DTREG Class Library

Phillip H. Sherrod
6430 Annandale Cove
Brentwood, TN 37027-6313 USA
phil@philsherrod.com

www.dtreg.com

Contents

Introduction

The DTREG class library makes it easy for .NET production applications to call DTREG as an "engine" to compute the predicted value for data records using a predictive model. You must use the GUI version of DTREG to construct a model before you can use it with the DTREG COM library to predict values.

Any type of model (Single Tree, TreeBoost, Decision Tree Forest, SVM, LDA, Logistic Regression, etc.) can be used with the DTREG 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 library.

Because of the standardization of the .NET interface, it is easy to call the DTREG class library from programs written in VB.NET, C#, Visual C++ and other languages.

Both 32-bit and 64-bit versions of the DTREG class library are available.

Data Types Used With the DTREG COM Methods

The descriptions of the DTREG methods show the recommended data types for the input and output arguments. For example, the SetContinuousValue method has two arguments:

int SetContinuousValue(int *VariableIndex*, double *Value*)

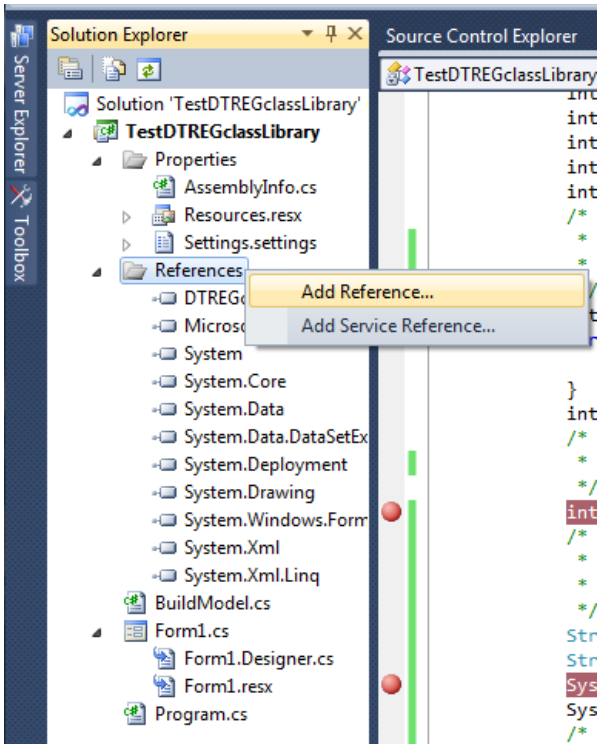
The recommended type for the *VariableIndex* argument is Integer and the recommended type for the *Value* argument is double precision. The value returned by SetVariableValue is integer.

Calling DTREG from C#

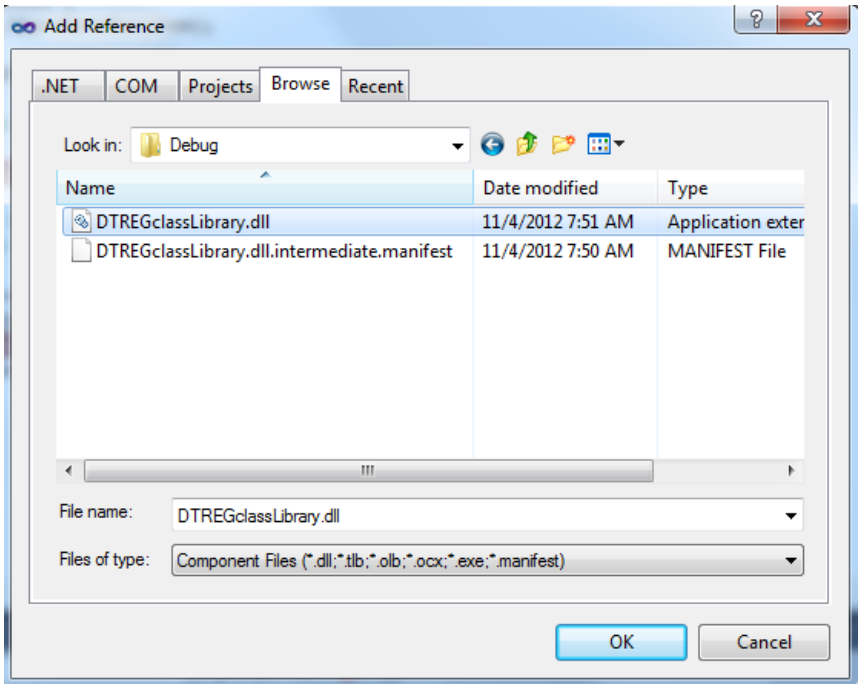
It is easy to call DTREG methods from C# .NET because C# and Developer's Studio have excellent support for class libraries.

Referencing DTREG from C# programs

The first step in using DTREG with a C# program is to add a reference to the DTREGclassLibrary to the project. To do this, start Developer's Studio and open the project, then right click on "References" in the Solution Explorer panel and select Add Reference from the popup menu.



Then select the Browse tab, navigate to the folder that has the DTREGclassLibrary.dll file, and select it as a reference for the project.



Within the C# program, you must add a using statement telling the program you will be referencing items in the DTREG class library:

```
using DTREGclassLibrary;
```

Within the executable code, you must instantiate a DTREGclass object and create a variable to reference it:

```
DTREGclass objDtreg = new DTREGclass();
```

Once a DTREGclass object variable has been created, you can reference routines within the DTREG library. For example:

```
intStatus = objDtreg.BeginTraining("Test model training");
```

Example C# program

Here is an example of a complete C# program:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DTREGclassLibrary;

namespace TestDTREGclassLibrary
{
    /*-----------------------------------------------------------------------------------
     *  Class to build a DTREG model.
     */
    class BuildModel
    {
        /*-----------------------------------
         * Routine that trains a DTREG model.
         */
        public int BuildTheModel()
        {
```

```
int intStatus;
/*
 *  Establish a reference to a DTREGclass object.
 */
DTREGclass objDtreg = new DTREGclass();
m_objDtreg = objDtreg;
/*
 *  Enter our registration information.
 */
intStatus = objDtreg.SetRegistration("registered name", "registration key");
if (CheckStatus(intStatus)) return (intStatus);
/*
 *  Initialize for a new model.
 */
intStatus = objDtreg.BeginTraining("Test model training");
if (CheckStatus(intStatus)) return(intStatus);
/*
 *  Set the type of model to build.
 *  1 = Single decision tree.
 *  2 = TreeBoost.
 *  3 = Decision tree forest.
 *  4 = Logistic regression.
 *  5 = SVM
 *  7 = LDA
 *  9 = Neural network
 *  10 = PNN/GRNN
 *  11 = RBF
 *  12 = Cascase correlation
 *  13 = GEP
 *  14 = Linear regression
 *  15 = K-Means
 *  16 = GMDH
 *  17 = Correlation, factor analysis.
 */
intStatus = objDtreg.SetModelType(2);
if (CheckStatus(intStatus)) return (intStatus);
/*
 *  Define 5 variables.  The first variable is the categorical target variable.
 *  The other 4 variables are continuous predictor variables.
 */
intStatus = objDtreg.BeginVariableDefinitions();
if (CheckStatus(intStatus)) return (intStatus);
intStatus = objDtreg.DefineVariable("Species", 2, true);
if (CheckStatus(intStatus)) return (intStatus);
intStatus = objDtreg.DefineVariable("Sepal length", 1, false);
if (CheckStatus(intStatus)) return (intStatus);
intStatus = objDtreg.DefineVariable("Sepal width", 1, false);
if (CheckStatus(intStatus)) return (intStatus);
intStatus = objDtreg.DefineVariable("Petal length", 1, false);
if (CheckStatus(intStatus)) return (intStatus);
intStatus = objDtreg.DefineVariable("Petal width", 1, false);
if (CheckStatus(intStatus)) return (intStatus);
intStatus = objDtreg.EndVariableDefinitions();
if (CheckStatus(intStatus)) return (intStatus);
/*
 *  Feed in data records.
 *  The column delimiter character is ","
 */
intStatus = objDtreg.BeginStoringData(",", 0);
if (CheckStatus(intStatus)) return (intStatus);
foreach (var item in DataRow)
{
    intStatus = objDtreg.StoreDataRow(item);
    if (CheckStatus(intStatus)) return (intStatus);
}
intStatus = objDtreg.EndOfData();
if (CheckStatus(intStatus)) return (intStatus);
/*
 *  Train the model.  The type of the model was set by SetModelType().
 */
intStatus = objDtreg.GenerateModel();
if (CheckStatus(intStatus)) return (intStatus);
/*
 *  Get analysis reports from the model, and write them to files.
 *  --- Change the folder for the files ---
 */
String txtReport = objDtreg.GetAnalysisReport();
String txtXMLReport = objDtreg.GetAnalysisReportXML();
System.IO.File.WriteAllText(@"C:\Test\AnalysisReport.txt", txtReport);
System.IO.File.WriteAllText(@"C:\Test\AnalysisReport.xml", txtXMLReport);
```

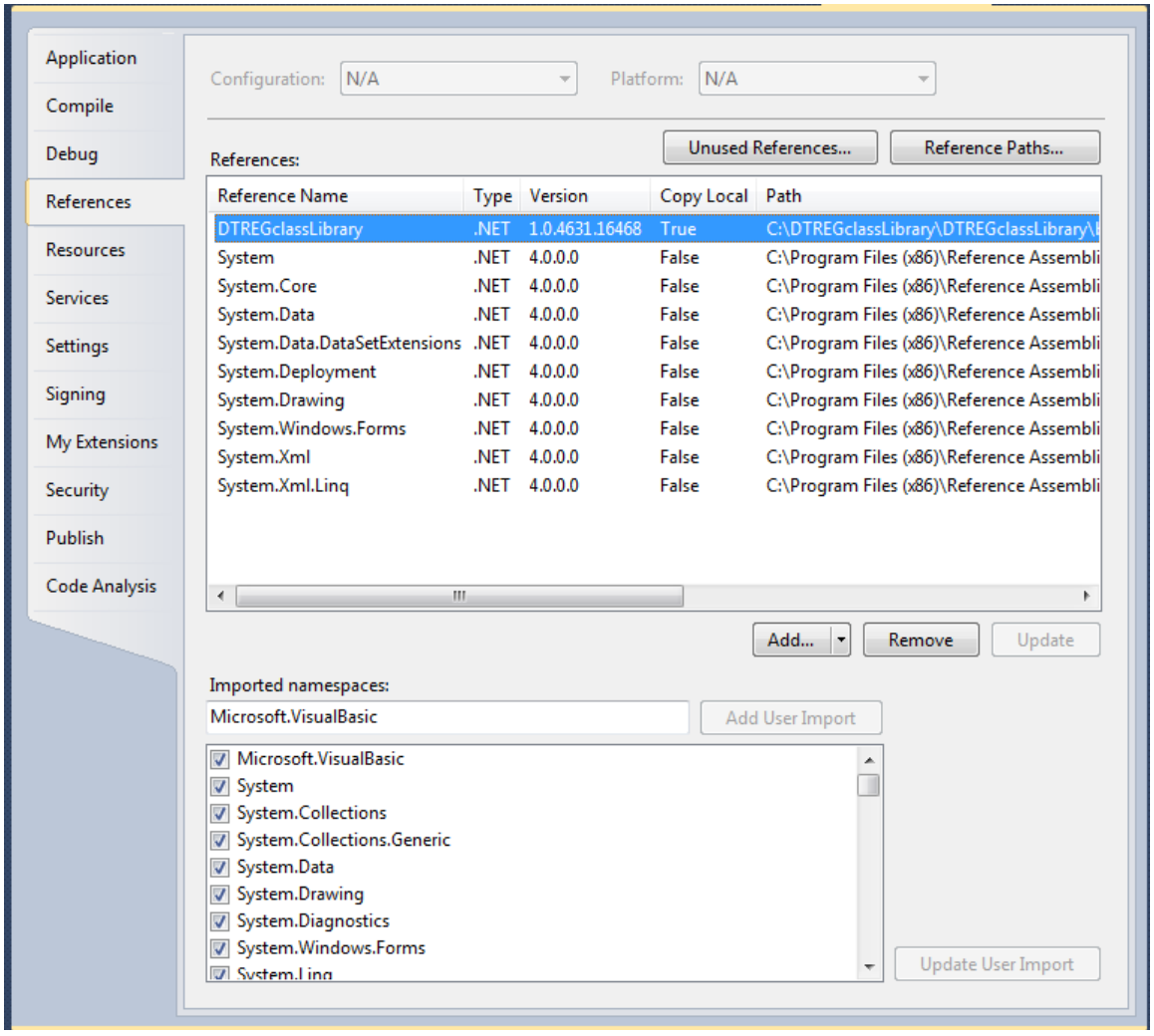
```
            /*
             *  Write the project to a file.
             */
            intStatus = objDtreg.SaveProject(@"C:\Test\Model.dtr");
            if (CheckStatus(intStatus)) return (intStatus);
            /*
             *  Finished
             */

            return(0);
        }

        /*---------------------------------------------------------------------------
         *  Check a status code and display a message box if there is an error.
         *  Return true if there is an error or false for success.
         */
        bool CheckStatus(int intStatus)
        {
            if (intStatus != 0) {
                MessageBox.Show(m_objDtreg.StatusMessage(intStatus), "Error");
                return(true);
            } else {
                return(false);
            }
        }
/*
 *  Data rows for model training.
 */
string []DataRow = {
  "Setosa,5.1,3.5,1.4,0.2",
  [… More data rows …]

 };
}

}
```

Calling DTREG from VB.NET

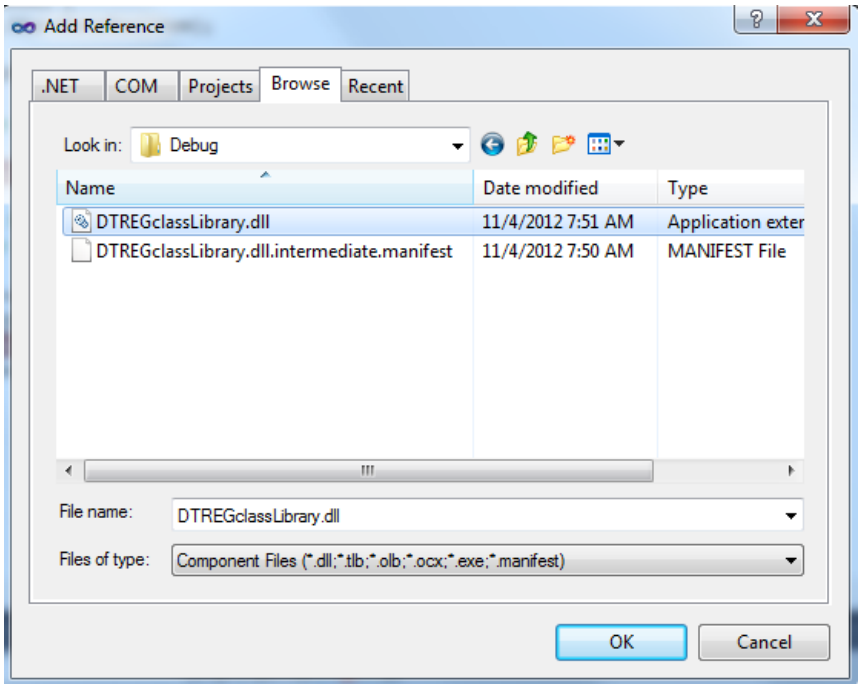
It is easy to call DTREG methods from VB.NET because Visual Basic has excellent support for class libraries.

Referencing DTREG from Visual Basic

The first step in using DTREG with a VB.NET program is to tell Visual Basic to reference DTREG. To do this, start Developer's Studio and open the project, then right click on the name of the project in the Solution Explorer panel and select "Properties" from the popup menu. Select the "References" tab, and then click "Add…". Use the "Browse" panel to find and select the DTREGclassLibrary.dll file. Note: VB.NET requires the use of the 32-bit version of the class library, not the 64-bit version.



Then select the Browse tab, navigate to the folder that has the DTREGclassLibrary.dll file, and select it as a reference for the project.



In addition to adding the reference, you also must add this statement to the top of your program:

```
Imports DTREGclassLibrary
```

Once you have established a reference to DTREG within your VB.NET program, you can add declarations and method calls to use the class library.

Declaring the DTREG Class Library object in a Visual Basic program

Each Visual Basic procedure that calls a DTREG method must contain the following declarations:

```
Dim dtreg As New DTREGclassLibrary.DTREGclass
```

This declares a variable named “dtreg” to reference the DTREG class library. Once you do that, you can then reference DTREG methods by typing “dtreg.*methodname*()”. (You do not have to name the variable “dtreg”, but it is recommended that you do so to make it clear that the methods are part of the DTREG library.)

Example Visual Basic program

Here is a complete example Visual Basic program that calls the DTREG library:

```
Imports DTREGclassLibrary

Private Sub btnRun_Click(sender As System.Object, e As System.EventArgs) Handles btnRun.Click
        '
        '  Miscellaneous variable declarations.
        '
        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 PredictedCategory As String
        Dim CatProb As Double
        Dim Mvalue As Double
        Dim intStatus As Integer
        '
        '  Create an object to access the DTREG class library.
        '
        Dim dtreg As New DTREGclassLibrary.DTREGclass
        '
        '  Enter registration key.
        '
        intStatus = dtreg.SetRegistration("registration name", "registration key")
        '
        '  Open a DTREG project file.
        '
        intStatus = dtreg.OpenModel("C:\DTREGtestFiles\Iris.dtr")
        If intStatus <> 0 Then
            MessageBox.Show("Project open status = " & intStatus.ToString)
        End If
        '
        '  Get the real value used to represent missing value.
        '
        Mvalue = dtreg.GetMissingValue()
        '
        '  Find out what type of model this is
        '
        ModelType = dtreg.GetModelType()
        '
        '  Find out how many variables are in the model.
        '
        NumVar = dtreg.GetNumberOfVariables()
        '
        '  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.SetContinuousValue(ixSepalLength, 5.1)
        status = dtreg.SetContinuousValue(ixSepalWidth, 3.5)
        status = dtreg.SetContinuousValue(ixPetalLength, 1.4)
        status = dtreg.SetContinuousValue(ixPetalWidth, 0.2)
        '
        '  Compute the predicted target category.
        '
        intStatus = dtreg.ComputeScore()
        '
        '  See if any error occurred during the computation.
        '
        If intStatus <> 0 Then
            MessageBox.Show("Error computing target: " & intStatus.ToString)
            Stop
        End If
        '
        '  Check the predicted category for the target.
        '
        PredictedCategory = dtreg.GetPredictedCategory()
        If PredictedCategory <> "Setosa" Then
            MessageBox.Show("Incorrect predicted category: " & PredictedCategory)
        End If
        '
        '  If we are using a TreeBoost model, check the probabilities
        '  for each of the target variable categories.
        '
        NumCat = dtreg.GetNumberOfCategories(ixSpecies)
        For index = 0 To NumCat - 1
            CatLabel = dtreg.GetCategoryLabel(ixSpecies, index)
            CatProb = dtreg.GetProbabilityScore(index)
        Next
    End Sub
End Class'
'  If we are using a TreeBoost model, check the probabilities
'  for each of the target variable categories.
'
If ModelType = 2 Then
    NumCat = dtreg.NumberOfCategories(ixSpecies)
    For index = 0 To NumCat - 1
        CatLabel = dtreg.CategoryLabel(ixSpecies, index)
        CatProb = dtreg.TargetCategoryProbability(index)
    Next
End If

End Sub
```

Procedures for Predicting Values

The procedures described in this section are used to compute predicted values after a DTREG model has been trained.

All references to procedures in the DTREG class library must use the object referencing the class. In this manual, that object is named “dtreg”, but you can use a different name if you wish.

Here is an example of a DTREG method:

```
Index = dtreg.VariableIndex(“age”)
```

LastStatus -- Status code for the last operation

int LastStatus()

The LastStatus function returns a status code indicating whether the last operation performed by the DTREG library was successful or not. If the last operation was completed successfully, a value of 0 (zero) is returned. If some error occurred on the last operation, the value indicates what type of error occurred.

List of status codes:

0	No error
1	Unable to open project file
2	The project file is corrupt
3	No DTREG project file is currently open
4	The variable name or index is not valid
5	Invalid category value
6	Unable to compute a score for the predictor values
7	No model has been computed for the project
8	Error in DTL program
9	Error setting up PCA transformation
10	A password must be provided to open this project
11	The supplied password is incorrect
12	Error while reading the data file
13	Demonstration period has expired
14	Argument string is too long
15	Unable to create chart output file
16	Error creating compressed PNG image
17	Error creating compressed JPEG image
18	Unable to allocate required memory
19	Invalid variable index number
20	Invalid row index number
21	No variables were defined
22	No target variable was defined
23	No data rows were specified
24	Text (non-numeric) data for continuous variable
25	Error while building the model
26	No data file specified
27	More than one target variable was specified
28	The cross validation control variable is not categorical
29	The validation row selection control variable is not categorical
30	Type of model you selected cannot be used for a time series
31	The type of model can only be used when the target variable is categorical

32	A time series cannot be built where the target variable is categorical
33	All of the rows have missing values on the target or weight variables
34	Unable to open the data file
35	The first row of the data file does not contain the names of variables
36	A variable with the same name appears in more than one column of the data file
37	Error setting up PCA variables
38	A variable used in the PCA transformation is not in the main project
39	A category for a variable in the PCA transformation is not
40	The number of records in the data file exceeds the limit for this version of DTREG
41	The input file for Score is empty
42	Unable to open output file for Score
43	An error occurred while running the DTL program
44	Unable to creat the output file
45	The length of the variable name is not valid
46	Analysis aborted
47	Unable to split the first node of the group
48	Invalid argument value
49	Unable to create output file
50	Error creating record describing a data row
51	Invalid registration key
52	DTREG registration has expired
53	Registration information has not been entered
54	The registration key is not for the Class Library version of DTREG
55	The registration key is not for the GUI version of DTREG
56	The registration key does not permit creating new DTREG models

StatusMessage -- Convert status code to descriptive message

String ^StatusMessage(int *StatusCode*)

The StatusMessage function takes a status code value as its argument and returns a string that is a description of what the status code means.

SetRegistration – Entering the registered name and registration key

Int SetRegistration(String ^*RegisteredName*, String ^*RegistrationKey*)

The SetRegistration() method must be called before calling either OpenModel() or BeginTraining(). It specifies the registered name for your DTREG installation and the associated registration key.

OpenModel -- Open a DTREG project file

int OpenModel(String ^*FileName* As String)

The OpenModel function opens a DTREG project file (usually with a .dtr file type), loads the information from it into memory and returns a status code indicating if the operation was successful or not (the status code also can be retrieved using the LastStatus property.

If a project file is open when you call OpenModel, the currently-open project is closed and then the new project is opened.

Arguments:

FileName = A string containing the name of the DTREG project file. It is best to provide a full file specification including the device name and directory.

Returned value:

The value 0 is returned by OpenModel() if the project file was successfully opened. A non-zero status code is returned if there were compile errors (see the LastStatus() for a list of the error code values).

CloseModel -- Close the current DTREG project file

void CloseModel()

The CloseModel procedure closes the currently open DTREG project file (if any) and releases the memory used by it. Note, if OpenModel is called while another project is open, it closes the current project before opening the new one.

GetModelBuildDate -- Get build date of project

String ^GetModelBuildDate()

The GetModelBuildDate function returns a value string that indicates when the project was built. Year, month, day, hour, minute and second based on UTC time are returned in a string with the format: *yyyymmddhhmmss*

GetModelType -- Get type of model

int GetModelType()

The GetModelType function returns a value that indicates what type of model is described by the current project file. These values can be returned:

1	Single tree model
2	TreeBoost model
3	Decision tree forest model
4	Logistic regression model
5	SVM model
7	Linear discriminant analysis model
9	Multi-layer feed-forward neural network
10	PNN or GRNN neural network
11	RBF neural network
12	Cascade correlation network
13	Gene Expression Programming (GEP)
14	Linear regression
15	K-Means clustering
16	GMDH model
17	PCA model

GetNumberOfVariables -- Get variable count from a project

int GetNumberOfVariables()

The GetNumberOfVariables function returns a count of the total number of variables defined in the project. This count includes the target variable, predictor variables, weight variable and any unused variables.

VariableIndex -- Get index number for a variable

int VariableIndex(String ^*VariableName*)

Many of the methods in DTREG use index numbers to identify variables. The VariableIndex method converts the name of a variable to its corresponding index number. The index number of the first variable is 0 (zero).

Arguments:

VariableName is a string containing the name of the variable whose index number is to be found.

Returned value:

The returned value of the method is the index number of the variable. A value of -1 is returned if no variable can be found with the specified name. Use the LastStatus property to check for errors after calling VariableIndex.

VariableName -- Get the name of a variable from its index number

String ^VariableName(int *VariableIndex*)

The VariableName function is the complement of the VariableIndex method. The VariableName method takes the index number of a variable as the argument and returns the name of the variable as its result.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

VariableClass -- Get the class of a variable

int VariableClass(int *inVariableIndex*)

The VariableClass method returns a number indicating the class of a variable. There are four possible class values:

0	Unused variable
1	Predictor variable
2	Target variable
3	Weight variable

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

VariableType -- Get the type of a variable

int VariableType(int *VariableIndex*)

The VariableType method returns a number indicating the type of a variable. There are two type values:

0	Continuous variable (e.g., age, income, height, weight).
1	Categorical variable (e.g., sex, race, religion)

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

VariableImportance -- Get the importance of a variable

double VariableImportance(int *VariableIndex*)

The VariableImportance method returns a double precision real number in the range 0.0 to 1.0 indicating the relative importance of a predictor variable. The most important variable has an importance score of 1.0. Other variables have relatively smaller importance scores. Note: in the report generated by DTREG, the importance scores are multiplied by 100.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number. The VariableImportance function should only be called with index numbers for predictor variables.

GetNumberOfCategories -- Get number of categories of a variable

int GetNumberOfCategories(int *VariableIndex*)

The GetNumberOfCategories method returns the number of categories for a categorical variable. For example, the number of categories for a “sex” variable will be 2.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

CategoryIndex -- Get index number for a category

int CategoryIndex(int *VariableIndex*, String ^*CategoryLabel*)

The CategoryIndex method returns an index number that identifies a category of a categorical variable. For example, a categorical "sex" variable might have two category labels, "male" and "female". The CategoryIndex method takes the index number of a variable and a category label as its arguments and returns the corresponding category index number. The index number of the first category is 0 (zero).

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

CategoryLabel is a string containing the label of the category whose index number is to be found.

The returned value of the method is the index number of the category. A value of -1 is returned if no category with the label is associated with the specified variable.

GetCategoryLabel -- Get the label of a category from its index number

String ^GetCategoryLabel(int *VariableIndex*, int *CategoryIndex*)

The CategoryLabel method is the complement of the CategoryIndex method. The CategoryLabel method takes the index number of a category label as the argument and returns the corresponding label string as its result.

Arguments:

VariableIndex is a numeric (long) value with the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

CategoryIndex is the index number of the category whose label is desired.

SetContinuousValue -- Set the value for a continuous variable

int SetContinuousValue(int *VariableIndex*, double *Value*)

The SetContinuousValue method is used to set the value of a continuous (numeric) predictor variable prior to computing the predicted target value. Use the SetCategoryValue method (described below) to set the value for a categorical variable.

You must use SetContinuousValue and/or SetCategoryValue to specify the values of all predictor variables before you reference the ComputeScore function to predict the target value based on the current set of predictor values.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

Value is a double-precision value that is to be set for the specified variable. Use the MissingValue property (see below) to get the value that you should specify to indicate a missing value for a variable.

The returned value of the method is 0 for success or a non-zero status code if an error occurs. See the description of the LastStatus method for information about status codes.

SetCategoryValue -- Set the category for a variable

int SetCategoryValue(int *VariableIndex*, String ^*CategoryLabel*)

The SetCategoryValue is used to set the value of a categorical predictor variable prior to computing the predicted target value. Use the SeContinuousValue method (described above) to set the value for a continuous (numeric) variable.

You must use SetContinuousValue and/or SetCategoryValue to specify the values of all predictor variables before you reference the ComputeScore function to predict the target value based on the current set of predictor values.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

CategoryLabel is a string value with the category label that is to be set for the specified variable. If the variable has numeric category values, you may specify a numeric value for the category label instead of a string.

For missing values, specify either a null (empty) string or a string consisting of a single question-mark (“?”) character.

The returned value of the method is 0 for success or a non-zero status code if an error occurs. See the description of the LastStatus method for information about status codes.

GetContinuousVariable -- Get value of a continuous variable

double GetContinuousVariable(int *VariableIndex*)

The GetContinuousVariable function returns the current value of a variable. This can be used after SetContinuousValue to check a value. It is especially useful when a DTL program computes values for some variables: You can set the input values and then call this function to determine what value DTL has computed for the generated variables. If the variable is a categorical variable with string values, you can assign the result to a string variable rather than a double variable to get the category label for the variable.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

GetCategoricalVariable -- Get value of a categorical variable

String ^GetCategoricalVariable(int *VariableIndex*)

The GetCategoricalVariable function returns the current value of a categorical variable. This can be used after SetCategoricalValue to check a value. It is especially useful when a DTL program computes values for some variables: You can set the input values and then call this function to determine what value DTL has computed for the generated variables. If the variable is a categorical variable with string values, you can assign the result to a string variable rather than a double variable to get the category label for the variable.

Arguments:

VariableIndex is the index number of the variable. You can use the VariableIndex method to convert the name of a variable to its index number.

GetMissingValue -- Numeric value to use for missing values

double GetMissingValue()

The GetMissingValue function returns a numeric value that can be used with the SetContinuousValue method to indicate a missing value for a continuous variable.

ComputeScore -- Compute predicted target value

int ComputeScore()

When you call the ComputeScore function, DTREG computes the predicted value of the target variable using the current model and the predictor variable values most recently set using the SetContinuousValue and/or SetCategoryValue methods. The function returns 0 if the calculation was successful or a status code if there was an error. See the description of the LastStatus function for a list of the status codes.

GetPredictedValue -- Get predicted continuous value

double GetPredictedValue()

The GetPredictedValue function can be called after ComputeScore to obtain the predicted value of a continuous target variable. Use GetPredictedCategory to get the predicted value of a categorical target variable.

GetPredictedCategory -- Get predicted category label

String ^GetPredictedCategory()

The GetPredictedCategory function can be called after ComputeScore to obtain the predicted category label value of a categorical target variable. Use GetPredictedValue to get the predicted value of a continuous target variable.

GetPredictedCategoryIndex -- Get predicted category index

Int GetPredictedCategoryIndex()

The GetPredictedCategoryIndex function can be called after ComputeScore() to obtain the index number of the predicted category of a categorical target variable. Use GetPredictedValue to get the predicted value of a continuous target variable.

GetProbabilityScore -- Get probability of target category

double GetProbabilityScore(int *CategoryIndex*)

You can call GetProbabilityScore after ComputeScore() to obtain the computed probability score for each possible category of the target variable. This method can be used only if the target variable is categorical. Note that you must call ComputeScore() to compute the predicted value before using this function. Probability values are in the range 0.00 to 1.00.

Arguments:

CategoryIndex is the index of the target category whose probability is being requested. You can use the CategoryIndex method to convert a category label to a category index number.

Procedures for Training Models

The procedures described in this section are used to train a DTREG model.

int BeginTraining(String ^*ModelTitle*)

You must call BeginTraining to start the process of training a new model. Any model that is open when this is called is closed, and a new model is started.

Arguments:

ModelTitle is a text string that will be used as the title of the project.

The returned value of the method is 0 for success or a non-zero status code if an error occurs. See the description of the LastStatus method for information about status codes.

int BeginVariableDefinitions()

Call this procedure to begin the process of defining variables. The DefineVariable() procedure can be called after this to define each variable.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int DefineVariable(String ^*VariableName,* int *Type,* bool *Categorical*)

Call DefineVariable() for each variable. The order in which the variables are specified must match the order of the columns in the data records.

Arguments:

VariableName is the name of the variable.
Type is the type of the variable: 0=unused, 1=predictor, 2=target, 3=weight.
Categorical is True if the variable is categorical or False if the variable is continuous

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int EndVariableDefinitions()

Call EndVariableDefinitions() after defining all of the variables.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int BeginStoringData(String ^*ColumnDelim,* int *EstimatedRecords*)

Call BeginStoringData() to start the process of feeding data records into DTREG to train a mode.

Arguments:

ColumnDelim is a string the first character of which specifies the character that will be used as the delimiter between columns in the data rows.
EstimatedRecords specifies the estimated number of data rows that will be entered. Specifying an estimate allows DTREG to preallocate memory. You can specify 0 (zero) in which case DTREG will allocate memory in blocks as data records are added.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int StoreDataRow(String ^*DataRecord*)

Call StoreDataRow() for each data row in the training file.

Arguments:

DataRecord is a string containing the data record to be stored. The columns must occur in the order in the variables were defined by DefineVariable(), and the column delimiter character must match the first argument of BeginStoringData().

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int EndOfData()

Call EndOfData() after the last data row has been specified.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int ReadTrainingData(String ^*FileName*, String ^*ColumnDelimiter*)

Call ReadTrainingData() to read an entire csv file with training data. *FileName* is the fully qualified name of the file to be read, and *ColumnDelimiter* is the character used to delimit the columns of data.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

Here is an example code block that reads a training file and sets the variable named “outcome” to be a categorical target variable and all other variables to be continuous predictor variables:

```
int intStatus,i,intNumVar;
string strVarName;
/* Read the csv training data file */
intStatus = objDtreg.ReadTrainingData(@"C:\Test\Data.csv", ",");
/* Set up information about the variables */
intNumVar = objDtreg.GetNumberOfVariables();
for (i = 0; i < intNumVar; i++)
{
    strVarName = objDtreg.VariableName(i);
    if (strVarName == "outcome")
    {
        /* Set target variable to be categorical */
        objDtreg.SetVariableType(i, 1);
        objDtreg.SetVariableClass(i, 2);
    }
    else
    {
--------/* Set predictor variable to be continuous */
        objDtreg.SetVariableType(i, 0);
        objDtreg.SetVariableClass(i, 1);
   }
}
```

SetVariableType -- Set the type of a variable

int SetVariableType(int *VariableIndex,* int *VariableType*)

The SetVariableType method sets the type of a variable. It is normally called after ReadTrainingData() to set the type of each variable read from the training file. There are two type values:

0	Continuous variable (e.g., age, income, height, weight).
1	Categorical variable (e.g., sex, race, religion)

Note, you can use the VariableIndex() method to convert the name of a variable to its index number.

SetVariableClass -- Set the class of a variable

int SetVariableClass(int *VariableIndex,* int *VariableClass*)

The SetVariableClass method sets the class of a variable. It is normally called after ReadTrainingData() to set the class of each variable read from the training file. There are four possible class values:

0	Unused variable
1	Predictor variable
2	Target variable
3	Weight variable

Note, you can use the VariableIndex() method to convert the name of a variable to its index number.

SetCategoryBalancingMethod – Set method for balancing target categories

Int SetCategoryBalancingMethod(int *BalancingMethod*)

Call this function to set the procedure to be used to balance the categories of the target variable. If used, this procedure must be called before EndOfData().

BalancingMethod may have one of these values:

0 = No balancing. Use the natural distribution of target categories.
3 = Adjust row weights so that weights for all categories are the same

SetValidationWeighting – Set validation row weighting

Int SetValidationWeighting(int *ValidationWeighting*)
Set whether the rows used for validation are to be weighted to balance the target categories.

ValidationWeighting may have one of these values:

0 = Do not do row weighting for validation
1 = Weight rows during validation to balance target categories

SetMisclassificationCostType – Method for selecting decision threshold

Int SetMisclassificationCostType(int *CostType*)
Set the procedure to be used to select the decision threshold.

CostType may have one of these values:
1 = Equal misclassification costs
2 = Use misclassification costs from cost matrix (see SetCostMatrix).
3 = Use probability threshold specified by SetMisclassificationThreshold().
4 = Minimize the total (unweighted) error
5 = Minimize the weighted error
6 = Select threshold to balance misclassification percents across target categories.

SetMisclassificationThreshold – Set decision probability threshold

Int SetMisclassificationThreshold(double *ProbabilityThreshold*)
If 3 is specified as the misclassification cost procedure by SetMisclassificationCostType(), then use this procedure to specify the true/false probability threshold for the decision point.

SetCostMatrix – Set value in misclassification cost matrix

Int SetCostMatrix(int *NumTargetCategories*, int *ActualTargetCategory*, int *PredictedTargetCategory*, double *Cost*)
If 2 is specified as the misclassification cost procedure by SetMisclassificationCostType(), then use this procedure to set values in the misclassification cost matrix. The "cost" is the penalty for misclassifying from a specified correct category to a specified incorrect category.

NumTargetCategories = Number of categories of the target variable.
ActualTargetCategory = Index of the actual (correct) category of the target variable.
PredictedTargetCategory = Index of the incorrect category of the target variable.
Cost = Cost of misclassification from the actual to the incorrect category.

SetPositiveTargetCategory – Set target category considered positive

Int SetPositiveTargetCategory(int CategoryIndex)
In some of the reports generated by DTREG such as sensitivity and specificity, one category of the target variable is considered to be the "positive" category. Use this procedure to specify the category index of the target variable to be treated as the positive category.

int GenerateModel()

Call GenerateModel() to do the actual model training after variables have been defined, model parameters have been specified and all data rows have been entered.

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

int SaveProject(String ^*FileName*)

Call the SaveProject() procedure to write a model to a .dtr project file.

Arguments:

FileName is the fully-qualified name of the file; for example, "C:\Test\NewModel.dtr".

The returned value of the method is 0 for success or a non-zero status code if an error occurs.

Decision Tree Parameters

The following procedures set options for building decision tree models. The returned value of all functions is 0 for success or a non-zero error code.

int SetDecisionTreeMinimumNodeRows(int *MinimumNodeRows*)

This procedure sets the minimum number of rows that will be place in any node of the tree.

int SetDecisionTreeMinimumRowsSplit(int *MinimumRows*)

This procedure sets the minimum number of rows that may be in a node for DTREG to consider the node eligible for splitting.

int SetDecisionTreeMaxTreeLevels(int *MaxLevels*)

This procedure sets the maximum number of levels to which a decision tree will be built. Note, pruning may reduce the number of levels in the final tree.

int SetDecisionTreeReportSplits(bool *ReportSplits*)

This procedure sets the option that controls whether a detailed report of each split in the tree is written to the analysis log.

int SetDecisionTreeValidationMethod(int *ValidationMethod*)

This procedure sets which validation method will be used.

The following values may be specified for *ValidationMethod*:
1 = Do not perform validation
2 = Use V-fold cross-validation
3 = Hold out a percentage of the training rows to use for validation
4 = Prune the tree to a specified number of fixed number of terminal nodes
7 = Use a variable to determine which rows to use for training and which for testing

int SetDecisionTreePruneNodes(int *PruneNodes*)

If option 4 is specified for SetDecisionTreeValidationMethod, then this procedure sets the number of nodes that the tree is pruned to.

int SetDecisionTreeSmoothSpikes(int *SmoothSpikes*)

Sets the number of entries in the tree size list that are evaluated when deciding the optimum prune size.

int SetDecisionTreePruneType(int *PruneType*)

Sets the procedure used when pruning the tree.

The following values may be specified for *PruneType*:
1 = Prune to the absolute minimum cross validation error
2 = Allow pruning to a smaller tree within 1 standard error of the minimum
3 = Prune to a specified number of standard error of the minimum
4 = Do not prune the tree

int SetDecisionTreePruneMargin(double *PruneMargin*)

If option 3 is specified for SetDecisionTreePruneType, this parameter specifies the number of standard errors from the minimum allowed for pruning.

TreeBoost Parameters

The following procedures set options for building TreeBoost models. The returned value of all functions is 0 for success or a non-zero error code.

int SetTreeBoostSeriesLength(int *NumTrees*)

This procedure sets the number of trees in the TreeBoost series. Note, pruning parameters that are described below may cause the series to be truncated to a smaller number of trees.

int SetTreeBoostMaxTreeDepth(int *MaxDepth*)

This procedure sets the maximum depth (i.e., number of levels) for each tree in the series.

int SetTreeBoostMinimumNodeSplit(int *MinimumRows*)

This procedure sets the minimum number of rows that may be located in a node of the tree to allow it to be considered for further splitting.

int SetTreeBoostTreeRowProportion(double *Proportion*)

This procedure sets the proportion of rows selected for consideration by each tree in the series.

int SetTreeBoostHubersQuantileCutoff(double *Cutoff*)

This procedure sets the value of Huber's quantile cutoff to limit the effects of extreme outliers. This parameter pertains only to models with continuous target variables.

int SetTreeBoostInfluenceTrimmingFactor(double T*rimmingFactor*)

This procedure sets the influence trimming factor used to exclude rows with limited remaining influence.

int SetTreeBoostSeriesLength(int *NumTrees*)

This procedure sets the number of trees in the TreeBoost series. Note, pruning parameters that

int SetTreeBoostAutoShrink(bool *AutoShrink*)

This procedure sets whether the shrinking factor should be set automatically.

int SetTreeBoostFixedShrinkFactor(double *ShrinkFactor*)

This procedure sets the shrinking factor that is used if automatic shrinking is disabled.

int SetTreeBoostMaxNodesPerTree(int *MaxNodes*)

This procedure sets the maximum number of nodes (leaves) for each tree in the series.

int SetTreeBoostPredictorSelectionMethod(int *PredictorSelection*)

This procedure sets the method used to select predictors for each tree in the series. The following values may be specified:

1 = Use a fixed number of predictors
2 = Use square root of the number of predictors
3 = Search for the optimum number of predictors
4 = Use all predictors
5 = Use a specified proportion of the predictors

int SetTreeBoostPredictorSelectionProportion(double *Proportion*)

This procedure sets the proportion of the predictors to use for each tree in the series.

int SetTreeBoostPredictorSelectionSearch(int *NumPredictorSearch*)

This procedure sets the number of trial series to build while searching for the optimal number of predictors.

int SetTreeBoostPredictorSelectionFixed(int *NumPredictors*)

This procedure sets a fixed number of predictors to use for each tree in the series.

int SetTreeBoostValidationMethod(int *ValidationMethod*)

This sets the method used to validate the model. The following values may be specified for ValidationMethod:

1 = Do not perform validation
2 = Use V-fold cross-validation
3 = Hold out a specified proportion of the rows for validation
7 = Use a variable to determine which rows to use for training and which for testing

int SetTreeBoostValidationPercent(int *ValidationPercent*)

Specify the percent of rows to be held out for validation if option 3 is specified with SetTreeBoostValidationMethod().

int SetTreeBoostCrossValidationFolds(int *ValidationFolds*)

This specifies how many cross-validation folds to use if option 2 is specified by SetTreeBoostValidationMethod().

int SetTreeBoostSmoothMinimumSpikes(int *SmoothCount*)

This specifies how many trees should be considered when determining the optimal pruning point.

int SetTreeBoostMinimumTrees(int *MinimumTrees*)

This specifies the minimum number of trees to be left in the series after pruning.

int SetTreeBoostPruneMethod(int *PruneMethod*)

This specifies the method used to prune the TreeBoost series. The following values may be specified:

1 = Prune to absolute minimum error
4 = Do not prune

int SetTreeBoostPruneTolerancePercent(double *PruneTolerance*)

This specifies the percentage tolerance from the absolute minimum error value allowed when pruning the TreeBoost series.

int SetTreeBoostPruneCrossValidate(bool *PruneCrossValidate*)

This specifies if the model is to be validated again using cross validation after pruning has been done.

int SetTreeBoostCrossValidationVariables(bool *CrossValidateImportance*)

This specifies if the variable importance is to be determined from the cross validation series rather than the primary training series.

Decision Tree Forest Parameters

The following procedures set options for building decision tree forest models. The returned value of all functions is 0 for success or a non-zero error code.

int SetForestSize(int *ForestSize*)

Sets the number of trees in the forest

int SetForestMinimumNodeSplit(int *MinimumNodeSize*)

A node in a tree with fewer than this number of rows will not be further split.

int SetForestMaxDepth(int *MaxDepth*)

Sets the maximum number of levels in each tree in the forest.

int SetForestPredictorControlMethod(int *PredictorMethod*)

Sets the method used to select the number of predictors to include in each tree of the forest. The following values may be specified for inPredictorMethod:

1 = Fixed number of predictors
2 = Use square root of total number of predictors
3 = Build trial forests and search for the optimal number
4 = Use all predictors
5 = Use a specified proportion of the predictors

int SetForestNumberOfTrialForests(int *NumTrials*)

If option 3 is selected for SetForestPredictorControlMethod(), then this procedure specifies the number of trial forests to create.

int SetForestNumberOfPredictors(int *NumPredictors*)

If option 1 is selected for SetForestPredictorControlMethod(), then this procedure specifies the number of predictors to use in each tree.

int SetForestMissingValueMethod(int *MissingValueMethod*)

This procedure determines how missing values are handled when building decision tree forest models. The following values may be specified for MissingValueMethod:

1 = Use surrogate variables
2 = Use the median value of the variable

int SetForestPredictorImportanceMethod(int *ImportanceMethod*)

This procedure sets which method should be used to estimate the importance of predictor variables for decision tree forests. The following values may be specified for ImportanceMethod:

0 = Do not compute predictor importance
1 = Use tree split information (fast)
2 = Use type-1 margins
3 = Use type-1 and 2 margins

Support Vector Machine (SVM) Parameters

The following procedures set options for building SVM models.

int SetSvmModelType(int *ModelType*)

This procedure sets the type of SVM model to be built. The following values may be specified for ModelType:

1 = C-SVC
2 = Nu-SVC
3 = Epsilon-SVR
4 = Nu-SVR

int SetSvmKernelType(int *KernelType*)

This procedure sets the type of kernel used by SVM models. The following values may be specified for KernelType:

1 = Linear
2 = Polynomial
3 = RBF
4 = Sigmoid

int SetSvmStoppingCriteria(double *StoppingCriteria*)

This procedure sets the stopping criteria value that determines when SVM considers a solution to have been reached.

int SetSvmCacheSize(double *CacheSize*)

This sets the number of MB of memory space used for support vector caching while building a SVM model.

int SetSvmUseShrinkingHeuristics(bool *UseShrinkingHeuristics*)

Specify a value of True to enable shrinking heuristics or False to disable the heuristics.

int SetSvmComputeVariableImportance(bool *ComputeImportance*)

Specify True to tell SVM to compute the estimated importance of predictors, or specify False to disable variable importance calculations.

int SetSvmComputeProbabilityEstimates(bool *ComputeProbabilities*)

Specify True if you want SVM to compute probability estimates for categorical target variables. Specify False if you want only categorical predictions without associated probabilities.

int SetSvmValidationMethod(int *ValidationMethod*)

This sets the method used to validate the SVM model. The following values may be specified for ValidationMethod:

1 = Do not perform validation
2 = Use V-fold cross-validation
3 = Hold out a specified proportion of the rows for validation
7 = Use a variable to determine which rows to use for training and which for testing

int SetSvmValidationPercent(int *ValidationPercent*)

Specify the percent of rows to be held out for validation if option 3 is specified with SetSvmValidationMethod().

int SetSvmCrossValidationFolds(int *ValidationFolds*)

This specifies how many cross-validation folds to use if option 2 is specified by SetSvmValidationMethod().

int SetSvmMissingValueMethod(int *MissingValueMethod*)

Set which method is used to handle missing predictor values. The following values may be specified for MissingValueMethod:

0 = Exclude rows that have any missing predictor variable values
1 = Replace missing values with the median value of the variable
3 = Use surrogate variables to impute missing values

int SetSvmGridSearchEnable(bool *EnableGridSearch*)

Specify True to enable the grid search for optimal parameter values. Specify False if you want to turn off the grid search and use specified parameter values.

int SetSvmGridIntervals(int *GridIntervals*)

Specify the number of grid intervals. This is the number of trial values for each SVM parameter.

int SetSvmGridRefinements(int *GridRefinements*)

Specify how many times progressively finer grid searches should be performed to improve the parameter value estimates.

int SetSvmPatternSearchEnable(bool *EnablePatternSearch*)

Specify True to enable a pattern search for optimal parameter values. Specify False if you want to disable the pattern search.

int SetSvmPatternSearchIntervals(int *PatternIntervals*)

Specify the number of pattern search intervals.

int SetSvmPatternTolerance(double *PatternTolerance*)

Set the tolerance (convergence goal) for the pattern search. The search stops when the parameter values converge to this.

int SetSvmGridRowPercent(double *RowPercent*)

Set the percent of the data rows to be used for the grid and pattern searches.

int SetSvmGridCrossValidationFolds (int *NumFolds*)

Set the number of cross validation folds to be used for the grid and pattern searches.

int SetSvmGridOptimizeMethod(int *OptimizeMethod*)

Specifies what error measure is to be optimized by the SVM model training. The following values may be specified:

1 = Total unweighted error
2 = Error weighted by class frequency
3 = Maximize AUC
4 = Maximize geometric mean of sensitivity and specificity

int SetSvmCurrentC(double *ParamValue*)

Set the current value of the C parameter.

int SetSvmMinimumC(double *ParamValue*)

Set the minimum value of the C parameter to be used for the search.

int SetSvmMaximumC(double *ParamValue*)

Set the maximum value of the C parameter to be used for the search.

int SetSvmCurrentNu(double *ParamValue*)

Set the current value of the Nu parameter.

int SetSvmMinimumNu(double *ParamValue*)

Set the minimum value of the Nu parameter to be used for the search.

int SetSvmMaximumNu(double *ParamValue*)

Set the maximum value of the Nu parameter to be used for the search.

int SetSvmCurrentGamma(double *ParamValue*)

Set the current value of the Gamma parameter.

int SetSvmMinimumGamma(double *ParamValue*)

Set the minimum value of the Gamma parameter to be used for the search.

int SetSvmMaximumGamma(double *ParamValue*)

Set the maximum value of the Gamma parameter to be used for the search.

int SetSvmUseDefaultGamma(bool *UseDefaultGamma*)

Set to True to use the default value of the Gamma parameter which is 1/K.

int SetSvmCurrentP(double *ParamValue*)

Set the current value of the P parameter.

int SetSvmMinimumP(double *ParamValue*)

Set the minimum value of the P parameter to be used for the search.

int SetSvmMaximumP(double *ParamValue*)

Set the maximum value of the P parameter to be used for the search.

int SetSvmCurrentCoef0(double *ParamValue*)

Set the current value of the Coef0 parameter.

int SetSvmMinimumCoef0(double *ParamValue*)

Set the minimum value of the Coef0 parameter to be used for the search.

int SetSvmMaximumCoef0(double *ParamValue*)

Set the maximum value of the Coef0 parameter to be used for the search.

int SetSvmCurrentDegree(double *ParamValue*)

Set the current value of the Degree parameter.

Multilayer Perceptron Neural Network Parameters

The following procedures set options for building multi-layer perceptron neural network models.

int SetMlpLayers(int *NumLayers*)

Set the total number of layers in the network. The parameter value must be 3 or 4. Specifying 3 total layers corresponds to 1 hidden layer. Specifying 4 total layers corresponds to 2 hidden layers.

int SetMlpL1Search(bool *DoSearch*)

Specify **true** to set DTREG to determine the number of neurons in hidden level 1 automatically by doing a search. Specify **false** if you are going to provide the number of neurons for hidden level 1.

int SetMlpL1Min(int *NumNeurons*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1, this is the minimum number of neurons to start the search at.

int SetMlpL1Max(int *NumNeurons*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1, this is the maximum number of neurons it will test.

int SetMlpL1Step(int *NumNeurons*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1, this is the number of neurons it will add to hidden level 1 during each search cycle.

int SetMlpL1MaxStepsFlat(int *NumSteps*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1 and it performs this many number of search steps without seeing an improvement in accuracy, it stops the search.

int SetMlpL1RowsProportion(double *Proportion*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1, this is the proportion of the total training rows that will be used during the search. Specify 1.0 to use all training rows.

int SetMlpL1Holdout(double *Proportion*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1 and you specified to hold out some percentage of the rows to use to test each step, this is the proportion of rows held out for testing.

int SetMlpL1SearchMethod(int *SearchMethod*)

If you set DTREG to search for the optimum number of neurons in hidden layer 1, this is the method that will be used to test each search step.

The following values may be specified for *SearchMethod*:
2 = Use cross validation
3 = Hold out a specified proportion of the rows for testing
6 = Use the training data for testing

int SetMlpL1Neurons(int *NumNeurons*)

If you set DTREG to use a fixed number of neurons for hidden level 1, this is the number of neurons to use.

int SetMlpL2Neurons(int *NumNeurons*)

This is the number of neurons to use for hidden layer 2 (if two layers are used).

int SetMlpDoEarlyStopping(bool *DoEarlyStopping*)

Specify **true** if you want DTREG to test for overfitting during the training process and stop the process early when it is detected. Specify **false** if you don't want to use early stopping.

int SetMlpEarlyStoppingRowsProportion(double *Proportion*)

This is proportion of the training rows to be held out to use for early stopping.

int SetMlpEarlyStoppingStepsFlat(int *NumSteps*)

If early stopping is used, the training process will stop if this many training iterations are performed without seeing improvement in the accuracy.

int SetMlpValidationMethod(int *ValidationMethod*)

This sets the method used to validate the model. The following values may be specified for ValidationMethod:

1 = Do not perform validation
2 = Use V-fold cross-validation
3 = Hold out a specified proportion of the rows for validation
5 = Use leave-one-out validation
7 = Use a variable to determine which rows to use for training and which for testing

int SetMlpValidationPercent(int *ValidationPercent*)

Specify the percent of rows to be held out for validation if option 3 is specified with SetMlpValidationMethod().

int SetMlpCrossValidationFolds(int *ValidationFolds*)

This specifies how many cross-validation folds to use if option 2 is specified by SetMlpValidationMethod().

int SetMlpMissingValueMethod(int *MissingValueMethod*)

Set which method is used to handle missing predictor values. The following values may be specified for MissingValueMethod:

0 = Exclude rows that have any missing predictor variable values
1 = Replace missing values with the median value of the variable
3 = Use surrogate variables to impute missing values

int SetMlpHiddenLayerFunction(int *FunctionIndex*)

Set the type of function to be used for the hidden layer(s):

1 = Linear
2 = Sigmoidal (logistic)

int SetMlpOutputLayerFunction(int *FunctionIndex*)

Set the type of function to be used for the output layer:

1 = Linear
2 = Sigmoidal (logistic)
3 = Softmax (classification problems only)

int SetMlpConvergenceTries(int *ConvergenceTries*)

Set the number of starting values to be used to try to achieve optimal convergence.

int SetMlpMaxIterations(int *MaxIterations*)

Set the maximum number of iterations allowed during the convergence process.

int SetMlpMaxIterationsFlat(int *MaxIterationsFlat*)

The convergence process will stop iterating when this number of iterations produce no improvement.

int SetMlpConvergenceTolerance(double *ConvergenceTolerance*)

Set the error tolerance that must be reached to stop the convergence iterations.

int SetMlpMinImprovementDelta(double *Delta*)

Set the amount that an iteration must improve the model to be counted as a positive step.

int SetMlpMinGradient(double *MinGradient*)

Set the minimum slope of the gradient to allow convergence iterations to continue.

int SetMlpMaxExecutionSeconds(double *MaxSeconds*)

Sets the maximum allowable execution time (in seconds) for the training process.

int SetMlpConjugateGradientType(int *CgType*)

Sets the type of conjugate gradient training method used. The following values may be specified:
1 = Scaled conjugate gradient (recommended)
2 = Traditional conjugate gradient

PNN/GRNN Model Parameters

The following procedures set options for building PNN and GRNN models.

int SetPnnSigmaType(int *SigmaType*)

Set the type of sigma values to be used for the model. The following values may be specified:

1 = Use a single sigma value for the whole model
2 = Use a separate sigma for each variable
3 = Use a separate sigma for each variable and target category

int SetPnnConstrainSigma(bool *ConstrainSigma*)

Specify True to constrain sigma values to the specified minimum and maximum.

int SetPnnMinimumSigma(double *MinimumSigma*)

Set the minimum allowable value for Sigma.

int SetPnnMaximumSigma(double *MaximumSigma*)

Set the maximum allowable value for Sigma.

int SetPnnReportSigmas(bool *ReportSigmas*)

Control whether the computed sigma values are written to the analysis report.

int SetPnnSearchSteps(int *SearchSteps*)

Set the number of steps to use when searching for the optimal sigma values.

int SetPnnPruneNeurons(bool *PruneNeurons*)

Specify True to prune (remove) unnecessary neurons after training the model.

int SetPnnPruneMethod(int *PruneMethod*)

Set the method to use to prune unnecessary neurons. The following values may be specified:

1 = Prune to the specified number of neurons
2 = Prune to the minimum error value
3 = Prune down to the minimum number of neurons

int SetPnnPruneNumNeurons(int *NumNeurons*)

Specify the number of neurons that the model should be pruned to.

int SetPnnPruneRetrain(bool *PruneRetrain*)

Specify True to retrain the model after pruning if finished.

int SetPnnValidationMethod(int *ValidationMethod*)

This sets the method used to validate the model. The following values may be specified for ValidationMethod:

1 = Do not perform validation
2 = Use V-fold cross-validation
3 = Hold out a specified proportion of the rows for validation
5 = Use leave-one-out validation
7 = Use a variable to determine which rows to use for training and which for testing

int SetPnnValidationPercent(int *ValidationPercent*)

Specify the percent of rows to be held out for validation if option 3 is specified with SetSvmValidationMethod().

int SetPnnCrossValidationFolds(int *ValidationFolds*)

This specifies how many cross-validation folds to use if option 2 is specified by SetPnnValidationMethod().

int SetPnnMissingValueMethod(int *MissingValueMethod*)

Set which method is used to handle missing predictor values. The following values may be specified for MissingValueMethod:

0 = Exclude rows that have any missing predictor variable values
1 = Replace missing values with the median value of the variable
3 = Use surrogate variables to impute missing values

int SetPnnKernelType(int *KernelType*)

Set the type of kernel function. The following values may be specified:

1 = Gaussian distribution
2 = Reciprocal of distance

int SetPnnComputeVariableImportance(bool *ComputeImportance*)

Specify true to cause DTREG to compute the importance of variables in a PNN model.

int SetPnnPriorsType(int *PriorsType*)

Set the assumed prior probability (“priors”) type. The following values may be specified:

1 = Equal (balance misclassifications)
2 = Use frequency distribution in the data set
3 = Use priors specified by category weights
4 = Use average of equal priors and distribution in data set

Logistic Regression Model Parameters

The following procedures set options for building logistic regression models.

int SetLogisticConvergenceTolerance(double *Tolerance*)

Set the error value that must be reached to detect convergence during the logistic regression.

int SetLogisticMaxIterations(int *MaxIterations*)

Set the maximum iterations to be performed in the logistic regression optimization procedure.

int SetLogisticConfidencePercent(double *Percent*)

Set the confidence interval to be displayed for logistic regression parameters.

int SetLogisticConvergenceTolerance(double *Tolerance*)

Set the error value that must be reached to detect convergence during the logistic regression.

int SetLogisticValidationMethod(int *ValidationMethod*)

This sets the method used to validate the model. The following values may be specified for ValidationMethod:

1 = Do not perform validation
2 = Use V-fold cross-validation
3 = Hold out a specified proportion of the rows for validation
7 = Use a variable to determine which rows to use for training and which for testing

int SetLogisticValidationPercent(int *ValidationPercent*)

Specify the percent of rows to be held out for validation if option 3 is specified with SetLogisticValidationMethod().

int SetLogisticCrossValidationFolds(int *ValidationFolds*)

This specifies how many cross-validation folds to use if option 2 is specified by SetLogisticValidationMethod().

int SetLogisticMissingValueMethod(int *MissingValueMethod*)

Set which method is used to handle missing predictor values. The following values may be specified for MissingValueMethod:

0 = Exclude rows that have any missing predictor variable values
1 = Replace missing values with the median value of the variable
3 = Use surrogate variables to impute missing values

int SetLogisticIncludeConstant(bool *IncludeConstant*)

Set whether to include the constant (intercept) term in the logistic function.

int SetLogisticUseFirthProcedure(bool *UseFirth*)

Set whether to use Firth's Procedure when fitting the logistic function.

int SetLogisticComputeLikelihoodRatio(bool *ComputeLikelihood*)

Set whether to compute likelihood ratio significance tests.

int SetLogisticComputeVariableImportance(bool *ComputeImportance*)

Set whether to compute the importance of the predictor variables.

Index

T

V