Automating NeuralTools with VBA

The purpose of this document is to introduce the NeuralTools Excel Developer Kit (XDK) and explain how you can use VBA to automate NeuralTools. The term “automate” simply means that you write a program in VBA to make NeuralTools do what you normally do through the NeuralTools user interface. For example, suppose you periodically obtain new data from a given source, append it to an existing data set, retrain an existing neural net, make new predictions, and create one or more custom reports. This requires a considerable number of steps—the same steps—each time you do it. Wouldn’t it be nice to click a button and have all of these steps magically happen? This is exactly the type of thing you will learn how to do with the NeuralTools XDK.

NeuralTools has extensive documentation for its XDK. However, before you dive into the details, you probably need some motivation and guidance, and you are probably asking questions such as the following:

- Why would anyone want to write code to automate NeuralTools?
- What is VBA all about? Is it difficult to learn?
- How does VBA for Excel (which you might know) differ from VBA for NeuralTools?

This document answers these questions in a simple and intuitive way, using actual VBA code for illustration. If you want to start programming for NeuralTools, this is the document you should read first. You can also view the introductory video, available under the Developer Kit (XDK) item of NeuralTools Help.

Introduction to VBA

Visual Basic for Applications, or VBA, is the programming language for Excel. It is also the programming language for other applications in Microsoft Office: Word, PowerPoint, and Outlook, for example. It is even the programming language for some non-Microsoft products. To understand how this works, it helps to separate the name VBA into two parts: “Visual Basic” and “for Applications.” You can think of Visual Basic, or VB, as the “backbone” programming language. It contains programming elements that all programmers use, regardless of the programming language or the type of development task. For example, it contains subroutines, variable types, logical constructs such as If-Then-Else, loops for repetitive tasks, arrays for storing lists, and much more. Actually, there are non-VBA versions of VB. For example, you might be familiar with VB6 or VB.NET. All of these versions of VB, including VBA, are extremely popular because VB is fairly easy to learn and quite powerful. In any case, to learn VBA for Excel or NeuralTools, you must first learn the backbone VB language.

The following is a typical segment of VB code. Even if you know nothing about programming, you will probably have little trouble understanding what this code accomplishes. Again, this is the attraction of VB. It is relatively easy to learn and understand. Note the lines that begin with an apostrophe. These lines are called comments. They are ignored by VBA when the program runs, but they are very useful for documentation.

Sub Invoice()

---

1 If you are already familiar with VBA for Excel, you can skip to the section “Introduction to the NeuralTools XDK” on page 5.
' Declare variables.
Dim nProducts As Integer, i As Integer
Dim total As Currency, subTotal As Currency
Dim nPurchased As Variant, unitPrice As Variant

' Define constants for this business.
Const taxRate = 0.06
Const cutoff1 = 50, cutoff2 = 100
Const discount1 = 0.05, discount2 = 0.1

' Enter information about this order
nProducts = 4
nPurchased = Array(5, 2, 1, 6)
unitPrice = Array(20, 10, 50, 30)
total = 0

' Loop over all products purchased.
For i = 1 To nProducts
    ' Calculate revenue, including possible discount, for this product.
    subTotal = nPurchased(i) * unitPrice(i)
    If subTotal >= cutoff2 Then
        subTotal = (1 - discount2) * subTotal
    ElseIf subTotal >= cutoff1 Then
        subTotal = (1 - discount1) * subTotal
    End If
    ' Add to total for this order.
    total = total + subTotal
Next

' Add tax.
total = (1 + taxRate) * total

' Display result.
MsgBox "The total for this order, including tax, is " & Format(total, "$#,##0.00")
End Sub

If you run this code in Excel, you will see the display (a message box) in Figure 1.

**Figure 1 Result of VBA Program**

![Image of Excel message box showing the total]

**Subroutines, Programs, and Macros**

Before proceeding, it is useful to discuss three terms you often hear: subroutine, program, and macro. A **subroutine** is any section of code that begins with the keyword Sub and ends with the keyword End Sub. Its purpose is typically to perform one specific task. A **program** is a collection of one or more related subroutines that achieves an overall goal. In practice, long programs are often broken up into smaller subroutines for enhanced readability and easier debugging. A **macro** is essentially the same as a program, and it is the term favored by many Excel programmers. However, if your programs are relatively short, consisting of a single subroutine, the terms subroutine, program, and macro are practically synonymous and are often used interchangeably.

**Object Models**

The above Invoice subroutine can be run in Excel, but it really has nothing to do with Excel. There are no references to ranges, formulas, worksheets, charts, or other items you use in Excel. The code is pure VB.
But now we come to the “for Applications” part of VBA. All of the familiar items in Excel, which will now be called **objects**, are part of an **object model** that Excel provides to programmers through VBA. This means that you as a programmer can reference these objects in your VBA code. In short, this object model allows you to “make things happen” through VBA code, rather than through the usual Excel user interface. In this way, you can automate Excel with VBA.

So what does an object model, or Excel’s object model in particular, look like? At the top level, it is simply a list of things—objects—that the software contains. Excel’s object model is quite large because everything you see in Excel is an object. One of the most common objects is the Range object: any range of cells (including a single cell). But there are over a hundred more objects, including the Worksheet object, the Chart object, the PivotTable object, and so on. There are also singular and plural objects. For example, there is the plural Worksheets object, the collection of all worksheets in an Excel workbook, and there is the singular Worksheet object, a particular worksheet in the workbook.

Figure 2 displays a partial list of the objects in Excel, plural (yellow) and singular (blue).

However, there is much more to an object model than the list of its objects. First, there is a **hierarchy** among many objects. Basically, this means that you often need to “drill down” to get to the object you want. A good example is a Range object, such as cell B3. The idea is that a range is contained in a worksheet, which is contained in a workbook. So the hierarchy is Workbook → Worksheet → Range, and the following line of code is typical:

```
ActiveWorkbook.Worksheets("Costs").Range("B3").Value = 10
```

You read this as: Starting in the active workbook, go to cell B3 of the worksheet named Costs, and enter the value 10. This concept of hierarchy is crucial to learning VBA for Excel or NeuralTools.

Second, most objects have **properties** and **methods**. (Methods are also called **functions**.) If you think of objects as nouns, then you can think of properties as adjectives and methods as verbs. Properties describe an object, and methods indicate what you can do to, or with, an object. For example, a Range object has a Value property, which was used in the above line of code. Similarly, a Font object has a Color property, a Size property, and many others. A typical object with methods is the Worksheet object. It has a Delete method, a Move method, and a few others.

Although less common, many objects also have **events** that they respond to. A good example is the Open event of a Workbook object. This event occurs—it “fires”—when you open the workbook. VBA allows you to write **event handlers** for the event. This is code that is executed when the event fires.

As you begin VBA programming for Excel, you gradually learn the most common objects, the hierarchies between them, their most common properties, methods, and events, and the syntax for using these in VBA code. It takes practice and perseverance, but if you want to make Excel “sing and dance” with the click of a button, this is time well spent.
By now, you should be starting to see the big VBA picture. Just as the Excel application can be automated through its object model, other applications can be automated through their object models. Microsoft Word is a typical example. Its objects are not the same as Excel’s. Instead of Worksheet, Range, and other typical Excel objects, Word has Sentence, Paragraph, and other objects that are useful in word processing, and these objects have their own hierarchies, properties, methods, and events. Fortunately, if you already know VBA for Excel and you want to learn VBA for Word, all you need to learn is the object model for Word. The backbone VB language is exactly the same in both applications. Admittedly, it is not a trivial task to learn a new application’s object model, but knowing VB for one application, like Excel, provides a big head start for learning VB for another application, like Word.
Using the Visual Basic Editor (VBE)

You will be doing all of your programming in the Visual Basic Editor (VBE). The easiest way is to open VBE from Excel is to press Alt+F11. Alternatively, if the Excel Developer tab is visible, you can click the Visual Basic button on its ribbon. The VBE window appears in Figure 3. In particular, the Project pane on the left shows a list of all open workbooks. (To get back to the Excel window, you can click the Excel icon on the toolbar below the menu bar.)

Figure 3 Visual Basic Editor

As you can see in the figure, the selected file contains a module. You will write all of your code in modules. (Actually, there are exceptions to this, but they aren’t discussed here.) By default, a file doesn’t contain a module, but you can insert one through the Insert menu. Then you will see a big white space on the right where you can start typing your code. Some of the code for this particular module appears in the figure.

Introduction to the NeuralTools XDK

Like Excel and Word, NeuralTools has its own object model, as shown in Figure 4. Assuming you are familiar with the NeuralTools add-in, there shouldn’t be any surprises in this object model. It can be used to change application settings, create data sets, train and test neural nets, predict, generate reports, and manage neural nets. The details of these possibilities are discussed in some detail later in this document and in the accompanying NeuralTools XDK example files.
Two questions probably come to mind. Do you really need to learn how to automate NeuralTools with VBA, and, if you do, how do you learn the language?

There are at least two reasons why you might want to automate NeuralTools with VBA. First, it lets you automate tasks that you perform repeatedly. For example, suppose you need to download the most current data from a website, append to an existing data set, use NeuralTools to train a neural net to the data, and create a custom report. If you do this once or twice, it is probably not worth the effort to write a VBA program to automate the process. However, if you do this repeatedly, a VBA program can replace many mouse clicks with a single click. Indeed, this is why VBA for Excel is so popular among users—it lets them automate repetitive tasks with the click of a button. This same motivation certainly applies to repetitive tasks in NeuralTools.

Second, suppose you are developing models for colleagues or clients who have little experience with NeuralTools. Instead of teaching them how the NeuralTools user interface works, it might be easier to develop a VBA application that allows them to simply click a button to make things happen. These “things” could include defining a NeuralTools data set, training a neural net on the data set, using the net for prediction on a different data set, and other NeuralTools tasks.

Figure 4 NeuralTools Object Model

Assuming you think VBA for NeuralTools is for you, how do you learn the language? First, you need some facility in VBA for Excel. As discussed earlier, this requires you to learn the fundamentals of the VB
“backbone” language and the basics of the Excel object model. There are a number of reference books on VBA for Excel, as well as Excel’s own online help. One recommendation is the first 10 chapters of *VBA for Modelers* by Albright. This book provides concise explanations and plenty of example code to get you up to speed in both the VB language and the Excel object model.

Once you are familiar with VBA for Excel, you have at least four options—which are not mutually exclusive—for extending your knowledge to VBA for NeuralTools.

1. You can continue reading this document. It provides an overview of what can be done, and it provides sample code for how to do it. Alternatively, you can watch the introductory video that covers much of the same material. It is available from the Developer Kit (XDK) item under NeuralTools Help.

2. You can do what all good programmers do—mimic code written by others. Examples of programs written for NeuralTools are provided in a number of NeuralTools XDK example files. (They can be found from the Developer Kit (XDK) item under NeuralTools Help.) Once you see how something is done, such as creating a NeuralTools data set or training a net, you will probably have little difficulty adapting the code for your own use.

3. You can open the Reference document from the Developer Kit (XDK) item under NeuralTools Help and start browsing. This document is quite long, and no sane person would ever read it from beginning to end, but it gives you a quick sense of the NeuralTools object model, and it often enables you to find the specific details you need for your program.

4. You can open the Object Browser that is part of the Visual Basic Editor. This is particularly useful if you are in the midst of writing a program and need information on one specific detail. The Object Browser is discussed later in this document.

By the way, if you are familiar with VBA for Excel, you might want to start by recording macros for NeuralTools, just as you can do for Excel. Unfortunately, this doesn’t work. If you turn on the recorder and start clicking NeuralTools buttons, you will get some recorded code, but it will be entirely irrelevant. To automate NeuralTools, you have to write the code.

**Setting the Library Reference**

Before you can access the NeuralTools XDK with VBA code, you must first set a reference to the NeuralTools library. You do this by selecting **References** in the **Tools** menu of the VBE and checking the NeuralTools item, as illustrated in Figure 5. This should be the first step before writing any VBA macros to automate NeuralTools.
NeuralTools Automation Guidelines

This section provides some guidelines for automating NeuralTools with VBA code. The guidelines in this section are purposely kept brief and are intended only to get you started. For a more pictorial set of guidelines for the NeuralTools object model, you should examine the file NeuralTools XDK - Object Model Diagrams.xlsx. To see complete applications of NeuralTools automation, you should look at the accompanying NeuralTools XDK example files. And you can always visit the NeuralTools XDK Reference under NeuralTools Help or the Object Browser in the Visual Basic Editor.

By the way, you will notice that all of the formal names of NeuralTools objects start with NT, as in NTApplicationSettings. However, there are typically referenced by a property or method that doesn’t include the NT prefix, such as ApplicationSettings.

Getting Started: The “Root” Object

You start with the “root” object NTTools, followed by any object down the hierarchy from it. (Don’t fall into the trap of spelling this root as NeuralTools; it is simply NTTools.) For example, the following line changes an application setting so that no reports will be generated automatically.

```vba
NTTools.ApplicationSettings.ReportSettings.ReportsToGenerate_DeselectAll
```

Changing Application Settings

You can change application settings with NTTools.ApplicationSettings, just as you can through the usual Utilities menu in NeuralTools. However, as indicated in Figure 6, the NTApplicationSettings object has only one method and two properties. Typically, you will use the ReportSettings property to drill down to the settings you want to change. This references an NTReportSettings object, with a number of settings,
some of which appear in Figure 7. For example, the following line changes the placement of a detailed report (the report to the right of a data set) so that columns from a previous detailed report are overwritten by the new report.

```vba
    NDetailedReportPlacement_Overwrite
```

As you will see in the XDK example files, if you use VBA to change any report settings, it is always a good programming practice to restore these settings to their original values at the end of the program.

Figure 6 NTApplicationSettings Object Members
Before NeuralTools can analyze data, a NeuralTools data set must be created. To do this, you need to supply the range of the data, give the data set a name, and specify the role of each variable in the data set, which can be any of the following:

- NTVariableType_Unused
- NTVariableType_Independent_Category
- NTVariableType_Dependent_Category
- NTVariableType_Independent_Numeric
- NTVariableType_Dependent_Numeric
- NTVariableType_Tag

The following code is typical. You first define an array of variable types, and then you use the DefineDataSet and SetVariableTypes methods of the NTDataSetManager object (obtained through NTools.DataSetManager) to create and define the data set.

```vbnet
Sub CreateDataSet()
    Dim varType(0 To 2) As NTVariableType
    ' Define data set with given variable types
    varType(0) = NTVariableType_IndependentCategory
```
varType(1) = NTVariableType_IndependentNumeric
varType(2) = NTVariableType_DependentCategory
With NTools.DataSetManager
    .DefineDataSet ActiveSheet.Range("B3:D3002"), "My data set"
    .SetVariableTypes "My data set", varType
End With
End Sub

There is also a **DeleteDataSet** method. For example, you might get more data and then need to redefine the data set with an expanded data range. The only way you can do this through VBA code is to delete the existing data set and then create a new one with the appropriate data range. You can use the following line to delete the existing data set.

NTools.DataSetManager.DeleteDataSet "My data set"

**Training**

Once you have defined a NeuralTools data set, the code for training it is straightforward, as illustrated by the following code. (This code assumes that a data set named “Loan Data” already exists.) You first use **NTools.TrainingSetting** to specify training settings. This is analogous to clicking Train on the NeuralTools ribbon and filling out the dialog box. Then you use **NTools.Train** to train the net.

```vba
Sub TrainNet()
    ' Training settings
    With NTools.TrainingSettings
        .DataSetName = "Loan Data"
        .NameOfNet = "PNN Net for Loan Outcome"
        .AutomaticallyTest = True
        .AutomaticallyTest_Percent = 30
        .AutomaticallyPredict = True
        .AutomaticallyPredict_DirectlyInDataSet = True
        .AutomaticallyPredict_EnableLivePrediction = True
        .CalculateVariableImpacts = False
        .NetConfigurationSettings.TypeOfNet = NTNetConfiguration_PnnGrnn
        With .RuntimeSettings
            .TimeSpanStoppingCondition_Selected = True
            .TimeSpanStoppingCondition_Hours = (5# / 60#)
        End With
    End With

    ' Train the net
    NTools.Train
End Sub
```

This code will actually train, test, and predict. Specifically, because of the “AutomaticallyTest” properties, it will train on 70% of the cases and test on the other 30%, and because of the “AutomaticallyPredict” properties, it will predict all cases with missing values of the dependent variable. Also, depending on your report settings, it will automatically generate the detailed and summary reports you request. Finally, the net “PNN Net for Loan Outcome” will be stored as part of the workbook for later use if you need it. As you can see, a lot happens with very little code.

If your training settings specify that a percentage of the data should be used for testing, then this testing happens automatically when you train the net, and the testing results appear in the reports. Alternatively, you can use the TestingOutput property discussed in the next section to create a custom report for the test cases.

**Testing**

You might also want to use a trained net to test a second data set, where the values of the dependent variable are known. These training and testing data sets could be in different worksheets or even
different workbooks. The following code illustrates how this can be done. This code assumes that the data sets, named “Training Data” and “Testing Data,” already exist. The top part of the code corresponds to clicking the Train button on the NeuralTools ribbon; the bottom part corresponds to clicking the Test button (twice). After training a net on the training data, you use the TestingSettings property to specify the trained net for testing and the data set to be tested, and then you use the Test method to perform the testing. The training/testing terminology requires some clarification. When you test, using the TestingSettings and TestingOutput properties and the Test method in VBA, you are actually testing any specified data set with a trained net. In particular, this data set can be the training set itself.

```vba
Sub TrainAndTest()
    ' Training settings
    With NTools.TrainingSettings
        .dataSetName = "Training Data"
        .NameOfNet = "PNN Net for Loan Outcome"
        .AutomaticallyTest = False
        .AutomaticallyPredict = False
        .CalculateVariableImpacts = False
        .NetConfigurationSettings.TypeOfNet = NTNetConfiguration_PnnGrnn
        With .RuntimeSettings
            .TimeSpanStoppingCondition_Selected = True
            .TimeSpanStoppingCondition_Hours = (5#/ 60#)
        End With
    End With

    ' Train the net
    NTools.Train

    ' Use the net to test the data on the Training Data sheet
    With NTools.TestingSettings
        .SpecifyNetToTest_NetInWorkbook "PNN Net for Loan Outcome"
        .dataSetName = "Training Data"
    End With

    ' Test from the trained net
    NTools.Test

    ' Use the net to test the data on the Testing Data sheet
    With NTools.TestingSettings
        .SpecifyNetToTest_NetInWorkbook "PNN Net for Loan Outcome"
        .dataSetName = "Testing Data"
    End With

    ' Test from the trained net
    NTools.Test
End Sub
```

The following code illustrates how you can use the TestingOutput property to create a custom report. A lot of this code deals with careful positioning of the items in the report and their formatting, but pay particular attention to the properties in the With NTools.TestingOutput block. For example, the ClassificationMatrix property returns an object with its own properties so that you can “build” the usual classification matrix.

```vba
Sub CreateReport(dataSetName As String, startCell As Range)
    Dim categories() As String
    Dim i As Integer, j As Integer

    ' Headings
    With startCell
        .Value = "Results of testing trained net on " & dataSetName
        .Font.Bold = True
        .Offset(1, 0).Value = "Number of cases"
        .Offset(2, 0).Value = "% bad predictions"
    End With
```

Prediction and Detailed Report Information

The code in the Training subsection automatically predicts cases with missing values of the dependent variable. However, there are situations where you define two data sets, one for training (and possibly testing), and one for prediction. The first has no missing values of the dependent variable, and the second has all missing values of the dependent variable. These two data sets could be in different worksheets or even different workbooks. You train a net on the first data set and then use the trained net to predict for the second data set.

The following code illustrates how this can be done. This code assumes that the data sets, named “Training Data” and “Prediction Data,” already exist. The top part of the code corresponds to clicking the Train button on the NeuralTools ribbon; the bottom part corresponds to clicking the Predict button. After training a net on the training data, you use the PredictionSettings property to specify the trained net for prediction and the data set to be predicted. Then you use the Predict method to make the predictions.

Sub TrainAndPredict()
' Training settings
With NTools.TrainingSettings
  .DataSetName = "Training Data"
  .NameOfNet = "PNN Net for Loan Outcome"
  .AutomaticallyTest = True
  .AutomaticallyTest_Percent = 30
  .AutomaticallyPredict = False
  .CalculateVariableImpacts = False
  .NetConfigurationSettings.TypeOfNet = NTNetConfiguration_PnnGrnn
With .RuntimeSettings
  .TimeSpanStoppingCondition_Selected = True
  .TimeSpanStoppingCondition_Hours = (5#/ 60#)
End With
End With

' Train the net
NTools.Train

With NTools.TestingOutput
  startCell.Offset(1, 1).Value = .NumberOfTestingCases
  startCell.Offset(2, 1).Value = .CategoryPredictor_PercentBadPredictions / 100
With .ClassificationMatrix
  startCell.Offset(4, .NumberOfCategories + 1).Value = "% bad"
  startCell.Offset(4, .NumberOfCategories + 1).HorizontalAlignment = xlRight
  .GetCategories categories
  For i = 0 To .NumberOfCategories - 1
    ' Category name
    startCell.Offset(5, 0).Offset(i, 0).Value = categories(i)
    For j = 0 To .NumberOfCategories - 1
      startCell.Offset(5, 0).Offset(i, j + 1).Value = _
        .GetCountOfCategoryPredictionsForCategoryMembers(categories(i), _
          categories(j))
    Next j
  Next i
End With
End With
End Sub
'Use the net to predict for the data on the Predictions sheet
With NTools.PredictionSettings
  .SpecifyNetToUse_NetInWorkbook "PNN Net for Loan Outcome"
  .DataSetName = "Prediction Data"
  .EnableLivePrediction = True
  .PlacePredictedValuesDirectlyInDataSet = True
  .PredictForWhichCases = NTPredictionCases_All
End With
'
'Make the predictions
NTools.Predict
End Sub

When you train a neural net, you typically get a “detailed report” to the right of the training data set. Of course, you have control (through the user interface or through VBA) over the columns in this report, or whether the report is created at all. However, you can also create your own customized report with NTools.DetailedReportInformation, followed by any of the following methods:

- GetDataRange_Predictions
- GetDataRange_ProbabilitiesOfPredictions
- GetDataRange_Tags
- GetRange_ProbabilitiesOfCategory

By using these methods, you don’t have to make guesses about the ranges that contain the prediction information; the methods return the correct ranges automatically. For example, the following code illustrates how you can grab the predictions and their associated probabilities for a custom report.

Sub CopyDetailedReportInfo()
' This assumes a detailed report with predictions exists next to the data set.
' The code below then copies selected columns of this report for a custom
' report on the MySummary sheet.
  ' Headings on MySummary sheet
With Worksheets("MySummary")
    .Range("A1").Value = "Prediction"
    .Range("B1").Value = "Prob of Prediction"
End With

  ' Copy two columns of detailed report to MySummary sheet
With NTools.DetailedReportInformation
  .GetDataRange_Predictions.Copy Worksheets("MySummary").Range("A2")
  .GetDataRange_ProbabilitiesOfPredictions Worksheets("MySummary").Range("B2")
End With
End Sub

The GetDataRange_Tags method can be used if you want to train or test another net, using the same training/testing split. You might recall that a tag variable is a column that specifies whether each case is used for training, testing, or prediction. However, even if you don’t use an explicit tag variable, the tags are created implicitly. The GetDataRange_Tags method allows you to grab these tags and paste them next to the original data set.

Neural Net Manager

NeuralTools has a Neural Net Manager that you can access from the Utilities dropdown on the NeuralTools ribbon. This allows you to view, copy, or remove existing neural nets in a workbook. You can do the same with VBA, as illustrated in the following code. Starting with NTools.NeuralNetManager, the GetNamesOfNetsInWorkbook method is used to return the number of nets and an array of their names. If there are any nets, the For loop searches for a net called “PNN 1.” If there is any such net, the DeleteNetInWorkbook method is used to delete it.
Sub UseNeuralNetManager()
    Dim nNets As Long, i As Long
    Dim netName() As String
    With NTools.NeuralNetManager
        .GetNamesOfNetsInWorkbook ActiveWorkbook, nNets, netName
        If nNets > 0 Then
            ' If a net with name "PNN 1" exists, delete it
            For i = 1 To nNets
                ' The array is 0-based
                If netName(i - 1) = "PNN 1" Then
                    .DeleteNetInWorkbook ActiveWorkbook, "PNN 1"
                    Exit For
                End If
            Next
        End If
    End With
End Sub

From NTools.NeuralNetManager, three other methods are available:

- CopyNetFromExternalFileToWorkbook
- CopyNetFromWorkbookToAnotherWorkbook
- CopyNetFromWorkbookToExternalFile

These methods have the obvious meanings and are not illustrated here.

Some General VBA Tips

This guide concludes with a few VBA tips that you should know regardless of whether you are automating NeuralTools or Excel.

File Format (xlsm)

If you save a workbook that includes VBA code, you must save it as a macro-enabled (.xlsm) file. This is true for any Excel file that contains VBA code; it is not specific to NeuralTools files. Then if you open this .xlsm file later on, you will be warned that it contains macros. Make sure you elect to enable the macros; otherwise the VBA macros won’t work.

Running a Macro

If you develop a VBA program for nontechnical users, you probably won’t want them to see your code, either for proprietary reasons or because it would be too intimidating. However, it is easy to create a simple user interface for running the program. To do so, activate a worksheet, insert a shape such as a rectangle, right-click it, select Assign Macro, and select your macro. You can also insert descriptive text, such as Run Program, in the shape. From then on, a user can simply click the shape to run the program.

ThisWorkbook

You can always reference a workbook by name, as in Workbooks("My Example File.xlsm"). Alternatively, you can refer to the active workbook with the built-in object ActiveWorkbook. However, an even safer reference is to ThisWorkbook, which always references the workbook containing the VBA code. (It is safer than ActiveWorkbook because the workbook containing the code might not be active when you run your program.)
Worksheet Code Names

The code in the example files sometimes references worksheets by their “code” names. A worksheet actually has two names, the name you see on its tab, and a code name, which can be set only in the Visual Basic Editor. This is illustrated in Figure 8 (see the highlighted line in the Properties section). In this example, the notation wsData (Data) in the Project section indicates that Data is the name on the tab and wsData is the code name. (Any names can be used for code names, but a common practice is to use the prefix ws.) One reason for using code names in VBA programs is that you don’t need to create a Worksheet object variable; you can simply write wsData.Range("A1"), for example. A second reason is that if your code refers to the worksheet by name, as in Worksheets("Data"), and someone changes the name on the worksheet tab, an innocent enough change to make, this will break your program. However, because code names can be changed only through the Visual Basic Editor, it is much less likely that anyone will change them.

Figure 8 Code Name for Worksheet
**With Blocks**

In the XDK example files, you will see a lot of **With** blocks, sections that start with the keyword **With** and end with the keyword **End With**. These are used by all good VBA programmers, so you should get used to them. They allow you to avoid a lot of repetitive code, as illustrated by the following block of code.

Inside the **With NTools.TrainingSettings** block, anything that starts with a period really has **NTools.TrainingSettings** to the left of it. For example, **.NameOfNet** is equivalent to **NTools.TrainingSettings.NameOfNet**. As you can see, these With blocks can be nested to avoid even more repetitive code. For example, **.TimeSpanStoppingCondition_Hours** is equivalent to **NTools.TrainingSettings.RuntimeSettings.TimeSpanStoppingCondition_Hours**. It is a very good idea to indent these blocks appropriately, as has been done here, for easier reading.

```vba
With NTools.TrainingSettings
    .DataSetName = "Loan Data"
    .NameOfNet = "PNN Net for Loan Outcome"
    .AutomaticallyTest = True
    .AutomaticallyTest_Percent = 30
    .AutomaticallyPredict = True
    .AutomaticallyPredict_DirectlyInDataSet = True
    .AutomaticallyPredict_EnableLivePrediction = True
    .CalculateVariableImpacts = False
    .NetConfigurationSettings.TypeOfNet = NTNetConfiguration_PnnGrnn
    With .RuntimeSettings
        .TimeSpanStoppingCondition_Selected = True
        .TimeSpanStoppingCondition_Hours = (5# / 60#)
    End With
End With
```

**Adding to a Collection**

Virtually all collection objects have an **Add** method for adding an item to the collection. For example, the **Worksheets** collection has an **Add** method for adding a worksheet to a workbook. The following code is typical. It adds a new worksheet after the last worksheet and gives it a couple properties. The **Add** line is compact in that it not only adds the worksheet (and makes it the *active* worksheet), but it returns a reference to the new worksheet so that the ws object variable can be “set.”

```vba
Dim ws As Worksheet
Set ws = Worksheets.Add(after:=Worksheets(Worksheets.Count))
With ws
    .Name = "Revenues"
    .Range("A1").Value = "Revenues for our new product"
End With
```

If the ws object variable is not needed later in the program, this code can be compacted even further, as follows. Now there is no ws object variable. The **With** line creates an implicit reference to the new worksheet, so that its properties can be referenced inside the **With** block.

```vba
With Worksheets.Add(after:=Worksheets(Worksheets.Count))
    .Name = "Revenues"
    .Range("A1").Value = "Revenues for our new product"
End With
```

**Intellisense**

The reference to the NeuralTools library mentioned earlier is not only required to make your NeuralTools code work properly, but it provides you with an invaluable programming aid: Intellisense. As an example, consider the following line:

```vba
NTools.TrainingSettings.NetConfigurationSettings.TypeOfNet = NTNetConfiguration_PnnGrnn
```
Could you remember all of this? Fortunately, you don’t have to. As soon as you type **NTools**. (including the period), you will see a list you can choose from. Then when you choose **TrainingSettings** from the list and type a period, you will see another list to choose from. And so on. To cap it off, when you have entered everything to the left of the equals sign and then type the equals sign, you will see a list of the appropriate NeuralTools constants for the **TypeOfNet** property that you can choose from. If you are already an Excel programmer, you undoubtedly appreciate the power of Intellisense, and you will be happy to know that it extends to VBA for NeuralTools. If you are not yet familiar with Intellisense, you are in for a big treat!

**Built-In Constants**

The line `.TypeOfNet = NTNetConfiguration_PnnGrnn` contains one of many built-in NeuralTools constants, in this case, NTNetConfiguration_PnnGrnn. All of these constants begin with NT, and they all have integer values, such as 3 or 445. However, because no person could ever remember these values, they are instead given meaningful names that programmers can use in their VBA code. You can find the names in the Object Browser (discussed next). For example, Figure 9 lists the constants (on the right) for the three possible settings of the **TypeOfNet** property. Together, this set of three constants is called an **enumeration**. (Strictly speaking, the **TypeOfNet** property returns the NTTypeOfNetConfiguration enumeration. This is only one of several enumerations available in NeuralTools, indicated by the double-yellow icons in the figure.²)

**Figure 9 NTTypeOfNetConfiguration Enumeration**

Excel also has many enumerations. Its built-in constants all begin with xl, as in xlDown, xlToRight, and so on.
Object Browser

A great source of information is the VBA Object Browser. To see this, make sure a file with a reference to the NeuralTools library is selected in Project pane. Then click the circled button in Figure 10, and select NeuralTools from the libraries dropdown list. This Object Browser shows all of the NeuralTools objects on the left and all of their properties and methods on the right. Also, when you select an item on the right, you get some limited help, such as the list of a function’s arguments, at the bottom of the window. Remember that such help is only a few mouse clicks away as you are doing your programming. It often provides just the detail you need to get you past a programming hurdle.

Figure 10 Visual Basic Object Browser