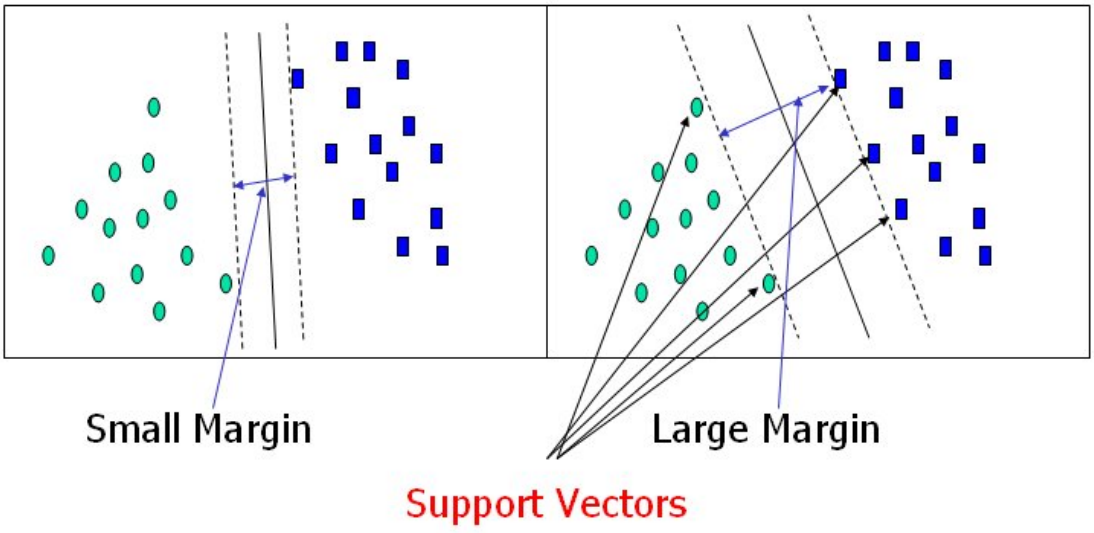
A Two-Dimensional Example

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.

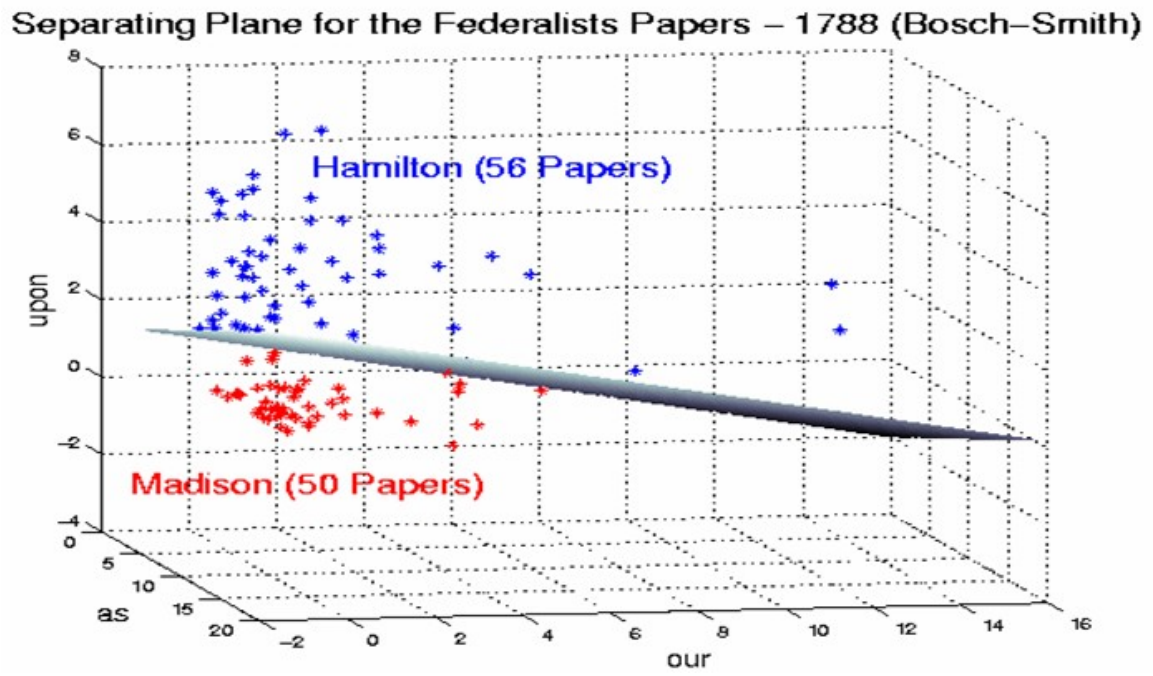


An SVM analysis finds the line (or, in general, hyperplane) that is oriented so that the margin between the support vectors is maximized. In the figure above, the line in the right panel is superior to the line in the left panel.

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

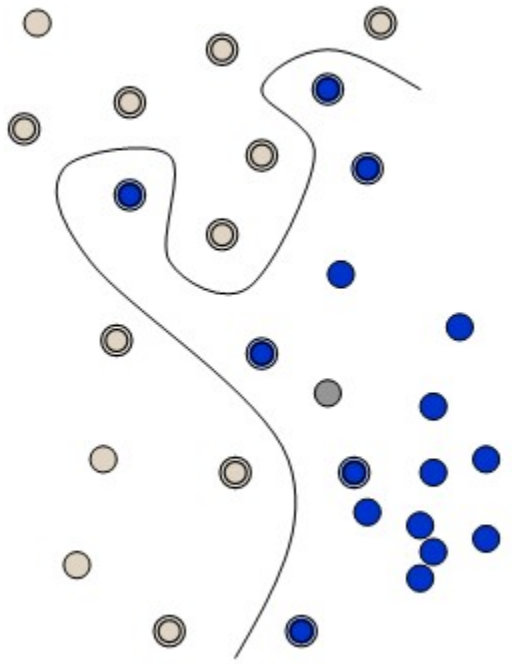
In the previous example, we had only two predictor variables, and we were able to plot the points on a 2-dimensional plane. If we add a third predictor variable, then we can use its value for a third dimension and plot the points in a 3-dimensional cube. Points on a 2-dimensional plane can be separated 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.

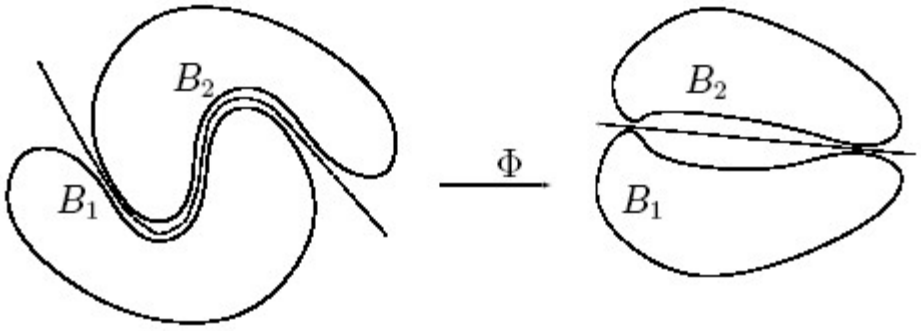
When Straight Lines Go Crooked

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?

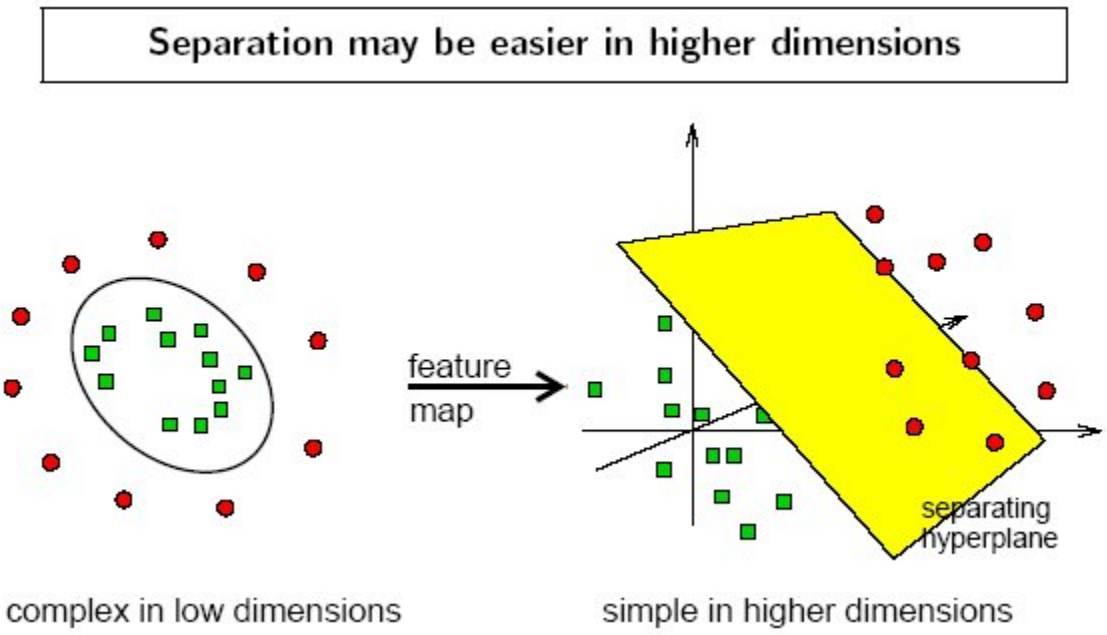


In this case we need a nonlinear dividing line.

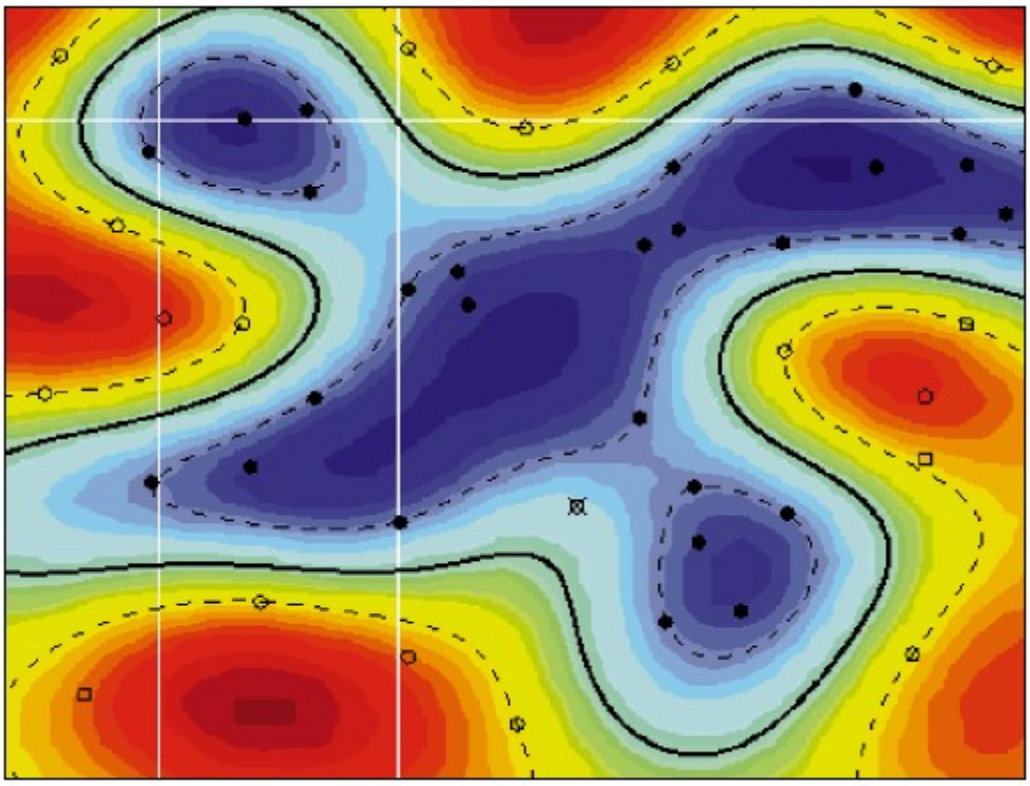
Rather than fitting nonlinear curves to the data, SVM handles this by using a *kernel function* to map the data into a different space where a hyperplane can be used to do the separation.



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.

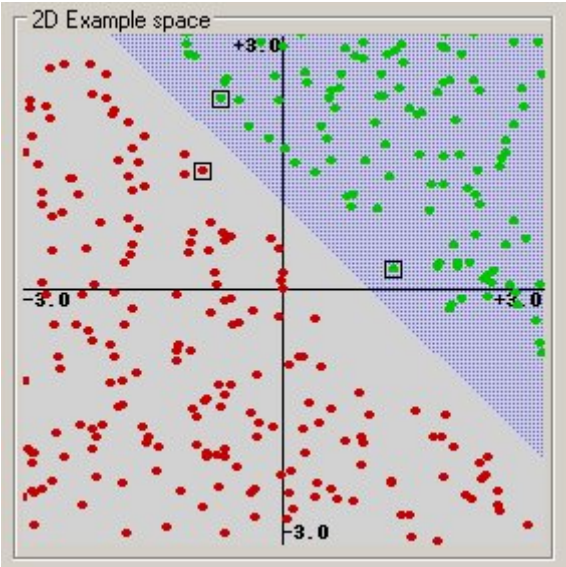


The Kernel Trick

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).

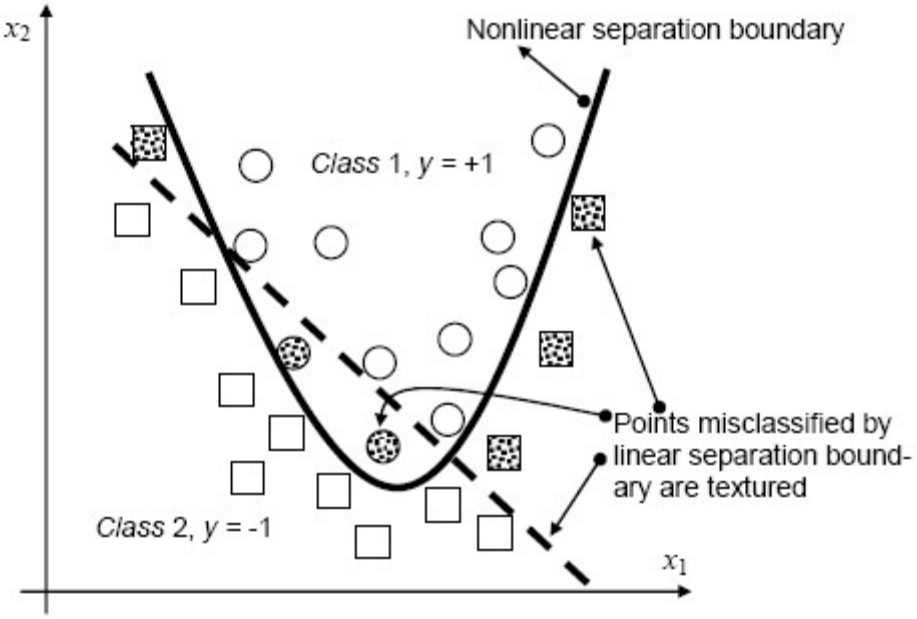
Kernel functions supported by DTREG:

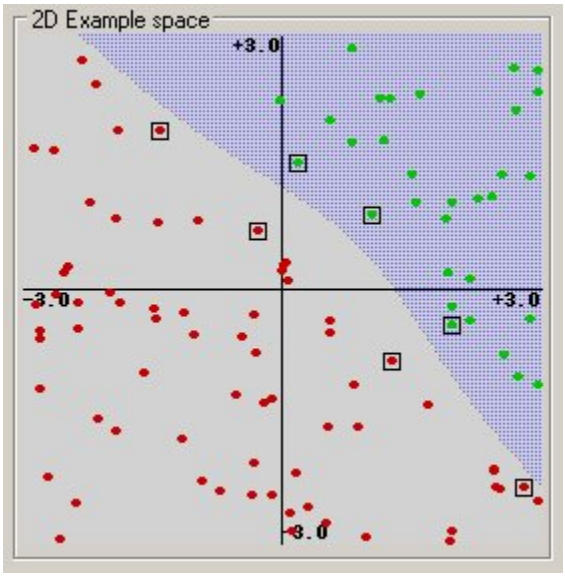
Linear: u'*v



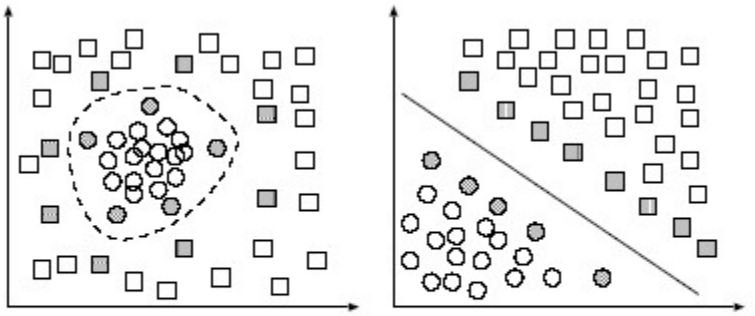
(This example was generated by pcSVMdemo:
http://www.procoders.net/en/Procoders/open_source/pcSVMdemo)

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



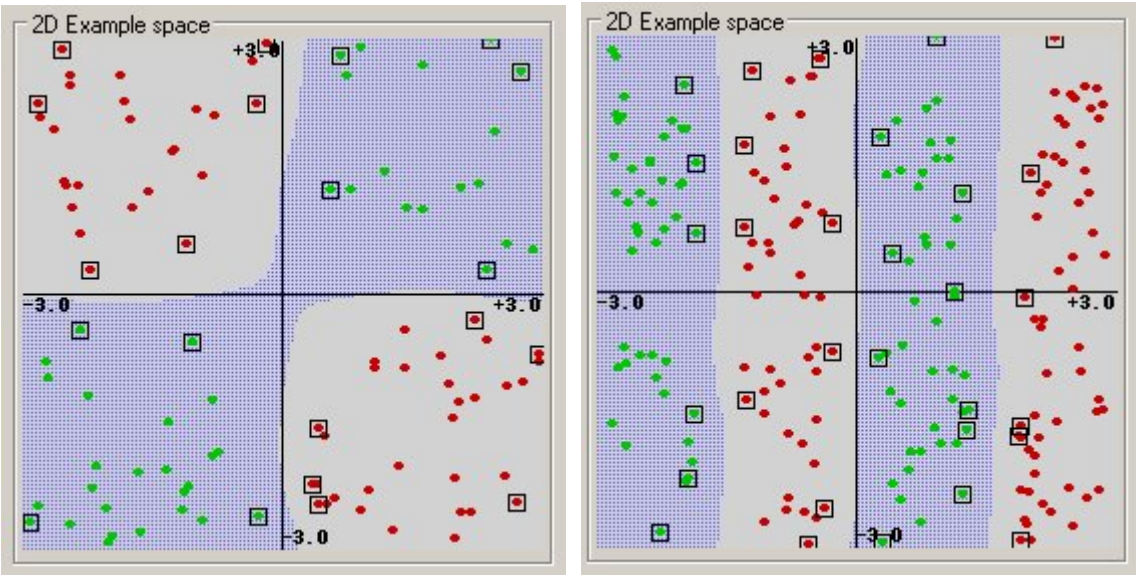


Radial basis function: exp(-gamma*|u-v|^2)
A Radial Basis Function (RBF) is the default and recommended kernel function. The RBF kernel non-linearly maps samples into a higher dimensional space, so it can handle nonlinear relationships between target categories and predictor attributes; a linear basis 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.

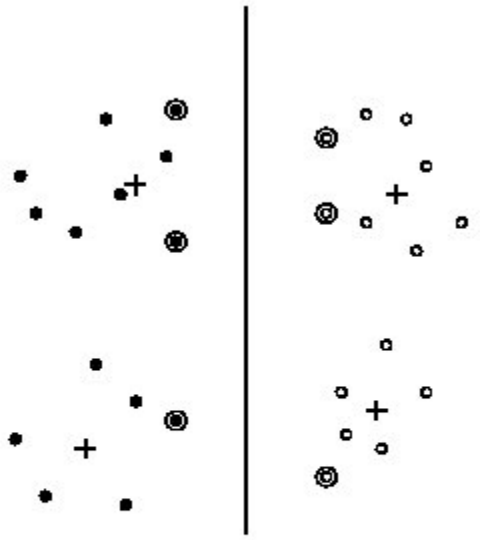


(a) Radial Basis Function (b) RBF mapping

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

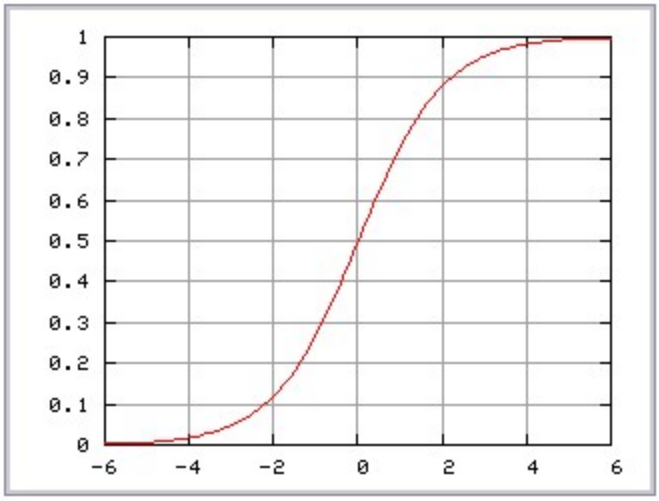


An SVM model using a radial basis function kernel has the architecture of an RBF network. However, the method for determining the number of nodes and their centers is different from standard RBF networks with the centers of the RBF notes on the support vectors (see the figure below from C. Campbell).



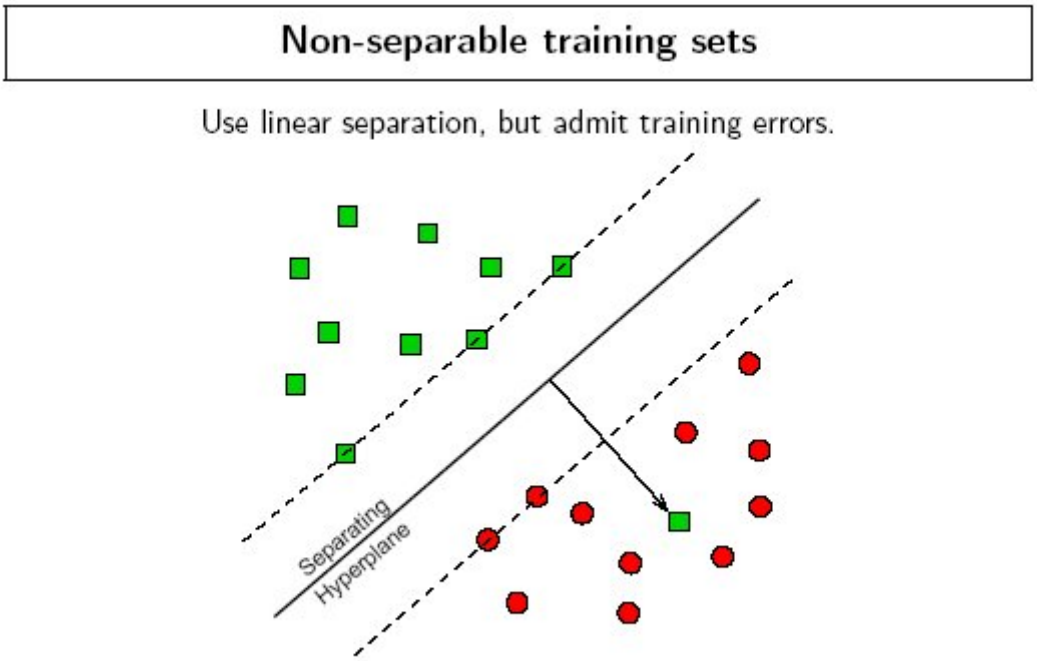
A classical RBF network finds the centers of RBF nodes by k-means clustering (marked by crosses). In contrast an SVM with RBF kernels uses RBF nodes centered on the support vectors (circled), i.e., the datapoints closest to the separating hyperplane (the vertical line illustrated).

Sigmoid (feed-forward neural network): 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[2] 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*.

[2] Technically, *C* is the cost of the sum of the distances of wrong-size points from the margins.

Classification With More Than Two Categories

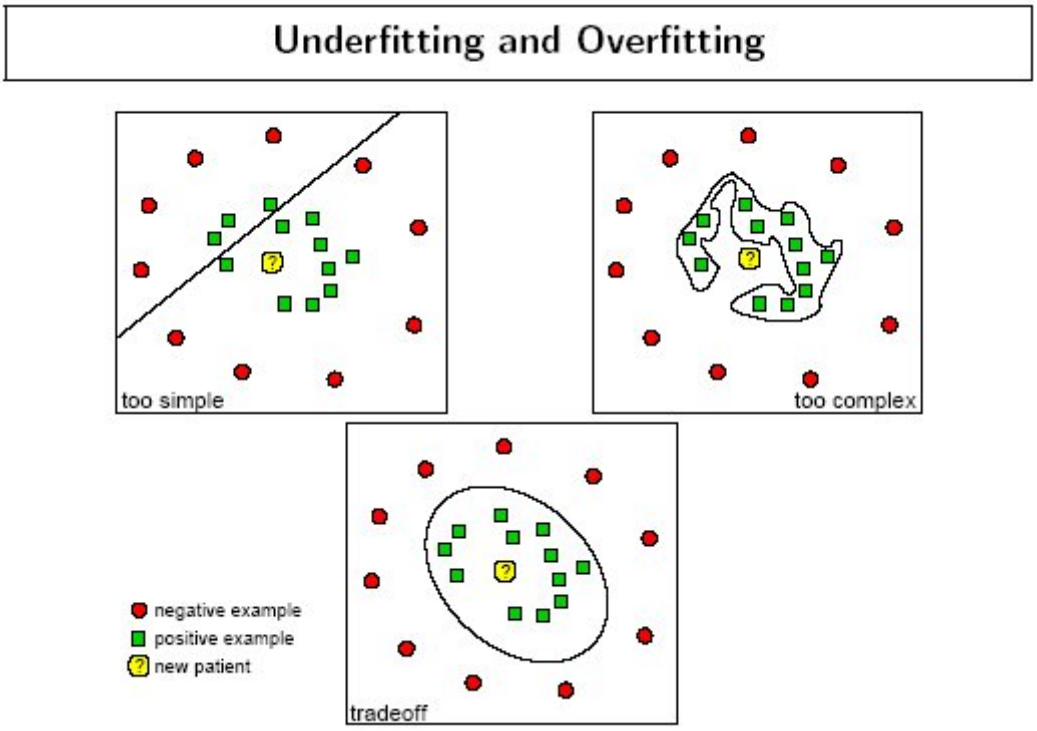
The idea of using a hyperplane to separate the feature vectors into two groups works well when there are only two target categories, but how does SVM handle the case where the target variable has more than two categories? Several approaches have been suggested, but two are the most popular: (1) "one 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:

LIBSVM: Copyright (c) 2000-2005 Chih-Chung Chang and Chih-Jen Lin
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither name of copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

“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.”

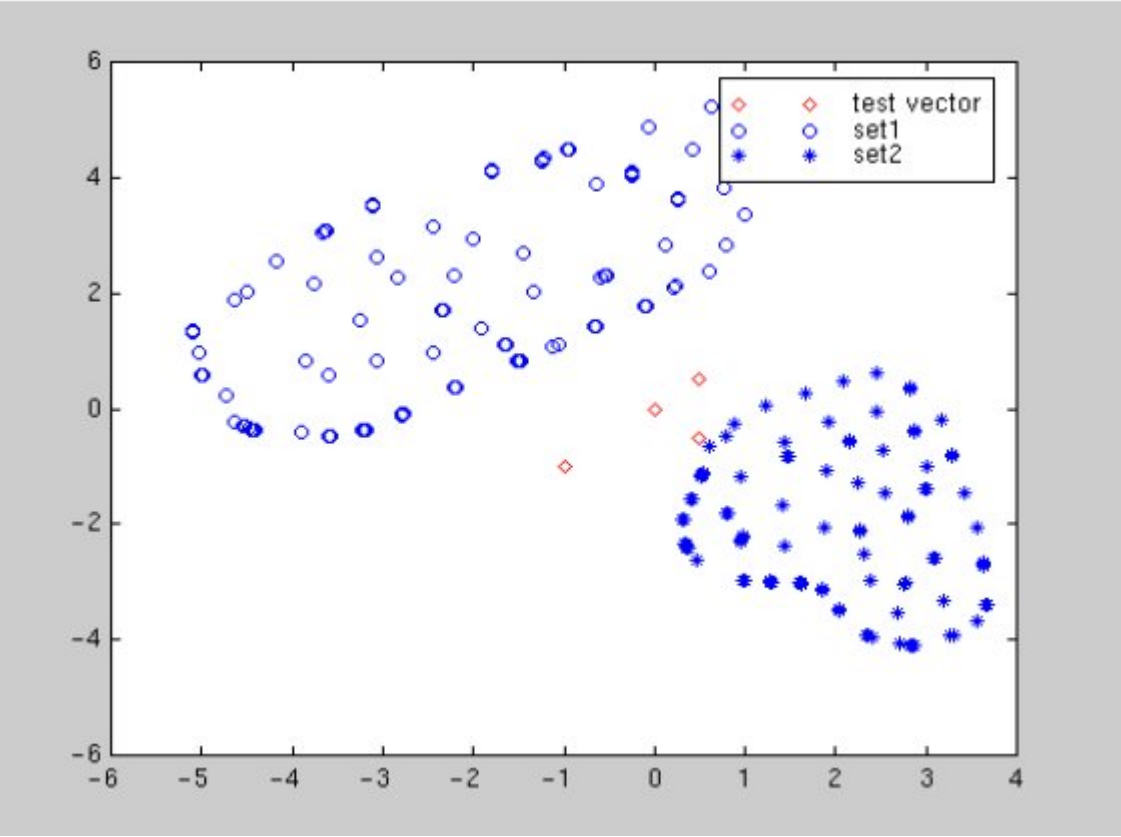
Discriminant Analysis

Introduction to 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:

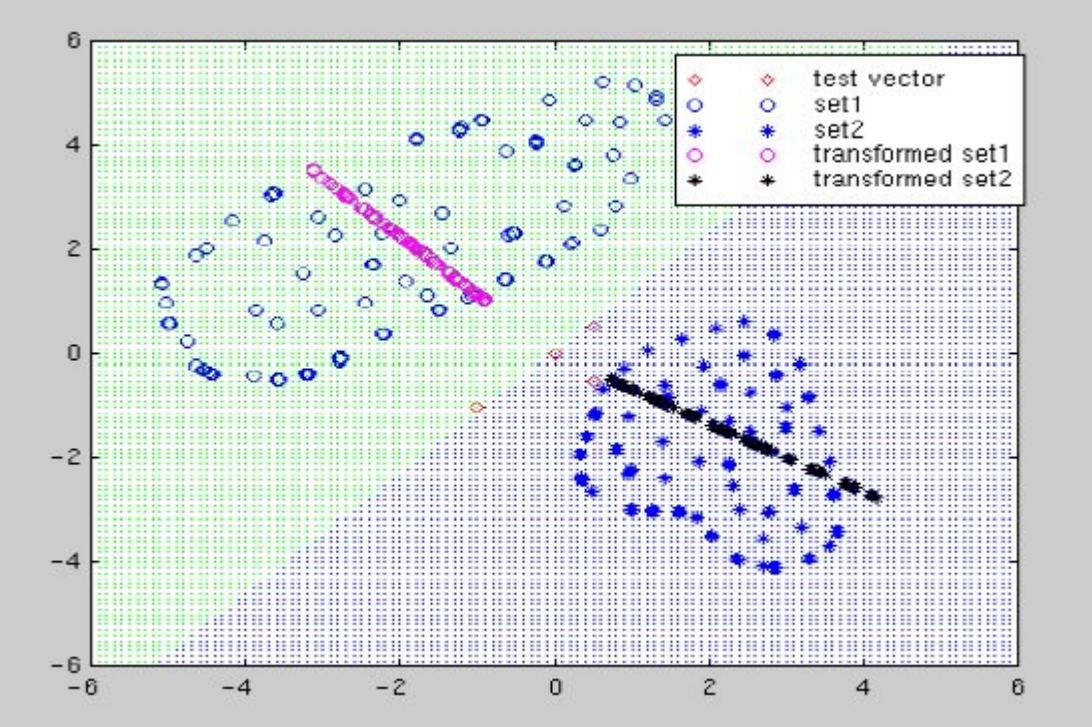


A visual inspection shows that category 1 objects (open circles) tend to have larger values of the predictor on the Y axis and smaller values on the X axis. However, there is overlap between the target categories on both axes, so we can't perform an accurate classification using only one of the predictors.

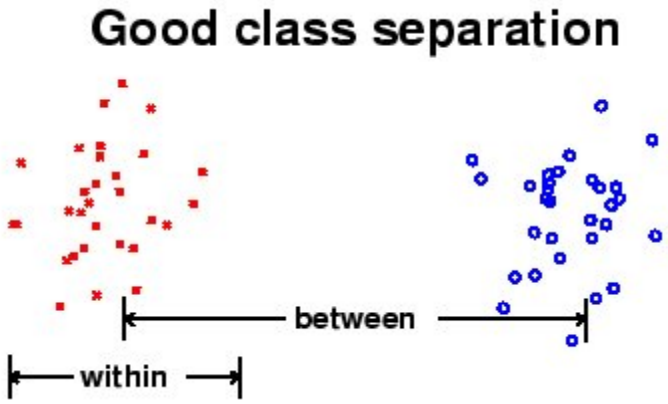
Linear discriminant analysis finds a linear transformation ("discriminant function") of the two predictors, X and Y, that yields a new set of transformed values that provides a more accurate discrimination than either predictor alone:

TransformedTarget = C1*X + C2*Y

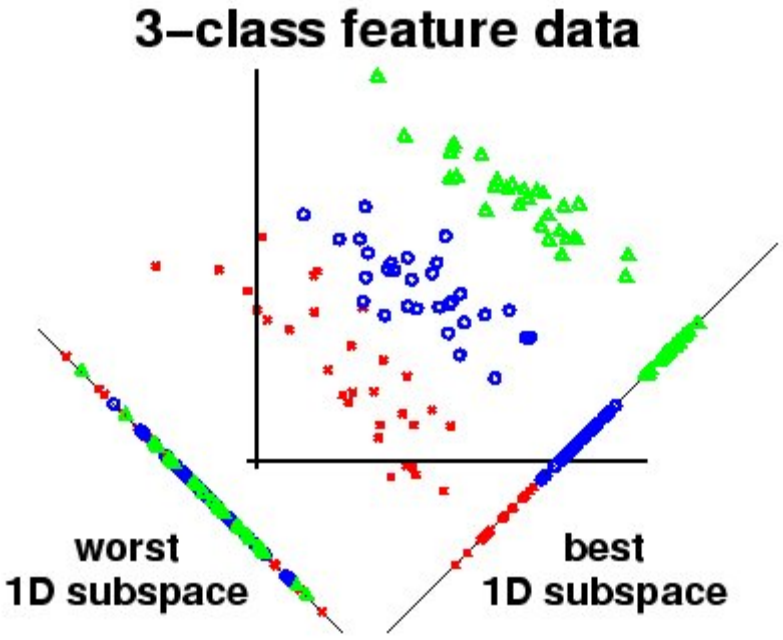
The following figure (also from Balakrishnama and Ganapathiraju) shows the partitioning done using the transformation function:



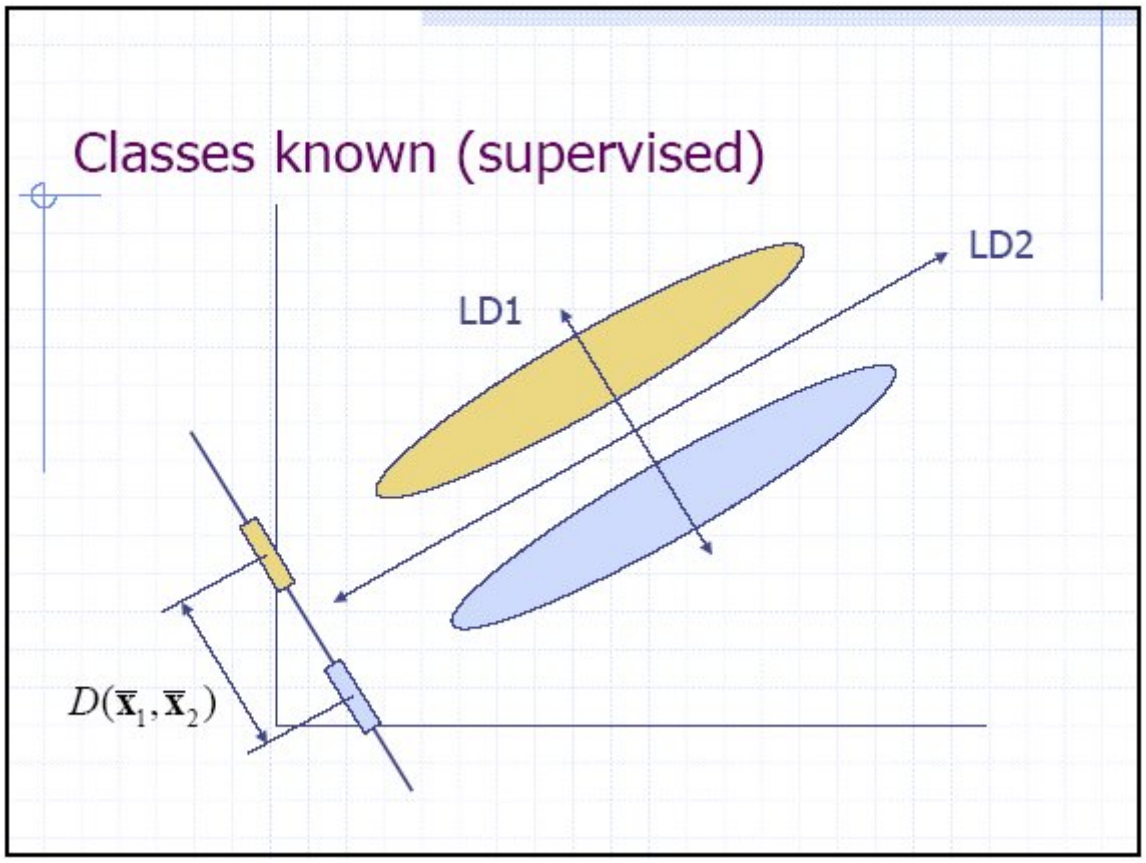
A transformation function is found that maximizes the ratio of between-class variance to within-class variance as illustrated by this figure produced by Ludwig Schwardt and Johan du Preez (Schwardt and Preez, 2005):



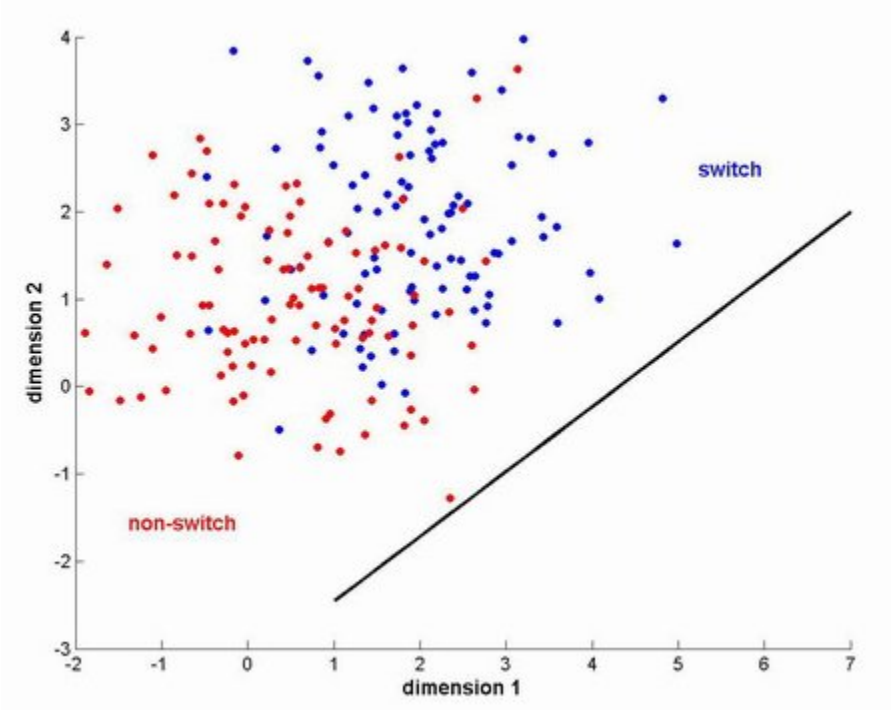
The transformation seeks to rotate the axes so that when the categories are projected on the new axes, the differences between the groups are maximized. The following figure (also by Schwardt and du Preez) shows two rotates axes. Projection to the lower right axis achieves the maximum separation between the categories; projection to the lower left axis yields the worst separation.



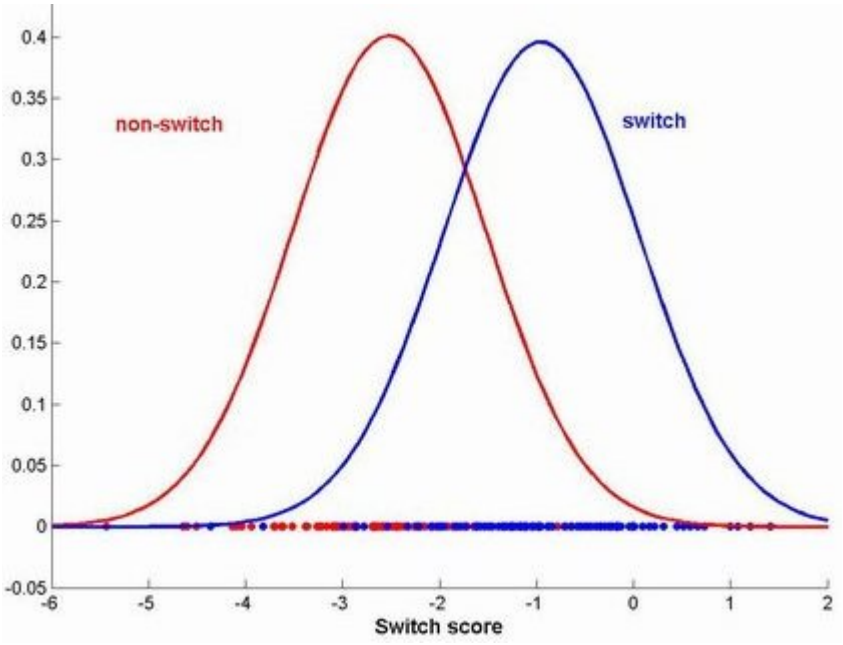
The following figure by Randy Julian (Julian, Lilly Labs) illustrates a distribution projected on the transformed axis labeled "D". Note that the projected values produce complete separation on the transformed axis, whereas there is overlap on both the original X and Y axes.



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.

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.

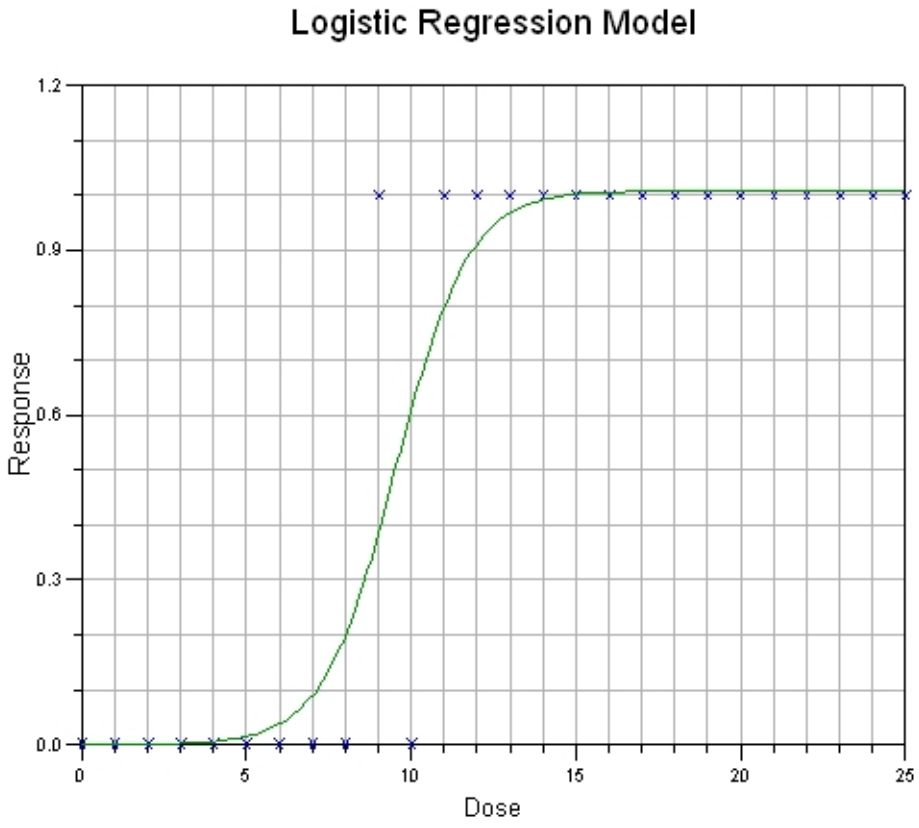
One obvious question is "Why not simply use linear regression?" In fact, many studies have done just that, but there are two significant problems:

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.

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 133 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 133 for information about generating a chart showing misclassification error rate versus model size.

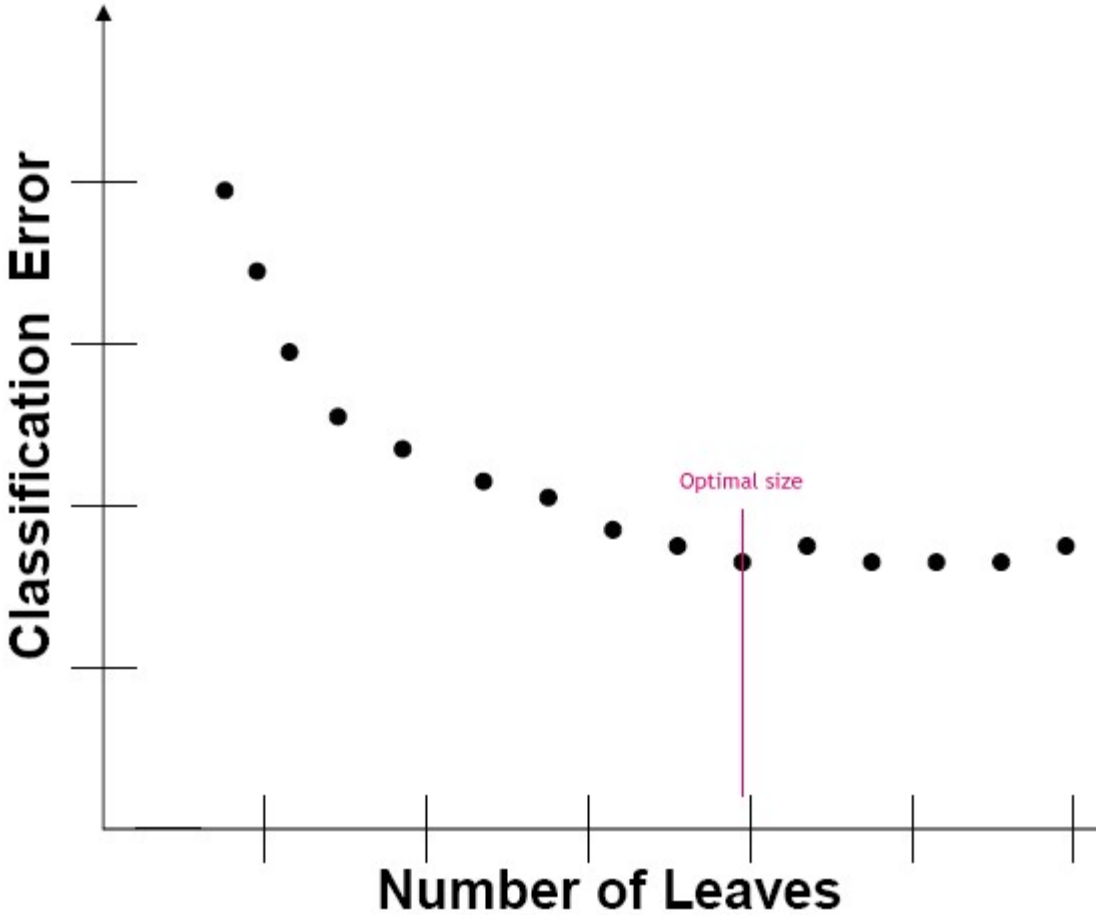
The primary goal of the pruning process is to generate the optimal size tree that can be generalized to other data beyond the learning dataset.

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.

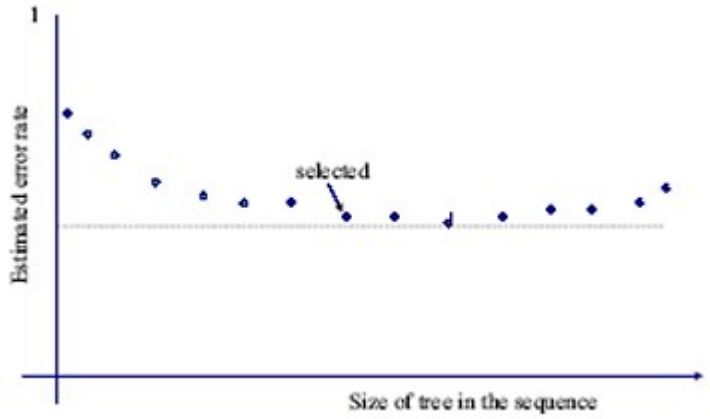
Once the 10 test trees have been built, their classification error rate as a function of tree size is averaged. This averaged error rate for a particular tree size is known as the "*Cross Validation cost*" (or "*CV cost"*). The cross validation cost for each size of the test trees is computed. The tree 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 109 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 109 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 109 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.

Decision Trees Compared To Other Modeling Methods

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.

Linear, Nonlinear and Logistic Regression

One of the simplest and most popular modeling methods is linear regression. **Linear regression** fits a straight line (known linear function) to a set of data values. The form of the function fitted by linear regression is:

$$y = a0 + a1*x1 + a2*x2 + \ldots$$

Where y is the dependent (target) variable, x1, x2, etc. are the independent (predictor) variables, and *a0*, *a1*, etc. are parameters whose values are determined so the function best fits the data. Linear regression is a popular modeling technique, and there are many programs available to perform linear regression. However, linear regression is appropriate only if the data can be modeled by a straight line function, which is often not the case. Also, linear regression cannot easily handle categorical variables nor is it easy to look for interactions between variables.

Nonlinear regression extends linear regression to fit general (nonlinear) functions of the form:

$$y = f(x1,x2,\ldots,a1,a2,\ldots)$$

Here are few examples of functions that can be modeled using nonlinear regression:

$$y = a0 + a1*\exp(x1)$$
$$y = a0 + a1*\sin(x1)$$

As with linear regression, nonlinear regression is not well suited for categorical variables or variables with interactions. The other challenge involved in using nonlinear regression analysis is that the form (model) of the function must be specified. For engineering and scientific problems, the function model may be dictated by theory, but for marketing, behavioral and medical problems, it can be very difficult to develop an appropriate nonlinear model. The program recommended for linear or nonlinear regression analysis is NLREG (http://www.nlreg.com).

Logistic regression is a variant of nonlinear regression that is appropriate when the target (dependent) variable has only two possible values (e.g., live/die, buy/don't-buy, infected/not-infected). Logistic regression fits an S-shaped logistic function to the data. As with general nonlinear regression, logistic regression is it good for detecting interactions between variables.

Neural Networks

Neural networks (also called "multilayered perceptron") provide models of data relationships through highly interconnected, simulated "neurons" that accept inputs, apply weighting coefficients and feed their output to other "neurons" which continue the process through the network to the eventual output. Some neurons may send feedback to earlier neurons in the network. Neural networks are "trained" to deliver the desired result by an iterative (and often lengthy) process where the weights applied to each input at each neuron are adjusted to optimize the desired output.

Support Vector Machine (SVM) models (see page 155) are a close cousin to classical neural networks. In fact, a SVM model using a sigmoid kernel function is equivalent to a two-layer, feed-forward neural network. 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.

SVM's are proven to get the optimal solution for a given set of training data. This is due to the fact that the space search is a convex function with a unique optimal solution, avoiding the local minima which are one of the pitfalls of perceptrons. The optimal solution is always found because the method is based on Lagrangian Variational Calculus.

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[3]. 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[4], and ID3 and, later, C4.5 by J. Ross Quinlan[5].

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*[6]. Much of their research was embedded in a program they developed called "CART"[7].

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.

---

[3] Morgan & Sonquist (1963) "Problems in the analysis of survey data and a proposal", JASA, 58, 415-434. (Original AID)

[4] Morgan & Messenger (1973) *THAID -- A sequential analysis program for the analysis of nominal scale dependent variables*, Survey Research Center, U of Michigan.

[5] Quinlan, J.R. (1993), *C4.5: Programs for Machine Learning*, Morgan Kaufman: San Mateo, CA.

[6] Breiman, L., Friedman, J.H., Olshen, R.A. & Stone, C.J. (1984), *Classification and Regression Trees*, Wadsworth: Belmont, CA.

[7] CART® is a registered trademark of Salford Systems.

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 COM Library

The optional DTREG COM (Component Object Model) library makes it easy for production applications to call DTREG as an "engine" to compute the predicted value for data records using a decision tree model. You must use the GUI version of DTREG to construct a decision tree model before you can use it with the DTREG COM library to predict values.

Any type of model (Single Tree, TreeBoost or Decision Tree Forest) can be used with the DTREG COM library to generate predicted values. All of the advanced scoring features such as the use of surrogate splitters to handle missing predictor values are used in the DTREG COM library.

Because of the standardization of the COM interface, it is easy to call the DTREG COM library from programs written in Visual Basic, Visual C++, VBA, Excel, Access, ASP and other languages. The DTREG COM library is designed to run as an in-process DLL for speed of execution.

An example Visual Basic program illustrating the use of the DTREG COM library is provided below. See the DTREG COM Library manual for details.

```
Private Sub RunTest_Click()
'
'  Reference the DTREG COM library.
'
Dim dtreg As DTREGCOMLib.dtreg
Set dtreg = New DTREGCOMLib.dtreg
'
'  Miscellaneous variable declarations.
'
Dim ProjectFile As String
Dim ModelType, status, index As Long
Dim NumVar, NumCat As Long
Dim VarClass, VarType As Long
Dim VarName, CatLabel As String
Dim ixSepalLength, ixSepalWidth As Long
Dim ixPetalLength, ixPetalWidth As Long
Dim ixSpecies As Long
Dim PredictedClass As String
Dim CatProb As Double
```

```
'
'  Open the DTREG project file (TreeBoostIris.dtr).
'
ProjectFile = "c:\DTREG\Test\TreeBoostIris.dtr"
status = dtreg.OpenProjectFile(ProjectFile)
If (status <> 0) Then
    boxStatus = "Error opening project file: " + Format(status, "##")
    Stop
End If
'
'  Find out what type of model this is
'
ModelType = dtreg.ModelType
'
'  Find out how many variables are in the model.
'
NumVar = dtreg.NumberOfVariables
'
'  Check the name and properties of each variable.
'
For index = 0 To NumVar - 1
    VarName = dtreg.VariableName(index)
    VarClass = dtreg.VariableClass(index)
    VarType = dtreg.VariableType(index)
Next
'
'  Get the index numbers of the variables variables.
'
ixSpecies = dtreg.VariableIndex("Species")
ixSepalLength = dtreg.VariableIndex("Sepal length")
ixSepalWidth = dtreg.VariableIndex("Sepal width")
ixPetalLength = dtreg.VariableIndex("Petal length")
ixPetalWidth = dtreg.VariableIndex("Petal width")
'
'  Set the values of the predictors we want to score.
'
status = dtreg.SetVariableValue(ixSepalLength, 5.1)
status = dtreg.SetVariableValue(ixSepalWidth, 3.5)
status = dtreg.SetVariableValue(ixPetalLength, 1.4)
status = dtreg.SetVariableValue(ixPetalWidth, 0.2)
'
'  Compute the predicted target category.
'
PredictedClass = dtreg.PredictedTargetCategory
```

```
'
'  See if any error occurred during the computation.
'
status = dtreg.LastStatus
If status <> 0 Then
    boxStatus = "Error computing target: " + Format(status, "##")
    Stop
End If

End Sub
```

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

Both the DTREG program and documentation are copyright © 1991-2004 by Phillip H. Sherrod. You are not authorized to modify the program or documentation. "DTREG" is a trademark of Phillip H. Sherrod.

Web page

Up-to-date information about DTREG can be found on the web page: http://www.dtreg.com

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*.

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.

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.

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, Chao, Andy Liaw, Leo Breiman, *Using Random Forest to Learn Imbalanced Data*.

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.

Fawcett, Tom. *ROC Graphs: Notes and Practical Considerations for Data Mining Researchers*. March 16, 2004.

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.

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.

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.

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.

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

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

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).

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)

Quinlan, J. Ross. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

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)

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.

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

.

.csv file type, 23, 35
.dtr file type, 23, 27

1

1 SE pruning, 43, 196

A

A priori probabilities, 69
Access, data for DTREG, 35
Actual versus predicted chart, 145
AdaBoost, 147, 148
Adjusting optimal tree size, 195
AID, Automatic Interaction Detection, 190, 199, 212
Akaike's Information Criterion, 180
Altered priors, 73, 188
Analysis of variance report, 115
Analysis report format, 107
Assigning categories to nodes, 188
Association direction, 114
Association of surrogate splitters, 114
AUC statistic, 121
Automatic Interaction Detection, 199
Average category weights, 70

B

Backward pruning, 190
Balanced category weights, 70
Bayesian Information Criterion, 180
Berk, Richard A., 209
Berra, Yogi, 9
Beta parameters, logistic regression, 180
Bibliography, 209
Binary split, 13
Blake, C. L., 209
Boosting, 147
Boston.dtr example, 201
Breiman, Leo, 151, 199, 209
Bridges.dtr example, 202
Building trees, 185

C

C code generation, 98, 100
C source code generation, 97
C++ code generation, 98
C4.5 program, 199
CART program, 199, 212
Categorical variables, 17
Categories for continuous variables, 34
Category labels property page, 65
Category weights, 69
Chang, Chih-Chung, 168
Charts and plots, 133
Christ, Jesus, 190
Class labels property page, 65
Classes of variables, 17
Classification trees, 19
ClevelandHeart14.dtr example, 202
Cluster analysis, 34, 186, 197
Column separator character, 23, 36
COM library, 203
Comma as decimal point, 23, 36
Comma separated value files, 35
Competitor splits, 114, 187
Complete separation, 183
Complexity measure, 195
Computer learning, 197
Confidence intervals, 181
Confusion matrix, 117
Continuous variable categories, 34
Continuous variables, 17
Convergence failure, 182
Copyright notice, 207
Cost complexity measure, 111, 195
Creating a new project, 22
Credit scoring, 9
Cross validation, 42, 47, 193
Cross validation cost, 110, 194
Cross validation cost standard error, 111
Cross-validation control variable, 40
csv file type, 23, 35
CSV files, 35
Cumulative gain, 123
Cumulative lift chart, 138
Custom category weights, 70
Custom pruning cutoff, 43, 196
Customer targeting, 9
CV cost, 194

D

Data file format, 35
Data mining, 9
Data modeling, 10
Data property page, 35
Data subset, 23, 35
Data Transformation Language (DTL), 81
Decimal point character, 23, 36
Decision Forests, 151
Decision Tree Forest, 151
Decision tree forest property page, 50
Decision tree forest size control, 51
Decision trees, 13
Depth of trees, 45
Design property page, 33
Deviance of log likelihood, 180
Dichotomous variables, 177
Disclaimer, 207
Discriminant analysis, 171
Discriminant analysis property page, 61
Dose-response curve, 178

DTL DataTransformation Language, 81
DTL reference manual, 82
dtr file type, 23, 27
DTREG COM library, 203
DTREG Web page, 207
DTREGcom.dll, 203
DTREGsetup.exe, 21

E

EndRun() function, 88
Ensemble tree methods, 147, 151
Entropy splitting method, 34, 187
Equal category weights, 70
Equal misclassification costs, 72
Equal priors, 70
Evaluating splits, 187
Example projects, 28, 201
Excel, data for DTREG, 35
Exhaustive search, 34, 186
Explained variance, 115
Explicit global variables, 84
Exploratory tree generation, 42

F

Factor analysis, 197
Feature selection, 155
Federalists Papers, 158
Feed-forward neural network, 57, 165
First row in data file, 36
Firth's procedure, 64, 182
Fisher, R.A., 171, 210
Fixed size pruning, 42
Focus category, 117
Focus Category Impurity chart, 134
Focus Category Loss chart, 135
Focus category, designating, 67
Focus category, Impurity, 117, 135
Focus category, Loss, 117, 136
Forcing the initial split, 68
Forward pruning, 190
Freund, Y., 210
Friedman, Jerome, 147, 199, 209, 210
Full tree generation, 42

G

Gain chart, 136, 137, 138
Gain table, 122
Generalization of trees, 191
Generating scoring code, 97
Gini splitting method, 34, 187
Global variables, 83
Gradient boosting, 147
Graphs and charts, 133
Grid search, 59

H

Hartigan, J.A., 210
Hessian matrix, 182, 183
Heterogeneity of nodes, 187
Homogeneity of nodes, 187
HorseColic.dtr example, 202
HouseVotes.dtr example, 201
Huber M-regression loss function, 148
Huber's quantile cutoff, 45
Hyperplane, 155, 158

I

ID3 program, 199
Implicit global variables, 83
Importance chart, 144
Improvement of split, 187
Impurity of focus category, 117, 135
Impurity of nodes, 187
Influence trimming factor, 45
Initial split property page, 67
Initial split variable, 68
Input data report section, 108
Installing DTREG, 21
Interactions between variables, 11
Interior nodes, 13
Interval variables, 17
Introduction, 13
Iris.dtr example, 24, 201

J

Jesus Christ, 190

K

Kernel function, 54, 159
Kernel trick, 162

L

lag function, 86
LandingControl.dtr example, 202
Leaf nodes, 13
Learning dataset, 14
Least squares criteria, 187
LIBSVM, 168, 209
License information, 207
Lift, 123
Lift and gain chart, 136
Lift chart, 138
Lift table, 122
Likelihood ratio significance test, 64
Likelihood ratio significance tests, 182
Lin, Chih-Jen, 168
Line search, 59, 167
Linear discriminant analysis, 171
Linear kernel function, 54, 162
Linear regression, 197
LiverDisorder.dtr example, 201
Log file, 33
Log likelihood function, 180
Logistic regression, 177

Logistic regression property page, 63
Loh, W.Y., 211
Loss of focus category, 117, 136

M

Machine learning, 197
Main screen, 21
main() function, 82
MART, 147
Maximum tree levels, 41
Merz, C. J., 209
Minimal cross-validated error, 43, 194, 196
Minimum CV, 194
Minimum node size, 41, 45
Minimum trees in TreeBoost series, 48
Minimum variance criteria, 187
Miscellaneous property page, 78
Misclassification cost, 71, 188
Misclassification cost property page, 71
Misclassification cost splitting method, 34
Misclassification matrix, 117
Misclassification summary table, 116
Missing data property page, 74
Missing value code, 86
Missing value indicator, 36
Missing values, 188
Missing values in data, 37
MissingValue implicit value, 86
Mix category weights, 70
Model size chart, 133
Monotonic variables, 17
Morgan, J.N., 199, 212
Most probable category in node, 188
M-regression loss function, 45, 148
Multilayered perceptron, 198
Multiple Additive Regression Trees, 147
Murphy, Patrick M., 212

N

Neural network, 155
Neural network kernel function, 57, 165
Neural networks, 148, 198
New project, 22
Newton-Raphson algorithm, 63
NLREG program, 198, 212
Node impurity, 187
Node split information, 113
Node splits report section, 111
Node summary report section, 111
Nodes in tree, 13
Nodes, interior, 13
Nodes, leaf, 13
Nodes, root, 13
Nodes, terminal, 13
Nominal variables, 17
Nonlinear regression, 197, 212
Number of trees in decision tree forest, 51

O

Odds Ratio, logistic regression, 181
Olshen, Richard, 199, 209
One standard error pruning, 43
Opening a project, 27
Optimal tree size, 191
Ordered variables, 17
Ordinal variables, 17
Output report, 107
Overall variable importance, 129

P

Pattern search, 59, 60, 167
Pazzani, Michael J., 212
pcSVMdemo, 162
Period as decimal point, 23, 36
Plots and charts, 133
Polynomial kernel function, 54, 55, 163
Predictor variable, 17
Preferred splitting variables, 68
Principle components analysis, 197
Prior probabilities, 69
Priors property page, 69
Probability scores, 93
Probability threshold balance chart, 143
Probability threshold chart, 141
Probability threshold report, 119
Probability threshold, balance misclassifications, 121
Probability threshold, minimize total error, 121
Probability threshold, minimize weighted errors, 72, 121, 144
Probability threshold, specifying, 73
Project log file, 33
Project parameters report section, 108
Project title, 33
Properties for a model, 31
Proportion of variance explained, 115
Pruning control, 43
Pruning tolerance, 48
Pruning trees, 190

Q

Quasi-complete separation, 183
Quinlan, J. Ross, 199, 212

R

Radial basis function, 56, 164
Radial basis kernel function, 54
Random Forests™, 151
Random number seeds, 78
Random rows validation, 42, 46
RBF kernel function, 54
RBF network, 56, 164
Receiver Operating Characteristic chart, 139
Recursive partitioning, 13
Reference tree, 194
References, 209
Registering DTREG, 207

Regression trees, 18
Residual chart, 145
Residual variance, 115
Resubstitution cost, 111, 191
Return on investment, 123
Return statement, 82
ROC chart, 139
ROC chart, area under, 121, 139
ROC chart, reference, 210
ROI, 123
Root node, 13
Running an analysis, 29

S

Salford Systems, 212
SAS code generation, 98, 103
Schapire, R., 210
Schwartz Criterion, 180
ScoreRecord function, 102
Scoring data, 91
Shakespeare, William, 155
Sherrod, Phillip H., 212
Shih, Y.S., 211
Shrinkage factor, 46
Sigmoid kernel function, 54, 57, 165
Sigmoidal dose-response curve, 178
Single Tree property page, 41
Singular Hessian matrix, 183
Slack variables, 166
Smooth minimum spikes, 42, 48
Soft margin, 166
Sonquist, J.A., 199, 212
Specifying category weights, 70
Specifying misclassification costs, 73
Split, 189
Split point, 14, 186
Splitting algorithm, 34
Splitting nodes, 185
Splitting variable, 14
Standard error of cross validation cost, 111
StartRun() function, 88
Static global variables, 87
Stochastic gradient boosting, 147
Stone, Charles, 199, 209
Stopping criteria, 190
StoreData() function, 87
Subset of data rows, 23, 35
Summary of categories report section, 109
Summary of variables report section, 109
Supervised learning, 197
Support of DTREG, 207
Support Vector Machine, 155
Surrogate splitters, 75, 95, 112, 114, 129, 188
Surrogate splitters association, 114
SVM, 155
SVM cache size, 57
SVM grid search, 59
SVM kernel function, 54
SVM line search, 59, 167
SVM pattern search, 59, 60, 167
SVM probability estimates, 58
SVM property page, 53
SVM shrinking heuristic, 58
SVM stopping criteria, 57

T

Target category distribution, 69
Target category distribution report, 112
Target variable, 17
Terminal node table, 128
Terminal nodes, 13
THAID program, 199
Threshold balance chart, 143
Threshold chart, 141
Threshold report, 119
Time series lag function, 86
Titanic passenger example, 201
Title for project, 33
Trademark notice, 207
Training data category weights, 70
Training dataset, 14
Translate property page, 98
Translation, 97
Tree fitting algorithm, 34
Tree level control, 41
Tree nodes, 13
Tree pruning control, 43
Tree size optimization, 191
Tree size report section, 109
TreeBoost, 147
TreeBoost cross validation, 47
TreeBoost probability scores, 93
TreeBoost property page, 43
TreeBoost series length, 44
Type 1 + 2 margins, 52
Type 1 margins, 52
Types of variables, 17

U

UCI Repository of Machine Learning Databases, 201, 209
Unexplained variance, 115
Unitary misclassification costs, 72
Unpruned tree, 43
Unsupervised learning, 197
Use and distribution, 207
Using a decision tree to predict values, 16

V

Validation cost, 110
Validation cost standard error, 111
Validation Statistics report section, 110
Values of nodes, 188
Variable classes, 17
Variable for initial split, 68
Variable importance chart, 144
Variable importance table, 129
Variable interactions, 11
Variable names in data file, 36
Variable types, 17

Variable weights property page, 77
Variables property page, 38
Variance splitting method, 34
V-fold cross validation, 42, 47, 193
Viewing the tree, 30, 130
Voter targeting, 9

W

Wald confidence intervals, 181
Web page, 207
Weight variable, 17
Weighted misclassification errors, 72, 121, 144
Wong, M.A., 210

Y

Yogi Berra, 9