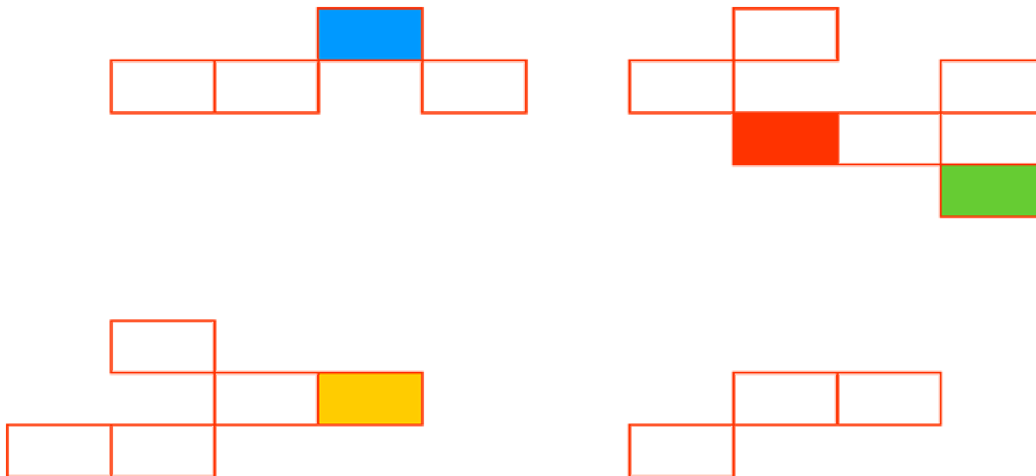
Online Books

Microsoft

Microsoft® SQL Server™ Series

Preparing and Mining Data with Microsoft® SQL Server™ 2000 and Analysis Services

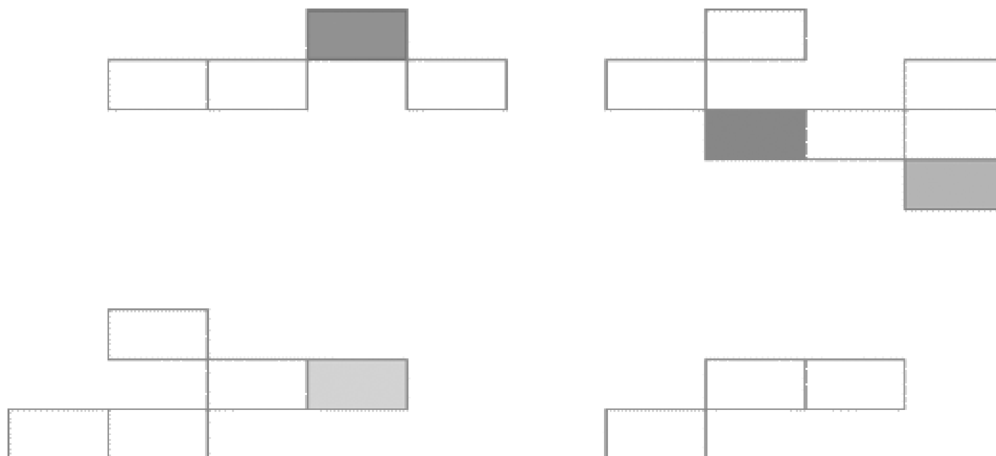


Seth Paul
Nitin Guatam
Raymond Balint

Microsoft .net Enterprise Servers

Microsoft®

Preparing and Mining Data with Microsoft® SQL Server™ 2000 and Analysis Services



Seth Paul
Nitin Guatam
Raymond Balint

Copyright

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

©2002 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, and Visual Basic are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Published: September 2002
Updated: September 2002
Applies To: Microsoft® SQL Server™ 2000

Editor: Margaret Sherman
Artist: Steven Fulgham
Production: Stephanie Schroeder

Table of Contents

Introduction 9
- Introducing the Data Mining Scenario 10
- What Will You Learn from this Book? 12
- Who Should Read This Book? 12
- What Technologies Does This Book Cover? 13
- How Is This Book Structured? 13
- Security 14

Chapter 1
Setup 15
- Reviewing the Setup Wizard Requirements 15
- Reviewing What the Setup Wizard Does 17
- Setting Up the SQL Server Database 18
- Setting Up the Analysis Server 19
- Connecting to the SQL Server Database 20
- Selecting a Table in the Data Mining Tool 21

Chapter 2
Data Mining Fundamentals 23
- What Is Data Mining? 23
  - Defining Data Mining 24
  - How Data Mining Works 25
    - The Raw Materials 25
    - The Process 26
    - The Product 26
- Translating the Data Mining Process into Steps 27
  - Step 1—Problem Definition 27
  - Step 2—Data Preparation 29
  - Step 3—Model Building 30
  - Step 4—Model Validation 31
  - Step 5—Deployment of the Model into Production 31
  - Step 6—Meta Data Management 31
- Implementing Data Mining with Microsoft Tools 31
  - Analysis Services 32
    - Decision Trees 33
    - Clustering 34

Chapter 3
Defining the Problem ... 35
Defining the Business Problem ... 35
Defining the Data Mining Problem ... 36
Deciding What Type of Analysis to Use ... 36
Determining Our Data Needs ... 36
Defining the Metrics ... 37
Creating the Formal Problem Definition ... 37

Chapter 4
Cleaning the Data ... 39
Targeting Inconsistencies ... 40
What About Those Null Columns? ... 41
Trying Out the Percent Null Tab ... 42
Looking at the Remove Null Columns Code ... 43
Calculating Null Values ... 44
Getting the Null Values ... 46
What About Those Table Properties? ... 48
Trying Out the Calculate Properties Tab ... 49
Looking at the Calculate Properties Code ... 52
Naming the Properties Table ... 52
Determining Whether the Properties Table Exists ... 53
Creating the Table and Calculating Properties ... 54
Displaying the Properties Table ... 58
Dropping a Column ... 58
What About Those Outliers? ... 59
Trying Out the Flag Outliers Tab ... 60
Looking at the Flag Outliers Code ... 63
Flagging the Outliers ... 63
Getting the Outliers ... 68
Displaying the Outliers ... 71
Replacing a Cell with Its Mean Value ... 72
Removing a Row ... 73

Chapter 5
Transforming the Data ... 75
Trying Out the DTS Import/Export Wizard ... 76
Looking at the Code Calling the Wizard ... 77
The Transformation Script ... 78

Chapter 6

Exploring the Data....................81

Visualizing Data with Histograms and Scatter Plots 82
Programming Challenges 83
Visualizing varchar Columns 83
Looking at the Code Behind Creating the varchar Histogram....................84
Visualizing Numeric Columns with a Histogram....................86
Looking at the Code Behind the Numeric Histogram 87
Visualizing Numeric Columns with a Scatter Plot 91
Looking at the Code Behind a Scatter Plot 92
Numerically Exploring Data with a Correlation Matrix 94
Looking at the Code Behind the Correlation Matrix Tab....................96
Setting Up for Two Loops 96
Creating a Table for Correlation Values....................97
Getting the Raw Data and Its Averages and Standard Deviations 98
Calculating the Correlations....................99

Chapter 7

Splitting the Data....................107

Trying Out Table Splitting....................109
Looking at the Code Used to Split the Table 111
Guaranteeing Uniqueness....................111
Removing Existing Training and Testing Tables....................113
Walking Through the Check_Table_Exist Function....................113
Calculating Percentages in the Original Table 114
Calling the Sampling Routine....................115
Random Sampling 116

Chapter 8

Building and Validating the Models 119

Building the Models 119
Column Parameters....................120
Model Parameters 121
Trying Out the Model Building Task 121
Looking at the Model-Building Code....................124
Create the Connection....................125
Defining the Model 126
Adding Columns to the Model....................128

Browsing the Models 130
Looking at the Browsing Code 133
Validating the Models 136
Trying Out the Validation Task 137
Looking at the Validation Code 140

Appendix
Managing Tables 145
Selecting Columns 145
Dropping Tables 146
Sampling a Table 146
Reducing Record Count 146
Looking at the Code for Sampling a Table 147
Increasing the Ratio of Responses in the Predictable Column 148
Looking at the Code for Over-Sampling a Table 149

Introduction

Data is a fact of life. As time goes by, we collect more and more data, making our original reason for collecting the data harder to accomplish. We don't collect data just to waste time or keep busy; we collect data so that we can gain knowledge, which can be used to improve the efficiency of our organization, improve profit margins, and on and on. The problem is that as we collect more data, it becomes harder for us to use the data to derive this knowledge. We are being suffocated by this raw data, yet we need to find a way to use it.

Organizations around the world realize that analyzing large amounts of data with traditional statistical methods is cumbersome and unmanageable, but what to do about it? Enter data mining. As both technology and data mining techniques continue to improve, the capability of data mining products to sort through the raw material, pulling out gems of knowledge, should make CEOs around the world jump up and clap their hands.

Before we get too far ahead of ourselves, realize that the success of any data mining project lies in the proper execution of specific steps. There is no magic box from which a data mining solution appears. We must work with the raw data and get to know what it contains. What we get out of a data mining solution is only as good as what we put into it.

The six steps for a data mining solution are as follows:

- Defining the problem
- Preparing the data
- Building the models
- Validating the models
- Deploying the models
- Managing the meta data associated with transforming and cleaning the data and building and validating the models

Each of these steps can further be subdivided into tasks. Only in working through each of these steps can we create the best data mining solution to solve a given problem. The most time-consuming task in the process is not creating the model, as you might think; instead, it's cleaning and exploring the data that takes up about 70 to 80 percent of the time spent on any data mining project. Creating the model is as simple as setting some parameters and clicking **Process**. Cleaning and exploring the data require data domain knowledge and a good feel for what you are doing. In the solution described in this book, we'll only go through the steps of creating and validating the model. Managing the meta data is beyond the scope of this book.

So what is it we're really trying to get out of a data mining solution? Well, we've talked about gaining knowledge, but that's pretty abstract. What gives us the capability to find this knowledge? The answer is to find the hidden patterns that exist in data, which can be stored in the form of a data mining model. A data mining model uses a specific algorithm to search through the data and find and store interesting patterns. We can then browse through a graphical representation of these patterns. Depending on the model, we can also create predictions based on the relationships that the model finds.

Microsoft supplies several great tools that allow us to create a complete data mining solution. The purpose of this book is to demonstrate how to apply these tools to the data mining process.

Throughout this book, we'll use a scenario to illustrate the steps in the data mining process. The following section describes this scenario.

Introducing the Data Mining Scenario

Your boss stops by your office and drops a dataset into your lap. He says that he's heard about data mining and wants to use it as part of a business objective to reduce costs.

Your company is well known throughout the industry for being very generous with charities. One of its endeavors is to send out requests for donations to the customers in its database. The idea is to bring everyone together in the donation process, letting the customers feel as though they are a bigger part of the company as a whole. But with the current economic situation, your boss wants to reduce the cost of the program while still optimizing the results. It's time to trim the fat!

As your boss explains, the dataset contains a large amount of demographic information and donation history as well as a column that describes whether each customer donated the last time the company sent out a mailing. He wants to use this information to try to predict who will donate this time, and then only send out solicitations to those people. Reducing the size of the mailing will, in turn, reduce costs and make everybody happy. You have at your disposal Microsoft® SQL Server™ 2000, including SQL Server Analysis Services,

and Microsoft Visual Basic® 6.0. Accompanying this book is a sample application, the Data Mining Tool, which you'll use to work through the steps in the book.

Note The data mining process described in this book does not include writing Visual Basic code. But because the Data Mining Tool is provided as non-compiled code, you'll need Visual Basic to view the code.

The following diagram shows the user interface of the Data Mining Tool. The steps shown on the user interface closely match the data mining process that you will be learning about in this book.

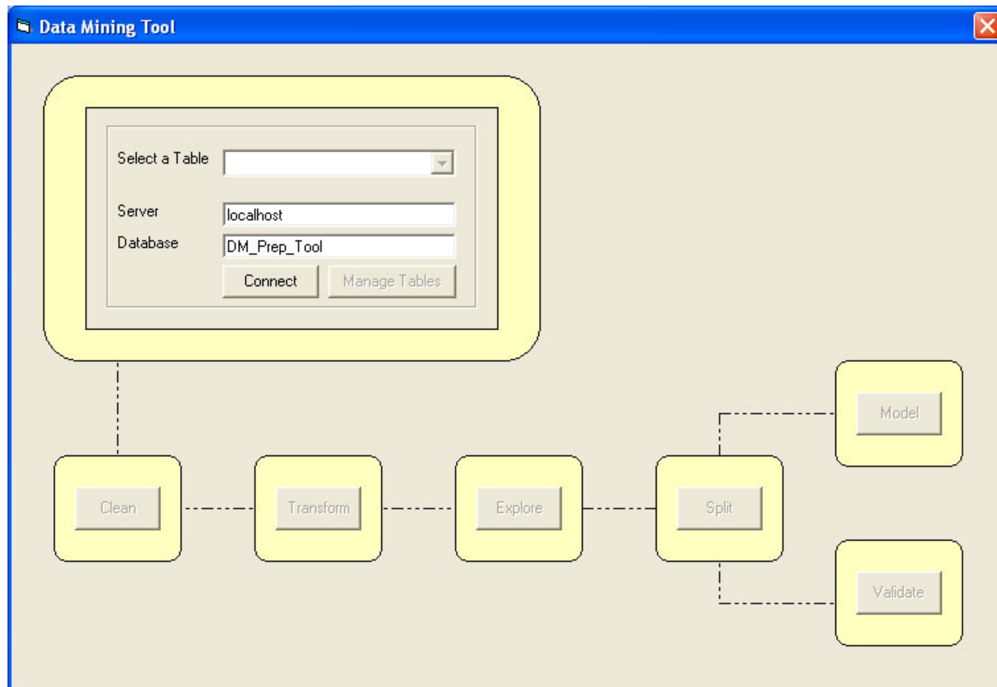


Figure i.1 The Data Mining Tool

This solution and the accompanying sample code were designed specifically for the targeted mailing dataset that is included with the code and text. You can use the Data Mining Tool with other datasets, but its functionality may be affected. However, the general concepts discussed in this book are applicable to most data mining projects. This solution is designed so that you can modify the code as necessary to fit within your individual project's goals—it's a place for you to start in designing your own solution.

Important The Data Mining Tool and the procedures associated with it may not work as described on non-English systems.

What Will You Learn from this Book?

Through the scenario, we'll apply the data mining process to a real-world situation, learning to:

- Think logically about how to use the data mining process.
- Create a succinct definition of the problem we are trying to solve.
- Prepare a real-world dataset for data mining by massaging and cleaning the data.
- Gain domain knowledge through data exploration and transformation.
- Create data mining models using the prepared data.
- Compare the data mining models and choose the one that best solves the given problem.

This book first introduces the concepts behind data mining, giving you a basic understanding of the process we are about to undertake. From there, the book dives right into creating the solution, with each chapter showing how to accomplish a step in the data mining process using Microsoft tools. In the final chapter, we'll build the models and choose one that performs the best. By the end, you will have learned about the data mining process and how to apply it to a real-world dataset.

Who Should Read This Book?

Although anyone with a technical background can benefit from the information presented in this book, the book is targeted toward enterprise developers, system architects, and information technology (IT) professionals already working with SQL Server:

- Enterprise developers—the people in an organization who are responsible for designing and implementing enterprise-wide solutions in and between organizations.
- System architects—the people in an organization who are responsible for planning and crafting overall business strategies and solutions.
- IT professionals—the people in an organization who are responsible for installing, maintaining, and administering software in the enterprise. IT professionals include managers, Webmasters, system engineers, and database administrators (DBAs).

Readers should also be familiar with Analysis Services, know the basics of how to navigate through Analysis Manager, and be proficient with Visual Basic.

What Technologies Does This Book Cover?

As you work through this book, you will use the following technologies:

SQL Server 2000

We'll use SQL Server 2000 to store the source data as well as create and maintain the multiple tables that are generated by the cleaning and exploration tasks. For additional information about SQL Server 2000, see SQL Server 2000 Books Online (http://go.microsoft.com/fwlink/?LinkID=6976).

Analysis Services

We'll use Analysis Services to create and store the data mining models and create predictions for testing the performance of the models. For additional information about Analysis Services, see SQL Server 2000 Books Online (http://go.microsoft.com/fwlink/?LinkID=6976).

Visual Basic 6.0

We'll use Visual Basic to view the code and compile the Data Mining Tool. This tool is not shipped as an executable, so you must have Visual Basic 6.0 with Service Pack 5 (SP5) installed on your computer to run the tool.

How Is This Book Structured?

This book has eight chapters, which step you through the data mining process. Each chapter describes how to use the Data Mining Tool to accomplish a specific step in the process, and then gives you a behind-the-scenes look at the code in the tool that makes the step work. Also included in this solution is source code for the Data Mining Tool and a sample dataset, which we'll use throughout the book.

Chapter 1, "Setup"

Lists the applications, system requirements, and database setup steps you need to perform before installing and running the Data Mining Tool. These steps include importing the text data file included with the project into a SQL Server database.

> **Important** Before working through the rest of the book, make sure you follow the procedures outlined in Chapter 1 If SQL Server or your Analysis server is not properly configured, or if you do not install the third-party controls, the sample will not function properly.

Chapter 2, “Data Mining Fundamentals”
Introduces the field of data mining and explains the process and concepts behind creating a data mining solution.

Chapter 3, “Defining the Problem”
Defines the data mining problem that we will solve .

Chapter 4, “Cleaning the Data”
Explains how to select and clean the data originating in the text file.

Chapter 5, “Transforming the Data”
Explains how to use the DTS Import/Export Wizard to create transformations on the table being cleaned.

Chapter 6, “Exploring the Data”
Explains how to explore the base table using the Data Mining Tool.

Chapter 7, “Splitting the Data”
Explains how to split the original table into two additional tables, which will be used separately to build and test the mining models.

Chapter 8, “Building and Validating the Models”
Explains how to build and test mining models based on the table that was cleaned and split in the previous steps.

Appendix, “Managing Tables”
Discusses the various table management tasks that the Data Mining Tool performs.

Security

Throughout this project, we assume that you’re using Windows Authentication to access the SQL Server and Analysis Services databases and that you have administrator privileges on the computer. Additionally, to create and modify data mining models, you must be a member of the OLAP Administrators group on the computer.

1

Setup

This chapter provides instructions on how to set up the Data Mining Tool. It also briefly describes the components of the Data Mining Tool and provides instructions on installing and configuring publicly available components.

Setup is a multistep process. First, you need to ensure that your computer is equipped with the required set of tools and technologies. Then you run the Microsoft® SQL Server™ 2000: Data Mining Setup Wizard (SQL2KDataMining.msi) to install the database, the Data Mining Tool, and other components.

We recommend that you install the software and run the Data Mining Tool on a computer that has at least 500 Megabytes (MB) of RAM and a 1.5 Gigahertz (GHz) processor. If you do not have a computer that meets these recommendations, the procedures that require processing will take a considerable period of time to complete. Optionally, you can use the sampling and column-selection techniques described later in this book to reduce the size of the original table, which will reduce processing time. If you choose to reduce the size of the original table, make sure that you do not eliminate the columns CONTROLN, TARGET_B, and TARGET_D. These columns are necessary for the data mining tasks we will be performing.

Reviewing the Setup Wizard Requirements

The SQL Server 2000: Data Mining Setup Wizard requires that you have administrator privileges on the computer on which you plan to run the wizard. Also, before running the wizard, you must install the tools and technologies listed in the following sections. It is strongly recommended that you install them in the order in which they're listed.

After your computer meets these requirements, you are ready to run the SQL Server 2000: Data Mining Setup Wizard.

Important If you do not have the required software installed before you run the SQL Server 2000: Data Mining Setup Wizard, the installation process will not succeed and the Data Mining Tool will not work.

Microsoft® Windows® XP Professional or Windows 2000 Service Pack 2 (SP2) with NTFS

To help ensure the security of your computer, install the latest Windows updates by going to the Microsoft Windows Update site (http://go.microsoft.com/fwlink/?LinkId=9609) and following the online installation instructions.

Microsoft SQL Server™ 2000 SP2 (Developer Edition recommended)

For more information about SQL Server 2000, go to the SQL Server Web site (http://go.microsoft.com/fwlink/?LinkId=6791).

Note The Data Mining Tool assumes that you are using Windows Authentication as your security protocol. If this is not the case, you must reconfigure your server to allow Windows Authentication.

SQL Server 2000 Analysis Services SP2 (Developer Edition recommended)

For more information about Analysis Services, go to the SQL Server Web site (http://go.microsoft.com/fwlink/?LinkId=6791).

MDAC 2.7

To run this solution, you need to make sure that MDAC 2.7 is installed on your computer. You can install MDAC 2.7 from the Microsoft Data Access Web site (http://go.microsoft.com/fwlink/?LinkId=9642).

Microsoft Visual Basic® 6.0 SP5

To view or manually compile the code used in the Data Mining Tool, you need to have Visual Basic installed on your computer.

Angoss Visualization Tools

To compile and run the sample code, you need to install the Angoss OLE DB for Data Mining Consumer Controls, which are available from the Microsoft OLE DB Web site (http://go.microsoft.com/fwlink/?LinkId=9610).

▶ To install the OLE DB for Data Mining Consumer Controls

1. From the Microsoft OLE DB Web site, click **Download the OLE DB for Data Mining Consumer Controls**.
2. Read the License Agreement, and then click **Download**.
3. Either click **Open** to run the installation file immediately, or click **Save** to save the installation file to your computer. You can then run the file locally.

   After you run the installation file, the consumer controls will be installed on your computer. In addition, Help files describing the controls and a sample application will be added to your **Start** menu.

Microsoft Windows Installer 2.0

Although Windows Installer 2.0 comes with Windows XP, Windows 2000 does not include this installer by default.

▶ To install Windows Installer 2.0

1. In Internet Explorer, go to the Windows Installer Version 2.0 page (http://go.microsoft.com/fwlink/?LinkId=7032)
2. In the Run-time Requirements section, click the link shown to download the redistributable file.

Reviewing What the Setup Wizard Does

After making sure that your computer meets the minimum software and hardware requirements, run the SQL Server 2000: Data Mining Setup Wizard.

Running the SQL Server 2000: Data Mining Setup Wizard:

- Installs the sample source code used in this book.
- Installs the documentation for *Preparing and Mining Data with Microsoft SQL Server 2000 and Analysis Services*.
- Adds a new database to your server, DM_Prep_Tool, and creates an empty table with the correct column information.

You can find the files associated with the source code in the C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample folder and the files associated with the database in the C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\Database folder.

Setting Up the SQL Server Database

Throughout this book, you will be using the Data Mining Tool to mine data in the DM_Prep_Tool database. This database contains data taken from the UCI Knowledge Discovery in Databases (KDD) Archive (http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html). A data dictionary describing the different columns included in the dataset can also be downloaded from the site. The data is available as a text file, with the rows delimited by linefeeds (lf) and the columns delimited by commas. The first row holds the column names. The dataset contains 481 columns and over 90,000 rows of data. Because the data is in a plain-text file, no column data types are included with the data; instead, they can be found in an accompanying document.

To make this solution easier to work with, we created the table structure in the database during setup, including setting the types of the columns. To work with the solution, you only have to import the data from the text file into the table using the DTS Import/Export Wizard.

The following procedure describes how to use the DTS Import/Export Wizard to populate the table with data.

▶ To populate the cup98lrn table with data

1. From the **Start** menu, point to **Programs**, point to **Microsoft SQL Server**, and then click **Import and Export Data**.

   The DTS Import/Export Wizard opens.
2. In the opening screen, click **Next**.
3. From the **Data Source** drop-down menu, select **Text File**.
4. Browse to **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample**, select **cup98lrn**, and then click **Next**.
5. Select the **Delimited** check box; in the **Row Delimiter** drop-down box, select {**LF**}; select the **First row has column names** check box; and then click **Next**.
6. Because **Comma** is already selected, click **Next**.
7. Select your server (or leave it as local host), select the **DM_Prep_Tool** database, and then click **Next**.
8. The correct table is already selected, so click **Next**.
9. Select **Run immediately**, and then click **Next**.

A DTS package is now built and run that imports the data from the text file into the existing cup98lrn table. The column data types were set during installation, so the new data is correctly typed.

Setting Up the Analysis Server

In order to create the mining models and work with them, you will need to set up a new database on your Analysis server.

▶ **To create a new Analysis Services database**

1. From the **Start** menu, open Analysis Manager.
2. In the tree view, navigate to your server.
3. Right-click your server, and then select **New Database**.
4. In the **Database name** text box, type **DM_OLAP**.

A new database has now been created on your Analysis server.

Within the new database, you also need to create a new data source from which the mining models can be built.

▶ **To create a new data source**

1. From the **Start** menu, open Analysis Manager.
2. In the tree view, navigate to your server, and then expand the node for the DM_OLAP database.
3. Right-click **Data Sources**, and then click **New Data Source**.
4. For the provider, select **Microsoft OLE DB for SQL Server**, and then click **Next**.
5. For the server, select **LocalHost**, select **Use Windows NT Integrated security**, and for the database, select **DM_Prep_Tool**.
6. Click **OK**.

A new data source named Localhost DM_Prep_Tool has been created on your Analysis server, which you can use as a source for building your models. The name that Analysis Services gives the data source is long and inconvenient, so let's rename it. The problem is that we have to use a funny work-around because Analysis Services provides no direct way to rename the data source through the user interface.

▶ **To rename a data source in Analysis Manager**

1. Right-click the **Localhost Data_Prep_Tool** data source, and then click **Copy**.
2. Right-click the **Data Sources** folder, and then click **Paste**.

   The **Duplicate Name** form appears, allowing you to choose a new name.
3. In the **Name** text box, type **cup98LRN**.

You now have a duplicate data source with a new name.

Connecting to the SQL Server Database

With the data properly stored in the database, we now need to set up a connection between the Data Mining Tool and the database. This step is critical, because no matter what kind of project you are working on, you need a connection to the database before you can start to mine the data.

When you first open the Data Mining Tool, the only button you can click is the **Connect** button. This button defines the server and database for a new connection string.

▶ To connect to the DM_Prep_Tool database

1. In Windows Explorer, browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click **DMFinal.vbp**.
2. In Visual Basic, on the **Standard** toolbar, click **Start**.

   This opens the Data Mining Tool.
3. In the Data Mining Tool, type the following information, and then click **Connect**:
   - In **Server**, type **Localhost**.
   - In **Database**, type **DM_Prep_Tool**.

After you click **Connect**, the Data Mining Tool uses the server and database information that you provided to create a connection between the tool and the database.

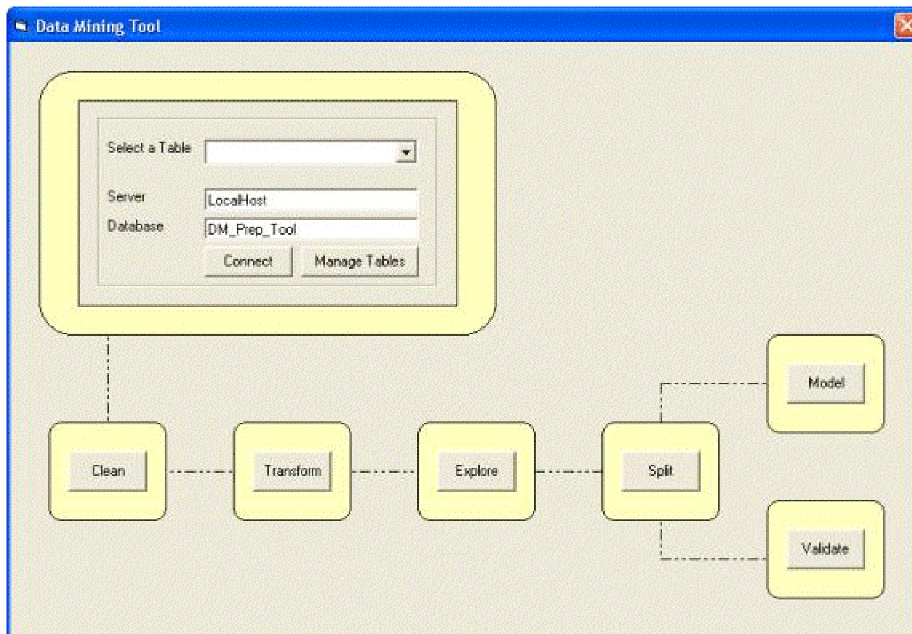


Figure 1.1 The Data Mining Tool after making a connection

Selecting a Table in the Data Mining Tool

We've just set a connection to the database. The next step is to decide which data we want to use in the data mining process. Because we do not want to affect the original data, we'll make a copy of the source data and store it in a new table. We will then mine the new table for information.

To select a table, click the **Manage Tables** button in the Data Mining Tool. This opens the **Manage Tables** form (Figure 1.2), where, in addition to copying entire tables, you can:

- Select specific columns to include in a new table.
- Drop tables from the database that are no longer useful.
- Create a copy of an existing table but include fewer rows (in other words, sample the table).
- Create a copy of an existing table but force the sampling algorithm to include a higher percentage of positive predicted values than actually existed in the original table.

Figure 1.2 The **Manage Tables** form

Depending on the data being mined, you might want to use any of the table management techniques listed here. In the course of this book, we'll focus on just a couple of these techniques. But if you're interested in the techniques we don't cover, see Appendix, "Managing Tables," for complete instructions on using the other techniques.

▶ To copy the table in the DM_Prep_Tool database

1. In the Data Mining Tool, click **Manage Tables**.
2. In the **Create a new table by copying an existing table** section, enter the following information:
   - In **Select a source table**, select **cup98LRN**.
   - In **Enter a table name**, type **cup98LRN_clean**.
3. Click **Copy Table**.

   The Data Mining Tool begins to copy the table. Depending on the speed of your computer, this process may take a little while. As you'll soon see, mining a large dataset not only requires an intimate familiarity with the data but a good deal of patience as well.
4. When a message box appears indicating that the table was created, click **OK**.
5. In the **Manage tables** form, click **Close**.

At this point, we are all set up to start mining data. But before we get too hasty, let's take the next couple of chapters to define exactly what data mining is and exactly what data mining problem we want to solve.

2

Data Mining Fundamentals

As mentioned in the Introduction, your boss wants you to use data mining to figure out which customers are most likely to respond to a request for a charitable donation.

Your first question is, "What is data mining?"

What Is Data Mining?

Every day, corporations throughout the world add billions of rows of data to their databases. As the amount of raw data increases exponentially, our ability to understand the data and extract the wealth of information that lies inside it plummets. Using SQL, we can generate queries that return lists of records, basically filtering the available data into smaller subsets. We can also create multidimensional aggregations using complex SQL statements—to answer questions like, how much did "so and so" sell in his district last year. These are valuable tools that help present and summarize data, but we can't develop a deep understanding of the data using these technologies. For instance, we can't use SQL and online analytical processing (OLAP) to predict the value of a column in a table based on the values of related columns in a database. Nor can we use these technologies to predict whether someone will donate money based on what we know about him or her. But we can use data mining to answer these complex questions, and in doing so, we can begin to make sense of the world of data that has accumulated around us.

Defining Data Mining

A technical definition of data mining is often stated as "the process of extracting valid, authentic, and actionable information from large databases." Notice that within this definition we are not extracting specific data, but instead we are deriving information that the data as a whole can provide us.

Now what does this definition of data mining really mean? Let's look a couple of examples.

Example of Data Mining Through Personal Experience

If you always walk down the same street on your way to work, you naturally observe and store things in your brain that you may not consciously realize:

- The bakery is always crowded at 8 A.M. and is always out of coffee by then.
- The bartender from the bar next door drops last night's bottles into the recycle bin about the time you reach the corner.
- The overhead train drowns out conversations at the bus stop.

When faced with decisions involving unknown factors, you use this stored information to make an educated guess. If you are in a hurry and need coffee, you most likely will skip the bakery and try the quick stop next door. You don't know for sure that the bakery will be out of coffee, but based on past experience you can make a good prediction. If you want to have a conversation with a friend while waiting for the bus, you will probably choose a stop that is not underneath the overhead train tracks. Again, you don't know for sure that the train will be there—maybe it broke down—but you have a good idea that it will. Though obvious, these examples help describe how data can be transformed into actionable explanations. You collect data and then later use that data to make a best guess as to what will happen in the future. The real power of data mining is that it can go beyond the obvious, finding hidden patterns you would otherwise not think to look for in large databases.

Example of Data Mining on a Corporate Level

For a more concrete example of data mining, consider this: you receive a credit card application in the mail and decide to apply. In the application, you give both personal and financial data. The bank issuing the credit card uses these few bits of personal and financial information to predict your credit risk. Much like you, the bank has learned from its experiences. There are many obvious reasons the bank might reject an application; for example, if the applicant is unemployed with a poor credit history, and has, more often than not, defaulted on his or her credit. Likewise, there are many obvious reasons the bank might accept an application; for example, if the applicant is married with two kids, and has a good job and a good credit history. You wouldn't necessarily need to use data mining to find these instances, but what about someone with a relatively good job and little credit history—are there any signs that this person

might be a bad credit risk? Data mining can help to uncover the hidden patterns that answer this question. Over the years that the bank has been issuing loans and credit cards, it has amassed a large store of observations that it can use to create mining models. The only difference between the bank and you is that the bank stores its observations in a large database. In the same way that you predict which store will be out of coffee, the bank predicts which person will pose a greater risk, but the bank's predictions will be based on models developed from large datasets.

Data mining is often compared to statistical analysis, yet it differs in a couple of ways. First, typical data mining algorithms deal with large datasets, while in the field of statistics, datasets consist of a sampled set of data taken from a larger population (though a data mining dataset can also be sampled). Second, statistical algorithms are typically hypotheses based. In creating a statistical solution, you are probably trying to answer a very specific question or prove or reject a hypothesis. In creating a data mining solution, you try to find general or hidden trends and relationships that exist in the data. Data mining draws from several fields, including artificial intelligence and statistics. Think of statistical analysis as approaching a problem in a top-down manner, while data mining approaches a problem from the bottom up. In other words, you don't know exactly what you'll find when you are data mining.

How Data Mining Works

Now let's talk about the specifics of data mining—how does it work? Creating a mining model can be compared to any manufacturing process. First you need the raw material—the data. You then pump that data through a mechanical process—the algorithm. This, in turn, produces a product—the mining model. The difference between the manufacturing process and the data mining process is that instead of creating the product through mechanical means, you are using mathematical means.

The Raw Materials

Where does the raw material—the data—come from? Basically, anywhere you can find it: text files (flat files), Microsoft® Excel files, online transaction processing (OLTP) databases, online analytical processing (OLAP) databases, and so on.

Typically, you also check this data for integrity. Data integrity is an important concept. Because you often pull data from multiple sources, you cannot assume that the data is always presented in the same manner. For example, dates from one dataset can be expressed in a M/D/YR format, while those from another as D/M/YR. Measurements can be expressed in different units from one dataset to another. To maintain consistency in the final data warehouse, you must identify these problems and resolve them.

The Process

As you might have noticed in the overview of the data mining process, the data mining algorithm is at the heart of this process. Technically speaking, data mining algorithms fall into the following categories:

- Classical statistical algorithms (that is, radial basis-functions and multivariate splines)
- Pattern recognition algorithms
- Genetic algorithms
- Classification and regression trees (CART)
- Other rule-based methods

Choosing the right algorithm can be complicated because each produces a different result, and some can produce more than one type of result. You can also use different algorithms to perform the same task. To further complicate matters, you don't have to use these algorithms independently; you can use multiple algorithms to solve a particular business problem. You use some algorithms as a means of exploring data, while you use others to predict a specific outcome based on the data. For example, you can use a regression tree algorithm to provide financial forecasting and a rule-based algorithm (a CART algorithm) in a market basket analysis. You can use the decision tree algorithm (a classification algorithm) both for prediction and as a way of reducing the number of columns in a dataset (by showing which columns do not affect the final model). You can use a clustering algorithm (a pattern recognition algorithm) to break data into groups that are more or less homogeneous, and then use the results to create a better decision tree model. Both a sequence clustering algorithm and a rule-based algorithm can be used in a click-stream analysis. However, remember that while choosing an appropriate algorithm is important, your true goal is to create a robust and accurate model that can be understood by users and deployed into production with minimal effort.

For this scenario, we will build decision tree models using the Microsoft Decision Trees algorithm. This algorithm allows us to build a good overall model with strong predictive capabilities. For more information about the Microsoft Decision Trees algorithm, see "Implementing Data Mining with Microsoft Tools" later in this chapter.

The Product

As you begin to work through your data mining solution, realize that it is a dynamic and iterative process—the solution evolves and grows over time. As you learn more about the data domain or add more data to the data warehouse, your mining model also changes. If you base your mining model on a dynamically changing data warehouse (which grows through scheduled updates), you need to develop a strategy for updating your mining model. Similarly, you may have originally built the mining models from a sparse data

source, but, as time has passed, your data source has become richer, allowing you now to create a more accurate model. To take advantage of this, you need to rebuild the model.

A key point here is that the model that you build is only as good as the raw material used to create it. However, it doesn't matter how good your data is if you do not understand it. You will be making tough decisions about which data should be included, how it should be cleaned and transformed, and what you eventually want to predict.

After you create your models, but before you put them into production, you need some metrics by which to calculate the effectiveness of your models. You do not want to put a model into production until you know how good it is.

To summarize, there are four things you must have to create an accurate mining model:

- A clear definition of the business problem
- A rich dataset related to the business problem
- A thorough understanding of the data domain
- A set of metrics with which to measure the success of the mining model

The usefulness of your mining model can be directly traced to the initial planning and thought dedicated to defining and clearly stating the problem you are trying to solve. How can you find the answer if you can't ask the right question?

Translating the Data Mining Process into Steps

As you've just learned, data mining is a process. Though the end step is clearly building a mining model, the steps leading up to the creation of the model determine the success of your solution. While there are a multitude of approaches to the data mining process, all of them roughly translate into the distinct steps and tasks shown in Figure 2.1.

Step 1—Problem Definition

Before you build a mining model, you need to understand the data you will work with and clearly define the business problem you are trying to solve. This includes analyzing the business requirements, defining the scope of the problem, defining the metrics by which the model will be evaluated, and defining the final objective for the data mining project. These tasks translate into questions like:

- What is your boss is looking for?
- Which attribute of the dataset do you want to try to predict?
- What types of relationships are you trying to find?
- Do you want to make predictions from the data mining model or just look for interesting patterns and associations?

- How is the data distributed?
- How are the columns related, or if there are multiple tables, how are the tables related?

These are the questions that you need to be able to answer before you can begin to work with the data. To find the answers, you may need to conduct a data availability study, investigating the needs of the business users with respect to the data available. If the data won't support what the users need to find out, you may need to redefine the project.

Step 2—Data Preparation

You've defined the problem that you are going to try to solve—now what? Well, first you need to find the raw data related to this business problem. Collecting the data can be a cumbersome task. This data is usually scattered across a company and stored in different formats. But do not narrow your focus! Find all data that is related to the business problem.

Often, the original data is collected through an OLTP system and contains inconsistencies. Entries are missing or flawed; for example, the data might show that a customer bought a product before she was born or shops regularly at a store 2,000 miles from her home. Before you begin to build the models, you need to fix these problems. In other words, you must "clean" the data. The problem is that cleaning the data is not a straightforward process. Maybe the person shopping 2,000 miles from her home has two residences and lives an equal amount of time at both. Usually, you are working with a very large dataset and can't look through every transaction personally. Therefore, you need to use some form of automation to explore the data and find the inconsistencies. Exploration techniques can include calculating the minimum and maximum values, calculating the mean and standard deviations, and looking at the distribution of the data. In the end, you need to decide which data seems flawed and devise a strategy for fixing the problem.

In preparing the data, you often have to transform columns of the dataset before building a mining model. For example, to determine whether your company's compensation strategy is equitable, you may try to predict salaries based on age, experience, length of time with the company, and other factors. The data you use to create your model contains a large number of possible values for the salary of an employee—in essence, it is a continuous attribute, a column with a large number of states. To make your final model more focused, you need to discretize the data. This simply means creating a limited number of buckets (salary ranges) such as low, medium, and high, and replacing the values in the column with the appropriate bucket name. You may also want to define a new column based on existing columns. For example, you may not have a column that details the total cost of retaining an employee, including such things as health insurance and other perks, but you could easily make one by adding up each cost and displaying it in a new column.

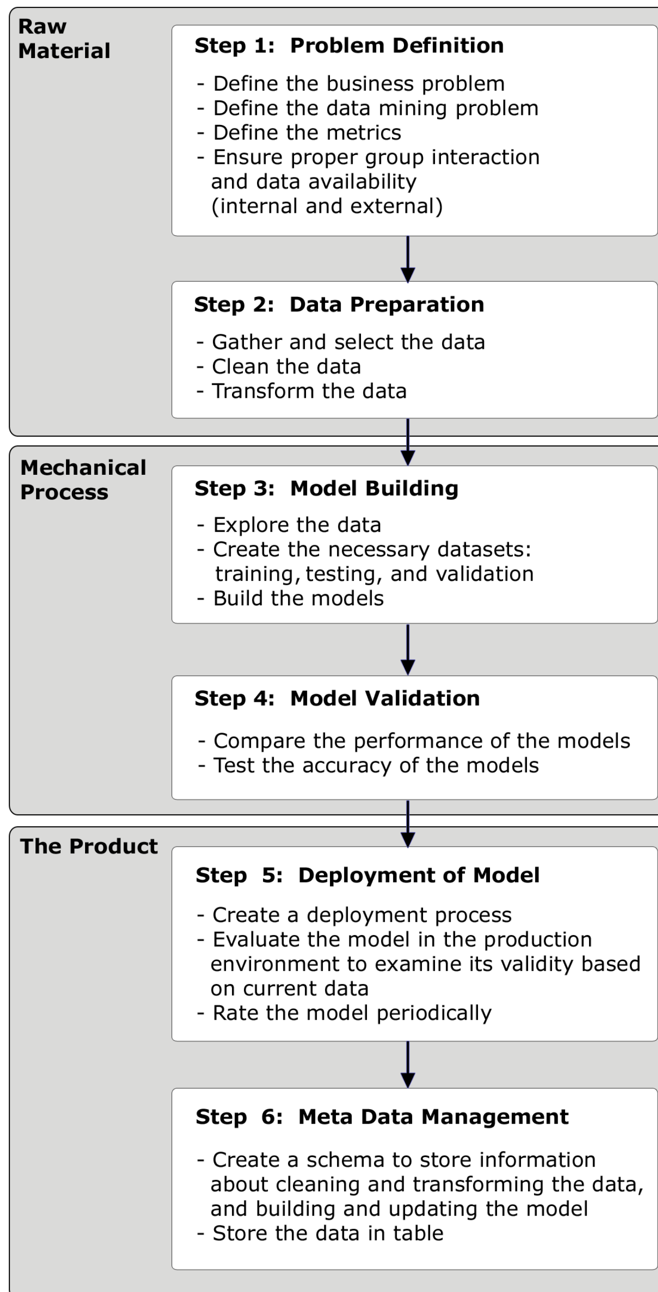


Figure 2.1 Steps in the data mining process

Step 3—Model Building

The most important concept in data mining is knowing your data. If you don't understand the structure of your dataset, how can you know what to ask, or which columns to include in your data mining model? Imagine that you are at an important business meeting but did not prepare. If you ask questions during the meeting, they will probably not make sense, reducing your effectiveness. The same holds true for data mining. If you build models without knowing your data, you will ask the wrong questions, reducing the model's effectiveness.

Before building the model, you need to randomly separate the original dataset into separate training (model-building) and testing (validation) datasets. You use the training data to build the model. Then you test the accuracy of the model by creating prediction queries against the testing dataset. Because you know the outcome of the predictions (the data comes from the same set used to train the model), you can calculate the accuracy of the model's performance.

Sometimes the attribute that you are trying to predict has a very high distribution of one state, and a very low distribution of another state. For example, in our dataset, the number of positive responses in the predictable column is about 5 percent, while the number of negative responses is about 95 percent. There is a chance that there are not enough occurrences of the positive response to generate the strong relationships that will allow us to create predictions. One way to solve this problem is to over-sample the data, which means that we artificially boost the number of positive responses but randomly remove a number of the records that correspond to negative responses. For more information about over-sampling, see the *SQL Server 2000 Resource Kit* and Appendix, "Managing Tables."

After you explore the data and select columns to include in the model, you can build your models using the training dataset. This process happens exactly the way it sounds—you pass the data through the algorithm to train the model. Each algorithm also contains adjustable parameters that can affect the outcome of the model. The result of the training process is a mathematical model you can either use to explore the data (as in the case of a clustering algorithm) or to create predictions (as in the case of a decision tree algorithm). How well you choose the columns to include in the model and how you alter parameters of the model ultimately determine the performance of the models. With that said, here are the steps for building the model:

- Select columns.
- Select a model.
- Adjust parameters.
- Train the model.

Step 4—Model Validation

After you build a model, you need to know how well it performs. You do not want to move the model into a production environment until you know how well it predicts. Often, you build several models and then compare how they perform against each other. This is where you use the testing dataset that you previously set aside.

Step 5—Deployment of the Model into Production

This is where all of your hard work begins to show results. After you build the models and measure their effectiveness, you can deploy them in a production environment, the place where the models will be used in the business decision-making process. Updating the model is part of the deployment strategy. As more data comes into the organization, you need to develop a process for rebuilding the models, thus improving their effectiveness.

Step 6—Meta Data Management

The information that is associated with exploring the data and building the models is useful for you to save. This includes columns that were removed, models that were previously built, and the effectiveness of those models. Managing this data can become a project in itself, but it is a very important step. Typically, you store this information in a database, where it is available through queries, like any other data.

Implementing Data Mining with Microsoft Tools

So how do we go about performing all of the tasks that we've been talking about? Well, luckily, Microsoft provides all of the tools, which, when used together, allow us to work all the way through the data mining process. Throughout this book, we will use:

- Microsoft® Visual Basic® 6.0 to view the code in the Data Mining Tool.
- Microsoft SQL Server™ 2000 to manipulate, manage, and store the data.
- SQL Server 2000 Analysis Services to build the mining models and make predictions.

In developing this solution, we found that the mechanisms to complete the tasks associated with each data mining step were divided between Analysis Services and SQL Server. Although each task can be performed individually in either SQL Server or Analysis Services, we felt that it would be nice to have an environment that tied it all together—the Data Mining Tool. There are a few advantages to this approach:

- The data mining process is exemplified as we work through the steps.
- All of the tasks can be accessed from a single program.
- Techniques for programmatically accessing SQL Server and Analysis Services functionality are demonstrated.

So how did we build this environment? Well, that's where Visual Basic comes into play. Visual Basic, along with the Microsoft ActiveX® Data Objects (ADO) and Decision Support Objects (DSO) programming interfaces, gives us the means to tie all of these technologies together. We take advantage of FlexGrid controls to look through data in the tables, chart controls to explore the data, and third-party modeling controls to view and compare data mining models.

In working through this solution, several tables are created, dropped, and modified, which implies a need for a mechanism to manage all of this data. SQL Server is perfect, providing all of the functionality we need to modify, store, and manage data. Additionally, several tools are provided with SQL Server that are useful in completing several of the tasks associated with the data mining process. Data Transformation Services (DTS) provides the mechanism for importing and transforming the data through the DTS Import/Export Wizard. We will use the wizard to import the raw data into the database and to transform the columns in the table we are cleaning. In using the wizard, we are actually creating a DTS package that can either be run immediately or saved and run later. DTS also includes the Data Mining Prediction Query Task, which can be used to create a package that creates a prediction based on a mining model and performs an action based on the results.

How do we actually build the models from the tables managed in SQL Server? This is where Analysis Services comes into play. Using Analysis Services, we'll build models based on the relational data source created with SQL Server. You can also build models based on multidimensional data sources, but that's a subject for another book.

Probably what is newest to you in this book is working with the data mining functionality in Analysis Services, so let's take a closer look at what Microsoft has done in this area.

Analysis Services

Microsoft included data mining functionality with the release of SQL Server 2000, coinciding with the release of the OLE DB for Data Mining 1.0 Specification version 1.0 (http://go.microsoft.com/fwlink/?LinkId=8631).

Historically, data mining has been restricted to users who can draw information from complicated statistical techniques and software. The Microsoft vision for data mining is to make it available not only to the power user, but also to the intermediate and naïve user. Microsoft does this by using technologies that developers already use and understand, such as ADO and schema rowsets.

Accordingly, Microsoft helped to define an industry standard API that allows you to create and modify data mining models, train these models, and then predict against them. The idea was to hide some of the more complicated details, letting you use a language similar to the Transact-SQL that you already know. You can build models using a CREATE statement, use an INSERT statement and train the models, use the SELECT statement to

get statistical information from the models, and use a PREDICTION JOIN statement to create predictions using the model and data when you don't know the outcome of the predictable column. Does this sound familiar—CREATE, INSERT, SELECT? The details about this language, as well as examples, can be found in the OLE DB for Data Mining 1.0 Specification.

The basis for data mining in Analysis Services is an object called the Data Mining Model (DMM). When you create a new model in Analysis Services using either the Mining Model Wizard or the language, you are actually building a container with a structure similar to a relational table. There is no information in this container except for a description of each column included in the model, as well as the algorithm type. By training the model, you fill the table with the information it needs to describe the model. The DMM stores the relationships and rules, but not the actual data.

A concept unique to Microsoft is how sparse data is handled. Often, companies store data in large flat files with each value in a row corresponding to a specific attribute (or column). To express many-to-one relationships, you add more columns to the table, leaving many cells with null values. For example, consider how your company stores information about your customers and the products that they buy. Your company probably uses tables such as Customer, Orders, and Order Details, where for each customer there are multiple orders, and for each order there are multiple columns describing the details of the order. Now, imagine flattening this into a single relational table. Imagine how many null values would exist. Each customer would end up taking up several rows in which only their specific columns would hold information. This table would contain one or more columns for each product in the catalog, which would make the table huge. Your table would be full of null values, making mining for data difficult and confusing. Microsoft has solved this problem by allowing you to define a column type as being a table; thus, allowing you to create many-to-one relationships within a single table.

The purpose of data mining is to create a model that expresses relationships. To accomplish this, Analysis Services includes the capability to build two types of mining models, a decision tree model and a clustering model. Let's take a closer look at these algorithms.

Decision Trees

The Microsoft Decision Trees algorithm creates a model that works well for predictive modeling. It looks at how each column in a dataset affects the result of the column whose values you are trying to predict, and then uses the columns with the strongest relationship to create a series of splits, which are called nodes. These splits can be visualized as a tree structure.

This may sound complicated, but it is really very simple to visualize. The top node describes the breakdown of the predicted attribute over the overall population. For example, you might be trying to define the credit risk of potential applicants. Over the

population, 20 percent of the applicants are considered to be a good risk while the remaining 80 percent are considered to be a bad risk. So we know that this is the worst case in creating predictions. It is now the job of the algorithm to try to improve the accuracy of the predictions. Suppose that the algorithm finds a strong relationship between marital status and risk potential—there are more cases of good credit when individuals are married than when they are not. The algorithm can then create a split based on this information, creating one dataset filled with only those individuals who are married and another with those who are not. After the split, you find that the percentages of positive and negative responses in each new dataset are more drastic, meaning that your ability to predict risk has been improved.

Now suppose that, for those who are married, job state is the next big factor. Of the people who are married and have a good job, 90 percent are low risk, while the remaining 10 percent of married people are high risk. By creating another split, your predictive ability has improved yet again. This process continues until the algorithm reaches a point in which an additional split does not improve the accuracy of the prediction. For a more detailed explanation of the Microsoft Decision Trees algorithm, see SQL Server 2000 Books Online.

Clustering

The Microsoft Clustering algorithm segments the data into groups that give you a better understanding of the relationships in the data. For this scenario, we have a dataset with a large number of attributes, or columns. How do these columns relate to one another? Is there an underlying pattern, something that relates the seemingly disparate columns? There may be natural groupings in the data that are impossible for you to find through casual observation. They would even be hard to spot if you are using basic graphing techniques. What kind of groupings are we talking about? To clarify, think about each record in the dataset relating back to a person. Consider the case where people who live in the same neighborhood, drive the same kind of car, eat the same kind of food, and all respond to the request for donations in a similar manner. This is a cluster of data. Another cluster may include people who go to the same restaurants, have similar salaries, and vacation twice a year outside the country. Seeing these clusters together, you can get a better handle on how the people in the dataset interact, and how that affects the outcome of our predictable attribute.

Now that we've talked about the tools we are going to use to create this solution, let's get into the process!

3

Defining the Problem

Now let's goto the beginning of the process—what are we trying to do? In this step, we mold the inherent vagueness of the boss's request into a data mining problem. First, we should ask, "What does the boss really want?" Then we need to clearly define this business problem and formulate an actionable goal that solves the problem defined in terms of data mining. In defining the data mining problem, we:

- Decide what type of analysis will solve the business problem. Are we simply exploring the data, or are we also trying to create a model that can predict the future?
- Determine our data needs. Does the data support the type of analysis that the problem requires? Or do we need to find additional data either internally or externally?

After defining the data mining problem, we need to define the metrics by which the model will be measured.

Defining the Business Problem

Right now, let's define the business problem in our current scenario—our company has partnered with a major charity to solicit donations from the community, and now the company wants to reduce the overall cost of the project while maximizing the results.

Over the years, the money that the company spends on mailing has increased significantly as the target audience for the mailing has grown. The problem is that the actual money brought in has not increased in proportion to the increased expense. With the current economic situation, the boss wants to reduce spending, but—and here's the catch—without adversely affecting the amount of money collected. He has come to us to find out how to do this. To aid us in this task, he has brought along several year's worth of historical data that describes the demographics and response rates of previous mailings.

Defining the Data Mining Problem

So it sounds like the boss is asking for a targeted mailing. By sending the mailing only to those people who are most likely to respond, we can make the process more efficient without reducing the amount of money coming in. Thus, our actionable goal becomes "to predict whether someone is likely to donate money to the company's volunteer effort based on the historical data collected over the years."

Now we need to decide what type of analysis will solve this problem and determine whether we have the necessary data.

Deciding What Type of Analysis to Use

In Analysis Services, we have a choice of two algorithms: the Microsoft Clustering algorithm and the Microsoft Decision Trees algorithm.

The Microsoft Clustering algorithm

We can use the Microsoft Clustering algorithm to describe how people in the dataset can be grouped based on similar demographic and donation patterns. Basically, the Microsoft Clustering algorithm is a diagnostic tool used for unsupervised learning.

The Microsoft Decision Trees algorithm

We can use the Microsoft Decision Trees algorithm to build a classification model to predict an output attribute.

For this scenario, we'll use the Microsoft Decision Trees algorithm to create the data mining models, but we could just as easily use the Microsoft Clustering algorithm to create the models and draw similar conclusions.

Determining Our Data Needs

Does the infrastructure of the dataset support the analysis we are trying to perform? The historical data that the boss provided contains two columns that are useful for predictions:

- A Boolean column (TARGET_B) that states whether each person donated money.
- A numeric column (TARGET_D) that stores the amount of money each donor gave.

Because the two columns are related, we need to be careful how we use them in our analysis. We can predict whether someone will donate based on whether a money amount exists in the TARGET_D column. Even though the predictive power of this model would be very accurate, it really doesn't tell us much about who will donate! Additionally, the dataset contains a large amount of demographic data and response history, which can be used by the model to predict which columns best describe people's donation patterns.

Defining the Metrics

Now that we've defined the data mining problem, how will we know our models work? We need to define what success is, which in this case, will be determined by whether the ratio of money collected to money spent on the project increases. One common method to determine the effectiveness of the model is to use a lift chart. To create a lift chart, we create a prediction query on a testing dataset and then compare the results to the known values in the dataset. This is possible because the testing dataset contains values for the columns that we are predicting. The lift chart displays the improvement the model provides in predicting the outcome of the predictable attribute as compared to a random guess. This difference is called "lift."

For example, suppose we have a database containing a record for each customer, and we have scored and ranked the customers based on how likely they are to donate (based on the models we developed). Now, if we take the top 10 percent and mail a request for donations to them, and then randomly take another 10 percent of the customers (records) not included in the first 10 percent, and mail requests to them, we can compare the response rates, and thus, rate the effectiveness of the model.

So how much lift should we expect to see? The lift provided by the models is ultimately limited by how good the data is. No matter how many models we build and what parameters we change, the models can only be as accurate as the data allows them to be.

We will talk more about lift charts in Chapter 8, "Building and Validating the Models."

Creating the Formal Problem Definition

The formal definition of the problem is as follows:

> *Predict which potential donors will respond to a mailing. To achieve this, we will build a decision tree model that will predict the outcome of TARGET_B, the Boolean column describing each person's donation history based on columns describing historical response and demographic data. This will allow the boss to create a targeted mailing, using the historical data the company has collected, and thus improve the response rate. The success of the model will be determined by the increased profitability of the ad campaign.*

4

Cleaning the Data

This can really be considered the most important stage of the project. It is in working with the data—exploring it, taking out unnecessary columns, and cleaning others—that we prepare for the process of creating a data mining model. While the boss has come up with the general idea for the project, we have to make it work, and by the end of the project, nobody will know the data better than we do.

So what is this "cleaning" stuff all about? At some point, data has to enter the computer. Depending on the error-checking procedures that are in place at the point of entry, it is likely that someone will make some mistakes—entering the wrong date, the wrong salary, the wrong address, and so on. If we aren't careful, these problems can potentially reduce the effectiveness of our models. It is our job to find and rectify as many of these mistakes as we can within an allowable period of time.

In the dataset, there are around 90,000 rows of data and 481 columns. That's something like 43,290,000 cells! Obviously, we can't investigate each cell individually, so we are going to have to devise some methods of automation that will find as many potentially inaccurate records as possible.

But before cleaning the dataset, we have to know the data. We'll be making decisions here that have far-reaching consequences on the accuracy and validity of the data mining models. We need to know what it is that we're trying to predict, how the column values are formatted, and what each column tells us.

After we know the dataset, we can target the inconsistencies that we want to eliminate before building our models.

Targeting Inconsistencies

Many types of inconsistencies can occur in a dataset, including those listed in the following table.

Problem	Example
Columns that contain a high number of null values	Sensitive information that the user might not have wanted to provide
Columns with too few or too many distinct states	Telephone numbers or other columns that have a one-to-one relationship with each case, or a column that only has a single case
Records that fall far outside the normal distribution of the column	A negative salary
Records that do not match a specified format	Different date formats
Records that do not make sense when compared to similar records of a different attribute	A product with a purchase date that is earlier than the purchaser's birth date

The way in which we resolve these problems depends on the situation, the requirements of the model, and the way in which we choose to approach the problem. For example, cells that are determined to be outliers can be replaced with a mean value, replaced with a value according to a specific distribution type, or be excluded (along with the rest of the row).

In this chapter, we'll use the following techniques to address the first three problems that are listed in the table:

- For columns with a high number of null values, we'll compute the percentage of null (missing) values for each attribute to determine if the attribute should be excluded from the model-building process.
- For columns with too few distinct states, we'll compute a mean, a minimum value (min), a maximum value (max), and a distinct count for each attribute. We will use this information to exclude columns that do not seem to be useful.
- For records that fall outside the normal distribution of the column, we'll compute outliers and flag the rows in which they reside. Then we'll decide, on an individual basis, how to handle them.

We can address these problems in the Data Mining Tool by using a tab control, where each tab represents one of the three cleaning tasks that we've chosen. This tab control resides on the **Clean Data** form (Figure 4.1).

▶ **To view the Clean Data form**

- In the Data Mining Tool, click **Clean**.

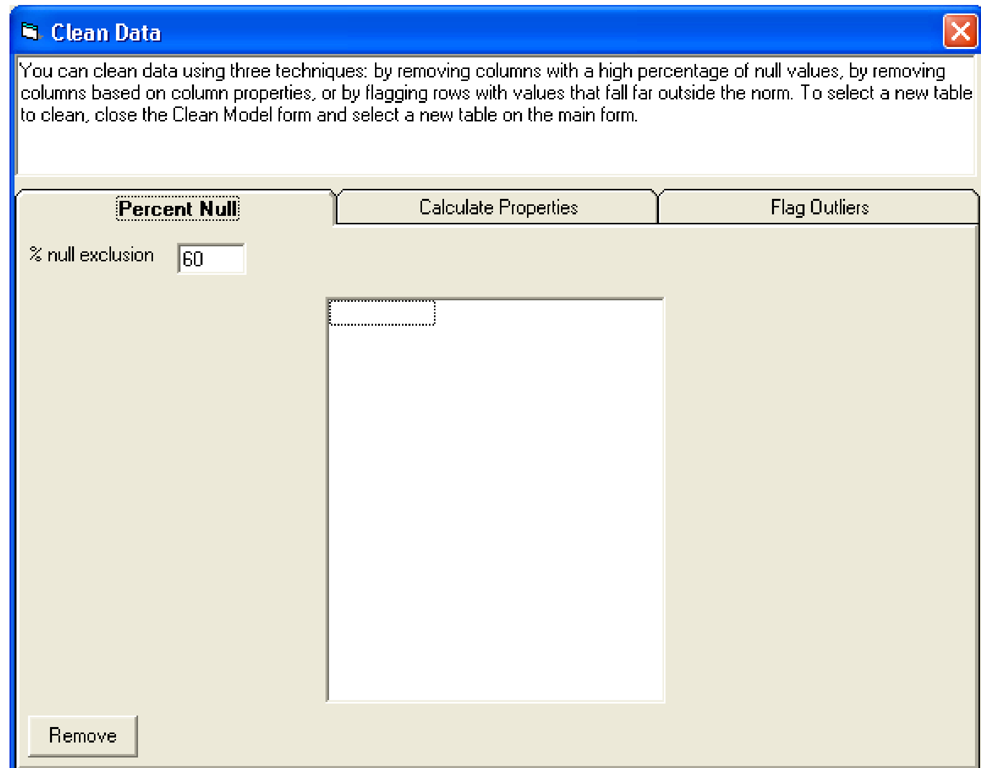


Figure 4.1 The Clean Data form

What About Those Null Columns?

The tab control on the **Clean Data** form defaults to the **Percent Null** tab, which by coincidence (or is it?) is our first step in cleaning the data. Calculating the number of null values in a column is the least time-consuming task of the three that we've selected. So it's only natural that we tackle this process first.

But why do we need to calculate the number of null values in a column, anyway?

The first thing you might have noticed about this dataset is that it has a large number of columns. Having so many columns can get extremely messy as we begin to work with the data. The more unnecessary columns we can remove now, the less time we will waste on computation in later tasks.

Columns with a high number of null values can, at best, have no effect on the final outcome of the model and, at worst, adversely affect the accuracy of the model. For this solution, we'll remove columns that have too many null values.

Trying Out the Percent Null Tab

The theory behind how the Data Mining Tool works with null columns is actually fairly simple. Because we preserve the source data, we can remove whatever columns we need to from the table being cleaned. To do this, we created a stored procedure, which is called from the tool, that calculates the number of null values in each column of the dataset and then divides that number by the total number of rows to get the percentage of null values. If this percentage is greater than a predetermined amount, the stored procedure drops the column from the table.

Let's give it a try. Because we have already selected a table to clean, we only need to input the percentage of null values that we want to allow to exist in each cleaned column.

Note The procedures in this chapter assume that the Solution_DB database is set up, a connection exists between the Solution_DB database and the Data Mining Tool, and the source data has been copied into a table to be used in the data mining process. If you have not yet completed these steps, proceed to Chapter 1, "Setup," and complete the installation procedures.

▶ **To remove null columns from the cup98LRN_clean table**

1. In the Data Mining Tool, click **Clean**.
2. On the **Percent Null** tab, for **% null exclusion**, type **60**.

   This means that if more than 60 percent of the values in a column are null, the Data Mining Tool targets that column for removal.
3. Click **Remove**.

   The Data Mining Tool now cycles through the columns in the table, removing those containing more than 60 percent null values. After removing the columns from the table, the tool displays them in a FlexGrid control on the **Percent Null** tab.

In looking at the form in Figure 4.2, we can see that the columns that were removed seem reasonable—none of them contain information that justify the number of null values that they contain.

Now that we've seen this functionality in action, let's look at the code behind all this cleaning business.

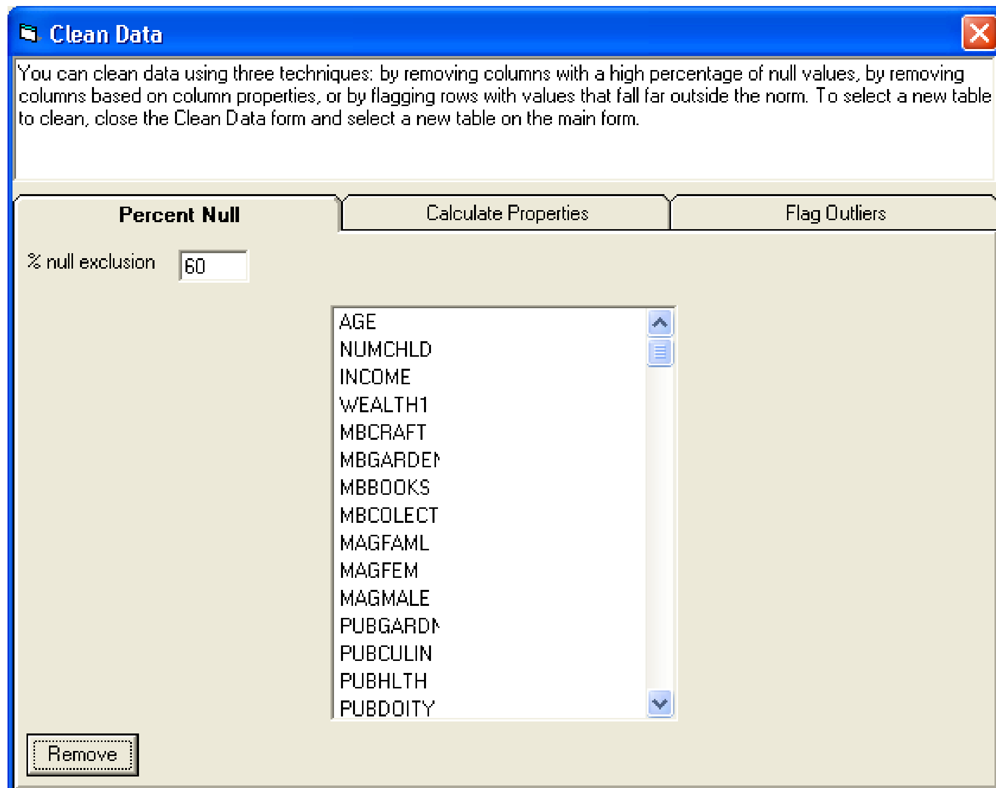


Figure 4.2 Removed columns in the FlexGrid control

Looking at the Remove Null Columns Code

To complete this cleaning task, we use three values:

- The percentage cutoff level
- The number of rows in the table
- The number of null values for each column in the table

The question is: how we should go about performing the calculations and removing the values?

We could split the work between the client and the server, grabbing a column of data from the server, moving it to the client, performing the test, and then going back to the server to drop the column from the table if it meets the criteria. Now what stands out about this solution? There is a lot of back and forth going on between the server and the client. When we're dealing with a table containing a small number of columns—say, 10—it's not so bad, only 10 trips back and forth are required. But look at our table. We start with 481 columns—that's a lot of round trips to the server! Each one of these round trips is a potential performance hit, which can cause a lot of waiting.

Alternatively, we could do all of the work on the server, creating a stored procedure that we call from the code. We tried both of these methods and found that the stored procedure processes the task about 20 percent faster! So let's see how we implemented the stored procedure that does all these heavy null value calculations.

Calculating Null Values

As with several of the tasks in this project, we have to cycle through each column in the table, performing the same operation on each one. If we did this in Microsoft® Visual Basic®, we would have had to create a recordset holding the column names and then cycle through each column, shooting a query back to the server to get the data and make necessary modifications to the table.

Let's look at how the `usp_KillNulls` stored procedure handles this challenge.

▶ To view the usp_KillNulls stored procedure

1. In Query Analyzer, expand the **DM_Prep_Tool** database, and then expand the **Stored Procedure** folder.
2. Right-click **dbo.usp_KillNulls**, and then click **Edit**.

To calculate the null values, the stored procedure requires two parameters: the name of the table being cleaned (`@strBaseTable`) and the percentage of allowable values (`@fltLimit`).

```
Create      procedure usp KillNulls
     @strBaseTable nvarchar(255),
     @fltLimit float
AS
```

We then create a new table that holds a list of all of the column names that were removed.

```
SET @strSQL='CREATE TABLE [' + @strNewTable + '] (Column Name
nvarchar(16))'
EXECUTE sp executesql @strSQL
```

In order to cycle through each column in the table, we declare a cursor that holds the column names for the base table.

```
DECLARE columns_cursor CURSOR FOR
SELECT [name] FROM syscolumns WHERE id =  OBJECT_ID(@strBaseTable)
```

The procedure then takes the number of rows from the table and stores this number in the `@iTotal` variable. We will use this value to calculate the percentage of nulls in the selected column.

```
SET @strSQL='SELECT @iTotalOut = COUNT(*) FROM ' + @strBaseTable
EXECUTE sp_executesql @strSQL,N'@iTotalOut int OUTPUT',@iTotalOut =
@iTotal OUTPUT
```

And now we start cycling through the columns! We first open the cursor, get the next available column name, and hold it in the local variable, `@strColName`.

```
OPEN columns cursor
FETCH NEXT FROM columns cursor
INTO @strColName
WHILE @@FETCH_STATUS = 0
```

Then, for each column, the procedure finds the number of null values that are present, divides that number by the total number of rows in the table (as held in the `@iTotal` variable), and compares the result against the allowable percentage set by the user. If the calculated percentage is greater than the user-defined cutoff percentage, the procedure drops the column from the table, and adds the column name to the table holding the list of columns that were removed.

```
BEGIN
    SET @strSQL='DECLARE @iNull int
                 SET @iNull=(SELECT Count(*) FROM ' + @strBaseTable
+' WHERE ' + @strColName + ' IS NULL)
                 IF cast(@iNull as float)/' + cast(@iTotal as
nvarchar(25)) + ' > ' + cast(@fltLimit as nvarchar(25)) +
            'BEGIN
                  INSERT INTO [' + @strBaseTable + ' removed] VALUES
(''' + @strColName + ''')
                  ALTER TABLE [' + @strBaseTable + '] DROP COLUMN ['
+ @strColName + ']
            END'
```

```
    EXECUTE sp executesql @strSQL
    FETCH NEXT FROM columns cursor
    INTO @strColName
END
CLOSE columns cursor
DEALLOCATE columns_cursor
```

Remember that the table being cleaned is a copy of the original table, cup98lrn. So we can make as many changes as we want without affecting the source data.

Getting the Null Values

Okay, that's how the stored procedure works. Now all we have to do is call this procedure from the code used to display and control the **Remove Nulls** tab. This code is located in the `cmdRemoveNulls_Click` subroutine. Open Visual Basic and follow along as we walk through this code.

▶ **To view the cmdRemoveNulls_Click subroutine**

1. In Windows Explorer, browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmClean** (**frmClean.frm**), and then click **View Code**.
4. Locate the `cmdRemoveNulls_Click` subroutine.

The `cmdRemoveNulls_Click` subroutine starts by declaring three new objects—a command to run the stored procedure on the server, and the two parameters that the procedure requires.

```
Dim objCommand As New ADODB.Command
Dim objParam_Table As New Parameter
Dim objParam_Limit As New Parameter
```

The routine then gathers the only user input (the cutoff percentage) that we need for the `usp_KillNulls` stored procedure.

```
sngPercentRemove = CSng(txtPercentRemove.Text)/100
```

The table that holds the names of the removed columns is then named and stored in the `strRemovedNullsTable` variable.

```
strRemovedNullsTable = frmMain.strCleanedTable & "_removed"
```

Having obtained the cutoff percentage value, the routine starts defining the parameters that the stored procedure requires.

```
...
With objParam_Table
    .Name = "@strBaseTable"
    .Direction = adParamInput
    .Type = adVarChar
    .Size = 255
    .Value = frmMain.strCleanedTable
End With

With objParam_Limit
    .Name = "@fltLimit"
    .Direction = adParamInput
    .Type = adDecimal
    .Precision = 2
    .Value = sngPercentRemove
End With
...
```

With the necessary parameters set, the routine prepares the Microsoft ActiveX® Data Objects (ADO) `Command` object and then calls the object's `Execute` method to run the stored procedure on the server.

```
...
With objCommand
    .ActiveConnection = cnDataPrep
    .CommandTimeout = 0
    .CommandText = "usp_KillNulls"
    .CommandType = adCmdStoredProc
```

```
     .Parameters.Append objParam Table
     .Parameters.Append objParam Limit
End With
...
objCommand.Execute
...
```

As we sit and wait (the time this task takes varies by computer), the stored procedure checks columns and removes those columns that exceed the cutoff percentage. By the time the procedure finishes, we have a table cleaned of those pesky null columns!

Now we just have to display those columns on the form. This allows the user to inspect the results and decide if he or she agrees with them. We grab the column names from the `strRemovedNullsTable` table that was created in the stored procedure.

```
...
Set rsData = mdlProperties.cnDataPrep.Execute("Select column_name
from " & strRemovedNullsTable & "")
Set hfgRemovedNulls.DataSource = rsData
Set hfgRemovedNulls.DataSource = rsData
...
```

If the procedure removes too many columns, the user can always go back, create a new copy of the source data, and redo this task with a different cutoff level. If it doesn't remove enough columns, the user can just re-run the procedure on the same table with a higher cutoff level.

And now, on to the next cleaning step!

What About Those Table Properties?

So far, we've removed the obvious columns—those that didn't have enough records to tell us anything—but we now have to look a little deeper. There are several properties of the columns that are easy to calculate but also tell us something about the data. These include the minimum value (min), the maximum value (max), the standard deviation (stdev), and number of unique values in the column (distinct count), which can obviously only be calculated for numeric columns.

We calculate these values for two reasons:

- To provide a better understanding of the distribution of records in each column of the dataset.
- To compute variables that will be used in exploring the data.

The more we know about the data, the more columns we can exclude, and the better choices we can make when it comes to building the models. For example, by calculating the distinct count, we can find and remove columns that only have one distinct value, and therefore, add no value to the accuracy of the model.

Trying Out the Calculate Properties Tab

The second tab of the **Clean Data** form, **Calculate Properties**, does...guess what? That's right—it calculates specific properties about each column. There are only three tasks that we can perform here: calculate properties, show a previously calculated properties table, or remove a column based on something we learned about the table from its properties. No user input is necessary. We just start clicking away!

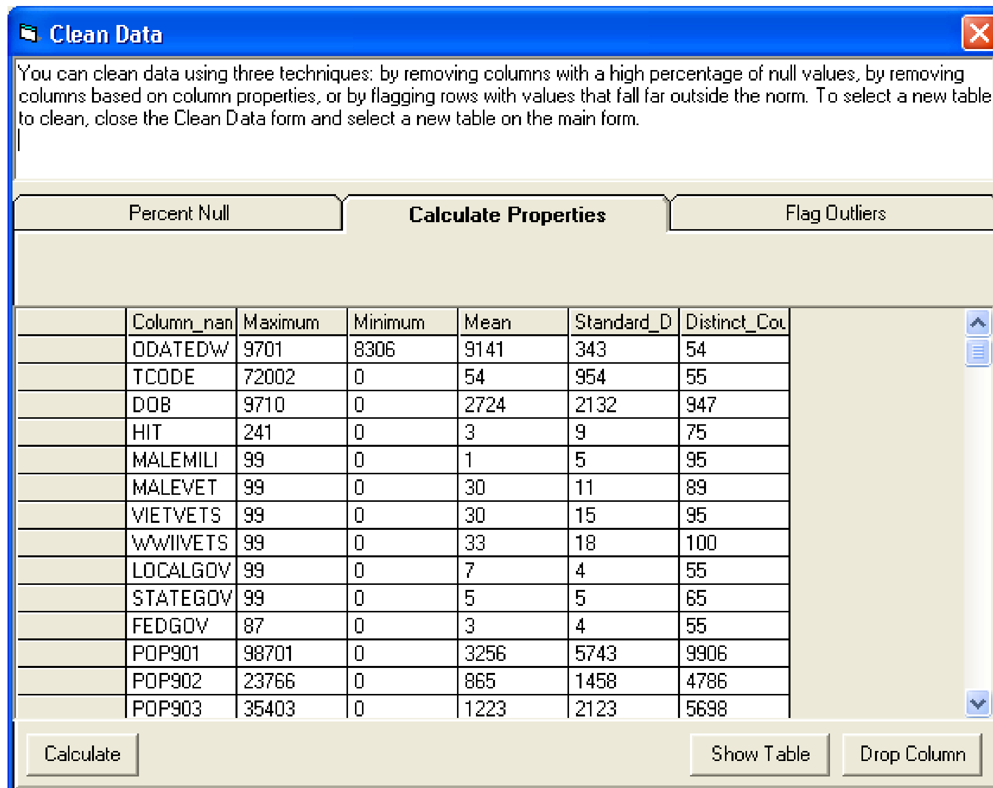
To calculate properties, the Data Mining Tool calls a stored procedure that loops through each numeric column in the table, calculates the column's properties, and stores these properties in a table. The tool then displays the results of all this hard work—the properties table—on the **Calculate Properties** tab by using a hierarchical FlexGrid control.

▶ To calculate the properties table for cup98LRN_clean

1. In the Data Mining Tool, click **Clean**.
2. On the **Calculate Properties** tab, click **Calculate**.

   Now, just relax and wait for the routine to finish—get a new cup of coffee and let the dog out. By that time, the properties will be calculated and we can begin to investigate the results shown in Figure 4.3.

   Figure 4.3 shows how the FlexGrid control on the **Calculate Properties** tab displays the properties table that we just created. Because we use these properties later when working with the data, the Data Mining Tool stores them in a table whose name is the name of the table being cleaned, appended with a _pr. In this way, anytime we need the properties associated with a table that is being cleaned, the tool can quickly find the appropriate properties table without having to set up a tracking mechanism that manages the association between each properties table and its corresponding cleaned table.



Column_nan	Maximum	Minimum	Mean	Standard_D	Distinct_Cou
ODATEDW	9701	8306	9141	343	54
TCODE	72002	0	54	954	55
DOB	9710	0	2724	2132	947
HIT	241	0	3	9	75
MALEMILI	99	0	1	5	95
MALEVET	99	0	30	11	89
VIETVETS	99	0	30	15	95
WWIIVETS	99	0	33	18	100
LOCALGOV	99	0	7	4	55
STATEGOV	99	0	5	5	65
FEDGOV	87	0	3	4	55
POP901	98701	0	3256	5743	9906
POP902	23766	0	865	1458	4786
POP903	35403	0	1223	2123	5698

Figure 4.3 The Calculate Properties tab

Let's look at what we can learn from these properties. The main thing we are trying to do in this task is to reduce the number of columns in the table. So how can this be done from the properties table?

There are a couple things we should look for when we decide which columns to exclude. Are there too many distinct states? If more than 90 percent of the values are distinct, maybe the column is not worth keeping. Ninety percent is about 85,870 rows. How many columns contain more than 85,870 distinct values? After some investigation we'll see that there is only one, CONTROLN, which also happens to be the key column. We obviously can't exclude the key column because it identifies each row.

Now let's take the opposite approach. How many columns contain a low number of distinct counts? Three columns have a distinct count of 2:

- ADATE_2
- TARGET_B
- HPHONE_D

If we hadn't removed all of those columns that contained null values in the previous step, we also would have found that the following columns contained very few distinct counts.

- ADATE_2
- ADATE_3
- ADATE_6
- ADATE_10
- ADATE_14
- ADATE_20
- ADATE_21
- ADATE_24

We can see that something is going on with the ADATE columns—what do these columns signify? If we look in the table description (from the Web site listed in Chapter 1, "Setup"), the number for each of the ADATE columns is the date (in YY/MM format) that a specific promotion was mailed out. It makes sense that these should have very few distinct counts because most likely the mailings were all mailed out at the same time of the year. The question is, should we include the remaining columns in the model? For an answer to this question, we need to explore the column and see how it looks. So let's keep these columns in mind and come back to them later in Chapter 6, "Exploring the Data."

The HPHONE_D column signifies whether the respondent has a published telephone number, so once again the fact that it only has two distinct states is normal. As for the last column, TARGET_B, it should have only two distinct states because it answers a yes/no question. Because this is our predictable column, we need to keep it.

Now look at DOB—it is an interesting column. Although it is listed as a numeric column in the database, it's not a numeric column that tracks a particular item, such as the cost of something. Instead, it contains a code, which in this case is the year and month that each customer was born (in YY/MM format). The mean is 2,724 and the standard deviation is 2,132—not exactly a normal distribution, which we need to keep in mind when we start to explore the columns. Although the date can tell us a lot, we also have an age column for which the calculated properties make much more sense.

After we explore the data, we will have a better idea of which columns we should drop. Here is the procedure we'll use eventually to drop columns from the table.

▶ To drop an unnecessary column

- On the **Calculate Properties** tab, select a column, and then click **Drop Column**.

We will come back to this procedure after we explore the columns in Chapter 6, "Exploring the Data." For now, let's look at how we constructed the code that calculates the properties.

Looking at the Calculate Properties Code

The code that makes the **Calculate Properties** tab work actually spans three different subroutines within the `frmClean` form:

- The `Form_Load` subroutine names the properties table associated with the table currently being cleaned.
- The `cmdCalculateProperties_Click` subroutine first determines whether the properties table exists, and if so, whether the user wants to re-create it. The subroutine then creates the table, calculates the property values for each column, and displays the resulting properties table on the tab.
- The `cmdDropColumn_Click` drops a column from the table being cleaned.

Let's look at the tasks each of these various subroutines accomplishes during this cleaning step. Open Visual Basic and follow along as we walk through the code.

▶ **To view the code used for the frmClean form**

1. Browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmClean** (**frmClean.frm**) and click **View Code**.

Naming the Properties Table

The point of the **Calculate Properties** tab is to produce a properties table by which the user gains insight into the contents of the table being cleaned. In addition to providing insight to the user, the properties stored in this table become essential later when we use the Data Mining Tool to transform and explore the data.

Knowing that we need this properties table, our first step is to name the table.

In the first iteration of this tool, we gave the user the opportunity to name (select) a properties table from a list of such tables wherever such a selection was important (for example, when generating a correlation matrix or graphing data). But this both cluttered the user interface and caused room for confusion (the user would have to remember which table to select from the list of tables). Because no one should ever need to select a properties table that is not associated with the table currently being cleaned, we removed that functionality and opted to set the properties table automatically for the user upon loading the **Clean Data** form.

In the Form_Load subroutine, we derive the name of the properties table by taking the name of the table being cleaned, appending _pr to the end of that name, and then storing the new name in the strTableProperties string.

```
Private Sub Form_Load()
    'strPropertiesTable is set equal to the table name the user
    'enters in the txtTableProperties textbox.
    strPropertiesTable = frmMain.strCleanedTable & " pr"
End Sub
```

Determining Whether the Properties Table Exists

We named the properties table. The next step is to actually determine whether this table exists. Why do we do this? You see, the user may have already created such a table and just wants to view that table from the **Calculate Properties** tab. If so, we don't want to overwrite the existing table. We just want to display it when the user clicks the **Show** button on the tab. Alternatively, the user may want to start over with this table, causing us to drop the existing table in preparation for creating a new one.

Thus, the first step in the cmdCalculateProperties_Click subroutine is to determine whether a properties table with the given name already exists.

```
    'Check to see if the table already exists in the database
    strSQLSelect = "SELECT TABLE NAME FROM INFORMATION SCHEMA.TABLES
WHERE TABLE_NAME = '" & strPropertiesTable & "' "
    Set rsTable = mdlProperties.cnDataPrep.Execute(strSQLSelect)


    'If the table already exists, the user can either recreate it or
exit the routine
    If rsTable.RecordCount <> 0 Then
        If MsgBox("You have already created a properties table for "
& frmMain.strCleanedTable & "." &
                "Do you want to recreate it?", vbYesNo) = vbYes Then
        Else
            GoTo Exit cmdCalculateProperties Click
        End If
```

As you can see from the code, the user has the choice of either re-creating the table or exiting the routine.

Creating the Table and Calculating Properties

Finally, we are ready to create the table and calculate the properties we've chosen for each of the columns in the table. Creating the table is not too difficult, but calculating those properties is a bit of a challenge. As with the **Percent Nulls** tab where we calculated null values for every column, we face a similar repetitive process here. This time we need to cycle through each column and calculate its properties. As before, we can either do this using a recordset in Visual Basic or through a stored procedure in Microsoft® SQL Server™.

In developing this solution, we tried both approaches. And it shouldn't come as a surprise that we determined the stored procedure approach (where the server does all the processing) improved processing time by about 40 percent over the recordset approach (where there is a lot of I/O between the client and server). Obviously, we opted to use the stored procedure.

To implement this stored procedure, we created a `Calculate_Properties` function that both creates the table and calculates the column properties. Once again, such a function lends itself to reuse. We are now able to quickly calculate table properties not only during this cleaning task, but also in creating the correlation matrix (see Chapter 6, "Exploring the Data").

By packing all this data manipulation into a function and a stored procedure, we didn't have to write lot of code for the `cmdCalculateProperties_Click` subroutine. The following line of code is all that's required.

```
Call mdlProperties.Calculate_Properties(frmMain.strCleanedTable,
strPropertiesTable)
```

Walking Through the Calculate_Properties Function

Before going into detail about the stored procedure, let's take a look what the `Calculate_Properties` function in the `mdlProperties` module does to call that procedure. As in the `cmdRemoveNulls_Click` subroutine for the **Percent Nulls** tab, the `Calculate_Properties` function first declares both the `Command` object that runs the stored procedure and the single input parameter (the table name) that is passed to the procedure.

```
Public Function Calculate_Properties(ByVal strTable As String, ByVal
strPropertiesTable As String)

    Dim objCommand As New ADODB.Command

    Dim objBase_Table As New Parameter
```

The function then defines the `objBase_Table` input parameter.

```
With objBase_Table
     .Name = "@strBaseTable"
     .Direction = adParamInput
     .Type = adVarChar
     .Size = 255
     .Value = strTable
End With
```

Next, the function prepares the `Command` object and runs it.

```
With objCommand
     .ActiveConnection = cnDataPrep
     .CommandTimeout = 0
     .CommandText = "usp Properties"
     .CommandType = adCmdStoredProc
     .Parameters.Append objBase Table
End With
objCommand.Execute
```

Here again, the code is pretty straightforward and not too complicated. Now let's look at the `usp_Properties` stored procedure that is called by the `objCommand` object.

Walking Through the usp_Properties Stored Procedure

Looking at the code so far, you've probably realized that the bulk of the computational load must be in the `usp_Properties` stored procedure. In this procedure, we first create a table to hold the calculated statistics, cycle through each numeric column, calculate the mean, min, max, standard deviation, and distinct count, and then write the values to the new properties table.

▶ To view the usp_Properties stored procedure

1. In Query Analyzer, expand the **DM_Prep_Tool** database and then expand the **Stored Procedure** folder.
2. Right-click **dbo.usp_Properties**, and then click **Edit**.

As pointed out in the discussion of the `Calculate Properties` function, the `usp_Properties` stored procedure takes only one input. This input is the name of the table for which the user wants to calculate properties. The `usp_Properties` procedure begins by establishing this parameter as a `varchar`.

```
CREATE       Procedure usp Properties
     @strBaseTable nvarchar(255)
```

Again, as previously mentioned, the procedure creates a name for the properties table by appending the name of the table being cleaned with _pr and storing this result in the local variable, `@strBaseTable`.

```
set @strNewTable = @strBaseTable + '_pr'
```

If you remember, in the beginning of the `cmdCalculateProperties_Click` subroutine, we gave the user the option of re-creating the table if it already exists. Accordingly, the stored procedure looks through the database to see if the table already exists, and drops the table if it does.

```
IF EXISTS(SELECT [name] FROM Sysobjects WHERE [name] = @strNewTable)
BEGIN
     SET @strDropSQL = 'DROP TABLE ' + @strNewTable
     EXECUTE sp executesql @strDropSQL
END
```

After any previous occurrence of the table has been dropped, the new table is created in the database.

```
SET @strCreateSQL = 'CREATE TABLE [' + @strNewTable + ']
(Column_name varchar(20) NULL, Maximum NUMERIC NULL, Minimum NUMERIC
NULL, Mean NUMERIC NULL, Standard Deviation NUMERIC NULL,
Distinct Count NUMERIC NULL)'
EXECUTE sp_executesql @strCreateSQL
```

As with the `usp_KillNulls` stored procedure used on the **Percent Nulls** tab, we get the system table ID for the newly-created properties table and then declare a cursor to hold the column names from the table being cleaned. This cursor differs from the cursor used in the `usp_KillNulls` procedure in that we are only interested in working with numeric columns. (It doesn’t make much sense to calculate the mean of `varchar` columns!) To

select only numeric columns, we select only columns of column type 108, which represents a numeric column.

```
Set @iTableID = (SELECT [id] from SysObjects where [name] =
@strBaseTable)


DECLARE Columns_Cursor CURSOR FOR
Select [name] from SysColumns where [id] = @iTableID AND [Type] =
108
```

Now that everything is set up and ready to go, we can begin to calculate the properties and write them to the new table. Using the cursor to iterate through the numeric columns, we create a Transact-SQL statement that calculates the max, min, mean, standard deviation, and distinct count, and then inserts them into the properties table.

```
OPEN Columns_Cursor
FETCH NEXT FROM Columns_Cursor
INTO @strColumn
WHILE @@FETCH_STATUS = 0
BEGIN

     SET @strInsertSQL = 'INSERT INTO [' + @strNewTable + '] ' +
                      'SELECT '''  + @strColumn +  ''' As
Column_name, ' +
                      'MAX([' + @strColumn + '])as Col_Max,' +
                         'MIN([' + @strColumn + ']) as Col_Min,' +
                       'AVG([' + @strColumn + ']) as Col_Avg,' +
                            'STDEV([' + @strColumn + ']) as
Col_STDev,' +
                            'COUNT(DISTINCT([' + @strColumn + ']))
as Col_Distinct ' +
                      'FROM [' + @strBaseTable + '] ' +
                            'WHERE [' + @strColumn + '] IS NOT NULL'
     EXECUTE sp_executesql @strInsertSQL
     FETCH NEXT FROM Columns_Cursor
     INTO @strColumn
END
```

Displaying the Properties Table

With the column properties calculated and stored, the user will most likely want to see these results in order to determine whether additional columns can be dropped from the table. To display these results, we use a hierarchical FlexGrid control. This means that we have to hook up the `DataSource` property of the FlexGrid control. Accordingly, we create a Transact-SQL statement that returns the contents of the properties table, runs it through a recordset, and sets the `DataSource` property to the recordset.

```
...
    ' Display properties data in the grid.
    strSQLSelect = "SELECT * FROM [" & strPropertiesTable & "]"
    Set rsData = cnDataPrep.Execute(strSQLSelect)
    hfgColumn.ColWidth(0) = 300
    Set hfgColumn.DataSource = rsData
...
```

Dropping a Column

After a user reviews the column properties in the FlexGrid control, it may become obvious that he or she should drop some of the columns (for instance, columns that have a distinct count of 1, indicating that these values are all the same).

To drop these types of columns, the user selects the column in the FlexGrid control and then clicks **Drop Column**. Clicking this button calls a separate routine (`cmdDropColumn_Click`) that displays a message box asking the user whether he or she really wants to drop the column. When the user clicks **Yes** in response to this message, the `cmdDropColumn_Click` subroutine uses a Transact-SQL statement to drop the corresponding column from the table being cleaned. The FlexGrid control is then refreshed.

```
Private Sub cmdDropColumn_Click()
...
    If MsgBox("Do you really want to drop " & hfgColumn.Text & "
from the table?"
                , vbYesNo) = vbYes Then
        strSQLString = "ALTER TABLE " & frmMain.strCleanedTable & "
DROP COLUMN " & hfgColumn.Text & ""
        cnDataPrep.Execute (strSQLString)
        cmdShow_Click...
```

And that wraps up the process of calculating table properties. Next up, the way in which we handle outlier values in the table.

What About Those Outliers?

Until now, we have been cleaning data at the column level—removing groups of data one column at a time. Here the focus shifts to looking at values at the individual cell level. Because the dataset is so large and it would be impossible to physically investigate every cell, the Data Mining Tool automates the process of flagging cells that stand out. Because it is not a good idea to just blindly remove these cells or their associated rows without some sort of validation, the Data Mining Tool has a built-in mechanism that allows us to review flagged cells and decide whether to remove them completely (including the entire row) or to replace them with the average value of the column. Figure 4.4 shows the **Flag Outliers** tab. We'll use this tab to specify which values we want the Data Mining Tool to flag as outliers.

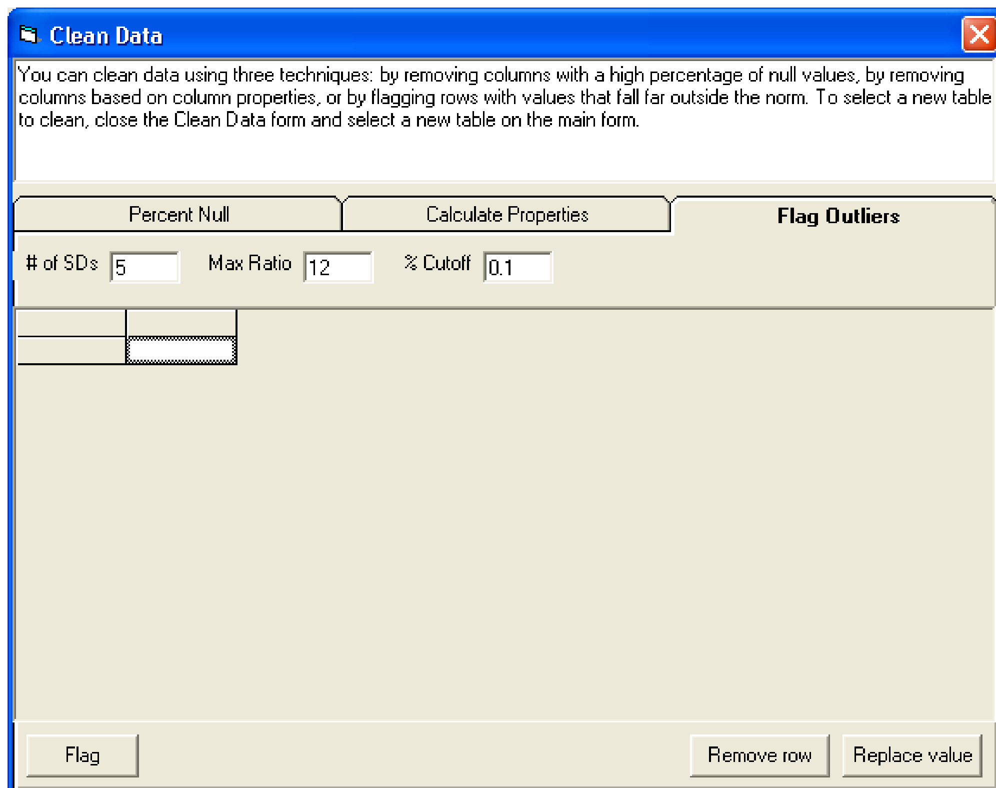


Figure 4.4 The Flag Outliers tab

Before we can flag outlier values, we must set three variables:

# of SDs
The number of standard deviations from the mean after which a value is considered to be an outlier.

Max Ratio
A ratio signifying that the maximum value is really out there, too far from the outlier cutoff to be considered a valid value.

% cutoff
The percentage of values past the outlier limit that we will allow before starting to wonder if they are really outliers or not. (Because we cannot assume that the distribution is normal, a large number of values a may be a significant distance from the mean. If this is true, we do not want to flag these values because they may be valid.)

We set these variables because the algorithms used to flag cells take into account both the distance away from a mean value and how often values of that distance occur.

Here is the first algorithm:

```
Outlier_Value= Mean + STDev*Number of STDev
```

To get this algorithm to work, we must first determine the number of standard deviations after which we consider values to be outliers. According to a normal distribution, three standard deviations should encompass 99.7 percent of the values. So the default is 3.

Having computed the `Outlier_Value`, the next algorithm looks at the ratio between the maximum value in the dataset and the outlier value:

```
Outlier_Ratio = MaxValue /OutlierValue
```

If this ratio is high (that is, greater than the value for the maximum ratio), we know that the maximum value is way beyond the outlier cutoff and is most likely an outlier. If it is low (less than the maximum ratio), we know that it is just barely beyond the cutoff and we won't consider those values beyond the cutoff to be valid outlier values.

We then calculate the percentage of values that lie beyond the cutoff. If it is high (greater than the cutoff percent value), we can assume that the values are not outliers (because so many exist). But if it is extremely low, we can assume that they are outliers.

Trying Out the Flag Outliers Tab

Finally, we come to the third tab (and the last cleaning task)—finding outlier values in each column. After identifying all of the outliers, the **Flag Outliers** tab displays the rows containing the outliers, highlighting outlier values in red. We then either remove a column containing an outlier from the table or replace it with the mean value of the column.

Let's give it a try. To find the outliers, we need to input three parameters, as explained earlier: the number of standard deviations that most of the data should be contained within, the maximum ratio that the maximum value should fall within before it is considered to be an outlier, and the percentage of values that are allowed beyond the outlier value before they are no longer considered to be outliers.

▶ To flag outliers in the cup98LRN_clean table

1. In the Data Mining Tool, click **Clean**.
2. On the **Flag Outliers** tab, type the following values:
   - For **# of SDs**, type **5**
   - For **Max Ratio**, type **12**
   - For **% Cutoff**, type **.1**.
3. Click **Flag**.

   Now the Data Mining Tool begins its work. After scouring the columns in the table for outliers, the tool displays those rows containing outliers in a FlexGrid control on the form. The cells containing outliers appear in red to make them easy to spot.

Clean Data

You can clean data using three techniques: by removing columns with a high percentage of null values, by removing columns based on column properties, or by flagging rows with values that fall far outside the norm. To select a new table to clean, close the Clean Data form and select a new table on the main form.

Percent Null | Calculate Properties | **Flag Outliers**

# of SDs 5 Max Ratio 12 % Cutoff 0.1

ODATEDW	OSOURCE	TCODE	STATE	ZIP	MAILCODE	PVASTATE	DOB	NC
8801	AHC	0	CO	81501			1602	0
9201	APP	1	CA	92675			1601	0
8701	LIS	0	MO	63028			0	0
8601	DRK	0	CA	94583			0	0
8601		0	FL	33311			0	0
8801	DNA	2	OR	97206			0	0
8601	MBC	1	NM	87111			1801	0
8601	BHG	0	IA	51041			6001	0
8601	MAT	28028	IL	60506-			2005	0
8801	STL	0	CA	90808			2701	0
9101	HCC	0	HI	96744			0	0
8801	AML	1	MI	48131			1801	0
8601	FRC	2	FL	34284			1505	0

Flag | Remove row | Replace value

Figure 4.5 The outliers in the DM_Prep_Tool data

The first time we ran this routine, we set values of 3, 10, and 0.4, respectively, and found that 20,000 rows contained outlier values. Does it seem right that almost 22 percent of the rows in the table contain outliers? Well, it didn't seem right to us either, so we revised the input variables to those in the procedure in this section.

In looking through the flagged values we can see that some really stick out—specifically those in the TCODE column. Most of the values fall within a very small range, say 1 through 10, but as shown in the Figure 4.5, some are as high as 28,028. Now before we go crazy and start dropping rows, let's look at the column definition and see if the values make sense. According to the column definition, TCODE stands for the respondent's title, with the title MSS. being represented by the number 28,028. Looking through the flagged values, we can see that each of the values flagged in the TCODE column actually corresponds to a valid state in the column definition.

Now look at AFC3. Several rows are selected because of the values in this column, but are they really outlier values? The AFC3 column describes the percentage of females who are active in the military in the respondent's neighborhood. In looking at the values we can see that they are all less than 100, which means that they are all potentially good values. But they keep popping up as outliers because the majority of the values are very small, but certain neighborhoods have a higher percentage, which causes the values to be flagged.

Of the values the routine marked as outliers, we could not find any that seemed to fit the definition of an outlier value. But suppose we did find an outlier value—if we click an outlier value in the table, we can use the Data Mining Tool to either remove the corresponding row from the table or replace the cell's value with the mean value of the column.

▶ **To remove or replace outliers in the DM_Prep_Tool data**

1. In the grid, click a cell containing an outlier value.
2. Do one of the following:
   - To remove the row containing the outlier, click **Remove row**.
   - To replace the value of the cell with the mean value of the column, click **Replace value**.

Now, lets look at the code that makes this all work.

Looking at the Flag Outliers Code

Looking through the Visual Basic code contained in the `cmdRemoveNulls_Click` subroutine, you will see that it consists of two main parts:

- Code that prepares the stored procedure that flags the outlier values and then runs that procedure
- Code that displays the combination of the outlier table and source table data in a hierarchical FlexGrid on the form.

In this section we'll take a closer look at each of these parts: the stored procedure, the code that runs the procedure, and the code that displays the outlier information.

Flagging the Outliers

When developing this tool, the way in which we flagged outliers became an interesting problem. How could we store the information about which cells contained outlier values?

To store it in the same table meant either creating an additional column for each numeric column in the table and checking off a flag, or somehow changing the original data to signify that a value is suspect. Neither of these options seemed like a good choice.

We decided to make a copy of the original table, leaving out the actual values. Every time we found an outlier value, we put an identifier in the corresponding cell of the new table. We also added a new column to the table that signifies whether a row contains outlier values. This made it possible to perform filtering on either table and to just look at the rows containing outliers. With these two tables, we can both preserve the original data and store outlier information. Anytime we want to see a combination of the two, we can use an inner join.

That's the theory behind flagging outliers. We then had to decide how to implement this theory. As with the other cleaning tasks, we had a choice of either using Visual Basic to loop through the table columns or running a stored procedure. As before, we chose the stored procedure.

Let's walk through the stored procedure and see how our theory translates into reality.

▶ To view the usp_Outliers stored procedure

1. In Query Analyzer, expand the **DM_Prep_Tool** database, and then expand the **Stored Procedures** folder.
2. Right-click **dbo.usp_Outliers**, and then click **Edit**.

The stored procedure requires six parameters:

- The source table (`@strBaseTable`)
- The key column of the table (`@strKeyID`)
- The properties table associated with the source table (`@strLookupTable`)
- The number of standard deviations from the mean (`@fltNumberSD`)
- A cutoff percentage (`@fltPercentageCutoff`)
- An outlier ratio (`@fltOutlierMax`)

These parameters are defined first in the procedure.

```
CREATE    Procedure usp Outliers
     @strBaseTable nvarchar(255),
     @strKeyID nvarchar(255),
     @strLookupTable nvarchar(255),
     @fltNumberSD float,
     @fltOutlierMax float,
     @fltPercentageCutoff float
```

The procedure then has to name the table that will record the location of the outlier values. It does this by appending the name of the source table with _ou.

```
SET @strNewTable = @strBaseTable + '_ou'
```

To calculate the percentage of values that lie outside of the cutoff value, we need to divide the number of outlier values by the total number of records in the column. We currently don't have this total, so we have the procedure get this count and store it in the `@intRecCount` variable.

```
SET @strLookupSQL = 'SELECT @return Count = (SELECT
COUNT([controln]) FROM [' + @strBaseTable + '])'
EXECUTE sp_executesql @strLookupSQL, N'@return_Count INT OUTPUT',
@return_Count = @intRecCount output
```

Next, the procedure queries the database to see if the table that will hold the outlier values (let's call it the outlier table) has already been created, and if so, drops it from the database.

```
IF EXISTS(SELECT [name] FROM Sysobjects WHERE [name] = @strNewTable)
BEGIN
     SET @strDropSQL = 'DROP TABLE ' + @strNewTable
     EXECUTE sp executesql @strDropSQL
END
```

The next query uses a SELECT INTO statement to make the outlier table an identical copy of the source table. Because we want the outlier table to be blank, we exclude the actual data from this table by including a WHERE clause that can never happen—the key being less than zero. Then the query copies the key column data from the source table into the outlier table. This ensures that the outlier table has the same number of rows as the source table and that they are in the same order, allowing us to later join the new table with the old table for display on the **Flag Outliers** tab. Last, the query adds a column, `FLAGGED_ROWS`. (Initially, the values in the `FLAGGED_ROWS` column are set to null. Later, the stored procedure replaces the null value with a 1 if one of the row's cells holds an outlier value.)

```
SET @strCreateSQL = 'SELECT * INTO [' + @strNewTable + '] FROM [' +
@strBaseTable + '] WHERE [' + @strKeyID + '] < 0 '  +
               'ALTER TABLE [' + @strNewTable + '] ADD flagged rows
INT NULL ' +
               'INSERT INTO [' + @strNewTable + '] ([' + @strKeyID
+ ']) ' +
               'SELECT [' + @strKeyID + '] FROM [' + @strBaseTable
+ ']'
EXECUTE sp_executesql @strCreateSQL
```

As with the previous two stored procedures, we need to get the ID of the source table in order to get the names of the columns in the table.

```
Set @iTableID = (SELECT [id] from SysObjects where [name] =
@strNewTable)
```

The procedure then declares the `Columns_Cursor` cursor, which will hold the numeric column names from the source table.

```
DECLARE Columns Cursor CURSOR FOR
Select [name] from SysColumns where [id] = @iTableID AND [xType] =
108
```

The procedure can now begin to loop through the columns in the table to look for outlier values.

```
OPEN Columns_Cursor
FETCH NEXT FROM Columns_Cursor
INTO @strColumn
WHILE @@FETCH STATUS = 0
BEGIN
```

The first step in the loop is to gather the table properties used to calculate the outlier values—the standard deviations, mean, and max values.

```
     SET @strLookupSQL =
            'Select @return STDev = (SELECT Standard Deviation from
' + @strLookupTable + ' WHERE Column_name = ''' + @strColumn + ''')
' +
            'Select @return Mean = (SELECT mean from ' +
@strLookupTable + ' WHERE Column name = ''' + @strColumn + ''') ' +
            'Select @return Max = (SELECT Maximum from ' +
@strLookupTable + ' WHERE Column_name = ''' + @strColumn + ''') '
     EXECUTE sp_executesql @strLookupSQL, N'@return_STDev float
OUTPUT,@return Mean float OUTPUT,@return Max float OUTPUT',
@return STDev = @fltCol STDev OUTPUT,@return Mean = @fltCol Avg
OUTPUT,@return_Max = @fltCol_Max OUTPUT
```

Using the number of standard deviations from the mean (as set by the user), the loop then calculates a cutoff value for outliers and uses the maximum value to determine a ratio for which the maximum is beyond the outlier—the `OutlierRatio`. To avoid dividing by zero, the loop calculates the maximum ratio only if the value for `sngOutlier` is greater than zero.

```
     SET @fltOutlier = (@fltCol_Avg + @fltNumberSD * @fltCol_STDev)
     IF @fltOutlier = 0
            BEGIN
```

```
                    SET @fltOutlierRatio = 0
             END
      ELSE
             BEGIN
                    SET @fltOutlierRatio = @fltCol Max / @fltOutlier
             END
```

Now the loop looks to see if the outlier ratio is greater than the ratio determined by the user. If it is, the procedure flags all of the values past the outlier value as outliers.

```
      IF @fltOutlierRatio > @fltOutlierMax
             BEGIN
                    SET @strLookupSQL = 'Select @return_Count =
(SELECT Count([' + @strColumn + ']) from [' + @strBaseTable + ']
where [' + @strColumn +  '] > ' + cast(@fltOutlier as nvarchar(25))
+ ')'
                    EXECUTE sp_executesql @strLookupSQL,
N'@return_Count decimal OUTPUT', @return_Count = @intOutlierCount
OUTPUT
```

By dividing the number of calculated outliers by the variable `@intRecCount` (the number of records in the table), the loop now finds the percentage of values in the column that are outliers. If this percentage is less than the percentage set by the user, the loop flags the row as containing an outlier for that column.

```
                    SET @fltPercentageOut = @intOutlierCount /
@intRecCount
                    IF @fltPercentageOut < @fltPercentageCutoff
                           BEGIN
```

Finally, the procedure replaces the cells in the outlier table corresponding to outlier values in the base table with a value of 2 to signify the presence of an outlier value. When a row contains an outlier, the procedure also updates the corresponding cell in the `flagged_row` column with a value of 1. This allows us to sort rows based on whether they contain outlier values.

```
                                  SET @strLookupSQL = 'UPDATE [' +
@strNewTable +  '] SET [' + @strColumn +  '] = 2, [flagged_rows] = 1
FROM [' + @strNewTable + '] t,[' + @strBaseTable + '] s WHERE t.[' +
```

```
@strKeyID + '] = s.[' + @strKeyID + '] AND s.[' + @strColumn + '] >
' + cast(@fltOutlier as nvarchar)
                                EXECUTE sp_executesql @strLookupSQL
                         END
           END
     FETCH NEXT FROM Columns_Cursor
     INTO @strColumn
END
```

Getting the Outliers

Okay, we now have a way to flag outliers. We just need to implement this stored procedure from the **Flag Outliers** tab. Here is how we do that within the `cmdCalculateOutliers_Click` subroutine.

As with the `usp_Outliers` stored procedure, the subroutine names the table that holds outlier information by appending _ou to the name of the source table.

```
    strOutlierTable = frmMain.strCleanedTable & "_ou"
```

As with the previous cleaning tasks, the subroutine must declare each parameter and the ADO `Command` object.

```
Dim objCommand As New ADODB.Command

Dim objBase Table As New Parameter
Dim objKeyID As New Parameter
Dim objLookupTable As New Parameter
Dim objNumberSD As New Parameter
Dim objOutlierMax As New Parameter
Dim objPercentageCutoff As New Parameter
```

The subroutine then prepares the various parameters to be used by the stored procedure.

```
    With objBase_Table
        .Name = "@strBaseTable"
        .Direction = adParamInput
        .Type = adVarChar
        .Size = 255
```

```
        .Value = frmMain.strCleanedTable
    End With

    With objKeyID
        .Name = "@strKeyID"
        .Direction = adParamInput
        .Type = adVarChar
        .Size = 255
        .Value = KEYID
    End With

    With objLookupTable
        .Name = "@strLookupTable"
        .Direction = adParamInput
        .Type = adVarChar
        .Size = 255
        .Value = strPropertiesTable
    End With

    With objNumberSD
        .Name = "@fltNumberSD"
        .Direction = adParamInput
        .Type = adSingle
        .Precision = 10
        .Value = CSng(txtNumberSD.Text)
    End With

    With objOutlierMax
        .Name = "@fltOutlierMax"
        .Direction = adParamInput
        .Type = adSingle
        .Precision = 10
        .Value = CSng(txtMaxRatio.Text)
    End With
```

```
With objPercentageCutoff
    .Name = "@fltPercentageCutoff"
    .Direction = adParamInput
    .Type = adSingle
    .Precision = 10
    .Value = CSng(txtPercentCutoff.Text)
End With
```

Next, the procedure passes all of this necessary information to the `Command` object.

```
With objCommand
    .ActiveConnection = cnDataPrep
    .CommandTimeout = 0
    .CommandText = "usp_Outliers"
    .CommandType = adCmdStoredProc
    .Parameters.Append objBase_Table
    .Parameters.Append objKeyID
    .Parameters.Append objLookupTable
    .Parameters.Append objNumberSD
    .Parameters.Append objOutlierMax
    .Parameters.Append objPercentageCutoff
End With
```

Finally, the procedure carries out the command and the stored procedure begins the process of flagging outliers.

```
objCommand.Execute
```

The `usp_Outlier` stored procedure takes the longest of the three to run—there are a lot of values to check! You may even have time to make lunch while you're waiting. But the good news is that by using a stored procedure, we were able to cut processing time from around 30 minutes to around 15 minutes on our computer.

Displaying the Outliers

Now that the outlier values have been calculated, let's look at how they are displayed to the user.

In addition to the challenge of tracking outlier values, the other challenge we faced in developing this solution was figuring out how to relay the information about outliers to the user. At first, we tried to display the actual outlier table in a FlexGrid control, which, when the user selected a row value, would display the corresponding row in an additional FlexGrid control. This would have allowed the user to quickly find the outlier values and view the row values to determine what action to take. We eventually scrapped this approach in favor of displaying only a single grid that conveys both pieces of information—displaying the real values in the grid but color-coding those values flagged as outliers.

To do this single-grid approach, the `cmdCalculateOutliers_Click` subroutine uses a Transact-SQL statement that creates an inner join between the outlier table and the source table, and then returns real values for each row containing an outlier value to the FlexGrid control.

```
...
    strSQLSelect = "SELECT c.* FROM [" & frmMain.strCleanedTable &
"] AS c INNER JOIN [" & strOutlierTable & "] AS t ON c.controln =
t.controln AND t.flagged_rows = 1 ORDER BY c.controln" Set rsData =
mdlProperties.cnDataPrep.Execute(strSQLSelect)
...
```

The FlexGrid control then displays this information.

```
...
Set hfgOutlier.DataSource = rsData
...
```

The tricky part comes in highlighting the outlier values within the grid. We first have to get a recordset filled with the outlier table data that marks the outlier values with a 2. We then cycle through the table and whenever we find an outlier value, we set the FlexGrid control to the corresponding value and highlight the cell in red.

```
...
    strSQLSelect = "SELECT * FROM [" & strOutlierTable & "] where
flagged_rows = 1 ORDER BY controln"
Set rsData = mdlProperties.cnDataPrep.Execute(strSQLSelect)
```

```
lngRow = 1
Do Until rsData.EOF
    For lngColumn = 0 To rsData.Fields.Count - 1
        If rsData(lngColumn) = 2 Then
            hfgOutlier.Col = lngColumn + 1
            hfgOutlier.Row = lngRow
            hfgOutlier.CellBackColor = vbRed
        End If
    Next lngColumn
    rsData.MoveNext
    lngRow = lngRow + 1
 Loop
 ...
```

The only two tasks left to do are to either replace a selected cell with its mean value or remove a selected row.

Replacing a Cell with Its Mean Value

If the user clicks **Replace value**, the code first displays a message box asking whether the user really wants to replace the value with the mean.

```
...
If MsgBox("Do you really want to replace the selected value in the
table?"
                , vbYesNo) = vbYes Then
...
```

If the user clicks **No**, the routine exits. Otherwise, the routine begins to determine where the cell is within the grid. The hardest part of this routine is figuring out how to find which column in the FlexGrid control holds the key to the table. The code does this by setting a variable, `strColumnID`, equal to the first column and first row in the FlexGrid control, and then cycling through the column values in the first row until they match the key column name for the dataset, “`controln`”.

```
...
strColumn = hfgOutlier.TextMatrix(0, hfgOutlier.Col)
    For lngColumnID = 0 To hfgOutlier.Cols
```

```
        If hfgOutlier.TextMatrix(0, lngColumnID) = UCase("controln")
Then Exit For
   Next lngColumnID
...
```

Using the selected column name in the `strColumn` string, along with the key identifier for the selected row, the routine then creates a SELECT statement that updates the table value with the mean value for the column.

```
...
strSQLString = "UPDATE " & frmTables.strCleanedTable & " SET " &
strColumn & " = " & sngMean & " where " & hfgOutlier.TextMatrix(0,
lngColumnID) & " = '" & hfgOutlier.TextMatrix(hfgOutlier.Row,
lngColumnID) & "'"
...
```

Removing a Row

The routine for removing a row is much the same, except that instead of replacing a column value, the routine creates a SELECT statement that removes the row containing the outlier value.

```
...
strSQLString = "Delete from " & frmTables.strCleanedTable & " where
" & hfgOutlier.TextMatrix(0, lngColumnID) & " = '" &
hfgOutlier.TextMatrix(hfgOutlier.Row, lngColumnID) & "'"
...
```

Now that our data should be in a fairly clean state, let's start looking at the types of transformations we can perform to further prepare the data for use in models.

5

Transforming the Data

Now that we've cleaned the data—getting rid of all of the trouble-causing columns and rows—it's time to transform some of it into a more desirable form. To do this, we will use the DTS Import/Export Wizard, which can run scripts that transform the data. This step is yet another example of why we need to understand our data. If we do not know how a column behaves, we can't determine how to transform it into a more usable state.

There are several types of transformations that we can perform. For example, it can be hard to find out how the states of the input columns affect the states of the output columns if the input columns have too many states. (This is the case with one of our input columns, POP901. This column defines the population of the neighborhood in which each respondent lives, and thus, consists of several thousand different states.) A way to solve this problem is to reduce the number of states in the input columns by creating buckets. Instead of having an infinite number of possibilities, we define five states (labeled 1 through 5) and replace the column's values with whatever designation is appropriate. This transformation makes it much easier to find relationships between the input and predictable columns. There are two places in the process where we can perform this transformation: either now using a script in the wizard or later during the model-building phase. We will demonstrate the first technique in this solution.

Another transformation converts a date into the number of months since an event occurred. In our table, the column ODATEW describes when each person first donated money. Instead of working with a date, we could work with a concrete number, such as the number of months, that we can then use to find things like the correlation and mean value.

In this chapter, we try both of these transformations on the POP901 and ODATEW columns, using just a single script. Keep in mind, though, that the number of transformations that you can perform is endless. You can easily modify the transformations in this chapter to create your own transformation, and see how it changes your models.

Note that we will not replace the existing column; instead, we add the transformed column to the table. This preserves the state of the original column and also allows us to look at the information contained within it in a different way.

Trying Out the DTS Import/Export Wizard

In the Data Mining Tool, clicking the **Transform** button opens the DTS Import/Export Wizard, allowing you to run through the steps of creating a transformation and then return to the Data Mining Tool.

▶ **To transform data using the DTS Import/Export Wizard**

1. In the Data Mining Tool, click **Transform**.
2. On the **Data Transformation Services Import/Export Wizard** page, click **Next**.
3. On the **Choose a Data Source** page, select the following options, and then click **Next**:
   - For **Data Source**, select **Microsoft OLE DB Provider for SQL Server**.
   - For **Server**, type (**local**).
   - Select **Use Windows NT Integrated Security**.
   - For **Database**, select **DM_Prep_Tool**.
4. On the **Choose a destination** page, select the following options and then click **Next**:
   - For **Destination**, select **Microsoft OLE DB Provider for SQL Server**.
   - For **Server**, type (**local**).
   - Select **Use Windows Authentication**.
   - For **Database**, select **DM_Prep_Tool**.

   Note These settings map the source database to itself. If you want to, you can also map the source database to a new database.

5. On the **Specify Table Copy or Query** page, select **Use a query to specify the data to transfer**, and then click **Next**.
6. On the **Type SQL Statement** page, type the following statement into the **Query statement** box:

```
select [cup98LRN_Clean].[ODATEDW], [cup98LRN_Clean].[POP901],
[cup98LRN Clean].[CONTROLN]
from [cup98LRN_Clean]
```

7. Click **Next**.
8. On the **Select Source Tables and Views** page, under **Transform**, click the browse (...) button.

9. On the **Transformations** tab, select **Transform information as it is copied to the destination,** and then click **Browse**.
10. In the **Open** dialog box, select the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample\transform_script.txt** file, and then click **Open**.

    Although you can type any script that you like in the text box, we have prepared one ahead of time, which is discussed later in this chapter in "The Transformation Script." All of the transformations are performed using just one script.
11. Click **OK**, and then click **Next**.
12. On the **Save, schedule, and replicate package** page, select **Run immediately**, and then click **Next**.

The POP901 and ODATEDW columns are now transformed and added to a new table named Results, along with the key column, CONTROLN. We now just have to run the following Transact-SQL scrip to copy the new columns into the cup98LRN_clean table.

Note If you use a non-English computer, the DTS Import/Export Wizard puts the transformed data into a table with a name other than Results. For example, on a German computer the DTS Import/Export Wizard puts the transformed data into a table named Ergebnisse. Check your database to find the proper name and replace the name Results with the correct name in the following script:

```
ALTER TABLE dbo.cup98LRN_Clean ADD ODATEDW2 NUMERIC(18,0) NULL,
POP901 2 NUMERIC(18,0) NULL
Go
UPDATE dbo.cup98LRN Clean
   SET ODATEDW2 = Results.ODATEDW, POP901 2 = Results.POP901
      FROM Results
         WHERE cup98LRN Clean.controln = Results.controln
Go
```

Looking at the Code Calling the Wizard

The design of this step has changed over time. During the initial discussions about creating this project, we designed a form that allowed users to create a custom Transact-SQL script to transform data and then apply this script to the data. As time passed, we realized that SQL Server had already created a perfectly good mechanism for doing this—the DTS Import/Export Wizard. So, instead of reinventing the wheel, we decided to take advantage of the wizard in showing you how to perform custom transformations on columns on a

table. Using the wizard, we are basically going to copy the specified columns, transform them, and put them back into the table.

Hooking up the DTS Import/Export Wizard to the Data Mining Tool is a simple task, requiring only the following code.

```
Shell Environ("ProgramFiles") & "\Microsoft SQL
Server\80\Tools\Binn\dtswiz.exe", vbNormalFocus
```

The Transformation Script

Using a Microsoft ®Visual Basic® Scripting Edition (VBScript) script, we can easily transform the data for the specified columns in the table. When the script runs, the database goes through each row in the source table and runs it through the script, performing the necessary transformation. The row is then copied into the new table, which in our case is the original table. In essence, we are pulling a row of data out of the table, transforming it, and putting it right back in.

To create the script, we copied the template that was already in the code editor (which just grabs each row and moves it without transforming it) and modified it. Let's look at the code we modified.

We first need to define local variables for the columns we are going to transform.

```
    Dim lvPOP901
    Dim lvODATEDW
    Dim Month
    Dim Year
```

We can now start to perform the transformations. In the first, we will change the variable, `lvODATEDW`, which tells us the date of the donor's first donations, from a date to the number of months since their first donations.

The data is formatted as YY/MM in the database. We first pull out the individual year and month for each row, and then recalculate them to be equal to the number of months (assuming the present year is 1998).

```
    Month = cdbl(right(DTSSource("ODATEDW"),2))
    Year = cdbl(left(DTSSource("ODATEDW"),2))

    lvODATEDW = (98-year)* 12 + month
```

In the next transformation, we convert the variable `lvPOP901` from a continuous variable to a discrete variable.

We first check to see if the value is null, and then convert it to a double, which allows us to use mathematical comparisons on the value. If the value is null, we set it equal to zero.

```
    If IsNull(DTSSource("POP901")) Then
          ' If null, then 0
          lvPOP901 = 0
    Else
          lvPOP901 = CDbl(DTSSource("POP901"))
    End If
```

We then start the discretization process.

```
    If lvPOP901  >= 0 And lvPOP901  <= 19740.2 Then
          lvPOP901  = 1
    ElseIf lvPOP901  > 19740.2 And lvPOP901  <= 39480.4 Then
          lvPOP901  = 2
    ElseIf lvPOP901  > 39480.4 And lvPOP901  <= 59220.6 Then
          lvPOP901  = 3
    ElseIf lvPOP901  > 59220.6 And lvPOP901  <= 78960.8 Then
          lvPOP901  = 4
    ElseIf lvPOP901  > 78960.8 Then
          lvPOP901  = 5
    End If
```

The column values are then added to the destination table.

```
    DTSDestination("CONTROLN") = DTSSource("CONTROLN")
    DTSDestination("ODATEDW") = lvODATEDW
    DTSDestination("POP901") = lvPOP901
...
    Main = DTSTransformStat OK
End Function
```

6

Exploring the Data

Data mining is more of an art than a science. No one can tell you exactly how to choose columns to include in your data mining models. There are no hard and fast rules you can follow in deciding which columns either help or hinder the final model. For this reason, it is important that you understand how the data behaves before beginning to mine it. The best way to achieve this level of understanding is to see how the data is distributed across columns and how the different columns relate to one another. This is the process of exploring the data.

When it comes to exploring data, both visual and numeric techniques offer unique perspectives:

- Visual techniques allow you to quickly look through a large number of columns and get a general feel for how they interact. To visualize non-numeric (`varchar`) data, you build histograms. To visualize numeric data, you can build both histograms and scatter plots. To see an example of a histogram, see Figure 6.1. To see an example of a scatter plot, see Figure 6.3.
- Numeric techniques, on the other hand, give you a more concrete understanding of how the data interacts. As the term "numeric" implies, you can use numeric techniques only with numeric columns. But by focusing only on the numeric columns, you can build a correlation matrix that shows the relationship between the numeric columns and the predictable column.

Because both visual and numeric techniques provide you with a deeper understanding of the data, the Data Mining Tool presented in this book includes both techniques.

In this chapter, we'll build histograms, scatter plots, and a correlation matrix, which we'll use to explore the data in our sample dataset. After completing each step, we'll look at the code in the Data Mining Tool that makes each step work.

Visualizing Data with Histograms and Scatter Plots

A histogram describes the distribution of the different states of a column with respect to the predictable column. For example, suppose there are two possible states for a column that describes the color of shirts that people wear in a particular situation: crimson and gray. When that column is compared to a column containing answers to a yes/no question, we find that those who wear crimson shirts answer "yes" to a question 25 percent of the time, while people who wear gray shirts may answer "yes" only 5 percent of the time.

We'll build two kinds of histograms, one based on `varchar` columns (like color) and one based on numeric columns (like salary). The main difference between the two is in the number of states that each column can contain. A `varchar` column typically has a finite number of states, such as crimson or gray, while a numeric column can have an infinite number of states, such as the time an event occurred, or a salary. To build a histogram, we need to compare a small number of states to a predictable column. This means that for numeric columns we have to create artificial buckets in which to group the data. Although a salary column can contain 10,000 states, we can transform that data into three new states: high, medium, and low, which we can then chart in a histogram. For a `varchar` column, we do not have to worry about creating buckets; instead, we can just work with the data in the column.

A scatter plot uses two axes to describe groupings of data in the predictable column with respect to the different states of the selected column. The x-axis holds the input column, which we are exploring, while the y-axis holds the column that we want to predict. By plotting these against each other, we can determine where the majority of the data lies. Do positive responses in the predictable column occur only for certain states of the input column? Or do the same number of positive responses tend to occur throughout the states, making no state unique?

So that's the insight that histograms and scatter plots can provide. Now it's time to get back to work and implement these visual techniques using the Data Mining Tool. Regardless of whether we're creating a histogram or a scatter plot, the basic methodology behind the **Charts** tab is simple—create a SELECT statement based on the selected column data, fill the recordset, and set the data source of the Microsoft Chart control so that it is equal to the recordset. The only thing that changes between chart types is the formatting of the chart, and the only thing that changes between column types is the formulation of the SELECT statement.

Programming Challenges

When trying to visually explore the data using charts, we encountered two challenges. The first was in working with the Microsoft Chart control—there is little documentation describing how to manipulate the control. Formatting the histogram control was not so difficult, because it is basically a two-dimensional bar chart. But in formatting the scatter plot, we had a lot of problems trying to get the individual points to appear without a line.

The second was in constructing the correct Transact-SQL statement to feed the chart control. For the histogram charts, we had to find a way to return a query that aggregates the states of the data with respect to the states of the predictable attribute.

Visualizing varchar Columns

When working with non-numeric or `varchar` columns, we have only one charting option open to us—a histogram. Because we only have to deal with a single type of chart when working with `varchar` data, and also because the `varchar` histogram is the simplest histogram to create (the data buckets are already defined), we'll start our visual exploration of the data here.

Selecting a `varchar` column on the **Charts** tab creates a SELECT statement to cull the `varchar` columns and group them by bucket. The Data Mining Tool then displays these results as a histogram on the screen.

We are looking for information that will help us decide whether to use specific columns in our data mining model. So let's grab one of the `varchar` columns and see how it looks plotted in a histogram.

▶ **To create a varchar histogram of the cup98LRN_cleantable**

1. In the Data Mining Tool, click **Explore**.
2. On the **Chart** tab, select the following options:
   - Select **Histogram**.
   - For **Predicted**, type **target_b**.
   - For **VarChar Column**, select **MAILCODE**.

   Figure 6.1 (on the next page) shows the histogram we just created.

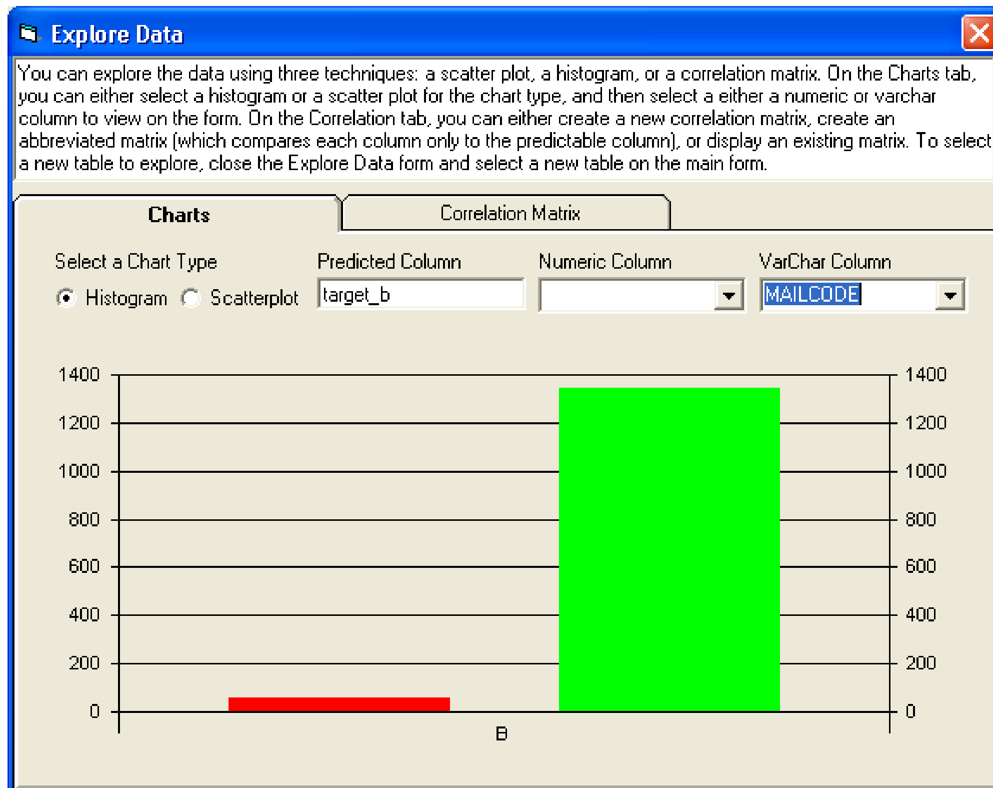


Figure 6.1 A `varchar` histogram

As we can see from the chart, the values for the column MAILCODE make up only a small percentage of the values in the table, and those that are present closely match the distributions found over the whole table. For this reason, we can exclude the column from the dataset. Let's drop this column now, using the procedure outlined in Chapter 4, "Cleaning the Data."

Now let's look at the code in the Data Mining Tool that creates the histogram.

Looking at the Code Behind Creating the varchar Histogram

A `varchar` histogram is generated through a single subroutine—`cbVarCharPlot_Click`. To put this subroutine in context with the other code on the **Charts** tab, open Microsoft® Visual Basic ® and look at this subroutine while we walk through the code.

▶ To view the cbVarCharPlot_Click subroutine

1. Browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmExplore** (**frmExplore.frm**), click **View Code**, and then locate the `cbVarCharPlot_Click` subroutine.

Selecting Varchar Data

When a user clicks an item in the **varchar Column** list, the `cbVarCharPlot_Click` subroutine is invoked. For the `varchar` data, we don't need to worry about separating values into buckets because a distinct number of buckets already exist. We do, however, need to differentiate between the states of the predictable column for each bucket. In our case, the predictable column contains two states, "Yes" if the person contributed last time (signified by a 1 in the column) and "No" if the person did not contribute (signified by a 0 in the column). For each bucket, we need to figure out how many positive ("Yes") and negative ("No") results are in the predictable column.

To do this, we sum the cases that are positive and those that are negative. After determining these counts, we can make a count for each bucket using a GROUP BY statement and order the information using an ORDER BY statement.

Here is the rather complicated Transact-SQL SELECT statement that results from all these calculations.

```
...
'The SQL statement that selects data from the table that is
appropriate for a histogram.
    strSQLSelect = "SELECT " & strHistInputVar & ", " &
                    "SUM(CASE WHEN " & strPredicted & " =1 THEN 1
ELSE 0 END)AS Match, " & _
                    "SUM(CASE WHEN " & strPredicted & " =0 THEN 1
ELSE 0 END) AS NoMatch FROM " & _
                    "[" & frmMain.strCleanedTable & "] GROUP BY " &
strHistInputVar & ""
Set rsData = mdlProperties.cnDataPrep.Execute(strSQLSelect)
...
```

Displaying the varchar Histogram

Now it is just a matter of formatting the chart control to display a two-dimensional bar chart (that is, a histogram) and setting its `datasource` property equal to the recordset holding the results of the SELECT statement.

```
...
'Initialize the chart control to display the data properly.
With mscExplore
    .Visible = True
    .chartType = VtChChartType2dBar
    .Plot.UniformAxis = False
End With

'Set the datasource property of the chart control to the recordset
returned by the Transact-SQL statement,
'and then refresh the chart.
Set mscExplore.DataSource = rsData
Me.Refresh
...
```

As you can see, creating and displaying a `varchar` histogram is relatively easy. Having done all the charting that we can with `varchar` columns, let's go back to the Data Mining Tool and create and display a numeric histogram.

Visualizing Numeric Columns with a Histogram

Now let's use a histogram to investigate the distribution of data in some of the numeric columns. If you remember from Chapter 4, we looked at the properties of several columns whose names begin with ADATE. From the properties we found that these columns contain only one or two distinct states. Will these states help us create a better mining model? Let's find out.

▶ **To create a numeric histogram of the Cup98LRN_clean data**

1. In the Data Mining Tool, click **Explore**.
2. On the **Charts** tab, select the following options:
   - Select **Histogram**.
   - For **Predicted**, type **target_b**.
   - For **Numeric Column**, select **ADATE_2**.

   Figure 6.2 shows the numeric histogram we just created.

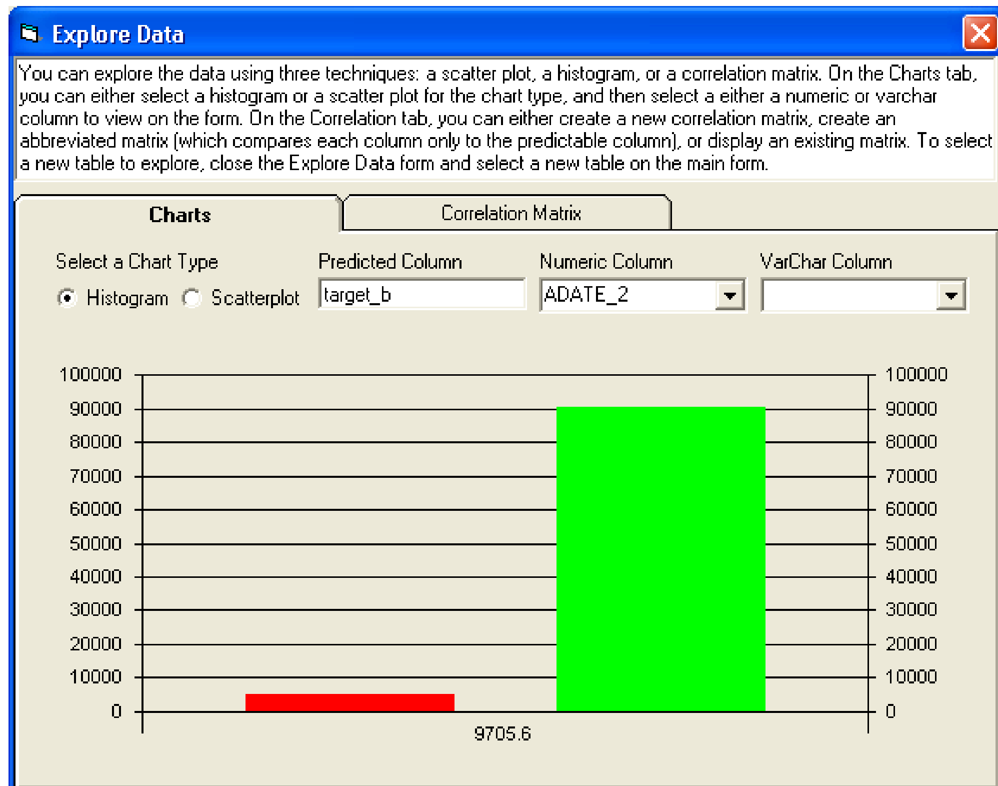


Figure 6.2 Numeric histogram

From the properties table, we found that the column ADATE_2 contains only two distinct states. We can see in Figure 6.2 that the ratio of positive to negative responses in the predictable column is the same for the column as for the entire population. If the proportions do not differ from the entire population, they do not add anything to the effectiveness of the model. Accordingly, we can remove ADATE_2 from the cup98LRN_clean table by using the procedure outlined in Chapter 4, "Cleaning the Data."

Looking at the Code Behind the Numeric Histogram

Working with numeric columns introduces a couple of issues that we didn't have to deal with when working only with `varchar` columns. The first issue is that that we can create two charts—histograms and scatter plots—from numeric columns. When a user just selects a numeric column on the **Charts** tab, it does not give us enough information to know which chart the user wants to generate. To deal with this issue, all we have to do is declare a Boolean variable that is set to True when the user selects a histogram and False otherwise. That takes care of the first issue.

The second issue has to do with the states represented by the numeric column. Numeric data does not exist in discrete states. Numeric data spans a nebulous range, and we need to define states for this data. Thus, when creating a numeric histogram, we need to determine the entire range of the data (subtract the minimum value from the maximum value) and divide that range into a user-defined number of states or buckets.

With those two issues out of the way, the code used to create a numeric histogram is remarkably similar to the code used to create a `varchar` histogram. Let's take this opportunity to see exactly how this code works.

As with charting `varchar` columns, a numeric histogram is generated through a single subroutine—`cbNumericPlot_Click`. Open Visual Basic and follow along as we walk through the code.

▶ To view the cbNumericPlot_Click subroutine

1. Browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmExplore** (**frmExplore.frm**), click **View Code**, and then locate the `cbNumericPlot_Click` subroutine.

Defining States for the Numeric Data

As with a `varchar` histogram, when a user clicks an item in the **Numeric Column** list, a subroutine is invoked. In this case, that subroutine is `cbNumericPlot_Click`. But unlike the subroutine used to create a `varchar` histogram, the `cbNumericPlot_Click` subroutine must first create a Boolean variable (`blnChartTypeHist`) that is set to True if the user selects a histogram and False if the user selects a scatter plot. Remember that we need to do this because numeric data can be plotted in more than one way.

The subroutine then uses the `blnChartTypeHist` variable in an If-Then-Else statement to determine which type of chart to create using the numeric data. When the value of this variable is True, the routine enters the Else section, which creates a numeric histogram.

In this section, the routine must first retrieve the maximum and minimum values for the column. (We'll use these values to define the various states for the data.) To get these values, the routine uses the `Select_Properties` function of the `mdlProperties` module and then places the values into the `sngCol_Max` and `sngCol_Min` variables.

```
...
Else
'If blnChartTypeHist is True, create a histogram.
```

```
'Get the min and max values to be used to define the states for
'grouping the numerical data.
    sngCol Max = mdlProperties.Select Properties(strPropertiesTable,
strNumericInput, Max)
    sngCol_Min = mdlProperties.Select_Properties(strPropertiesTable,
strNumericInput, Min)
...
```

With the maximum and minimum values retrieved, we can now define the size for each state. The routine determines the entire range of values (which is stored in `sngDivision`) by subtracting the minimum value from the maximum value and then dividing this range by the value (the user-defined number of buckets) stored in the `NOD` variable.

```
...
'Define the size of each state.
    sngDivision = (sngCol_Max - sngCol_Min) / NOD
    sngAdd = 0
...
```

So far, so good. Now comes the task of using these states to define a SELECT statement that gathers the information for each state.

Building a SELECT Statement

Using the range stored in `sngDivision`, the routine calculates an upper and lower bound for each bucket, and stores these values in two arrays. Because the number of divisions is a variable that the user sets, the routine never knows how many divisions to expect when constructing the Transact-SQL statement. So, to make the program as versatile as possible, the routine also constructs the case statement of the Transact-SQL SELECT (`strSQLSubSelect`) within a loop. The `strSQLSubSelect` statement created within this loop basically divides the data based on where it fits within a range.

```
...
'For each division create an upper and lower bound and store them in
the
'upper(i) and lower(i) arrays. Also, build the case portion of the
SQL statement.

    For i = 1 To NOD
```

```
         Lower(i) = sngCol Min + sngAdd
         Upper(i) = Lower(i) + sngDivision
         sngAdd = sngDivision * i
         strSQLSubSelect = strSQLSubSelect & "when " &
strNumericInput & " between " & Lower(i) & " and " & Upper(i) & "
then '" & Lower(i) & "'"
    Next i
...
```

The routine then finishes the `strSQLSubSelect` statement with the following code.

```
...
'Finish the sub select of the SQL statement.
    strSQLSubSelect = strSQLSubSelect & " end As
AggregatedName,target b from " & frmMain.strCleanedTable & ""
...
```

Having created the `strSQLSubSelect` statement, we can now add this statement to the final SELECT statement (`strSQLSelect`). As you might recall from the SELECT statement used to create the `varchar` histogram, this final SELECT statement makes a count for each bucket using a GROUP BY statement and then orders this information using an ORDER BY statement.

```
...
'Construct the final Transact-SQL statement using the sub select.
    strSQLSelect = "select AggregatedName, sum(case when target b=1
then 1 else 0 end)" & _
                   "as Match, sum(case when target_b=0 then 1 else
0 end) as NoMatch " & _
                   "from (" & strSQLSubSelect & ") as a group by
AggregatedName order by AggregatedName"
...
```

All that's left to do now is to run this statement and display the results on the screen.

Displaying the Numeric Histogram

Not too surprisingly, we display the numeric histogram the same way we display a `varchar` histogram. Here is the code to do so.

```
    'Initialize the chart control to display the histogram.
    With mscExplore
        .Visible = True
        .chartType = VtChChartType2dBar
        .Plot.UniformAxis = False
    End With
End If

'Set the chart datasource of the chart control equal to the data
extracted from the table.
Set mscExplore.DataSource = rsData
mscExplore.Refresh
```

Visualizing Numeric Columns with a Scatter Plot

A scatter plot tells us how the states of the predictable column are distributed across the selected column. By looking at this distribution, we can get a better idea of which columns to include when we build the data mining models. If the states of the predictable column are evenly scattered across the selected column, we know that the selected column does not tell us anything significant about the predictable column. But if the states of the predictable column are grouped together, the selected column probably can be useful in determining the outcome of the predictable column.

▶ To create a scatter plot of the Cup98LRN_clean data

1. In the Data Mining Tool, click **Explore**.
2. On the **Charts** tab, do the following:
   - Select **Scatterplot**.
   - For **Predicted**, type **target_d**.
   - For **Numeric Column**, select **HC19**.

   Figure 6.3 shows the scatter plot we just created.

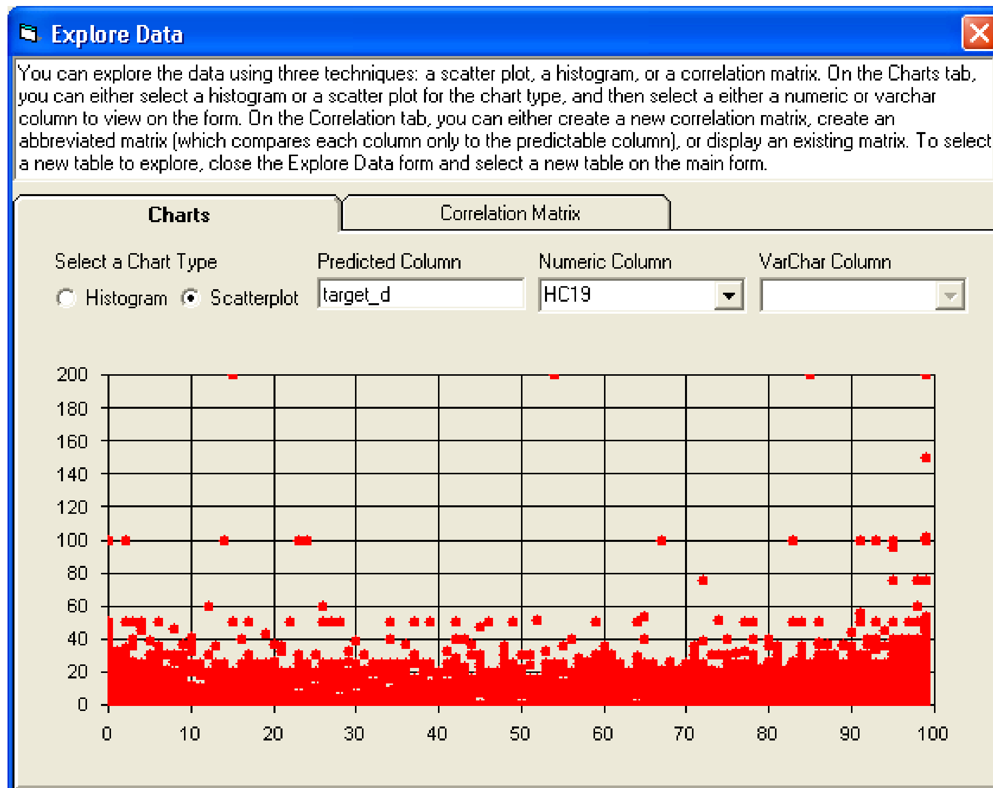


Figure 6.3 Scatter plot

In this procedure we selected TARGET_D, which describes how much money was donated, as the predictable attribute. If we used TARGET_B, which indicates whether customers donated, the chart would contain only two straight lines, while TARGET_D shows more of the grouping pattern. We can see from Figure 6.3 that the data is distributed across the states of the selected column fairly evenly, reducing the column's effectiveness in a data mining model. Accordingly, we'll drop the column HC19 by using the procedure outlined in Chapter 4, "Cleaning the Data."

Looking at the Code Behind a Scatter Plot

Luckily, when it comes to scatter plots of numeric data, we don't have to worry about dividing the data into discrete buckets. Therefore, scatter plots are much easier to code.

We only have to retrieve the values for the selected numeric column and the predictable column, and then chart the results on a scatter plot. Of course, formatting this chart takes a bit more work than the two-dimensional charts we've been using until now.

The code used to create a scatter plot is located in the If-Then portion of the If-Then-Else statement used in the `cbNumericPlot_Click` subroutine. For information about how to view this code, see "Looking at the Code Behind a Numeric Histogram" earlier in this chapter.

Compared to the SELECT statements used to create histograms, the SELECT statement for a scatter plot is much easier to construct. The `cbNumericPlot_Click` subroutine just creates a SELECT statement that returns both the selected numeric column and the predictable column.

```
...
'If the user chooses to create a scatter plot from numeric data,
'the Boolean variable, blnChartHist, is False.
If blnChartTypeHist = False Then


'Get the data from the table to create the scatter plot.
    Set rsData = mdlProperties.cnDataPrep.Execute("select " &
strNumericInput & " as input, " & strPredicted & " as predicted FROM
[" & frmMain.strCleanedTable & "] ")
...
```

It's more of a problem to format the chart control. To display the data points without a line, the `Showline` property must be set to **False** and the `Style`, `Size`, and `Visible` properties of the `Marker` must be set as shown in the following code.

```
...
'Initialize the chart control to display the data in a scatter plot.
    With mscExplore
        .Visible = True
        .chartType = VtChChartType2dXY
        .Plot.UniformAxis = False
        .Plot.SeriesCollection(1).ShowLine = False
        .Plot.SeriesCollection(1).SeriesMarker.Auto = False
        .Plot.SeriesCollection(1).DataPoints(-1).Marker.Style =
VtMarkerStyleFilledCircle
        .Plot.SeriesCollection(1).DataPoints(-1).Marker.Size = 80
        .Plot.SeriesCollection(1).DataPoints(-1).Marker.Visible =
True
...
```

Like the histogram code we've already seen, all we have to do after selecting the data and formatting the chart is to display it using the following familiar code.

```
...
Set mscExplore.DataSource = rsData
mscExplore.Refresh
...
```

Numerically Exploring Data with a Correlation Matrix

While graphical explorations give us a good feel for the different columns in the table, it is also nice to have a more concrete way of seeing how the columns interact. Using a correlation matrix, we can investigate the relationships between the columns, and, more important, the relationship between each column and the predictable column.

A correlation describes how one column changes in comparison to another column. For example, an increase in a gas tax correlates with an increase in the overall cost of gasoline. The value of a calculated correlation lies between –1 and 1, depending on the direction of the correlation. If an increase in the values in one column corresponds to an increase in the values in the second column, the correlation is positive. If the increase in the values in the first column corresponds to a decrease in the values in the second column, the correlation is negative. As the calculated value of correlation approaches –1 or 1, the correlation between the attributes becomes stronger. A perfect correlation returns a value of 1 or –1.

To build a correlation matrix, we'll construct a table that has a row and column for each column in the source table. We'll then fill in the calculated values for the correlations so that we have a value for each column against each other column. A pattern should appear where a line of ones down the middle corresponds to each column being compared to itself. The values on either side are mirror images of each other because the same columns are being compared, just in a different order.

To calculate correlation, we use the following formula:

$$r = \frac{1}{n-1}\sum_i \left(\frac{x_i - \bar{x}}{\sigma_x}\right)\left(\frac{y_i - \bar{y}}{\sigma_y}\right)$$

The problem with creating a correlation matrix is that it can be extremely resource-intensive, especially with a dataset as large as ours. If we work with all 317 numeric columns, we'll have to work through that equation 90,000 times! This can be done, but it takes a long time. As an alternative, the Data Mining Tool gives us the choice of either

calculating the entire matrix, or just one row: each column versus the predictable column. Because we already know which column the predictable column is, we are really most interested in how each other column relates to it, because this will help us choose which columns to include in the final data mining model. Because there is value in looking at how all columns relate to one another, the Data Mining Tool also provides this option.

Now that we understand what a correlation is and how to calculate it, let's see what kind of correlations we find using the Data Mining Tool.

▶ To create an abbreviated correlation matrix of the Cup98LRN_clean data

1. In the Data Mining Tool, click **Explore**.
2. On the **Correlation Matrix** tab, type **Small_Matrix** to name the matrix.

Note Make sure that the **Calculate the entire matrix** check box is cleared.

3. Click **Calculate**.

   The abbreviated matrix is calculated and displayed on the **Correlation Matrix** tab, as shown in Figure 6.4.

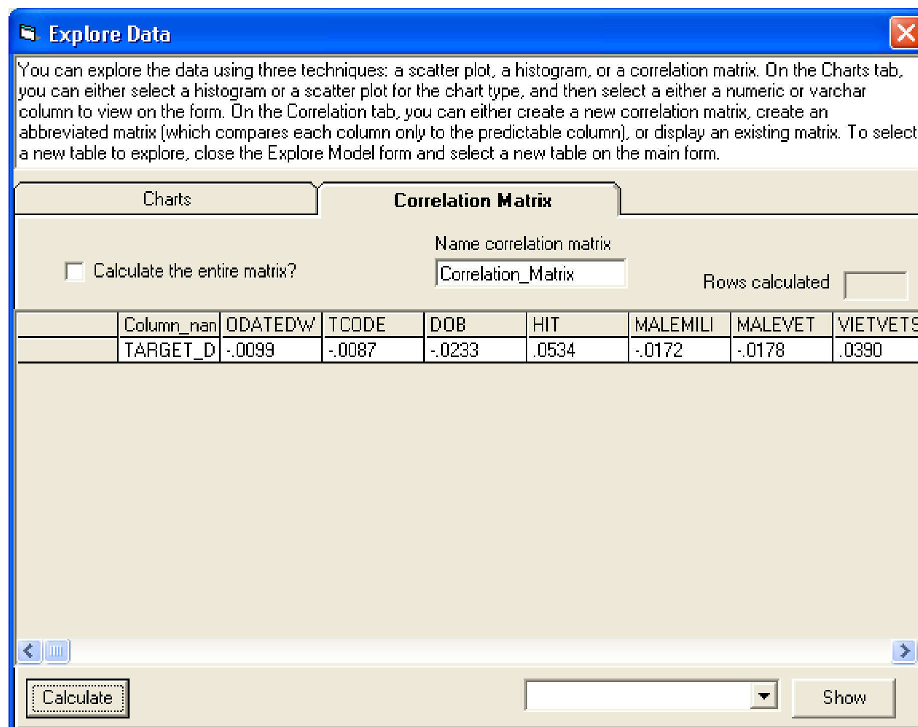


Figure 6.4 Correlation matrix

In the correlation matrix, we can see which columns correlate most closely to the predictable column, and which do not seem to be affected by the change in values of the predictable column. You may notice that some of the values do not fall within the expected range of –1 to 1, or a column compared to itself does not return a value of 1. This is most likely explained by the fact that not all columns contain data that is distributed normally, which is an assumption of the correlation calculation.

Looking through the matrix in Figure 6.4, we can disregard columns such as:

- LFC5 (-0.0006)
- OEDC1 (-0.0036)
- EC7 (0.001)

In fact, all of the ADATE columns have a very low correlation, which is related to the low number of states that they contain. This matrix should be a good guide in choosing which numeric columns to include in the model-building process. Note that each time we run this routine, slightly different values are returned. This is expected because the routine is using a sampled version of the original table to calculate the correlations.

If you want to look at the entire matrix, you can go ahead and calculate it now, but be prepared to wait a while! By calculating the entire matrix, you will be able to see how columns are not only related to the predictable column, but also how they are related to the other columns in the table.

Looking at the Code Behind the Correlation Matrix Tab

The code for the correlation matrix is in the `cmdCalculateCorrelation_Click` subroutine. Open Visual Basic and follow along as we walk through the code.

▶ To view the cmdCalculateCorrelation_Click subroutine

1. Browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmExplore (frmExplore.frm)**, click **View Code**, and then locate the `cmdCalculateCorrelation_Click` subroutine.

Setting Up for Two Loops

Our goal is to translate the correlation equation into workable code. That means we must identify the various parts of the equation and determine how we can either obtain that information or generate it.

Looking at the equation, we see that we need a record count for the original table. That's pretty easy to calculate. We also see that we'll need the mean and standard deviation for each column. That too is easy to calculate. We also see that we need some way to sum the product of the values of every column against every other column. That is a bit more complicated and requires embedding a loop within a loop. The outer loop selects a column from the data, while the inner loop calculates the correlation value for that column against all other columns.

The code used in the `cmdCalculateCorrelation_Click` subroutine tackles all these tasks, but not exactly in the order listed. The subroutine first creates two recordsets, `rsColumns1` and `rsColumns2`, with identical schema information about each column in the table. We do this by first creating a recordset, as usual, and then using the `Clone` function to copy all of the information from the first recordset into the second. Because we are working only with numeric columns, we set a filter on the recordset so that only numeric columns are visible.

```
...
    Set rsColumns1 =
mdlProperties.cnDataPrep.OpenSchema(adSchemaColumns, Array(Empty,
Empty, "[" & frmMain.strCleanedTable & "]"))
    rsColumns1.Filter = "[data_type] = 131"

    Set rsColumns2 = rsColumns1.Clone
    rsColumns2.Filter = "[data_type] = 131"
...
```

As we'll see later in the code, these twin recordsets are the key to making the embedded-loop approach work. They enable us to select a column from the `rsColumns1` recordset in the outer loop, and then, within the inner loop, use the columns in the `rsColumn2` recordset to calculate the associated correlation values.

Creating a Table for Correlation Values

Although we now have the recordsets we need to compare every column against every other column, we don't have a place to store the correlation results. Thus, we need to create a table to hold this matrix of calculated correlations. This table requires identical numeric column names in both the first row and the first column.

To create the table, we need a string holding all of the column names, which can be incorporated into a Transact-SQL SELECT statement. Because we are already filtering the recordset by data type, we simply have to loop through the recordset, adding the column

names to the strColNames string. At the end of the routine, the column names form a comma-separated list, with a trailing comma at the end.

```
...
Do Until rsColumns1.EOF
    strColNames = strColNames & rsColumns1!COLUMN_NAME & ", "
    lngColumnCount = lngColumnCount + 1
    rsColumns1.MoveNext
Loop
...
```

Then, to clean up the strColNames string for use in a Transact-SQL statement, we need to remove the final comma in the string. Also, because the loop to create the strColNames string cycles through to the end of the recordset, we'll need to return to the beginning of the recordset before starting another loop.

```
...
If Right(strColNames, 2) = ", " Then strColNames = Left(strColNames,
Len(strColNames) - 2)

rsColumns.MoveFirst
...
```

With the final set of columns selected, the next step is to create the correlation matrix table, setting aside the first column to hold the column names and then using the strColNames string to populate the rest of the table.

```
strCreateTable = "CREATE TABLE [" & strCorrelationTable & "]" & _
                "(Column_name varchar(20), " & Replace(strColNames,
", ", " numeric(5,4), ") & _
                " numeric(5,4)" & " )"
mdlProperties.cnDataPrep.Execute (strCreateTable)
```

Getting the Raw Data and Its Averages and Standard Deviations

As mentioned earlier, there are more than 300 numeric columns in the table. Counterbalance this with the fact that there are more than 90,000 rows, and you can see that we are working with a huge amount of data. Because calculating the correlations is heavily resource intensive, this could take a very long time! One way to solve this problem is to sample the original table down to 1,000 rows, and then use this new table as the source for the correlation matrix. Although we are working with less data, the results will

still be accurate enough to be able to compare columns based on their correlations. Because we have sampled the original table, we also need to create new values for the mean and standard deviations, which will be used in the correlation matrix calculations.

In order to sample the table, we call the same sampling routine that is used to split the table and create an over-sampled table. For a description of this subroutine, see Chapter 7, “Splitting the Data.”

```
mdlSample.Create_Table "Correlation_Sample",
frmMain.strCleanedTable, strSQLWhere, N...
```

Okay, we have the sampled table of raw data, the recordsets that make the embedded loops possible, and someplace to store the correlation values. We’re almost ready to start the number crunching. But note the “almost.” We are still missing the averages and standard deviations required by our correlation equation for the new table we just created. We know the last necessary value, the record count, because it is just the size of the new table.

We have to recalculate the mean and standard deviations because these values have changed from the original table. If we do not update these values, our correlation calculations will be inaccurate. Because we only use these values locally in this subroutine, we can hold them in a local array. To calculate the values, we call the same `Calculate_Properties` routine we used to calculate the properties in Chapter 4. The properties are then stored in a local array, so that they can be easily accessed later in the routine.

```
mdlProperties.Calculate Properties "Correlation Sample",
"strPropertiesTable"
Set rsProperties = cnDataPrep.Execute("SELECT * FROM
correlation_sample_pr")


'Insert the values from the properties table into a local array
arrPropertiesTable = rsProperties.GetRows()
```

Calculating the Correlations

Before we calculate the correlations, we need a place to store them. Because we previously created a table in the SQL Server database, we now only have to open it in a recordset and disconnect the recordset. Remember that at this point the table does not hold any data, so the table will hold only the column names with no rows. Accordingly, we will add a new row to the recordset each time the code iterates through the outer loop, populating the recordset with values.

The following code opens the new recordset, and then disconnect it.

```
With rsTable
     .ActiveConnection = cnDataPrep
     .CursorLocation = adUseClient
     .LockType = adLockBatchOptimistic
     .CursorType = adOpenStatic
     .Source = "SELECT * FROM " & strCorrelationTable & ""
     .Open
     Set .ActiveConnection = Nothing
End With
```

Originally we updated the table in the database each time the code iterated through the loop, but with all of the calculations we are doing, this requires a lot of I/O. Performance-wise, it is much better to perform all of the calculations, filling the disconnected recordset with values, and then update the table all at one time.

Now, because we have the option of creating either the a full matrix or just an abbreviated version, the calculation section of the routine is split between the two options using the Boolean variable `blnTotalMatrix`, and an If-Then-Else statement. The Boolean variable is set in the click event of the `chkTotalMatrix` check box.

If the check box is selected, `blnTotalMatrix` is set to True, and a full matrix is calculated.

Calculating a Full Matrix

Now it's time to start crunching some numbers!

First we have to get the data from which we will calculate the correlations and store it in the `rsData` recordset.

```
strSQLSelect = "SELECT " & strColNames & " FROM Correlation Sample"
Set rsData = cnDataPrep.Execute(strSQLSelect)
```

The routine begins to cycle through the numeric columns in the table, using the `rsColumns` recordset. Each time the code iterates through the outer loop, a new row is added to the `rsTable` recordset and the first column, COLUMN_NAME, is populated with the name of the selected column.

```
Do Until rsColumns1.EOF
        rsTable.AddNew
        rsTable("Column_name") = rsColumns1!COLUMN_NAME
```

To calculate the correlation, we need the mean and standard deviation for the selected column, which we get from the `arrPropertiesTable` array and store in local variables. We first have to check to make sure the calculated values for the mean and standard deviation are not null; otherwise, an error will be thrown later in the routine. The numbers 3 and 4 signify the location in the properties table for the mean and standard deviation.

```
        sngXAvg = IIf(IsNull(arrPropertiesTable(3, lngColumn1)),
sngXAvg, arrPropertiesTable(3, lngColumn1))
        sngXDev = IIf(IsNull(arrPropertiesTable(4, lngColumn1)), 0,
arrPropertiesTable(4, lngColumn1))
```

Notice the `lngColumn1` variable. This defines the position of both the mean and standard deviation values, and the column ID in the `rsData` recordset, which holds the raw data that will be used for the calculations. It is incremented each time the code cycles through the `rsColumns1` recordset, for each column. There is an equivalent variable, `lngColumn2`, for the `rsColumns2` recordset.

After getting the properties for the column in the `rsColumns1`, the code begins to cycle through the columns in the second recordset, `rsColumns2`. In this way, the code calculates the correlation between the selected column in `rsColumns1` and every column in `rsColumns2`. The code also gets the mean and standard deviation from the `arrPropertiesTable` array for the selected column in `rsColumns2`, and stores them in a local variable.

```
        Do Until rsColumns2.EOF
                sngYAvg = IIf(IsNull(arrPropertiesTable(3,
lngColumn2)), sngYAvg, arrPropertiesTable(3, lngColumn2))
                sngYDev = IIf(IsNull(arrPropertiesTable(4,
lngColumn2)), 0, arrPropertiesTable(4, lngColumn2))
```

Now we have everything we need to calculate the correlations. But before we do so, we need to make sure that the standard deviation for both columns is not equal to zero. If it is, we will be dividing by zero in the calculations, which causes an error. If both deviations are not zero, the code begins the loop to calculate the correlations. Because calculating the correlations involves a summation, we create a loop that cycles through each row in the recordset, `rsData`, grabbing the necessary values and adding them to the values already calculated in the `sngCorrelation` variable.

```
                If sngXDev = 0 Or sngYDev = 0 Then
                    sngCorrelation = 0
                Else
                    Do Until rsData.EOF
```

```
                      sngXValue = IIf(IsNull(rsData(lngColumn1)),
sngXAvg, rsData(lngColumn1))
                      sngYValue = IIf(IsNull(rsData(lngColumn2)),
sngYAvg, rsData(lngColumn2))
                      sngCorrelation = sngCorrelation + ((sngXValue
- sngXAvg) / sngXDev) * ((sngYValue - sngYAvg) / sngYDev)
                       rsData.MoveNext
                  Loop
              End If
```

Within the loop we check to see if the selected value is null. If the value is null, it is replaced with the average value, effectively negating its contribution to the correlation calculation. If we didn't do this, an error would be raised, because a mathematical calculation cannot be performed on a null value.

Outside the loop we then insert the correlation value into the disconnected `rsTable` recordset. Notice that we increment the variable `lngColumns2` by 1, because this column already holds the column name for the row of the matrix. We also move to the next row in the `rsColumns` recordset, return to the first row in the `rsData` recordset, and increment the `lngColumn2` variable, which specifies the column that we are working with in both the `arrPropertiesTable` array and the `rsData` recordset.

```
                rsData.MoveFirst
                sngCorrelation = sngCorrelation / (N - 1)

                rsTable((lngColumn2 + 1)) = round(sngCorrelation,4)

                lngColumn2 = lngColumn2 + 1
                sngCorrelation = 0
                rsColumns2.MoveNext
                rsData.MoveFirst
            Loop
```

Outside of the inner loop, we move to the first row in the `rsColumns2` recordset, set the `lngColumn2` variable to zero, increment the `lngColumn1` variable by 1, move to the next row in the `rsColumns1` recordset, and move to the next row in the `rsTable` recordset to be filled with correlation calculations.

```
        lngColumn2 = 0
        lngColumn1 = lngColumn1 + 1

        rsColumns2.MoveFirst
        rsColumns1.MoveNext
        rsTable.MoveNext
```

The code also displays the number of rows inserted into the correlation matrix on the form to keep the user updated.

```
    txtRowCount = CVar(lngCount)
    txtRowCount.Refresh
Loop
```

This continues until the matrix is filled with all of the correlation values.

Now let's look at how we create an abbreviated matrix. If the `chkTotalMatrix` check box is cleared, the value of the `blnTotalMatrix` variable is False, and an abbreviated matrix is calculated.

Calculating an Abbreviated Matrix

If a user is working with a very large table and doesn't want to wait for the complete matrix to be filled, we create an abbreviated matrix displaying the correlation of each column against only the predictable column, TARGET_B.

There is not much difference between these two versions of the matrix. Instead of using two embedded loops, we only need to re-create the inner loop from the previous section, which compares each column to the predictable column. In order to know where the predictable column lies in the `arrPropertiesTable` array and `rsData` recordset, we create a new variable, `lngPredict`, that is set in the loop that creates a string of columns that will be in the table. Within the loop, we look for the column with the same name as the predictable column, and then store the corresponding ID.

```
If rsColumns1!COLUMN_NAME = PREDICT Then lngPredict = lngCount
```

Using the column position, we can then get the mean and standard deviation for the predictable column and store them in local variables.

```
sngYAvg = arrPropertiesTable(3, lngPredict)
sngYDev = arrPropertiesTable(4, lngPredict)
```

We then add the only new row to the `rsTable` recordset, because we are only calculating the correlation values against the single column.

```
rsTable.AddNew
rsTable("Column_name") = PREDICT
```

The way in which we calculate and store the correlation values is exactly the same as the inner loop in the previous method; we just do it once instead of cycling through all of the columns a second time.

```
Do Until rsColumns1.EOF
    sngXAvg = IIf(IsNull(arrPropertiesTable(3, lngColumn1)),
sngXAvg, arrPropertiesTable(3, lngColumn1))
    sngXDev = IIf(IsNull(arrPropertiesTable(4, lngColumn1)), 0,
arrPropertiesTable(4, lngColumn1))
    If sngXDev = 0 Or sngYDev = 0 Then
        sngCorrelation = 0
    Else
        Do Until rsData.EOF
            sngXValue = IIf(IsNull(rsData(lngColumn1)), sngXAvg,
rsData(lngColumn1))
            sngYValue = IIf(IsNull(rsData(lngPredict)), sngYAvg,
rsData(lngPredict))            sngCorrelation = sngCorrelation +
((sngXValue - sngXAvg) / sngXDev) * ((sngYValue - sngYAvg) /
sngYDev)
            rsData.MoveNext
        Loop

        sngCorrelation = sngCorrelation / (N - 1)
        rsTable((lngColumn1 + 1)) = Round(sngCorrelation, 4)
rsTable((lngColumn1 + 1)) = sngCorrelation
        rsData.MoveFirst
    End If
    sngCorrelation = 0
    lngColumn1 = lngColumn1 + 1
    rsColumns1.MoveNext
Loop
```

We now have the correlations for each column against the predictable column. Using this will help us to choose which columns to include in the final mining model, because we only want those columns that correlate most closely with the predictable column.

After completing the calculations, using a FlexGrid control, we display the data stored in the correlation table on the form.

```
strSQLSelect = "SELECT * FROM [" & strCorrelationMatrix & "]"
Set rsData = mdlProperties.cnDataPrep.Execute(strSQLSelect)
Set hfgCorrelation.DataSource = rsData
```

7

Splitting the Data

Okay. We've cleaned, transformed, and explored the data. Now what? Build data mining models? Not quite yet—there are still a couple of things we need to do with the original table.

First, we need to create a new table to hold the columns that we will use to create the models. In the previous two chapters we cleaned and explored the data. We now have to take what we learned and use it in deciding which columns to use in the model. To save time, the following 20 columns have been preselected:

AVGGIFT	GENDER	NUMPROM
POP901_2	HOMEOWNR	PETS
CARDGIFT	LASTGIFT	ODATEDW2
CARDPM12	MAJOR	RAMNTALL
CARDPROM	MAXRAMNT	TARGET_B
CONTROLN	MINRAMNT	VETERANS
DOB	NGIFTALL	

Through your explorations, you may decide on different columns, or you may choose a subset of these columns. Feel free to experiment and see how your models differ from the ones we build in this solution. To create a new table, we will use the functionality in the **Manage Tables** form that allows us to select specific columns and copy them into a new table. To create the new table, select the columns listed here from the cup98LRN_clean table, and insert them into a new table called cup98LRN_Select.

Second, if you remember from Chapter 2, modeling and predicting depend on having both a training (model-building) and testing (validation) dataset. With the training table we apply a data mining algorithm to learn the hidden patterns in the data to accomplish a given objective. Using the testing table, we find out how well the trained model performs in its ability to predict whether someone will donate money. Because you often have only a single table or dataset with which to work (as in our scenario), at some point you have to artificially create the training and testing tables from the original dataset. We have now reached this point.

The goal when splitting a table is to separate that table into two unique tables that each accurately represents the original table. To achieve this objective, we need to make sure that:

- Both tables have the same structure—the same columns must exist in both datasets and the columns must have the same names.
- The rows in the training table do not also exist in the testing table—rows are unique to each table.
- A sufficient number of rows exist in each table to allow us to build a good model, yet still have enough data left to perform a good validation. For this solution, we'll split 60 percent of the data into the training table and the remaining 40 percent into the testing table.
- The distribution of data in the input and predictable columns is approximately the same for both tables.

So how will we split the tables without introducing a bias? If we just choose the top 60 percent of the rows for one table and the bottom 40 percent of the rows for the other, we can never be sure that each table holds a true representation of the original data. Do we know that the data was randomly distributed throughout the sample? What if the first 1,000 rows only contain information about people from Washington (WA), while the last rows contain information about people from Alaska (AK)? What if the rows were inserted alphabetically or by the date of their creation? In any of these cases, just siphoning off the first chunk of data into a training table excludes vital information that exists in later rows, and, therefore, can create an inaccurate model. If we then tried to use the remaining data as the testing table to test the model, the information excluded from the training table could throw off the testing results.

The third task we will need to perform in this chapter is to create an over-sampled version of the training table. An over-sampled table artificially increases the state of the column that we want to predict. The positive result of the predictable column is fairly underrepresented in all of our tables, so we might be able to make a better model by creating an over-sampled version of the training table. The best way to determine how much of a difference this makes is to create a model based on both versions of the table, and investigate the predictive differences between the two, which is what we will do in the model-building section.

In this chapter, we use the Data Mining Tool to split the table. Then we'll look behind the scenes at the code in the Data Mining Tool that makes this work.

Note For an explanation of the **Manage Tables** form and for more information about over-sampling, see Appendix, "Managing Tables."

Trying Out Table Splitting

To split the original table correctly, there are two critical tasks that we need to do—track the rows, so that we can ensure the uniqueness of each table, and select rows randomly for each table. As you'll soon see in detail, we'll find some pretty good ways for Microsoft® SQL Server™ to accomplish each of these tasks. We first have SQL Server create a new table that we'll use to track which rows we use in the training and testing tables. We then start to populate those tables using a cool little Transact-SQL statement that randomly selects rows from the original table.

Saving the more detailed discussion of these tasks for later, let's try the code out and split the original table in two.

▶ **To split a cleaned table into two separate tables**

1. In the Data Mining Tool, click **Split**.
2. For **Original Table**, type **cup98LRN_select**.
3. In the **Table used to create the models** section, type the following options:
   - For **Table name**, type **cup98LRN_Model**.
   - For # **Rows**, type **57,000**. This is the number of records to be included in the training table.
4. In the **Table used to validate the models** section, enter the following options:
   - For **Table name**, type **cup98LRN_Test**.
   - For # **Rows**, type **38,000**. This is the number of records to be included in the testing table.
5. Click **Sample**.

At this point, the Data Mining Tool creates the two new tables based on the information we just entered. Then the tool calculates the percentage of positive and negative responses in the predictable column and displays these results on the form.

The percentages shown allow us to see how well the new tables represent the data in the original table. As Figure 7.1 shows, the percentages in the new tables are very close to those in the original table and should work fine.

Split

To split the data into two tables, select a source table, type names for the two new tables, and select the number of rows to include in each new table.

Predicted attribute distribution

Original table		% Positive	% Negative
cup98LRN_clean		5.075934	94.92406

Table used to create the models

Table name	# Rows	% Positive	% Negative
cup98LRN_model	57000	5.0263157	94.97369

Table used to validate the models

Table name	# Rows	% Positive	% Negative
cup98LRN_test	38000	5.15	94.85

Split

Figure 7.1 The calculated percentages from splitting the table

After you have created the training table, create an over-sampled version of the training table, forcing the table to contain 80 percent negative values and 20 percent positive values in the predictable attribute. We use this in the model-building stage.

▶ To create an over-sampled table

1. In the Data Mining Tool, click **Manage Tables**.
2. Click **Sample**.
3. On the **Small Table** form, do the following:
   - Select **Create an over-sample of a table**.
   - For **% Positive**, type **20**.
   - For **Original table**, type **cup98LRN_Model**.
   - For **New table**, type **cup98LRN_ModelOS**.
4. Click **Sample**.

Let's now look at the code that splits the table.

Looking at the Code Used to Split the Table

So we know that we can't just take a chunk of data from the original table and move it to the new table, but what are we going to do? We can use the idea of random sampling to randomly select rows from the source table and insert them into the training table. Between Microsoft Visual Basic® and SQL Server, we have all the tools we need to do this. Visual Basic provides a way to define how we want to create the tables. SQL Server provides a way to create a query that randomly compiles rows from a destination table and inserts them into a new table.

In this section, we'll first look at the code used to create, maintain, and calculate percentages for the various tables used within the splitting process. All of this code resides within the `cmdSample_Click` subroutine. The way in which we populate these tables with data is discussed later in this chapter. To see this code in context, open Visual Basic and walk through the code as we talk about it.

▶ **To view the cmdSample_Click subroutine**

1. Browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmSplit** (**frmSplit.frm**), click **View Code**, and then locate the `cmdSample_Click` subroutine.

Guaranteeing Uniqueness

To create two tables, each originating from the same source table, we need to ensure that we don't insert the same record into both tables. That is, we need to guarantee the uniqueness of records between the tables. If we don't do this, some of the data used to create the model could also be used to test it, and that is not very scientific.

To perform this random sampling, we need:

- A unique identifier for each row in the source table.
- A pool of data from the source table from which to draw the rows.
- A way to keep track of which columns have already been used in each table.
- A way to randomly select columns from the source table.

To guarantee this uniqueness, we need to store information, such as the row ID (`controln`), for each table that is created by splitting the original table. For example, when randomly selecting a row for the validation table, we need to ensure that the pool of data being sampled does not include rows already inserted into the training table.

To keep track of the data used by the training and testing tables, we create another table, Userids. This new table contains two columns:

- The USER_ID column contains the `controln` value for each row randomly selected from the original table.
- The USED column contains a value of 1 if we use the corresponding `controln` value to creating either the training or testing table.

The second column is necessary when we are trying to create multiple tables from the same source data.

To explain how we use the Userids table, let's look at what happens to the table as we progress through the splitting routine. Upon entering the routine, the table is created, and is therefore empty. As we sample rows for the training table, the Userids table is slowly populated with the `controln` values corresponding to the selected rows. When it comes time to actually create the training table, a join is created between the Userids table and the source table, populating the training table with data. When we create the testing table, we do not want to use columns that already have been used by the training table, so we need to save the previously used `controln` values. But at the same time, when we perform the join we want to include only the new IDs that have been added to the Userids table for the testing table. For this reason we populate the USED column of the table with a value for each `controln` used in the training table. When the Userids table is populated with `controln` values for the testing table, we can still join the two tables to create the testing table by ignoring those IDs that have a corresponding value of 1 in the USED column. In this way, the USED column guarantees uniqueness between the two tables.

Here is the Transact-SQL statement used to create the Userids table.

```
strSQLCreate = "IF EXISTS(SELECT TABLE_NAME FROM
INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME = 'UserIds') DROP TABLE
UserIds CREATE TABLE UserIds (User_ID Numeric NULL," & _
                    "Used VARCHAR NULL)"
mdlProperties.cnDataPrep.Execute (strSQLCreate)
```

As you can see, the statement first checks to see if the table exists, and if so, drops and re-creates it. We are able to drop an existing Userids table because we only have to worry about uniqueness when creating the training and testing tables. If the Userids table already exists, a set of training and testing tables exist and already contain unique rows. The user can either use these existing training and testing tables or create new ones, in which case we will be populating a fresh Userids table anyway.

Removing Existing Training and Testing Tables

With the Userids table all set to manage uniqueness between tables, we now check to see whether either the training or testing table exists in the database. If so, we give the user the option of either replacing the contents of the existing tables or returning to the form to enter new table names and create new tables. We will check to see if tables exist throughout this project, so we made this into a separate function that does all of the work.

Because we check for the existence of more than one table, we cycle through control array `txtSampleTable(j)` using the index, `j`.

```
For j = 0 To (lngTableCount - 1)
    If Not mdlProperties.Check Table Exist(txtSampleTable(j).Text)
Then
        GoTo Exit cmdSample Click
    End If
Next j
```

Let's look at how that `Check_Table_Exist` function works.

Walking Through the Check_Table_Exist Function

This function, returns either a value of False (meaning that the table exists and we should exit the routine) or a value of True (meaning that the table does not exist or has been dropped and we can continue). The first step is to initialize the function to False.

```
Check_Table_Exist = False
```

The function then calls another function, `Check_Table_Created`, which searches the database for the table name. If it returns a value of True, the `Check_Table_Exist` function displays a message box asking if the user would like to drop the table or exit the routine.

```
    If Check_Table_Created(strTable) = True Then
        If MsgBox("" & strTable & " table already exists in the
database. Would you like to drop the table and recreate it?." _
                , vbYesNo) = vbYes Then
            strSQLDrop = "Drop Table [" & strTable & "]"
            cnDataPrep.Execute (strSQLDrop)
            Check Table Exist = True
        End If
```

```
    Else
        Check Table Exist = True
    End If
```

If the table does not exist or if the user opts to drop the table, the function is set to True; otherwise, it remains False. The calling routine then uses the state of the function to determine whether to exit the routine or not.

Calculating Percentages in the Original Table

Because we need to measure how well the new tables represent the original table, we calculate the percentage of "yes" (1) and "no" (0) values in the TARGET_B column (the predictable column) of the original table, and display these values on the form. (Later on, we'll calculate these same values for each of the new tables.)

To get the percentage, we first find the number of records in the original table as well as a count of the records in the PREDICTED column (set to TARGET_B) that have a positive response of 1. We do this by using a recordset and a SELECT statement.

```
    'Get the original percentages of yes and no responses in the
table.
    strSQLSelect = "SELECT COUNT(controln) AS count1," & _
                    "(SELECT COUNT(" & PREDICTED & ") FROM [" &
strTable & "] " & _
                    "WHERE " & PREDICTED & " = 1) AS count2 FROM ["
& strTable & "] "

    Set rsRecordCount =
mdlProperties.cnDataPrep.Execute(strSQLSelect)
    lngRecordCount = rsRecordCount!Count1
```

At this point, we have both the total number of rows in the original table and the number of rows with a positive response in the PREDICTED column. All we have to do is divide the number of rows with a positive response by the total number of rows, multiply by 100, and get the percentage of "yes" responses in the original database. For the percentage of "no" responses, we just subtract the percentage of "yes" responses from 100. Here's the code that does all of that.

```
    sngPercentYes = (rsRecordCount!Count2 / lngRecordCount) * 100
    sngPercentNo = 100 - sngPercentYes    txtTableYes.Text =
CVar(sngPercentYes)
    txtTableNo.Text = CVar(sngPercentNo)
```

Pretty simple, really. It starts to get a bit more complicated (but not too much) when we have to calculate these percentages for both the training and testing tables.

Calling the Sampling Routine

Because we use the sampling routine in several places, it is more efficient to create a function that randomly samples a selected table, which can then be called as necessary. For this reason, the code is split into two sections, one that manages the tables and calls the sampling function and one that describes the sampling function being called.

All right, it's time to actually start creating the sampled tables!

To do this, we set up a loop that calls the `Create_Table` function (found in the `mdlSamples` module) for both the training table and testing table. The `Create_Table` function allows us to randomly select a row for insertion into the table currently being created. We'll discuss this function in detail in a bit. For now, all we need to be aware of is the need to pass the following variables to the `Create_Table` function:

- `txtSampleTable(j).text`—the name of the table we are creating
- `strTable`—the name of the original table
- `strSQLWhere`—the WHERE clause of the Transact-SQL statement that returns the pool of available data
- `txtSampleNumber(j).text`—The number of rows to include in the new table

The reason for all of these should be obvious, except for the WHERE clause. As you will soon see when we describe the sampling function, a Transact-SQL SELECT statement is built that returns the pool of available data. The needs of the pooled data change for each sampled table that is created—in one case we may want to create a sampled table based on the entire original table, while in other cases we may want to exclude rows based on the outcome of a specific column. Regardless, we need a mechanism for choosing which data to include, and this is provided by the WHERE clause. Using the WHERE clause, we specify constraints that we want to impose on how data is chosen from the original table. If it is left empty, the routine assumes that no constraints exist and uses the entire source table as the pool of data.

In this case, the WHERE clause is empty because no additional constraints are required.

```
    strSQLWhere = ""
```

So let's start the loop that creates a sampled table for both the training and testing tables. The first step in the loop calls the `Create_Table` function, passing in the appropriate values.

```
For j = 0 To (lngTableCount - 1)
    mdlSample.Create_Table txtSampleTable(j).Text, strTable,
strSQLWhere, CLng(txtSampleNumber(j).Text)
```

Just as in the original table, we now need to calculate and display the percentage of positive and negative values in the PREDICTED column to check how each new table compares to the original table.

```
    strSQLSelect = "SELECT COUNT(" & PREDICTED & ") FROM [" &
txtSampleTable(j).Text & "] WHERE " & PREDICTED & " = 1"
    Set rsData = mdlProperties.cnDataPrep.Execute(strSQLSelect)

    txtYes(j) = (rsData(0) / CVar(txtSampleNumber(j).Text)) * 100
    txtNo(j) = 100 – CSNG(txtYes(j).text)
Next j
```

And that's it for the `cmdSample_Click` subroutine. Now let's look at how the `Create_Table` sampling function called by this subroutine works.

Random Sampling

The sampling function is called for three different purposes throughout this solution:

- To create a smaller version of a table—one with fewer rows
- To create an over-sampled table
- To split a table into two smaller tables

If you remember from Chapter 1, we had the opportunity to create a sampled version of table—one with fewer rows. We will also use the over-sampling functionality later on in the model-building phase. As far as we are concerned now, the routine is being used to split the source table in two.

Recall that the function takes four parameters—the new table name, the source table name, the Transact-SQL WHERE statement that defines the pool of available data, and the number of sampled rows to insert into the new table. So how do we create the new table?

▶ To view the Create_Table public function

1. Browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Modules**.
3. Right-click **mdlSample** (**mdlSample.bas**), and then click **View Code**.

There are actually a couple of ways to sample the table, one more typical and the other new and exciting. First let's talk about the more common way.

The concept is simple—we need to randomly select a row from a source table and insert it into a new table. The typical way to do this, either using a stored procedure or in Visual Basic, is to use a random-number generator to generate a column key identifier, grab the associated row, insert the row into a new table, and then store the key column identifier to ensure that the row is not selected again. Going row by row, you can see that this would turn into a very time-consuming process. We first created a procedure to do this in Visual Basic using a disconnected recordset. As the project progressed, we wanted to move more of the resource-intensive calculations to the server, and we decided to change this into a stored procedure. In researching the stored procedure, we found a new way to create a random table using just a few lines of Transact-SQL. Now be careful when you use this, because it only works with SQL Server 2000.

```
SELECT TOP 1000 controln user id
INTO test1
FROM  dbo.cup98LRN
ORDER BY NEWID()
```

With just four lines of code we have created a new, randomly sampled table. The way this works is pretty cool. The `NEWID()` function creates a unique, random identifier for each row in the table. By using an ORDER BY clause, those random numbers become ordered, which effectively randomizes the order of the rows in the table. By taking the top *n* number of rows and inserting them into a new table, we have effectively created a new, random table with *n* number of rows. This little bit of Transact-SQL code effectively takes a pretty complicated routine and simplifies it down to creating a Transact-SQL statement to send to the server.

So the first thing we do is create the following statement.

```
strSQLUserID = "INSERT INTO Userids SELECT TOP " & lngSampleNumber &
" controln " &
                "FROM  [" & strTable & "] WHERE " & strSQLWhere & "
not exists " & _
```

```
                "(SELECT * FROM Userids WHERE user id = [" &
strTable & "].controln) " & _
                "ORDER BY NEWID() "
```

Notice that we are only grabbing the random `controln` values for the rows and inserting them into the Userids table. Later we will create a join between the Userids table and the source table to create the final sampled table. Notice that the USED column in the Userids table remains null. This signifies that `controln` has not yet been used to create a new table in this routine.

So now we have a statement that creates a table holding two columns, a column holding the randomly selected `controln` values and a column holding an identifier signifying whether the row has already been used somewhere within the calling routine. And we have the source data from the original table. We now only have to create a Transact-SQL statement that joins the two tables together on the row ID where the USED column in the Userids table is null. This returns all of the rows from the source table that were randomly selected in the statement we used earlier. Here is the Transact-SQL statement that joins the two tables.

```
strSQLSelect = "SELECT * INTO [" & strNewTable & "] " & _
                "FROM [" & strTable & "] AS OT JOIN UserIds AS " & _
                "UI ON OT.controln = UI.User_Id " & _
                "WHERE ui.used IS NULL ALTER TABLE [" & strNewTable
& "] " &
                "DROP COLUMN user id, used " &
                "UPDATE userids SET used = 1 WHERE used IS NULL"
```

Notice that the last line of the statement updates the Userid table so that all of the rows we just inserted into the new table are marked as used. It does this by updating to 1 the values in the USED column that were previously null.

We then combine the two statements into one statement to reduce trips to the server, and execute the statement.

```
strSQLFull = strSQLUserID & strSQLSelect
cnDataPrep.Execute (strSQLFull)
```

A sampled table has now been created from the source data containing as many rows as specified by the user.

8

Building and Validating the Models

We've finally made it! After all the work we've done cleaning and organizing—making sense out of that mountainous collection of data, it's time to build some data mining models. For all of its importance, this is probably the easiest task in the project to complete. We only need to select the columns we want to include in the model, give them appropriate parameters, and process the new model.

We will build two models, each using different attributes and different algorithm parameters, which means that we will also need to compare how effective the models are in predicting which customers will donate money. It would not be good to put a model into production without testing its predictive ability. We will do this using a lift chart, which was briefly explained in Chapter 3, "Defining the Problem."

Now let's build some models!

Building the Models

Splitting the data was an important step because it allows us to create the models and test them using data derived from the same source. We will now take advantage of this, using the training table we created in Chapter 5, to build the models.

Our goal for the project is to build a mining model that will allow us to fulfill the objective outlined in Chapter 3, "Defining the Problem." It is important to keep in mind that we want to build a model that predicts the values in the TARGET_B column—whether someone will donate money in response to a mailing. This in turn will reduce mailing costs and save the boss money. Accordingly, in this step we will build two models: one from a regular dataset and one from an over-sampled dataset.

Through all of the cleaning and exploring tasks, we have been able to eliminate many of the original columns in the base table. Of the remaining columns, we need to choose those that will most likely aid us in getting the information that we want to predict. While building a model using all 317 remaining columns is possible, it would be resource intensive to process and difficult to understand. It is better to choose columns that we can justify as being interesting to include in the model. For example, a mailing code and full address is redundant information; for our purposes, just the mailing code is sufficient.

Each column that we include in the model can be parameterized in several ways, which can have a profound impact on how the model is created. Let's look a little closer at how we can parameterize a column.

Column Parameters

When we add a column to the model, we need to decide what type of a column it is. Will it be used to identify records (a key column) or as the final column we want to predict? Are its values continuous or discrete? How is the data distributed within the column?

Here are the properties we can use to define a column:

- `Data type`
- `Usage`
- `Related to`
- `Distribution`
- `Content type`
- `Modeling flags`

For this solution, we will only work with the `Data type`, `Usage`, and `Content type` column properties. The `Data type` property describes the kind of data that is in the column, either numeric (single), or for this solution, `varchar`. The `Usage` property signifies whether the column is an input column, a predictable column, or both. The `Content type` property describes data in the columns, which in this solution means they are either continuous (numeric columns) or discrete (`varchar` columns). Also, remember that the predictable column must be discrete in order for us to be able to build the model—if its type is anything other than discrete, an error is raised. For more information about these properties and the ones that we did not use, see SQL Server Books Online.

Now let's see how the algorithm can be parameterized.

Model Parameters

We will build the models using the Microsoft Decision Trees algorithm, which has two adjustable parameters: COMPLEXITY_PENALTY and MINIMUM_LEAF_CASES.

The COMPLEXITY_PENALTY parameter inhibits the growth of the decision tree. A low value decreases the likelihood of a split, while a high value increases the likelihood. The default value is based on the number of columns for a given model:

- For 1 to 9 columns, the value is 0.5.
- For 10 to 99 columns, the value is 0.9.
- For 100 or more columns, the value is 0.99.

Increasing the complexity penalty moves the model from being more general to more detailed. A higher complexity penalty slows the growth of the tree, making it harder for the algorithm to generate more branches. The complexity penalty raises the bar on whether a split should occur at a certain point.

The MINIMUM_LEAF_CASES parameter determines the minimum number of leaf cases required to generate a split in the decision tree. The default number of cases is 10. This means that a split cannot be generated based on a single value—if only one person in the dataset is from Alaska, the algorithm cannot use this as a reason to create a split.

For more information about the algorithms and their parameters, see the *SQL Server 2000 Resource Kit*, SQL Server Books Online, and *Preparing and Mining Data with Microsoft SQL Server 2000 and Analysis Services*. There are also several good third-party books about data mining that give a more in-depth look into using decision trees.

Trying Out the Model Building Task

As we look at the **Create Mining Model** form, we can see that it is divided into three sections. In the first section, we'll select an Analysis server, a data source, and a model name. Here we are defining the type of model we're building and where the data is coming from to build it. We don't actually populate the model with data until we add columns to the model and process it. The following steps describe how to build the models.

▶ **To select an Analysis Server and create an empty model**

1. In the Data Mining Tool, click **Model**.
2. In the **OLAP server** text box, type *Localhost*.
3. In the **Database** text box, type **DM_ OLAP**.
4. For **Data source name**, type **cup98LRN**.
5. For **Model name**, type **DM_Tree**.
6. Select **Decision tree**, and then click **Create**.

   An empty decision tree shell named DM_Tree has been created on the Analysis server and added to the tree control on the form. We can now begin to populate it with columns and process it.

Notice that for these models we did not change any model parameters using the **Model parameters** text box. When you build your own models, use this text box to change model parameters and see how they affect the models. Enter the parameter and its new value as you would in Analysis Manager. For example, you can change the complexity penalty by typing COMPLEXITY_PENALTY = 0.6 in the text box.

For more information about changing these parameters, see SQL Server Books Online.

In the second section of the **Create Mining Model** form, we'll select the table that holds the columns we're adding to the model, and then we'll select a column. When we set the data source in the first section, we were only directing the model to the appropriate database; now we have to choose a table from that database—the cup98LRN_Model table. We'll then define how the column is used by the mining model, and we'll add it to the model.

▶ **To select a table and column from the data source and parameterize the column**

1. In **Select an origin table**, select cup98LRN_Model.
2. In **Select a column**, select **AVGGIFT**.
3. Parameterize the column:
   - For **Data type**, select **single**.
   - For **Usage**, select **input**.
   - For **Content type**, select **continuous**.
4. Click ->.

   The column is now added to the tree control, under the model name DM_Tree. Repeat this procedure, adding each column in the following table, and parameterize the columns as described.

   Note When parameterizing the CONTROLN column, you must also select the **is case key** checkbox because this column is the key column.

Column name	Data type	Usage	Content type
AVGGIFT	Single	Input	Continuous
POP901_2	Single	Input	Continuous
CARDGIFT	Single	Input	Continuous
CARDPM12	Single	Input	Continuous
CARDPROM	Single	Input	Continuous
CONTROLN	Single		
DOB	Single	Input	Continuous
GENDER	Char	Input	Discrete
HOMEOWNR	Char	Input	Discrete
LASTGIFT	Single	Input	Continuous
MAJOR	Char	Input	Discrete
MAXRAMNT	Single	Input	Continuous
MINRAMNT	Single	Input	Continuous
NGIFTALL	Single	Input	Continuous
NUMPROM	Single	Input	Continuous
PETS	Char	Input	Discrete
ODATEDW2	Single	Input	Continuous
RAMNTALL	Single	Input	Continuous
TARGET_B	Single	Input and predictable	Discrete
VETERANS	Char	Input	Discrete

In the first step we created the shell for the model, but by clicking **Process** we are actually passing the data through the shell and creating the relationships that define the model.

▶ **To process the model**

- Click **Process** to train the mining model.

Now that the model is processed, we have a working model that we can browse through and base predictions on. After we have browsed through this first model, we will repeat the process, using the over-sampled table as the data source. We can then test the effectiveness of the models using the validation dataset we created in Chapter 7, "Splitting the Data," and the lift chart.

Now we'll use the same procedures to make another model, which is based on the over-sampled table we created in the Chapter 7, "Splitting the Data." To make the model, we will use the same Analysis server, the same data source, but a different source table (cup98LRN_Model_OS) and model name (DM_Tree_OS). Select and parameterize the same columns that are displayed in the table earlier in this section.

Later on, we'll use graphical representations to compare these models. But before we do that, let's take a look behind the scenes at the code that makes all of this work.

Looking at the Model-Building Code

Let's look at the tasks that various subroutines accomplish during the model-building step. Open Visual Basic and follow along as we walk through the code.

▶ **To view the form frmCreateModel.frm**

1. In Windows Explorer, browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmCreateModel** (**frmCreateModel.frm**), and then click **View Code**.

The code is broken up into four main areas:

- Creating a connection
- Building the model shell
- Adding columns to the model shell
- Browsing the model

Let's first look at what we have to declare before we get heavily into the code.

The mapTree2Column collection stores the columns that are added to the model. This allows us to see a visual representation of the columns that we have added to the model.

```
Private mapTree2Column As Collection
```

Because we are now working against an Analysis server instead of SQL Server, we open a new connection to the Analysis server using the DSO.Server object. The DSO.Server object is the only Decision Support Objects (DSO) object that can be instantiated as new.

```
Private srv As New DSO.Server
```

The next variables deal with defining the new mining model.

The database and other objects cannot be accessed directly, but instead are referenced through the MDStore object. For example, a new database must be created through the MDStore object.

```
Private mds As DSO.MDStore
```

We also need to define a new mining model and mining model data source. If you remember, we set up a new Analysis server database and data source in Chapter 1, “Setup.” This new dsmm object references that data source.

```
Private mm As DSO.MiningModel
Private root As Node, parentNode As Node
Private dsmm As DSO.DataSource
```

The last two variables are used in creating the DSO connection strings. Because different providers can user different characters to distinguish the open and closed quote delimiters, we need to get them from the dsmm data source and store them for later use.

```
Private LQuote As String, RQuote As String
```

Create the Connection

The first thing we have to do is create a connection to the server. Remember that we are working with two different servers: the SQL Sever that is holding and organizing our raw data, and the Analysis server, where the models are built and administered. So, before we create the models, we need to connect to the previously created Analysis Services database.

We first connect to the server. Because Analysis Services uses Windows Authentication for security, we only need to provide the server name.

```
srv.Connect txtServer.Text
```

Next we set the `mds` variable equal to the data source. Remember that we cannot reference the database directly. Instead, we have to reference it through the `MDStores` object.

```
Set mds = srv.MDStores(txtDatabase.Text)
```

After setting the proper connections, we give the user access to the controls that he or she can use to create the model.

```
txtDataSource.Enabled = True
txtModelName.Enabled = True

optC.Enabled = True
optDT.Enabled = True

btnCreateModel.Enabled = True
```

The user can now begin defining and adding columns to the model and create the model.

Defining the Model

We will now define the type of model to build (clustering or decision tree) and add it to the Analysis Services database. Remember that we are actually just creating the shell for the model. We still need to define the content by adding columns to the model. Even then the model will not be ready to use until it has been processed.

The first step in creating the model is to check to see whether it already exists in the database. If the model already exists, we remove it; otherwise, we create it.

```
If Not (mds.MiningModels(txtModelName) Is Nothing) Then
    mds.MiningModels.Remove txtModelName
End If
Set mm = mds.MiningModels.AddNew(txtModelName, sbclsRelational)
```

Next, we create a data source for the new model that is equal to the one that was created when the Analysis server was set up.

```
mm.DataSources.AddNew txtDataSource
Set dsmm = mm.DataSources(txtDataSource)
```

```
    LQuote = dsmm.OpenQuoteChar
    RQuote = dsmm.CloseQuoteChar
```

The last step in modifying the new mining model is to set the model type. Analysis Services includes algorithms for both a decision tree model and a clustering model.

```
    If (optDT.Value = True) Then
        mm.MiningAlgorithm = "Microsoft_Decision_Trees"
    End If

    If (optC.Value = True) Then
        mm.MiningAlgorithm = "Microsoft_Clustering"
        mm.Parameters = "CLUSTER_COUNT=" + txtNoClusters
    End If
```

After all of the updates are implemented, we commit the transactions to the database.

```
    mds.CommitTrans
```

We are going to reuse the `cnDataPrep` connection that we used earlier in this application, but we need to make sure that it is set to the same data source that the mining model is using. Accordingly, we first need to check to see whether `cnDataPrep` is an open connection, and if it is, close it.

```
    Dim ds As DSO.DataSource
    If (cnDataPrep.State = adStateOpen) Then
        cnDataPrep.Close
    Else
        Set cnDataPrep = New ADODB.Connection
    End If
```

We set the `cnDataPrep` connection equal to the connection string used by the mining model for its data source.

```
    cnDataPrep.ConnectionString =
mds.DataSources(txtDataSource).ConnectionString
    cnDataPrep.Open
```

We can then populate the cbTable drop-down box on the form with available tables from that data source. This table will be the source of columns that the user will add to the mining model.

```
Fill_Combobox cbTable
```

The last step is to clear the tree viewer control, which displays a graphical representation of any previous model. We want to make sure that the user is viewing the current model.

```
tvModel.Nodes.Clear
Set root = tvModel.Nodes.Add(, , "root_node", mm.Name)
```

Adding Columns to the Model

Now that we have created the empty shell of a mining model, we need to start populating it with columns. Associated with each column are several properties, which must be set before the column is created.

We first declare an object for the new column.

```
Dim mc As DSO.Column
```

To be safe, we then make sure that the data source for the mining model is set correctly.

```
Set dsmm = mm.DataSources(txtDataSource)
LQuote = dsmm.OpenQuoteChar
RQuote = dsmm.CloseQuoteChar
```

Mining models themselves do not allow transactions, so we actually have to reference the parent—or database—to begin the transaction.

```
mm.Parent.BeginTrans
```

We can now add a new column to the mining model, using the same name as the column in the table from the data source.

```
Set mc = mm.Columns.AddNew(txtColumnName)
```

Now we set properties of the column. The column case parameter is very important because it identifies the different rows, or cases, in the table. Each mining model must have a case column.

```
With mc
    .IsKey = chkCase.Value
```

Next, we set the data type.

```
        Select Case cbDataType.Text
            Case "Char"
                .DataType = adChar
            Case "Integer"
                .DataType = adInteger
            Case "Single"
                .DataType = adSingle
        End Select
```

The column type is determined. If the column is not a case column, it can be either input, predictable, or both input and predictable.

```
        If (.IsKey) Then
        Else
            Select Case cbUsage.Text
               Case "Input"
                    .IsInput = True
                    .IsPredictable = False
               Case "Input and Predictable"
                    .IsInput = True
                    .IsPredictable = True
               Case "Predictable"
                    .IsInput = False
                    .IsPredictable = True
            End Select
            .Distribution = cbDistribution.Text
            .ModelingFlags = cbModelingFlags.Text
            .RelatedColumn = cbRelatedTo.Text
            .IsDisabled = False
```

If the column is a predictable column, it must have a column type of discrete. If the selected column is a continuous variable, we use the following code to make it discrete.

```
            If cbContentType = "DISCRETIZED" Then
                .ContentType = cbContentType & "(" &
cbDiscretizationMethod & ", " & txtBuckets & ")"
```

```
            Else
                .ContentType = cbContentType
            End If
        End If
```

Finally, we create the column statement. Here is a good example of using the `LQuote` and `Rquote` variables.

```
        .SourceColumn = .FromClause & "." & LQuote & cbColumns.Text
& RQuote

    End With
```

The mining model is updated, and the transaction is committed to the database.

```
    mm.Update
    mm.Parent.CommitTrans

    Dim currentNode As Node
```

The Angoss viewer control and the tree view control are updated with the new column. For more information about the Angoss tree viewer control, see “Browsing the Models” later in this chapter.

```
    Set currentNode = tvModel.Nodes.Add("root node", tvwChild, ,
mc.Name)

    mapTree2Column.Add currentNode, CStr(currentNode.Index)
    tvModel.Refresh
```

Browsing the Models

A good way to evaluate a model is to look at a visual representation of it. After all, what is easier to understand—a table full of mathematical relationships or a graphic displaying a decision tree with all of its splits and branches?

Although Analysis Services provides a viewer for both the Decision Trees and Clustering algorithms, we will use a different method. Because all of the tasks are pretty much contained within the sample application, we don’t want to go back and forth between the sample application and Analysis Manager building models, then viewing models, then building models, and so on. So what to do—is there a way to display a tree viewer on a

form within the Data Mining Tool? The answer is a resounding yes—which brings us to another point—it is very easy to view and compare the models using third-party viewers. We'll view the models with the Angoss Consumer Controls, which were installed during setup.

▶ To browse a model

- On the Create Mining Model form, click **Browse**.

  The appropriate viewer opens, displaying the newly-created model.

Figure 8.1 shows the decision tree for the model that we created—this is what all of that work was for! Of course, this is just the first model of many that we could make from this dataset. We can see that out of the columns we selected, the following had the most effect on the outcome of the predictable column (in order of importance).

- LASTGIFT
- CARDPROM
- ODATEDW2

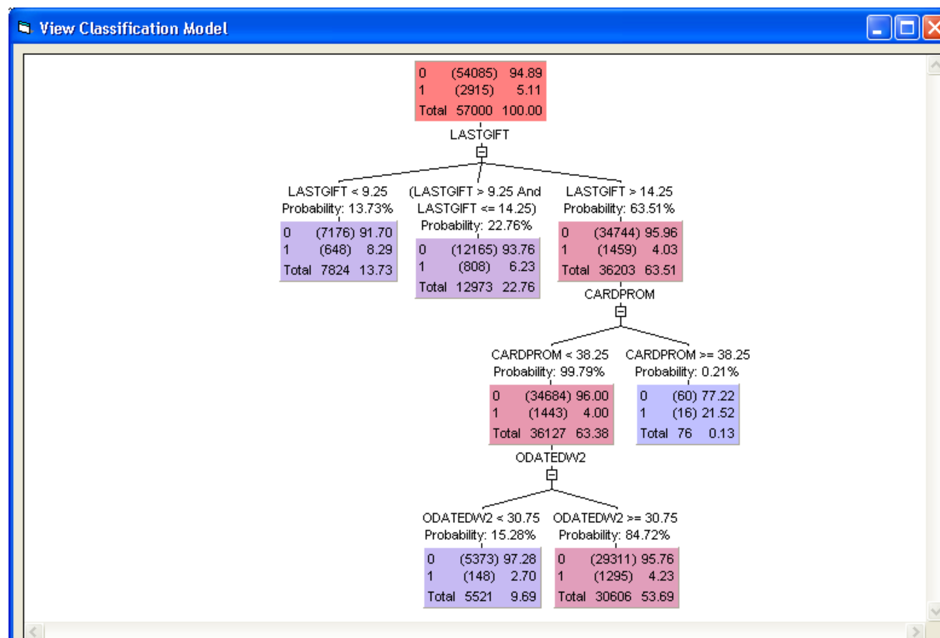


Figure 8.1 Graphical representation of the DM_Tree model

Over the entire population, LASTGFT was the greatest factor in determining where the first split in the data occurs—a place where one state was preferred over the other state as compared to the percentage of states in the entire population.

Remember all of that work we did to transform the data? Well, it actually paid off with ODATEDW2! In testing this application, we built several models that included the ODATEDW column (and not the transformed ODATEDW2 column) and the ODATEDW column had no impact on the model. But as you can see in Figure 8.1, the transformations we performed on ODATEDW caused a split that had not appeared previously. This shows how transforming a single column can have a profound impact on the final model.

Of course, for all of the columns we choose to include in the model, only a few played an important role, which is how it goes in data mining. For all of the assumptions we make about which columns should be the most important, we really never know what we are going to find. The reason for this is that even though empirically, we may come to conclusions about how the data is related, the decision tree finds mathematical relationships, which it uses to create the model. And this is really the reason that we use data mining, to find the relationships that are less obvious. If we only found relationships that we could have deduced simply by perusing the data, what use would we have for data mining?

In looking at the improved percentages in the final splits, we can see that there is not a huge change from the original percentages in the dataset, but remember it doesn't take a huge change to save the company money. For every percentage point of improvement, fewer envelopes have to be mailed out to receive a good response.

Now let's look at the model we created from the over-sampled table (Figure 8.2).

As you can see, by over-sampling the table, we created a model that is only slightly different from the DM_Tree model. The main difference is that ODATEDW2 and CARDPROM switched positions on the tree. Looking at the outcomes, we can also see that the breakdown of "yes" and "no" percentages is a little more drastic as compared to the original 80 percent and 20 percent from the original table. Will this translate into better predictions? We'll find out later when we build some lift charts.

First, let's look at the code behind the browsing functionality.

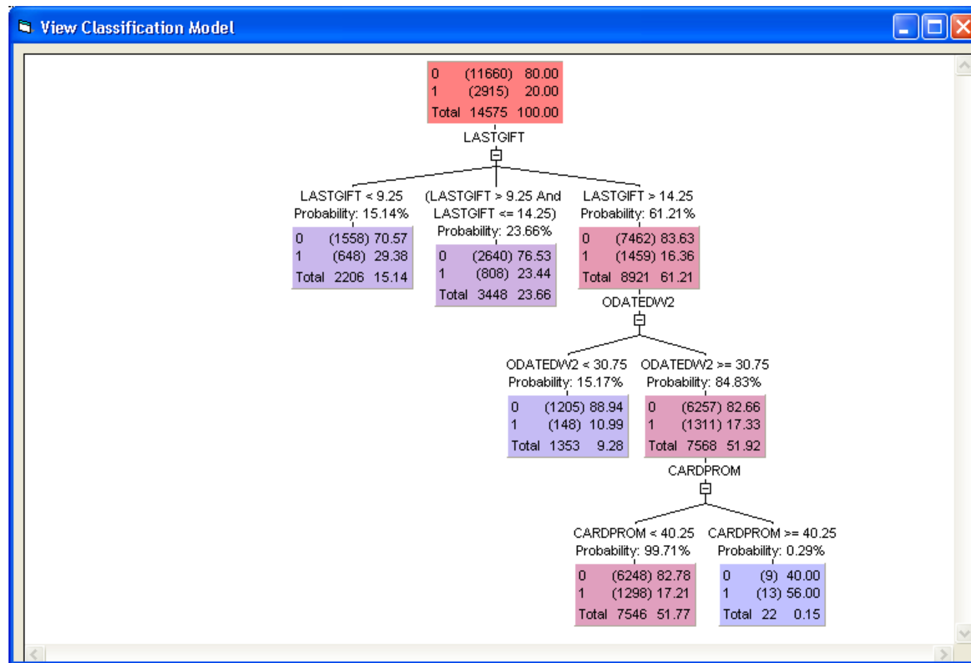


Figure 8.2 Graphical representation of the DM_TREE_OS model

Looking at the Browsing Code

Let's look at the tasks that various subroutines accomplish during the model-browsing step. Open Visual Basic and follow along as we walk through the code.

▶ To view the form frmBrowseModelClustering.frm

1. In Windows Explorer, browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmBrowseModelClustering** (**frmBrowseModelClustering.frm**), and then click **View Code**.

▶ To view the form frmBrowseModelClassification.frm

1. In Windows Explorer, browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmBrowseModelClassification** (**frmBrowseModelClassification.frm**), and then click **View Code**.

A company called Angoss has created three useful Visual Basic controls that work well with Analysis Services and that are available for download in their SDK. The first two controls, a tree view and a cluster viewer, are immediately useful at this stage, while the third, a lift chart viewer, will show up later in this chapter when we compare the different models. The controls are simple to use; viewing a model is as easy as pointing the control to the model and setting a few parameters. Before you can use the controls, you must add the following references to your project:

- ANGOSS Decision Tree Viewer (OLEDB DM)
- ANGOSS Segment Viewer (OLEDB DM)
- ANGOSS LiftChart Control (OLEDB DM)

Remember that these controls only work with existing mining models—the controls give you the capability to browse through models that already exist, not to create new models. For more information about using the controls, see the Help file that is installed with the consumer controls.

To display either type of model, we create two forms—one for decision tree models and one for clustering models.

Let's look at the decision tree form. We first define a new tree control and connection, which is set to the Analysis Services database.

```
    Dim tree As New DecisionTreeViewerLibCtl.DTVTree
    Dim cn As New ADODB.Connection
```

We then set up the tree viewer to display the model.

```
     With Me.DTViewer.NodeDetail
         .NodeDisplayFlags = .NodeDisplayFlags Or
dtvNodeDisplayColorGradient _
                                                 Or
dtvNodeDisplayPlusMinus
```

```
        .InputAttributeDisplayFlags = .InputAttributeDisplayFlags Or
dtvIADisplayProbability
        .PredictAttributeColorGradient.FromColor = &HFFC0C0
        .PredictAttributeColorGradient.ToColor = &H8080FF
    End With
```

Next, we pass the viewer the information it needs to display the control.

```
   cn.Open "Provider=MSOLAP.2;Data Source=" & Server & ";Initial
Catalog=" & Database
   tree.Connection = cn
   tree.ModelName = Model_name
```

Then, to display the viewer, we set the tree control to the model object and refresh it.

```
   DTViewer.tree = tree
   DTViewer.tree.Refresh
```

The control is resized in a separate routine so that it fits nicely within the form.

```
Private Sub Form_Resize()
    DTViewer.Width = Me.Width - (DTViewer.Left + 150)
    DTViewer.Height = Me.Height - (DTViewer.Top + 450)End Sub
```

The clustering form is much easier to write. We only have to create the correct connection string, and pass that information into the viewer, along with the model name.

```
   cn.Open "Provider=MSOLAP;Data Source=" & Server & ";Initial
Catalog=" & Model_name
   SegViewer.InitFromOLEDBDM cn, Model_name
```

And then plot the data.

```
   SegViewer.PlotData
```

And that's it. As with the tree viewer, we used resizing code to allow the user to resize the form without losing information. Now let's look at how we validate the models.

Validating the Models

Now that we've built the models, how do we know if they are any good? Well, in our case, by using a lift chart viewer, which is supplied in the Angoss SDK.

A lift chart calculates the accuracy of the predictions created by a specific mining model. It does this by predicting a column in a set of testing data, and then comparing it to the actual value. Then the predicted value and the actual value are displayed graphically.

A lift chart measures the effectiveness of the model by comparing the results that are achieved with and without the predictive model. For example, if we randomly select 20,000 people to whom we will send requests for donations, we should expect to receive a positive response from about 5 percent of the people (in keeping with the distribution in our current database). But if we use our models to choose which people to send the requests to, we would hopefully see an increase in the response rate—say, to 20 percent. The difference between the two response rates is called lift. By charting the response rate for the different mailings, both the random mailing and the targeted mailing, we can create a lift chart.

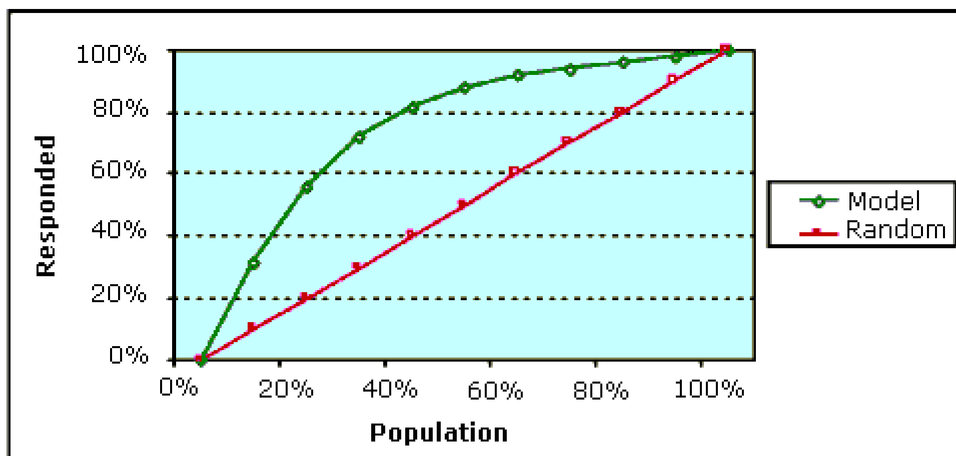


Figure 8.3 Lift chart

Note the characteristic curve. This happens because the total number of donations in the population is always the same. Out of our 90,000 records, we will always have a return rate of around 5 percent, or around 5,000 responses. Our goal is to reduce the number of mailings we send out, but still keep the number of responses returned at 5,000, thus increasing the percentage of mailings returned. Another way of saying this is that we should expect to get 50 percent of the possible positive responses by contacting 50 percent of the possible people randomly. But by using the model, we would hope to increase this to, say,

85 percent of the possible positive responses by contacting 50 percent of the possible people.

To implement the lift chart, we need:

- A trained model
- A testing dataset from which the model can create predictions
- A mapping between the input data and the structure of the mining model

Now let's look at how we use the lift chart.

Trying Out the Validation Task

To use the **Validation** tab, we have to specify a model and the Analysis server on which it's located, a source for the testing data, the predictable column, and the state of the predictable column that we are trying to find. We then construct the prediction query, which creates predictions for the testing data based on the model, and feeds the results into the lift chart.

▶ To create a lift chart

1. In the **Server** text box, type **Localhost**.
2. In the **Database** text box, type **DM_ OLAP**.
3. From the **Models** drop-down menu, select **DM_Tree**.
4. In **Predicted column**, select **target_b**.
5. In **Predicted state**, select **1**.
6. In the **Prediction query** text box, type the prediction query corresponding to the model you want to look at.

   We have included a prediction query for each of the models stored as a text file located in **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder. The file names are Query_Dmtree.txt and Query_Dmtreeos.txt. To speed up the process, you can just copy and paste each query into the text box.
7. Click **Plot liftchart**.

   A new lift chart appears in the lift chart control.

Figure 8.4 shows the lift chart we just created.

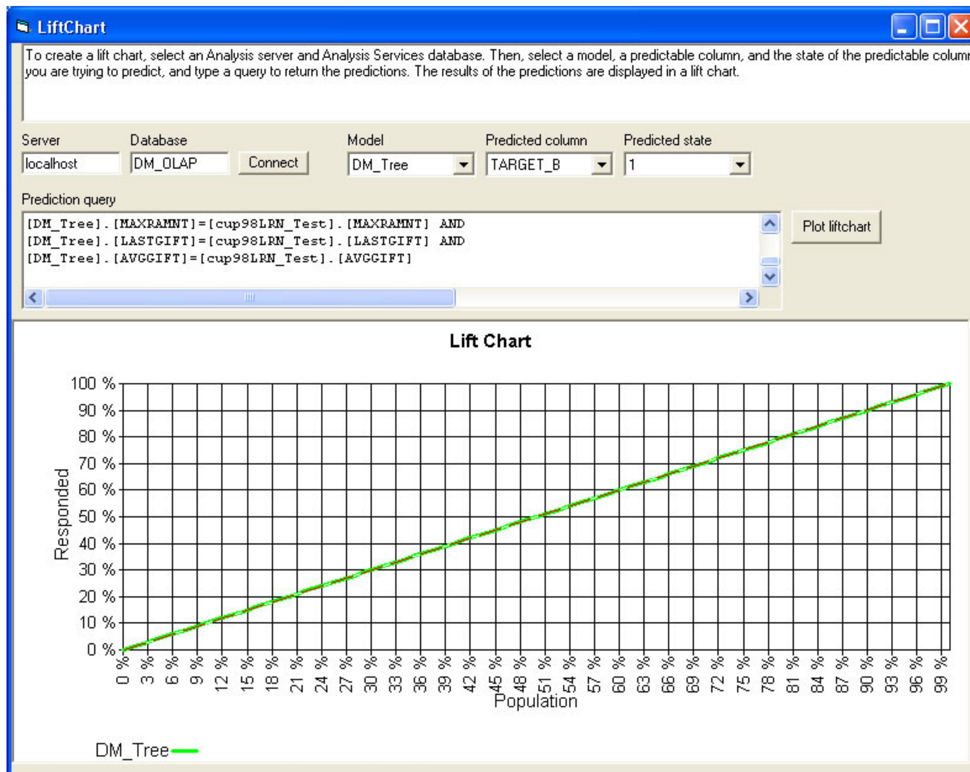


Figure 8.4 Lift chart with the DM_Tree mining model

This lift chart is not very exciting; it's hard to tell if the model's predictions are an improvement over random selection. Let's create another lift chart based on the model we created based on the over-sampled data. To do this, use the same server and the same predictable column and predicted state—just change the model to DM_Tree_OS, and copy and paste the query named `Query_dmtreeos`.

Figure 8.5 shows the lift chart that we just created.

Once again, we do not see much difference. What is this telling us? Are our models not good enough, or is this about what we should expect? Looking at the models in Figure 8.1 and Figure 8.2, we can see that even when the tree splits, the improvement in the number of positive to negative responses in the data is not that great—the probability that someone will not donate is still greater than the probability that someone will donate. This means that the model does not predict whether someone will donate, and therefore, we do not see any improvement in the lift charts.

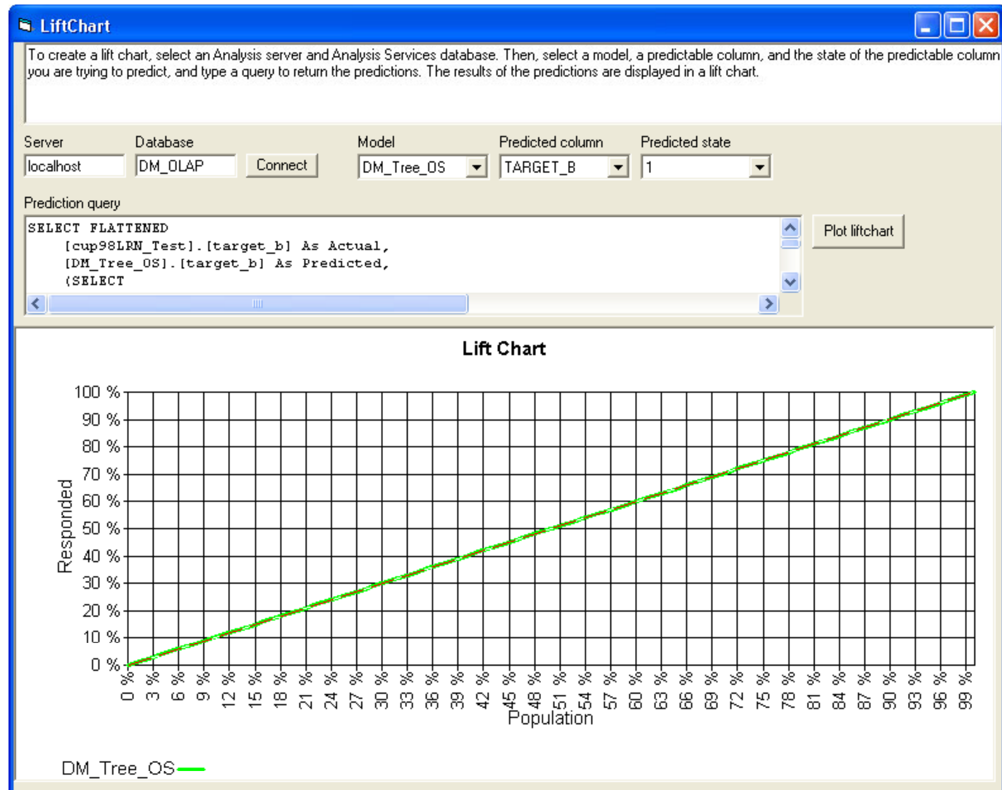


Figure 8.5 Lift chart with the DM_Tree_OS mining model

The key point here is that data mining is not an exact science, and it's not an easy process. Even with all the work we did to create these models, we did not get the results we expected. Maybe better transformations would improve our results. Maybe better column selection, achieved through more in-depth exploration, would create better models. Or maybe creating an over-sampled table with an even higher ratio of positive to negative responses would generate better models.

So now it's your turn…we've started you off with the process and the tools. What can you achieve? Work through the process a few more times and see if you can improve the results. If you find something interesting, send us your results.

Good luck, and have fun!

Now, before you go running off on your own, let's see the code that makes this validation task possible.

Looking at the Validation Code

Let's look at the tasks that various subroutines accomplish during the model-validation step. Open Visual Basic and follow along as we walk through the code.

▶ **To view the form frmLiftChart.frm**

1. In Windows Explorer, browse to the **C:\Program Files\Microsoft NESBooks\SQLServer2000\Data Mining\DM Sample** folder, and then double-click the **DMFinal.vbp** file.
2. In Visual Basic, in the **Project Explorer** window, expand the project, and then expand **Forms**.
3. Right-click **frmLiftChart** (**frmLiftChart.frm**), and then click **View Code**.

Once again, we will be using an Angoss control to create the lift chart. For more information about using the control, see the Angoss documentation that was installed with the controls.

The first steps are to create a new connection to the Analysis server and create a recordset that will hold the models that exist in the Analysis database that we are connecting to.

```
    Set cn = New ADODB.Connection

    Dim rsModels As ADODB.Recordset

    cn.Open "Provider=MSOLAP;Data Source=" & txtServer & ";Initial
Catalog=" & txtDatabase
    Set rsModels = cn.OpenSchema(adSchemaProviderSpecific, ,
schemaModels)
```

We now populate the `cbModels` combo box with the model names held in the `rsModels` recordset. We first clear the combo box and then cycle through the recordset, adding model names to the combo box.

```
    cbModels.Clear

    While Not rsModels.EOF
        cbModels.AddItem CStr(rsModels.Fields("MODEL_NAME").Value)
        rsModels.MoveNext
    Wend
```

Next, the cbColumns combo box is populated with the attributes labeled as predictable for the model selected in the cbModels combo box. This routine is called on the cbModels click event.

We first open a recordset that will be used to hold the schema information for the columns. We will use this information to find the predictable columns and display them in the cbColumns combo box.

```
    Set rs = cn.OpenSchema(adSchemaProviderSpecific,
Array(txtDatabase.Text, Empty, cbModels.Text), schemaColumns)
```

The last step in this subroutine is to cycle through the rs recordset, search for the predictable columns, and add them to the combo box.

```
    cbColumns.Clear
    While Not rs.EOF
        If rs.Fields("IS_PREDICTABLE").Value Then
            cbColumns.AddItem CStr(rs.Fields("COLUMN_NAME").Value)
        End If
        rs.MoveNext
    Wend
```

We only have one more subroutine to get through before setting up and displaying the lift chart—we need to populate the cbPredictedState combo box with all of the possible predicted states of the predictable column.

We do this much the same as before, filling a recordset with each distinct state of the column select in the cbColumns combo box. This routine is called on the cbColumns click event.

```
    Set rs = cn.Execute("SELECT DISTINCT [" & cbColumns.Text & "]
FROM [" & cbModels.Text & "]")
```

The states of the selected predictable column are then added to the cbPredictedState combo box.

```
    cbPredictedState.Clear
    While Not rs.EOF
        cbPredictedState.AddItem CStr(rs.Fields(0).Value)
        rs.MoveNext
    Wend
```

Everything is now ready to begin the code for creating the lift chart, which is called on the `btnDoLiftChar` click event.

We first open a recordset, using the query we copied into the `txtQuery` text box and the connection we crated in the first subroutine.

```
    rs.Open txtQuery.Text, cn
```

We then set the `LiftchartModel` object to the lift chart control, and point it to the model selected in the `cbModels` combo box and the recordset we just created.

The query in the recordset defines the prediction query that will be used to compare the states predicted by the model to the actual states in the testing table.

```
    Set LiftchartModel = LiftChart1.Models.Add(cbModels.Text, rs,
"")


    If LiftchartModel Is Nothing Then
        MsgBox "Model has not been created." & vbCrLf & "Current
limit for number of models is 2.", vbCritical
    Else
```

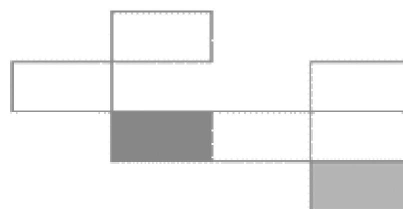
Next, we parameterize the lift chart using values inserted into the `rs` recordset by the MDX query.

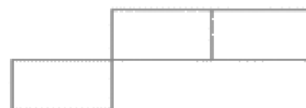
```
        LiftchartModel.ActualColumn = "Actual"
        LiftchartModel.PredictColumn = "Predicted"
        LiftchartModel.ChartedValue = cbPredictedState.Text
        LiftchartModel.ProbabilityColumn = "Certainty"
```

We then compute the values that will be displayed in the lift chart and refresh the control to display the results to the screen.

```
        LiftchartModel.Compute
        LiftchartModel.Visible = True
```

Appendix





Appendix

Managing Tables

In addition to copying an existing table, you can also perform the following tasks using the **Manage Tables** form:

- Select specific columns to be included in a new table.
- Drop tables from the database that are no longer useful.
- Create a copy of an existing table but with fewer rows (or sample the table).
- Create a copy of an existing table, but force the sampling algorithm to include a higher percentage of positive predictable values than actually existed in the original table.

Selecting Columns

You may want to investigate specific columns in a table without having to work with the entire table—especially with a table as large as the one we use in this book. Also, you may find some columns that you want to exclude from your analysis. For example, if your data includes both a postal code and street address, you may decide that the street address is unnecessary because you already have the postal code.

▶ **To create a table with selected columns**

1. In the Data Mining Tool, click **Manage Tables**.
2. In the **Create a new table by selecting columns from an existing table** section, enter the following information:
   - For **Select a source table**, select the table that holds the columns you want to include in the new table.
   - For **Enter a table name**, type a name for the new table.
3. To include specific columns in the new table, select a column from the list associated with the source table, and then click the > button. Repeat this step for each column to be included. (To include all columns in the new table, click the >>button.)
4. To remove a column from the new table, select the column from the list associated with the new table, and then click the < button. (To remove all columns from the new table, click the << button.)
5. Once you have selected the columns that you want to include in the new table, click **Create Table**. This adds a new table containing the selected columns to the database.

Dropping Tables

Eventually, with your own data, you will create several tables. If you later decide that a table has lost its usefulness or if you want to redo a step, you can drop the table from the database.

▶ To drop a table from the database

1. In the Data Mining Tool, click **Manage Tables**.
2. In **Select a table to drop**, select the table that you want to drop.
3. Click **Drop Table**.

Sampling a Table

Often a dataset contains too many rows, making it difficult and time consuming to perform an analysis. Accordingly, the Data Mining Tool includes an option to reduce the size of the table by reducing the number of rows that are in it. To do this, the Data Mining Tool randomly selects the number of rows you specify and creates a new table.

Additionally, there are times when the predictable attribute is so skewed toward one state that it is hard to create a model that accurately predicts it. In this case, you can use the Data Mining Tool to create an over-sampled table, meaning that you can artificially increase the ratio of positive to negative responses in the predictable attribute.

In this section, we'll first use the Data Mining Tool to create a table with a reduced record count, and then we'll look at the code in the Data Mining Tool that makes this work. Next, we'll create an over-sampled table and then look at the code behind it.

Reducing Record Count

If the table you are working with is very large, you may want to reduce the number of records in your table so that the cleaning tasks are less time consuming. This process is called creating a sampled table.

▶ To create a sampled table

1. In the Data Mining Tool, click **Manage Tables**, and then click **Sample**.
2. In the **Small Table** form, make the following selections and then click **Sample**:
   - Select **Create a sample of a table**.
   - For **Number of records**, type the number of records to be included in your new table.
   - For **Original table**, select the table that you want to sample.
   - For **New table**, type a name for your new table.

Figure A.1 Creating a sampled table

Looking at the Code for Sampling a Table

Within the routine, the user has the option of either creating a sampled table or an oversampled table. If the user selects a sampled table, the variable `blnSampleBasic` is set to True, and the code in this section is used.

This procedure is similar to the one used in splitting the original table into the training and testing tables. As with that routine, we first check to see whether the table already exists, and if it does, we give the user the option of either dropping and re-creating it or choosing a new name for the table. Next, we get the percentage of positive and negative responses in the predictable column so that they can be displayed on the form.

Because we are creating a single random table from the original table, there is no restriction on the rows that can be used from the original table. This means that the `strSQLWhere` statement is left blank.

```
strSQLWhere = ""
```

We then only have to call the `Create Table` function and the table is created. For more information about using the `Create Table` function, see Chapter 7, "Splitting the Data."

```
mdlSample.Create Table txtSampledTable.Text, strTable, strSQLWhere,
lngRecordCount
```

Retrieving the final percentages of states in the predictable column and writing them to the form finishes off this section of the routine.

```
strSQLSelect = "SELECT COUNT(" & PREDICTED & ") as count FROM [" &
txtSampledTable.Text & "] WHERE " & PREDICTED & " = 1"
Set rsRecordCount = mdlProperties.cnDataPrep.Execute(strSQLSelect)

sngPercentYes = (rsRecordCount!Count / lngRecordCount) * 100
sngPercentNo = 100 - sngPercentYes

txtSampledYes.Text = CVar(sngPercentYes)
txtSampledNo.Text = CVar(sngPercentNo)
```

Now let's look at how to over-sample a table.

Increasing the Ratio of Responses in the Predictable Column

As explained in the introduction, it may be useful to highlight a state of the predictable column, especially if it is underrepresented. In the data used in this book, a positive occurrence of the predictable attribute only occurs about 5 percent of the time. By artificially forcing the ratio of positive to negative values to be higher, such as 80 percent positive to 20 percent negative, we can often find stronger relationships and create better models. Although this seems like we're manipulating the data in a bad way, it actually can improve the effectiveness of the final model. The final goal is to build a good model, and how we change the data to achieve that does not matter.

▶ **To create an over-sampled table**

1. In the Data Mining Tool, click **Manage Tables**.
2. Click **Sample**.
3. On the **Small Table** form, make the following selections and then click **Sample**:
   - Select **Create an over-sampling of a table**.
   - For **% Positive**, type the percentage of positive predictable values that you want to exist in the new table.
   - For **Original table**, select the table that you want to sample.
   - For **New table**, type a name for your new table.

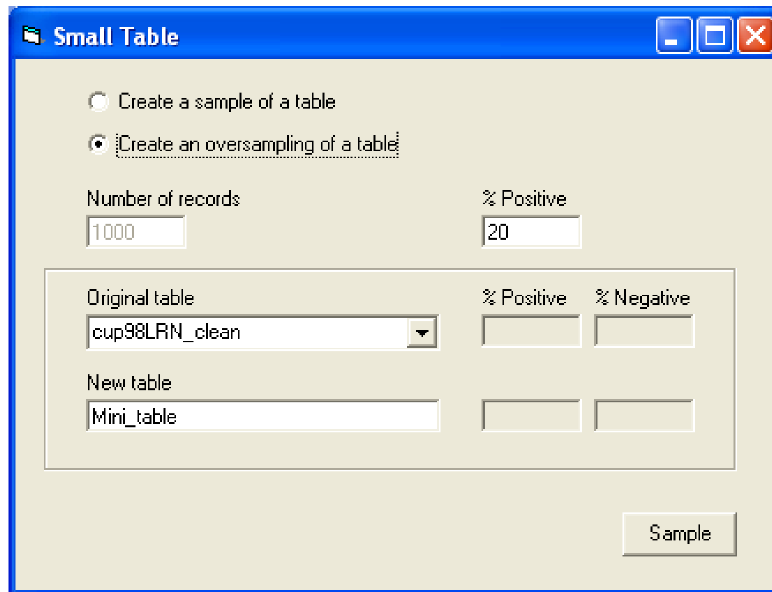


Figure A.2 Creating an over-sampled table

Looking at the Code for Over-Sampling a Table

If the user opts to create an over-sampled table, the code in this section is used.

This is only slightly more complicated than a simple sampling of the table. In order to make this work, we made the assumption that if the user wants to over-sample the table, he or she will want to keep all of the positive responses and just add to them enough of the negative responses to create the specified percentages.

The first step is to find out how many values we need to sample. Unlike the previous case, the user is not selecting the number of rows to include in the table, but instead the percentage ratio of "yes" to "no" values in the predictable column. For this reason, the first step is to get the number of positive responses in the table.

```
Set rsTrueData = cnDataPrep.Execute("SELECT COUNT(" & PREDICTED & ")
as count FROM " & strTable & " WHERE " & PREDICTED & " = 1
")lngSampleTrue = rsTrueData!Count
```

We then calculate the number of negative results that we need to add to the positive results to achieve the percentages the user specified.

```
lngSampleFalse = lngSampleTrue / (CLng(txtPercentYes) / 100) -
lngSampleTrue
```

Because we are keeping all of the positive responses, we will include only the negative responses in the pool of data that we're extracting using the WHERE clause.

```
strSQLWhere = " WHERE " & PREDICTED & " = 0 "
```

The contents of `strSQLWhere` are then passed into the `Create Table` function.

```
mdlSample.Create_Table txtSampledTable.Text, strTable, strSQLWhere,
lngSampleFalse
```

We now have a table filled with randomly selected negative responses. To finish, we use the following SQL statement to populate the table with the positive responses from the original table.

```
strSQLSelect = "INSERT INTO " & txtSampledTable.Text & " SELECT *
FROM " & strTable & " WHERE " & PREDICTED & " = 1"

cnDataPrep.Execute (strSQLSelect)
```

To get the record count we combine the number of positive responses and the number of rows we sampled in the original table.

```
lngRecordCount = lngSampleTrue + lngSampleFalse
```

As the last step, we calculate the percentages of yes and no values in the new table and display them to the form.

And now we are finished—an over-sampled table now exists in the database.