Microsoft

Data Science with Microsoft SQL Server 2016

Buck Woody, Danielle Dean, Debraj GuhaThakurta
Gagan Bansal, Matt Conners, Wee-Hyong Tok

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

ISBN: 978-1-5093-0431-8

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Acquisitions Editor: Kim Spilker
Developmental Editor: Bob Russell, Octal Publishing, Inc.
Editorial Production: Dianne Russell, Octal Publishing, Inc.
Copyeditor: Bob Russell

Contents

Foreword ..... **viii**

Introduction ..... **xi**

How this book is organized xii

Who this book is for xii

Acknowledgements xiii

Free ebooks from Microsoft Press xiii

Errata, updates, & book support xiii

We want to hear from you xiv

Stay in touch xv

Chapter 1: Using this book **1**

For the data science or R professional 1

Solution example: customer churn 3

Solution example: predictive maintenance and the Internet of Things 4

Solution example: forecasting 5

For those new to R and data science 7

Step one: the math 8

Step two: SQL Server and Transact-SQL 11

Step three: the R programming language and environment 14

Chapter 2: Microsoft SQL Server R Services 16

The advantages of R on SQL Server 16

A brief overview of the SQL Server R Services architecture 21

SQL Server R Services 21

Preparing to use SQL Server R Services 24

Installing and configuring 24

Server 25

Client 28

Making your solution operational 36

Using SQL Server R Services as a compute context 36

Using stored procedures with R Code 40

Chapter 3: An end-to-end data science process example 43

The data science process: an overview 44

The data science process in SQL Server R Services: a walk-through for R and SQL developers 47

Data and the modeling task 48

Preparing the infrastructure, environment, and tools 51
Input data and SQLServerData object 65
Exploratory analysis 68
Data summarization 68
Data visualization 70
Creating a new feature (feature engineering) 76
Using R functions 77
Using a SQL function 80
Creating and saving models 83
Using an R environment 84
Using T-SQL 86
Model consumption: scoring data with a saved model 89
Evaluating model accuracy 95
Summary 97
Chapter 4: Building a customer churn solution 98
Overview 99
Understanding the data 101
Building the customer churn model 105
Step-by-step 108

Summary 118

Chapter 5: Predictive maintenance and the Internet of Things 120

What is the Internet of Things? 122

Predictive maintenance in the era of the IoT 124

Example predictive maintenance use cases 127

Before beginning a predictive maintenance project 129

The data science process using SQL Server R Services 132

Define objective 136

Identify data sources 137

Explore data 140

Create analytics dataset 142

Create machine learning model 155

Evaluate, tune the model 157

Deploy the model 161

Summary 165

Chapter 6: Forecasting 167

Introduction to forecasting 169

Financial forecasting 169

Demand forecasting 170
Supply forecasting 171
Forecasting accuracy 171
Forecasting tools 173
Statistical models for forecasting 174
Time–series analysis 174
Time–series forecasting 179
Forecasting by using SQL Server R Services 183
Upload data to SQL Server 183
Splitting data into training and testing 185
Training and scoring time–series forecasting models 186
Generate accuracy metrics 189
Summary 190
About the authors 191

Foreword

The world around us—every business and nearly every industry—is being transformed by technology. This disruption is driven in part by the intersection of three trends: a massive explosion of data, intelligence from machine learning and advanced analytics, and the economics and agility of cloud computing.

Although databases power nearly every aspect of business today, they were not originally designed with this disruption in mind. Traditional databases were about recording and retrieving transactions such as orders and payments. They were designed to make reliable, secure, mission-critical transactional applications possible at small to medium scale, in on-premises datacenters.

Databases built to get ahead of today's disruptions do very fast analyses of live data in-memory as transactions are being recorded or queried. They support very low latency advanced analytics and machine learning, such as forecasting and predictive models, on the same data, so that applications can easily embed data-driven intelligence. In this manner, databases can be offered as a fully managed service in the

cloud, making it easy to build and deploy intelligent Software as a Service (SaaS) apps.

These databases also provide innovative security features built for a world in which a majority of data is accessible over the Internet. They support 24 × 7 high-availability, efficient management, and database administration across platforms. They therefore make possible *mission-critical intelligent applications* to be built and managed both in the cloud and on-premises. They are exciting harbingers of a new world of ambient intelligence.

SQL Server 2016 was built for this new world and to help businesses get ahead of today's disruptions. It supports hybrid transactional/analytical processing, advanced analytics and machine learning, mobile BI, data integration, always-encrypted query processing capabilities, and in-memory transactions with persistence. It integrates advanced analytics into the database, providing revolutionary capabilities to build intelligent, high-performance transactional applications.

Imagine a core enterprise application built with a database such as SQL Server. What if you could embed intelligence such as advanced analytics algorithms plus data transformations within the database itself, making every transaction

intelligent in real time? That's now possible for the first time with R and machine learning built in to SQL Server 2016. By combining the performance of SQL Server in-memory Online Transaction Processing (OLTP) technology as well as in-memory columnstores with R and machine learning, applications can achieve extraordinary analytical performance in production, all while taking advantage of the throughput, parallelism, security, reliability, compliance certifications, and manageability of an industrial-strength database engine.

This ebook is the first to truly describe how you can create intelligent applications by using SQL Server and R. It is an exciting document that will empower developers to unleash the strength of data-driven intelligence in their organization.

Joseph Sirosh
Corporate Vice President
Data Group, Microsoft

Introduction

R is one of the most popular, powerful data analytics languages and environments in use by data scientists. Actionable business data is often stored in Relational Database Management Systems (RDBMS), and one of the most widely used RDBMS is Microsoft SQL Server. Much more than a database server, it's a rich ecostructure with advanced analytic capabilities. Microsoft SQL Server R Services combines these environments, allowing direct interaction between the data on the RDBMS and the R language, all while preserving the security and safety the RDBMS contains. In this book, you'll learn how Microsoft has combined these two environments, how a data scientist can use this new capability, and practical, hands-on examples of using SQL Server R Services to create real-world solutions.

How this book is organized

This book breaks down into three primary sections: an introduction to the SQL Server R Services and SQL Server in general, a description and explanation of how a data scientist works in this new environment (useful, given that many data scientists work in "silos," and this new way of working brings them in to the business development process), and practical, hands-on examples of working through real-world solutions. The reader can either review the examples, or work through them with the chapters.

Who this book is for

The intended audience for this book is technical—specifically, the data scientist—and is assumed to be familiar with the R language and environment. We do, however, introduce data science and the R language briefly, with many resources for the reader to go learn those disciplines, as well, which puts this book within the reach of database administrators, developers, and other data professionals. Although we do not cover the totality of SQL Server in this book, references are provided and some concepts are

explained in case you are not familiar with SQL Server, as is often the case with data scientists.

Acknowledgements

Brad Severtson, Fang Zhou, Gopi Kumar, Hang Zhang, and Xibin Gao contributed to the development and publication of the content in Chapters 3 and 4.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You

can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

https://aka.ms/IntroSQLServerR/errata

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to http://support.microsoft.com.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: http://twitter.com/MicrosoftPress.

CHAPTER 1

Using this book

In this book, you'll learn how to install, configure, and use Microsoft's SQL Server R Services in data science projects. We're assuming that you have familiarity with data science and, most important, the R language. But if you don't, we've added a section here to help you get started with this powerful data-analysis environment.

For the data science or R professional

"Data science" is a relatively new term, and it has a few definitions. For this book, we'll use the

name itself to define it. Thus a data science professional is a technical professional who uses a scientific approach (asks a question, creates a hypothesis—or more accurately a *model*—tests the hypothesis, and then communicates the results) in the data-analytics process, whether using structured or unstructured data, or perhaps both.

We're assuming that you have a background in general mathematics, some linear algebra, and, of course, an in-depth familiarity with statistics. We're also assuming that you know the R language and its processing environment and are familiar with how to load various packages, and that you understand when to use R for a given data solution. But even if you don't have those skills, read on; we have some resources that you can use.

Even if you have a deep background in statistics and R, Microsoft's SQL Server might be new to you. To learn how to work with it, take a look at the section "SQL Server and Transact-SQL" later in this chapter. In this book, we'll assume that you have a working knowledge of how SQL Server operates, and how to read and write Transact-SQL—the dialect of the SQL language that Microsoft implements in SQL Server.

In the two chapters that follow, we'll show you what SQL Server R Services is all about and how you can install it. You'll learn the client tools and the way to work with R Services, and we'll follow that up with a walk-through using the data science process.

One of the best ways to learn to work with a product is to deconstruct some practical examples in which it is used. In the rest of this book, we've put together representative, real-world use cases that demonstrate an end-to-end solution for a typical data science project. These are examples you'll find in other data science tools, so you should be able to extrapolate the concepts of what you already know to how you can do the same thing in SQL Server using R Services—we think you'll find it has some real advantages to using a standard R platform.

Solution example: customer churn

One of the most canonical uses for prediction science is customer churn. Customer churn is defined as the number of lost customers divided by the number of new customers gained. As long as you're gaining new customers faster than you're losing them, that's a good thing, right? Actually, it's not—for multiple reasons. The

primary reason customer churn is a bad thing is that it costs far more to gain a customer, or regain a lost one, than it does to keep an existing customer. Over time, too much customer churn can slowly drain the profits from a company. Identifying customer churn and the factors that cause it are essential tasks for a company to stay profitable.

Interestingly, customer churn extrapolates out to other uses, as well. For instance, in a hospital, you *want* customers to churn—to not come back. You want them to stay healthy after their hospital visit.

In this example, we'll show you how to calculate and locate customer churn by using R and SQL Server data.

Solution example: predictive maintenance and the Internet of Things

It is critical for businesses operating or utilizing equipment to keep those components running as effectively as possible because equipment downtime or failure can have a negative impact beyond just the cost of repair. Predictive maintenance is defined as a technique to forecast when an in-service machine will fail so

that maintenance can be planned in advance. It includes more general techniques that involve understanding faults, failures, and timing of maintenance. It is widely used across a variety of industries, such as aerospace, energy, manufacturing, and transportation and logistics.

New predictive maintenance techniques include time-varying features and are not as bound to model-driven processes. The emerging Internet of Things (IoT) technologies have opened up the door to a world of opportunities in this area, with more sensors being installed on devices and more data being collected about these devices. As a result, data-driven techniques now promise to unleash the potential of using data to understand when to perform maintenance.

In this example, we'll show you different ways of formulating a predictive maintenance problem and then show you how to solve them by using R and SQL Server.

Solution example: forecasting

Forecasting is defined as the process of making future predictions by using historical data, including trends, seasonal patterns, exogenous factors, and any available future data. It is widely used in many applications and critical business decisions depend on having an accurate

forecast. Meteorologists use it to generate weather predictions; CFOs use it to generate revenue forecasts; Wall Street analysts use it to predict stock prices; and inventory managers use it to forecast demand and supply of materials.

Many businesses today use qualitative judgement–based forecasting methods and typically manage their forecasts in Microsoft Excel, or locally on an R workstation. Organizations face significant challenges with this approach because the amount and availability of relevant data has grown exponentially. Using SQL Server R Services, it is possible to create statistically reliable forecasts in an automated fashion giving organizations greater confidence and business responsiveness.

In this section, we will introduce basic forecasting concepts and scenarios and then illustrate how to generate forecasts by using SQL Server R Services.

For those new to R and data science

If you are new to R and you're interested in learning more before you dive in to these examples, read on. You have a few things to learn, but it isn't too difficult if you stick with it. As our favorite philosopher, Andy Griffith, would say, "Ain't nothing good, easy." Although that might not be grammatically correct, the sentiment is that you're about to embark on a journey with a very powerful tool, and with great power comes great responsibility. It will take time and effort on your part to learn to use this tool correctly.

R is used to process data, and has powerful statistical capabilities. In most cases, when you run a statistical formula on a set of numbers, you'll get *an* answer—which isn't always true of many languages. But when you process statistical data, you're often left with an additional set of steps involving *interpreting* and then *applying* the answer to a decision. This means that not only are your coding skills at stake, your professional reputation is, as well.

But, not to fear: there are many low-cost and even free options to bring you up to speed. If you're a motivated self-learner, you're in luck.

Step one: the math

There's no getting away from math when you're working with R. To fully make use of the R language, you'll need three disciplines covered: general math, linear algebra, and first- to second-year level experience with statistics.

General math

Let's begin with an understanding of basic math, which includes the following concepts:

- **Numbers** Counting (natural), whole, real, integers, rational, imaginary, complex, binary, fractions and scientific
- **Operations** Add, subtract, divide, multiply, conversions, working with fractions in those operations

We are big fans of the Khan Academy. You can find a good course on general math at https://www.khanacademy.org/math. You also can go to http://www.webmath.com/index2.html and use Discovery Education for a general math course. And a quick web search using the term **Basic Math Skills** will turn up even more resources in your geographic area. Even if you're sure about your skills, it can be fun and useful to bone up quickly on these basic skills.

Linear Algebra

Linear algebra covers vector spaces and linear mappings between them. You'll need to focus especially on the matrices equations and also understand the following:

- Vector spaces
- Matrices
- Linear transformations
- Eigenvalues and eigenvectors
- Least-squares fitting
- Fourier transforms and other transform operations

If you're new to algebra, check out the aforementioned Khan Academy courses. After that, move on to Linear Algebra courses, which you can find at https://www.khanacademy.org/math/linear-algebra.

You also can find a good course on linear algebra at the Massachusetts Institute of Technology's Open Courseware at http://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/index.htm. And, of course, a quick web search using **Learning Linear Algebra** yields even more results.

Statistics

Descriptive and predictive statistics are essential tools for the data scientist, and you'll need a solid grounding in these concepts to effectively use the R language for data processing. You'll probably spend most of your time learning statistics, more so than any other skill in data science. Here are the primary concepts and specific processes you need to understand in statistics:

- Descriptive statistical methods
- Predictive statistical methods
- Probability and combinatorics
- A focus on inference and representation statistical methods
- Time-series forecasting models

- Regression models (linear systems and eigensystems, multivariate, and nonlinear regression, as well)

Again, the Khan Academy has a wide range of breadth and depth courses on statistics. You can find its list at https://www.khanacademy.org/math/probability. Sat Trek (http://stattrek.com/) is another free tutorial site with a good introduction to statistics. Because statistics is a very mature science, a quick search yields multiple sources for learning from books, videos, and tutorials.

Step two: SQL Server and Transact-SQL

In the late 1960s and the early 1970s, working with data usually meant using ASCII or binary-encoded "flat" files with columns and rows. Programs such as COBOL would "link" these files together using various methods. If these links were broken, the files were no longer able to be combined, or joined. There were also issues around the size of the files, the speed (or lack thereof) with which you could reference and open them, and locking.

To solve these issues, a relational calculus was implemented over an engine to insert, update,

delete, and read data over a designated file format—thus, the Relational Database Management System (RDBMS) was born. Most RDBMS implementations used the Structured Query Language (SQL), a functional language, to query and alter data. This language and the RDBMS engines are among the most widely used data processing and storage mechanisms in use today, and so the data scientist is almost always asked to be familiar with using SQL.

Microsoft's SQL Server is an RDBMS, but it also serves as a larger platform for Business Intelligence (BI), data mining, reporting, an Extract, Transform, and Load (ETL) system, and much more—including the R language integration. It uses a dialect of the SQL language called Transact-SQL (T-SQL). To effectively use the R integration demonstrated in this book, you'll need to understand how to use T-SQL, including the following:

- Basic Create, Read, Update, and Delete (CRUD) operations
- Database and database object creation: Data Definition Language (DDL) statements
- Multi-join operations
- Recursive SELECT statements

- Grouping, combining, and consolidating Data Manipulation Language (DML) statements
- SQL Server architecture and general operation

There is a litany of courses you can take for SQL in general, and T-SQL specifically. Here are a few:

- Learn SQL is a great site to get started with general SQL: http://www.sqlcourse.com/
- Codeacademy is another great place to get started: https://www.codecademy.com/learn/learn-sql
- To learn the basics of the T-SQL dialect, try this resource: http://www.tsql.info/
- Microsoft has a tutorial on getting started with T-SQL: https://msdn.microsoft.com/en-us/library/ms365303.aspx

Next, you'll need to understand SQL Server's architecture and features. For that, use the information in Books Online at https://msdn.microsoft.com/library/ms130214.aspx.

Step three: the R programming language and environment

R is a language and platform used to work with data, most often by using statistical methods. It's very mature and is used by many data professionals around the world. It's extended with a "package," which is code that can reference using *dot notation* and function calls.

If you know SQL, T-SQL, or a scripting language like Windows PowerShell, you'll be familiar with the basic structure of an R program. It's an *interpreted* language, and one of the interesting things about the way it works is in how it stores computational data. When you work with R, everything is stored in an ordered collection called a *vector*. This is both a strength and a weakness of the R system, one that Microsoft addresses with its enhancements to the R platform.

To learn more about R, you have a very wide array (pun intended) of choices:

- There's a full course you can take on R at DataCamp: https://www.datacamp.com/
- The primary resource you can use for learning R on SQL Server is here:

https://msdn.microsoft.com/library/mt674876.aspx

- And you can find tutorials on R for SQL Server here: https://msdn.microsoft.com/library/mt591993.aspx

You can also find out more about data science and working with R at my blog, which you can view at https://buckwoody.wordpress.com/. You'll find a rich list of resources there to help you continue in your learning journey. If you want to go further and learn more about data science, check out https://buckwoody.wordpress.com/2015/09/16/the-amateur-data-science-body-of-knowledge/.

Now, on to R with SQL Server…

Microsoft SQL Server R Services

This chapter presents an overview of the SQL Server R Services, how it works, and where you can get it. We also show you how to make your solutions operational and where you can learn more about R on SQL Server.

The advantages of R on SQL Server

In a 2011 study,[1] Erik Brynjolfsson of the Massachusetts Institute of Technology Sloan

[1] See http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1819486

School of Management showed a link between firms that use *Data-Driven Decision Making* and higher performance. Organizations are moving ever closer to using more and more data interpretation in their operations. And much of that data lives in Relational Database Management Systems (RDMBS) like Microsoft SQL Server.

R has long been a popular data-processing language. It has thousands of external packages, is relatively easy to read and understand, and has rich data-processing features. R is used in thousands of organizations around the world by data-analysis professionals.

Note If you're not familiar with R, check out the resources provided in Chapter 1.

A statistical programmer versed in R often accesses data stored in a database by using a package that calls the Open Database Connectivity (ODBC) Application Programming Interface (API), which serves as a conduit to the RDBMS to retrieve data. R then receives that data as a *data.frame* object. The results from the database server are either pushed back across the network to the RDBMS, or the data professional saves the results locally in tabular or other form. Using this approach, all of the

processing of the data happens locally, with the exception of the SQL statement used to gather the initial set of data. Data is rarely sent back to the RDBMS—it is most often a receive operation.

The Structured Query Language (SQL) is another data-processing language designed specifically for working within an RDBMS. Its roots involve relational algebra and relational calculus, and it is used in multiple database systems. Most vendors extend the basic SQL constructs to take advantage of the platform it runs on, and in the case of Microsoft SQL Server, this dialect is called *Transact-SQL* (T-SQL). T-SQL is used to query, update, and delete data, along with many other functions.

In both R and T-SQL, the developer types commands in a step-wise fashion in an editor window or at a command-line interface (CLI). But the path of operations is different from that point on. R is an *interpreted* language, which means a set of binaries local to the command environment processes the operations and returns the result directly to the calling program. In SQL Server, the client is separate from the processing engine. The installation of SQL Server listens on a network interface, and the client software puts the commands on the network path in a particular protocol. The server receives this packet with the T-SQL statements only if the

packet is "well formed." The commands are run on the server, and the results, along with any messages the server sends (such as the number of rows) and any error messages, are returned to the client over the same protocol. The primary load in this approach is on the server rather than the workstation. Of course, the workstation might then further process the data—using Java, C#, or some other local language—but often the business logic is done at the server level, with its security, performance, and other advantages and controls.

But SQL Server is much more than just a data store. It's a rich ecostructure of services, tools, and an advanced language to deal with data of almost any shape and massive size. Many organizations store the vast amount of their actionable data within SQL Server by using custom and commercial software. It has more than 36 data types, and gives you the ability to define more.

SQL Server also has fine-grained security features. When these are applied, the data professional can simply query the data, and only the allowed datasets are returned. This facilitates good separation of duties, which is highly important in large, complex systems for which one group of professionals might handle the

security of data, and another handles the querying and processing of the data.

SQL Server also has advanced performance features, such as a column-based index, which can provide extremely fast search and query functions over very large sets of data.

Using R on SQL Server combines the power of the R language (and its many packages) and the advantages of the SQL Server platform by *placing the computation over the data*. This means that you aren't moving the data to the R system, involving networking, memory on two systems, CPU power on each side, and other disadvantages—the code operates on the same system as the application data.

Combining R and SQL Server means that the R environment gains not only the functions and features in the R language, but also the ecostructure, security, and performance of SQL Server, as well as increased scale. And using R directly on SQL Server means that the R code can save the results of the operation to a new or existing table for other queries to access and update.

A brief overview of the SQL Server R Services architecture

The native implementation of open-source R reads data into a data-frame structure, all of which is held in memory. This means that R is limited to working with data sizes that will fit into the RAM on the system that processes the data. Another limitation in R is within a few of the core packages that process certain algorithms, most notably dealing with linear regression math. These native calls can perform slowly.

SQL Server R Services

To address these limitations (and others), Microsoft R Server brings several major enhancements to the R platform—Microsoft R Server is what is used in SQL Server R Services. The first enhancement is the *ScaleR* library, which allows MRS to "chunk" data stored on permanent storage in either comma-separated-value files, databases, and many other data sources into manageable sets. These libraries also offer increased parallelization, which makes

it possible for the R code to process data more efficiently.

Microsoft R uses a binary storage format called an XDF, which handles data frames in a more efficient pattern, allowing advantages such as appending data to the end of a file, and other performance improvements.

Another set of enhancements involves replacing some of the core calls to some of the math libraries in the open-source version of R, with much higher performance. Other enhancements involve extending the scaling features of R to distribute the workload across multiple servers.

R Server is available on multiple platforms, from Windows to Linux, and has multiple editions. Microsoft also has combined the R Server code in its other platforms, including HDInsight (Hadoop) and with the release of SQL Server 2016. In this book, we'll deal with the implementation in SQL Server 2016, called SQL Server R Services.

A SQL Server installation, called an *instance*, contains the binaries required to run the various RDBMS engine functions, Business Intelligence (BI) features, and other engines. The instance also instantiates entries into an internal Windows database construct called the *Windows Registry*, and a few SQL Server databases to configure and

secure the RDBMS environment. The binaries run as Windows Services (equivalent to a Linux Daemon), regardless of whether someone is signed in to the server. These Windows Services listen on networking ports for proper calls from client software.

In SQL Server 2016 and later, Microsoft combines the two environments by installing the Microsoft R Server binaries along with the SQL Server installation. Changes in the SQL Server base code allows the two environments to communicate securely in the same space and makes it possible for the two services to be upgraded without affecting each other, within certain parameters. This architecture means that you have the purest possible form of both servers, while allowing SQL Server the complete access to the R environment.

To use R code in this architecture, you must configure the SQL Server instance to allow an *external scripts* setting (which can be secured) so that the T-SQL code can make calls to the R Server. Data is passed as a data.frame object to the R code directly from SQL Server, and SQL Server interprets the results from the R code as a tabular or other format, depending on the data returned. In this manner, the T-SQL and R code can interoperate on the same data, all while using the features and functions in each

language. Because the call stays within the constructs of SQL Server, the security and performance of that environment is maintained.

Preparing to use SQL Server R Services

After the installation and configuration of SQL Server R Services, you can begin to use your R code in two ways: by executing the code interactively, or, more commonly, by saving your R code within the body of a script that executes on SQL Server, called a *stored procedure*. The stored procedure can contain T-SQL *and* R code, and each can pass variables and data to the other. Before you can run your code, you'll need to install SQL Server R Services.

Installing and configuring

You can install R Services on an initial installation of a SQL Server 2016 instance. You also can add R Services later by using the installation source. The installation or addition process will install the R server and client libraries onto the SQL Server.

Note There are various considerations for installing R Services on SQL Server, and if you're setting up a production system you should follow a complete installation planning process with your entire IT team. You can read the full installation instructions for R Services on SQL Server at https://msdn.microsoft.com/en-us/library/mt696069.aspx.

For your research, and for any SQL Server developer, there's a simplified installer for the free Developer Edition, which we describe in a moment.

Server

SQL Server comes in *versions* and *editions*. A version is a dated release of the software based on a complete set of features; it has a product name such as SQL Server 2016. SQL Server R Services is included with SQL Server Version 2016 and later.

An edition of SQL Server is a version with an included set of capabilities. These range from Microsoft SQL Server Express (a free offering), which provides a limited amount of memory, capabilities, and database size, to several other Editions up to SQL Server Enterprise, which contains all capabilities in the platform and can

use the maximum resources the system can provide.

More info You can learn more about which editions support each capability at https://www.microsoft.com/cloud-platform/sql-server-pricing.

In a production environment, your IT team should help you research and decide on the proper edition of SQL Server to install. If you are installing a copy for yourself or for a development environment, the SQL Server Developer Edition is often your best choice. It's a free, single-user edition that contains all of the features and capabilities in SQL Server, and you can use it to work through all of the examples in this book. You can find the download for SQL Server Developer Edition at https://www.microsoft.com/cloud-platform/sql-server-editions-developers, and you can start the installation process on your workstation or in a virtual server. But there's a new method of installing the Developer Edition that's even simpler: to download and install the software, go to https://blogs.msdn.microsoft.com/bobsql/2016/

07/13/the-sql-server-basic-installer-just-install-it-2/.

If you have a previous installation of SQL Server 2016, you can add Microsoft R Server capabilities. During the installation, on the Installation tab, click New SQL Server Stand-Alone Installation Or Add Features To An Existing Installation. On the Feature Selection page, select the options Database Engine Services and R Services (In-Database). This will configure the database services used by R jobs and install all extensions that support external scripts and processes.

Whether you're installing for the first time or after a previous installation, there are a few steps you need to take to allow the server to run R code. You can either follow these steps yourself or get the assistance of the database administrator.

Open the SQL Server Management Studio. Note that you can install SQL Server Management Studio directly from the installation media. Connect to the instance where you installed R Services (In-Database), which is by default the "Default Instance," and then type and run (Press

the **F5** key) the following commands to turn on R Services:

```
exec sp_configure  'external scripts enabled', 1
reconfigure  with  override
Restart the SQL Server service for the SQL Server
instance, using the Services applet in the Windows
Control Panel, or by using SQL Server Configuration
Manager. Once the service restarts, you can check to
make sure the setting is enabled by running this
command in SSMS:
exec sp_configure  'external scripts enabled'
```

Now you can run a simple R script within SQL Server Management Studio:

```
exec sp_execute_external_script  @language =N'R',
@script=N'OutputDataSet<-InputDataSet',
@input_data_1 =N'select 1 as helloworld'
with result sets (([helloworld] int not null));
go
```

Client

When you install the R Services for SQL Server, the server contains the Microsoft R environment, including a client. However, you'll most often use a local client environment to develop and use your R code, separate from the server.

You can use a set of ScaleR functions to set the *compute context* to instruct the code to run on the SQL Server instance. This method makes it possible for the data professional to use the

power of the SQL Server 2016 system to compute the data, with the added performance benefits of enhanced scale and putting the compute code directly over the data.

To set the compute context, you'll need the Microsoft R Client software installed on the developer or data scientist's workstation. You can learn more about how to do that and more about the ScaleR functions at https://msdn.microsoft.com/microsoft-r/install-r-client-windows?tduid=%2874674bbb9257612d8927ec3c206c5172%29%28256380%29%282459594%29%28TnL5HPStwNw-VRuyHJhNp2D7.E7Jtg1Fiw%29%28%29&f=255&MSPPError=-2147217396.

When you install the Microsoft R Client, whether remotely or on the server, several base packages are included by default (https://mran.microsoft.com/rro/installed/):

- stats
- graphics
- grDevices
- utils

- datasets
- methods
- base

Some packages (listed here) are included, but not loaded at startup. tools

- compiler
- parallel
- splines
- tcltk
- grid

To load these packages, use the following command:

```
library("packagename")
```

Another method is to develop your R code locally and then send it to the database administrator or developer to incorporate into a solution as a stored procedure—this is code that runs in the context of the SQL Server engine. We'll explore this more in a moment.

You have many client software options for writing and executing R code. Let's take a quick

look at how to set up each of these to perform the examples in this book.

Microsoft R Client

The Microsoft R Client contains a full R environment, similar to installing open-source R from CRAN. It also contains the Microsoft R *ScaleR* functions that not only increase performance for many R operations, but make it possible for you to set the compute context of the code to run on the Microsoft R Server or SQL Server R Services. You can read more about that function at https://msdn.microsoft.com/microsoft-r/scaler/rxcomputecontext. If you're using a client such as RStudio or R Tools for Microsoft Visual Studio, you'll want to install this software so that you can have your code execute on the server and return the results to your client workstation.

If you want the SQL Server instance to process the R code directly within T-SQL, you have two choices. Your first option is to use "Dynamic SQL" statements, which means that the client software (such as SQL Server Management Studio or SQL Server Data Tools in Visual Studio

or some other SQL Server client tool) simply sets the language for interpretation by using the `sp_execute_external_script @language =N'R'` internal stored procedure in SQL Server. The second, more common, option is to write a stored procedure in SQL Server that contains those calls to the R code, as you'll see demonstrated later in this book. You can find a more complete explanation at https://msdn.microsoft.com/library/mt591996.aspx

RStudio

You can use the RStudio environment to connect to Microsoft R Server as well as to SQL Server and SQL Server with R Services. You'll require the Microsoft R Client software (see the previous subsection) if you want to interact directly with SQL Server R Services or MRS.

You also can create and edit R scripts locally and then send scripts to the SQL Server development team in your organization to include within the body of a T-SQL stored procedure. If you follow the latter route, you'll need to assist that team in making the changes for using SQL Server for input data frames to your R code, obtaining the data you want from SQL Server, and other

changes to make full use of the R environment in SQL Server.

More info You can read more about this latter approach from the RStudio team by going to https://support.rstudio.com/hc/articles/214510788-Setting-up-R-to-connect-to-SQL-Server-.

R Tools for Visual Studio

Visual Studio is Microsoft's development environment for almost any programming language. It contains a sophisticated Integrated Development Environment (IDE), team integration features using Git or Team Foundation Server (among others), and is highly extensible and configurable. There are paid and free editions, each with various capabilities. For this book and in many production environments, the free Community Edition is the right choice.

Microsoft has created a set of tools called R Tools for Visual Studio (RTVS) with which you can work within the R environment, both locally and by using the Microsoft R Server and SQL Server R Services. RTVS also can configure your Visual Studio environment to have similar

shortcuts to RStudio, if you are familiar with that environment.

You can follow a simple step-by-step installation guide for the free Community Edition of Visual Studio with R Tools at https://www.visualstudio.com/features/rtvs-vs.aspx. And there's a video you can watch for a class on using RTVS: https://channel9.msdn.com/Events/Build/2016/B884.

More info You can learn more about Visual Studio at https://msdn.microsoft.com/library/dd831853(v=vs.140).aspx.

SQL Server Management Studio

SQL Server Management Studio (SSMS), which you can install on the SQL Server or on a client machine, is a management and development environment for SQL Server. You can find the installation for SSMS at https://msdn.microsoft.com/library/mt238290.aspx

SSMS works in a *connected* fashion, which means that you connect to an instance of SQL Server prior to running the code, sending T-SQL code,

or interactively as you navigate the various objects in SQL Server represented graphically. You can create stored procedures using SSMS that contain R Code.

For a walk-through of SSMS, visit https://msdn.microsoft.com/library/bb934498.aspx.

More info To read more on this method of interaction with SSMS and R code, go to https://msdn.microsoft.com/library/mt591996.aspx.

SQL Server Data Tools

SQL Server Data Tools (SSDT) is another extension to Visual Studio. It works in a *disconnected* fashion to the SQL Server instance, which means that you can develop and test T-SQL code locally (it includes an Express Edition of SQL Server) and then deploy that solution to SQL Server after your testing is complete, or incorporate your code changes into a version control system such as Git or Team Foundation Server.

You follow the same process for working in this manner as you would in SQL Server

Management Studio, but you need to upgrade the SQL Server Express Edition to 2016 to obtain an R environment for local development.

More info You can find out more about SSDT at http://msdn.microsoft.com/data/tools.aspx.

Making your solution operational

As mentioned earlier, you have two options for using R code with SQL Server R Services. The first option is to use your local client to create R scripts that will call out to SQL Server R Services and use the compute and data resources on that system to obtain data, run the R code, and return the results to the local workstation. The second option is to include the R code in SQL Server stored procedures, which are stored and run on the SQL Server.

Using SQL Server R Services as a compute context

The process you follow for using SQL Server R Services as your compute context is largely the same as your normal R development process. You will, however, need to install the Microsoft R

Client software so that you have the Microsoft R ScaleR functions that can send code to a Microsoft SQL Server with R Services system for execution and processing.

You'll then create a connection to the SQL Server R Services instance, and then you can use the ScaleR library to access it. Depending on the code you run, you might need to create a local location to store temporary data. You'll see this in examples in this book and on the Microsoft documentation sites.

The remote functions in the ScaleR library also give you the ability process T-SQL code remotely, and allows those calls to interact with the R code. Following are the primary functions you'll use with SQL Server and a remote Microsoft R Client:

- `rxSqlServerTableExists` Checks for the existence of a database table or object.
- `rxExecuteSQLDDL` Execute a command to define, manipulate, or control SQL data objects, such as a table. This function does not return data.
- `RxSqlServerData` This function defines a SQL Server data source object—this is the primary method to return data to your R code from SQL Server.

After you have the data object, you can use it as a data source. The primary functions for that are listed here:

- `rxOpen` Opens a data source for reading.
- `rxReadNext` Reads data from a source.
- `rxWriteNext` Writes data to the target.
- `rxClose` After you run your code, use this function to close the data source and release the resources it has been using.

To use the SQL Server R Services with the data, you create and manage the compute context. Here are the primary functions to do that:

- `RxComputeContext` Create a compute context.
- `rxInSqlServer` Generates a SQL Server compute context that lets ScaleR functions run on SQL Server R Services.
- `rxGetComputeContext` Shows you the current compute context.
- `rxSetComputeContext` Sets which compute context to use so that your code can switch between local and server operations, or even other MRS or SQL Server with R Services systems.

To read the full documentation on each of these functions, which you'll see used throughout this book, go to https://msdn.microsoft.com/library/mt732681.aspx.

Let's see an annotated R example of how this would work in a simple script. You'll see more complex examples later in the book. This example connects to a SQL Server R Services instance, runs a T-SQL statement using that server, and then returns the data into a variable:

```
# Create a variable for the SQL Server Connection
String
connStr <- "Driver=SQL
Server;Server=ServerName;Database=DatabaseName;Uid=U
serName;Pwd=Password"
# Create a variable to store the data returned from
the SQL Server, with the user's name,
# a variable for the parameters to pass to the SQL
Server,
# the values you can pass to the RxSQLServerdata
constructor
sqlShareDir <-
paste("C:\\temp\\",Sys.getenv("USERNAME"),sep="")
sqlWait <- TRUE
sqlConsoleOutput <- FALSE
# Now we'll set the compute context for the data
object, using all the variables
# we just created.
cc <- RxInSqlServer(connectionString = connStr,
shareDir = sqlShareDir, wait = sqlWait,
consoleOutput = sqlConsoleOutput)
# Next we can set the compute context to point to
SQL Server R Services, defined earlier.
rxSetComputeContext(cc)
# We can then construct the T-SQL query. This one
simply brings back three columns.
sampleDataQuery <- "select Col1, Col2, Col3 from
```

```
MyTableName"
# Finally we run the query, using all of the objects
set up in the script.
# Note that we're using a colClasses variable to
convert the data types to something
# R understands, since SQL Server has more datatypes
than R, and we're reading 500 rows
# at a time.
inDataSource <- RxSqlServerData(sqlQuery =
sampleDataQuery, connectionString = connStr,
colClasses = c(Col1 = "numeric", Col2 = "numeric",
Col3 = "numeric"), rowsPerRead=500)
```

You now can use the `inDataSource` object obtained from SQL Server R Services in your R code for further processing.

Using stored procedures with R Code

Another method that you can use to operationalize your solution is to take advantage of SQL Server stored procedures. Stored procedures in SQL Server are similar to code-block type procedures in other languages. You can either develop the stored procedures yourself or work with the data programming team to incorporate your R code into the business logic in the application that uses SQL Server stored procedures.

Note If you're new to SQL Server stored procedures, you can learn more about them at https://msdn.microsoft.com/en-

us/library/ms187926.aspx?f=255&MSPPError=-2147217396.

In general, your stored procedure will perform the following steps:

1. Call the external script SQL Server stored procedure and set the language to R.
2. Set a variable for the R code.
3. Call input data from SQL Server by using T-SQL.
4. Return data from the R code operation.

Here's an annotated example. Let's assume that you have a table called "MyTable" with a single column of integers. You want to pass all of the data into an R script that simply returns the same data, but with a different column name:

```
-- Call the external script execution – note, must
be enabled already
execute sp_execute_external_script
-- Set the language to R
      @language = N'R'
-- Set a variable for the R code, in this case
simply making output equal to input
    , @script = N' OutputDataSet <- InputDataSet;'
-- Set a variable for the T-SQL statement that will
obtain the data
    , @input_data_1 = N' SELECT * FROM MyTable;'
-- Return the data – in this case, a set of integers
with a column name
    WITH RESULT SETS (([NewCollumnName] int NOT
NULL));
```

There are many more complex operations that you can perform in this manner, which you can read about at https://msdn.microsoft.com/library/mt591996.aspx.

In the scenarios that follow, you'll see a mix of these methods to develop, deploy, and use your solution. First, let's take a look at how you can use the data-science process to create an end-to-end solution.

CHAPTER 3

An end-to-end data science process example

In this chapter, we take you on a systematic walk-through for performing data science and building intelligent applications using Microsoft SQL Server R Services. You'll see a sequence of steps for developing and deploying predictive models using the R and Transact-SQL (T-SQL) environments.

The data science process: an overview

A data science process for building and deploying a predictive solution typically involves the following steps (see also Figure 3-1):

1. Defining the business problem, identifying the technologies suitable for the solution, and establishing key performance indicators (KPIs) for measuring success of the solution.
2. Planning and preparing the platform and environment on which the solution will be built (for example, SQL Server R services).
3. Data ingestion from a source to the environment. (Data cleansing is often needed.) Considerations for data ingestion include the following:
    - Data: on-premises or cloud; database or files; small, medium, and big data
    - Pipeline: streaming or batch; low or high frequency
    - Format: structured or unstructured; data validation and clean-up
    - Analytics: on-premises or cloud; database or data lake

4. Exploratory data analysis, summarization, and visualization. Methods can include the following:

    - Data dimensions, types, statistical summary, missing values
    - Distribution, histogram, boxplot, relationships, and so on
    - Statistical significance (t-test), fit (chi-squared test), and so on

5. Identifying the dependent (target) and independent variables (also referred to as predictors or features). Generating and/or selecting the features on which a predictive model will be created.

6. Creating predictive models using statistical and/or machine learning algorithms. Evaluating such models for accuracy. If the accuracy is not appropriate for deploying the model, you can reiterate steps 4, 5, and 6.

7. Saving and deploying the model into a predictive service for consumption.

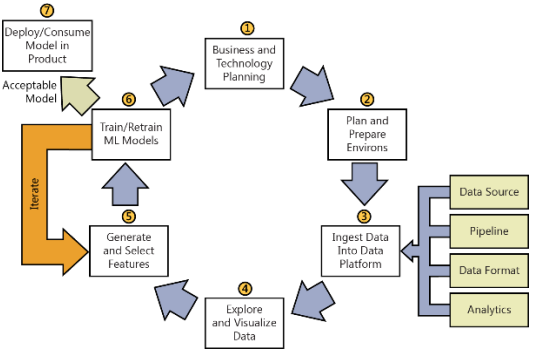


Figure 3-1: Lifecycle of a data science process.[2]

SQL Server R Services provides a platform and environment for building and deploying predictive services. In this chapter, we cover steps 1 through 7 of the data science process. You can modify this walk-through to fit your own business scenarios, datasets, and predictive tasks.

[2] Data science process: https://azure.microsoft.com/documentation/articles/data-science-process-overview

Note Much of the detail of this process is published online[3,4] so that you can download the specific code if you like.

The data science process in SQL Server R Services: a walk-through for R and SQL developers

In the first two chapters in this book, we introduced you to the power of R in SQL Server. We also covered the process for installing SQL Server R Services as well as a discussion of the client environment and the tools that you can use. In this chapter, we put those tools to use and show you a walk-through that R professionals and SQL developers can follow.[2,3] Wherever an activity can be performed by using

[3] Data Science End-to-End Walkthrough for R Developers: https://msdn.microsoft.com/en-us/library/mt612857.aspx

[4] In-Database Advanced Analytics for SQL Developers: https://msdn.microsoft.com/library/mt683480.aspx

R scripts in an R development environment or by using T-SQL with tools such as SQL Server Management Studio (SSMS), both approaches are shown. You can download the R and SQL scripts we show you here from a public GitHub repository,[5] where they are described in detail.[2,3]

Data and the modeling task

The first step of a data science process is to clearly understand the problem you're trying to solve. In this example, we want to predict whether a taxi driver in New York City will be given a tip, based on features such as trip distance, pickup time, number of passengers, and so on.

To accomplish that goal, we move on to the next steps in the data science process. We'll set up our environment, tools, and the servers we need to create the model. Next we'll vet and obtain a representative set of data to create the model.

---

[5] Public GitHub repository for walk-through scripts: https://github.com/Azure/Azure-MachineLearning-DataScience/tree/master/Misc/RSQL

Data: New York City taxi trip and fare

The data we'll use is a representative sampling of the *2013 New York City taxi trip and fare* dataset, which contains records of more than 173 million individual trips in 2013, including the fares and tip amounts paid for each trip. For more information about this data, go to http://chriswhong.com/open-data/foil_nyc_taxi.

To make the data easier and faster to work with for this example, we'll sample it to get just one percent of the data. This data has been shared in a public blob storage container in Microsoft Azure, in .csv format (http://getgoing.blob.core.windows.net/public/nyctaxi1pct.csv). The source data is an uncompressed file, just a little less than 350 MB in size. This will make it a bit quicker for you to download and follow along with this example.

It's important to understand that although we have a ready-made set of data to work with, this isn't often the case. According to recent polls,[6] a majority of the data scientist's time is spent on

[6] "For Big-Data Scientists, 'Janitor Work' is Key Hurdle to Insights." The *New York Times*. http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html.

finding, curating, vetting, obtaining, and cleaning source data for a model. That's where another advantage to working with SQL Server comes into play. The platform contains a very comprehensive, mature data sourcing and conditioning environment called SQL Server Integration Services (SSIS) that you can use to source and transform your data. This relieves a lot of the work that your R code previously had to do—although that's still an option, of course.

More info You can learn more about SQL Server Integration Services at https://msdn.microsoft.com/library/ms141026.aspx.

Modeling task: predicting whether a trip was tipped

The modeling task at hand is to predict whether a taxi trip was tipped (a binary, 1 or 0 outcome), based on *features* such as distance of the trip, the duration of the trip, number of passengers in the taxi for that trip, and other factors. Features are columns of data that have a potential relationship to another column, frequently referred to as the *Label or Target*—the answer that we are looking for. The dataset we have contains past information about the trips, the

passengers, and other data (features), and it includes the tip (the label).

Preparing the infrastructure, environment, and tools

Now we're ready to move on to the steps necessary for creating the infrastructure and environment for executing a data science process on SQL Server R Services.

SQL Server 2016 with SQL Server R Services

You must have access to an instance of SQL Server 2016 with SQL Server R Services installed[7] (see Chapter 2). You must be using SQL Server 2016 CTP3 or later. Previous versions of SQL Server do not support integration with R, though you can use SQL databases as an Open Database Connectivity (ODBC) data source. You must have a valid sign-in on the SQL database server for creating tables, loading data, and querying.

[7] Set up SQL Server R Services, https://msdn.microsoft.com/library/mt696069.aspx

SSMS

SSMS[8] should be installed on your testing environment—if you set up the Developer Edition, you'll probably run this code there, so you should install SSMS as described in Chapter 2. SSMS is an integrated environment for accessing, configuring, managing, administering, and developing all components of SQL Server. SSMS combines a broad group of graphical tools with a number of rich script editors to provide developers and administrators of all skill levels access to SQL Server. If SSMS is also installed on the client workstation, you can connect to SQL Server from the client machine using SSMS and run SQL scripts to perform database activities.

R integrated development environment

For development in R, you will need a suitable R integrated development environment (R-IDE) or command-line tool that can run R commands, such as Microsoft R-client (see Chapter 2),

[8] SSMS: https://msdn.microsoft.com/en-us/library/mt238290.aspx

Rstudio,[9] or R-Tools for Visual Studio (RTVS).[10] Using these tools, you can connect to an instance of SQL Server (with a valid sign-in with appropriate privileges) and run R scripts. You also can use the one at C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\R_SERVICES\bin\x64.

In this directory, clicking the **Rgui.exe** icon will invoke the Microsoft R Server; you can use this as the environment for developing and running R code.

Note Development and testing of the actual R code is often performed by using an R-IDE rather than SSMS. If the R code that you embed in a stored procedure has any problems, the information that is returned from the stored procedure might not be descriptive enough of the R steps for you to understand the root cause of the error. However, after the solution has been created, you can easily deploy it to SQL Server by using T-SQL stored procedures via SSMS.

---

[9] RStudio installation: https://www.rstudio.com/products/RStudio

[10] R tools for visual studio (RTVS): https://www.visualstudio.com/en-us/features/rtvs-vs.aspx

If you are using R-IDE on a client machine, your client will need to have an installation of the Microsoft R Server (https://www.microsoft.com/cloud-platform/r-server) as described in Chapter 2. The version of Microsoft R Server on your client will need to be compatible with the one installed on the SQL Server with R services.

R libraries

We need a few additional R libraries for this example. If you are using a client, you'll install these packages both on the client and on the SQL Server:

- **R libraries on the client** On the client, run the following to install the required libraries for this example:

```
if (!('ggmap' %in%
rownames(installed.packages()))){
  install.packages('ggmap')
}
if (!('mapproj' %in%
rownames(installed.packages()))){
  install.packages('mapproj')
}
if (!('ROCR' %in%
rownames(installed.packages()))){
  install.packages('ROCR')
}
if (!('RODBC' %in%
rownames(installed.packages()))){
  install.packages('RODBC')
}
```

- **R libraries on the SQL Server** On the SQL Server instance, open the Rgui.exe tool as an administrator. If you have installed SQL Server R Services using the defaults, you can find RGui.exe at C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\R_SERVICES\bin\x64.

At an R prompt, run the following R commands:

```
install.packages("ggmap", lib=grep("Program Files",
.libPaths(), value=TRUE)[1])

install.packages("mapproj", lib=grep("Program
Files", .libPaths(), value=TRUE)[1])
```

SQL server compute context

Typically, when you are using R, all operations run in memory on your computer. However, in R Services (in-database) you can specify that R operations take place on the SQL Server instance, which might have much more memory and other compute resources. You can do this by defining and then using a compute context. The compute context is by default set to "local," until you specify otherwise (for example, a SQL Server). When using T-SQL from within a SQL Server, the compute context is by default SQL Server.

The script that follows shows how to set a SQL Server compute context and define data objects and data sources when using an R-IDE. For this, you will need to ensure that your R development environment is using the library that includes the `RevoScaleR` package, and then load the package.

> **Note** The exact path will depend on the version of R services that you are using.

```
# Set the library path
.libPaths(c(.libPaths(),"C:\\Program
Files\\Microsoft SQL
Server\\MSSQL13.MSSQLSERVER\\R_SERVICES \\library"))

# Load RevoScaleR package
library(RevoScaleR)

# Define the connection string
# This walkthrough requires SQL authentication
connStr <- "Driver=SQL
Server;Server=<SQL_instance_name>;Database=<database
_name>;Uid=<user_name>;
    Pwd=<user password>"

# Set ComputeContext
sqlShareDir <-
paste("C:\\AllShare\\",Sys.getenv("USERNAME"),sep=""
)
sqlWait <- TRUE
sqlConsoleOutput <- FALSE
cc <- RxInSqlServer(connectionString = connStr,
shareDir = sqlShareDir,
                    wait = sqlWait, consoleOutput =
sqlConsoleOutput)
rxSetComputeContext(cc)
```

Now, we have our infrastructure and environments set up; let's move forward with the subsequent steps of the data science process.

Scripts for creating tables, stored procedures, and functions

There's another tool, called Windows PowerShell, that you can use to set up your data science environment and automate the entire process. Windows Powershell is Microsoft's scripting environment, like Perl but much more powerful and versatile—Windows Powershell also can work with any of the .NET libraries in the Microsoft ecostructure as well as just about anything that runs in the Microsoft Windows environment. You don't need to install anything to make this work on your server: all modern versions of Windows come with Windows PowerShell installed by default.

> **Note** Although you don't need to learn all about Windows PowerShell for this example, if you'd like to explore it further, go to https://technet.microsoft.com/library/bb978526.aspx.

Downloading scripts and data

For this example, we've provided Windows PowerShell and T-SQL scripts to download the

data and perform the necessary SQL Server operations, create the necessary tables, and load the data into SQL Server.[3,4,5] The Windows PowerShell script RunSQL_R_Walkthrough.ps1 uses other T-SQL scripts to create the database, tables, stored procedures, and functions—it even loads data into the data table.

On the computer where you are doing development—typically a client workstation with R-IDE installed—open a Windows PowerShell command prompt as an administrator. If you have not run Windows PowerShell before on this instance, or you do not have permission to run scripts, you might encounter an error. If so, run the following command before running the script, to temporarily allow scripts without changing system defaults:

```
Set-ExecutionPolicy Unrestricted -Scope Process -
Force
```

Run the command that follows (see Figure 3-2) to download the script files to a local directory. If you do not specify a different directory, by default folder c:\tempR is created and all files are saved there. If you want to save the files to a different directory, edit the values of the parameter `DestDir` to a folder on your computer. If you specify a folder name that does not exist, the Windows PowerShell script will create the folder for you.

```
$source =
'https://raw.githubusercontent.com/Azure/Azure-
MachineLearning-
DataScience/master/Misc/RSQL/Download_Scripts_R_Walk
through.ps1'
$ps1_dest =
"$pwd\Download_Scripts_R_Walkthrough.ps1"
$wc = New-Object System.Net.WebClient
$wc.DownloadFile($source, $ps1_dest)
.\Download_Scripts_R_Walkthrough.ps1 –DestDir
'C:\tempR'
```

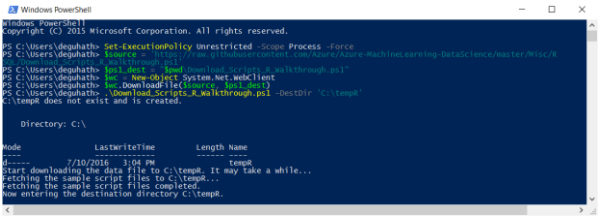


Figure 3-2: Windows PowerShell commands for downloading scripts and data for the end-to-end data science walk-through.

After you download and run this script and sign in to the SQL Server by using SSMS, you'll see the database, tables, functions, and stored procedures that were created (Figure 3-3). These tables and functions are used in subsequent steps of the walk-through.

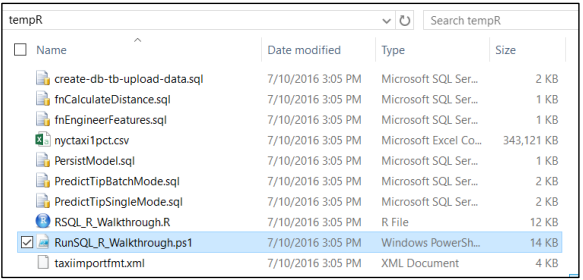


Figure 3-3: A list of files downloaded after running the Windows PowerShell script. The files contain data to be loaded to the database (nytaxi1pct.csv), several SQL (.sql) script files, and an R-script (.R) file.

Creating tables, stored procedures, and functions

To set up the SQL Server data, run the Windows PowerShell script RunSQL_R_Walkthrough.ps1 (highlighted in Figure 3-3). This script creates the tables, stored procedures, and functions that you need to prepare the model. Figure 3-4 shows the resulting RSQL_Walkthrough database. Unless specified in the command line as options, the script will prompt the user to input the database name, password, and path to the data file (nyctaxi1pct.csv) to be loaded. By default, we're connecting to SQL Server using the Named Pipes protocol.

The script performs these actions:

- Checks whether the SQL Native Client and command-line utilities for SQL Server are installed
- Connects to the specified instance of SQL Server and runs some T-SQL scripts that configure the database and create the tables for the model and data
- Runs a SQL script to create several stored procedures
- Loads the data you downloaded previously into the table `nyctaxi_sample`
- Rewrites the arguments in the R script file to use the database name that you specify

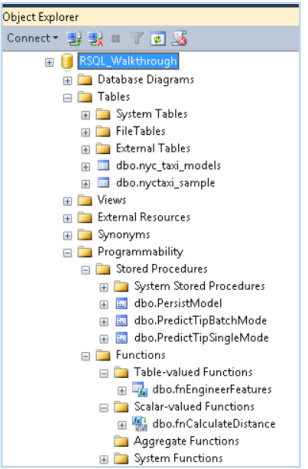


Figure 3-4: Tables, stored procedures, and functions that are created in the database after running the Windows PowerShell script.

The following tables, stored procedures, and functions are created in the database:

Tables:

- `nyctaxi_sample` Contains the main NYC Taxi dataset. A clustered columnstore index is added to the table to improve storage and query performance. The one-percent sample of the NYC Taxi dataset will be inserted into this table.

- `nyc_taxi_models` Used to persist the trained models.

Stored procedures:

- `PersistModel` Called to save a trained model. The stored procedure takes a model that has been serialized in a varbinary data type and writes it to the specified table.
- `PredictTipBatchMode` Calls the trained model to create predictions using the model. The stored procedure accepts a query as its input parameter and returns a column of numeric values containing the scores for the input rows.
- `PredictTipSingleMode` Calls the trained model to create predictions using the model. This stored procedure accepts a new observation as input, with individual feature values passed as in-line parameters and returns a value that predicts the outcome for the new observation.

Functions:

- `fnCalculateDistance` Creates a scalar-valued function that calculates the direct distance between pickup and dropoff locations.

- `fnEngineerFeatures` Creates a table-valued function that creates new data features for model training.

An example of running the script with parameters is presented here:

```
.\RunSQL_R_Walkthrough.ps1 -server
SQLinstance.subnet.domain.com -dbname MyDB –u
SqlUserName –p SqlUsersPassword -csvfilepath
C:\tempR\nyctaxi1pct.csv
```

The preceding example does the following:

- Connects to the specified instance and database using the credentials of `SqlUserName`.
- Gets data from the file C:\tempR\nyctaxi1pct.csv.
- Loads the data in nyctaxi1pct.csv into the table `nyctaxi_sample`, in the database `MyDB` on the SQL Server instance named `SQLinstance`.

Note If the database objects already exist, they cannot be created again. If a table already exists, data will be appended, not overwritten. Therefore, be sure to drop any existing objects before running the scripts.

Input data and SQLServerData object

Now, let's look at the input data and the data objects that we'll use for building our models.

In the example that follows, you can see the T-SQL script used to create the table (see Figure 3-5) for hosting the NYC Taxi data, called `nyctaxi_sample`. In the script, the binary classification target column (tipped, dependent variable with binary 0 or 1 values—our label) is highlighted.

```
-- Create nyctaxi_sample table
       CREATE TABLE [dbo].[nyctaxi_sample](
        [medallion] [varchar](50) NOT NULL,
        [hack_license] [varchar](50) NOT NULL,
        [vendor_id] [char](3) NULL,
        [rate_code] [char](3) NULL,
        [store_and_fwd_flag] [char](3) NULL,
        [pickup_datetime] [datetime] NOT NULL,
        [dropoff_datetime] [datetime] NULL,
        [passenger_count] [int] NULL,
        [trip_time_in_secs] [bigint] NULL,
        [trip_distance] [float] NULL,
        [pickup_longitude] [varchar](30) NULL,
        [pickup_latitude] [varchar](30) NULL,
        [dropoff_longitude] [varchar](30) NULL,
        [dropoff_latitude] [varchar](30) NULL,
        [payment_type] [char](3) NULL,
        [fare_amount] [float] NULL,
        [surcharge] [float] NULL,
        [mta_tax] [float] NULL,
        [tolls_amount] [float] NULL,
        [total_amount] [float] NULL,
        [tip_amount] [float] NULL,
        [tipped] [int] NULL,
```

```
        [tip_class] [int] NULL
) ON [PRIMARY]
```

MLADS16SQLR.nyct...bo.nyctaxi_sample

Column Name	Data Type	Allow Nulls
medallion	varchar(50)	☐
hack_license	varchar(50)	☐
vendor_id	char(3)	☑
rate_code	char(3)	☑
store_and_fwd_flag	char(3)	☑
pickup_datetime	datetime	☐
dropoff_datetime	datetime	☑
passenger_count	int	☑
trip_time_in_secs	bigint	☑
trip_distance	float	☑
pickup_longitude	varchar(30)	☑
pickup_latitude	varchar(30)	☑
dropoff_longitude	varchar(30)	☑
dropoff_latitude	varchar(30)	☑
payment_type	char(3)	☑
fare_amount	float	☑
surcharge	float	☑
mta_tax	float	☑
tolls_amount	float	☑
total_amount	float	☑
tip_amount	float	☑
tipped	int	☑
tip_class	int	☑

Figure 3-5: The data table, `nyctaxi_sample`, where the sampled NYC Taxi data from http://getgoing.blob.core.windows.net/public/nyctaxi1pct.csv can be loaded.

Note In this table, `pickup_longitude`, `pickup_latitude`, `dropoff_longitude`, and

`dropoff_latitude` are loaded as `varchar(30)` data types. We will convert these data types to `float` for performing computations with these variables. For example, we'll use this in the query `sampleDataQuery` that defines the input data, `inDataSource`.

Next, let's look at the `SQLServerData` object. The `SQLServerData` data object combines a connection string with a data source definition. After the `SQLServerData` object has been created, you can use it as many times as you need, to get basic information about the data, to manipulate and transform the data, or for training a model with it. You can run the following scripts in an R-IDE to define a `SQLServerData` object by using a sample of the data from the `nyctaxi_sample` table:

```
# Define a DataSource with a query (sample 1% of
data and take 1000 observations
# from that sample)

sampleDataQuery <- " select top 1000 tipped,
tip_amount, fare_amount,
    passenger_count,trip_time_in_secs,trip_distance,
    pickup_datetime, dropoff_datetime,
    cast(pickup_longitude as float) as
pickup_longitude,
    cast(pickup_latitude as float) as
pickup_latitude,
    cast(dropoff_longitude as float) as
dropoff_longitude,
    cast(dropoff_latitude as float)  as
dropoff_latitude,
payment_type from nyctaxi_sample
tablesample (1 percent) repeatable (98052) "
```

```
ptypeColInfo <- list(
  payment_type = list(
    type = "factor",
    levels = c("CSH", "CRD", "DIS", "NOC", "UNK"),
    newLevels= c("CSH", "CRD", "DIS", "NOC", "UNK")
  )
)

inDataSource <- RxSqlServerData(sqlQuery =
sampleDataQuery, connectionString = connStr,
            colInfo = ptypeColInfo,
            colClasses = c(pickup_longitude =
"numeric", pickup_latitude = "numeric",
                           dropoff_longitude =
"numeric", dropoff_latitude = "numeric"),
                rowsPerRead=500)
```

Exploratory analysis

Developing a data science solution using our process includes data exploration and data visualization, prior to feature selection and building models. You need to understand the data prior to working with it. In the following steps, we'll summarize the sampled NYC Taxi data, and generate plots by using R functions.

Data summarization

You can apply enhanced R functions in SQL Server R Services to summarize data from a SQL Server data source by using the provided `ScaleR` libraries. In the following steps, we're using an R environment to run R scripts.

- `rxGetVarInfo` Use this function to get information such as the range of values, the variable types in columns, and the number of levels in factors in variable columns. You should consider running this function after any kind of data input, feature transformation, or feature engineering. By doing so, you can ensure that all of the variables are of the expected data types and are within expected ranges.

```
> rxGetVarInfo(data = inDataSource)

Output:
Var 1: tipped, Type: integer
Var 2: tip_amount, Type: numeric
Var 3: fare_amount, Type: numeric
Var 4: passenger_count, Type: integer
Var 5: trip_time_in_secs, Type: numeric,
Storage: int64
Var 6: trip_distance, Type: numeric
Var 7: pickup_datetime, Type: character
Var 8: dropoff_datetime, Type: character
Var 9: pickup_longitude, Type: numeric
Var 10: pickup_latitude, Type: numeric
Var 11: dropoff_longitude, Type: numeric
Var 12: dropoff_latitude, Type: numeric
Var 13: payment_type, Type: factor, no factor
levels available
```

- `rxSummary` Use this function to get detailed statistics about individual variables, compute summaries by factor levels, and to save the summaries. In the following example, statistical summaries for `fare_amount` are shown by `passenger_count`.

```
> rxSummary(~fare_amount:F(passenger_count,1,6),
data = inDataSource)

Output:
Call:
rxSummary(formula =
~fare_amount:F(passenger_count, 1, 6), data =
inDataSource)

Summary Statistics Results for:
~fare_amount:F(passenger_count, 1, 6)
Data: inDataSource (RxSqlServerData Data Source)
Number of valid observations: 1000

 Name                                  Mean
StdDev   Min Max ValidObs MissingObs
 fare_amount:F_passenger_count_1_6_T 11.294
7.409316 2.5 52  1000     0

Statistics by category (6 categories):

 Category
F_passenger_count_1_6_T Means    StdDev   Min
Max  ValidObs
 fare_amount for F(passenger_count,1,6,T)=1 1
11.26151 7.358224 2.5 52.0 717
 fare_amount for F(passenger_count,1,6,T)=2 2
11.40323 8.303608 4.0 52.0 124
 fare_amount for F(passenger_count,1,6,T)=3 3
11.45238 8.292525 4.0 41.5  42
 fare_amount for F(passenger_count,1,6,T)=4 4
11.58333 5.727257 6.5 28.0  18
 fare_amount for F(passenger_count,1,6,T)=5 5
10.90000 5.613093 4.0 28.0  45

fare_amount for F(passenger_count,1,6,T)=6 6
11.58333 7.289013 4.0 36.5  54
```

Data visualization

The saying goes that a picture is worth a thousand words, and in the case of exploring large sets of data, it's true. Looking at the layout

of the data in graphical form is a key part of exploring it.

Creating a histogram

To create a histogram, you can use the SQL Server data source you defined earlier, together with the `rxHistogram` function provided by SQL Server R Services. The `rxHistogram` function is one of many functions in the `RevoScaleR` package that provides functionality similar to that in open-source R packages, but with the ability to run in a remote compute context and with input data objects that are different from R data frames (such as a `SQLServerData` object). Running the following script from your R-IDE will create the histogram illustrated Figure 3-6 in the plot window.

```
# Plot fare amount on SQL Server and return the plot
to RStudio
> rxHistogram(~fare_amount, data = inDataSource,
title = "Fare Amount Histogram")
```

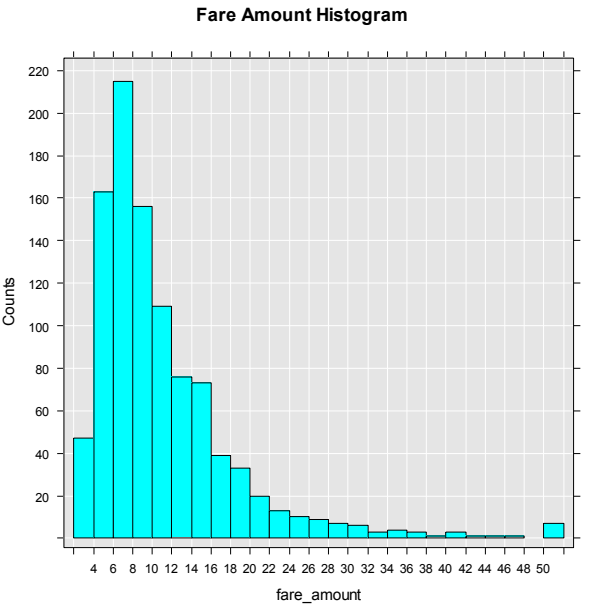


Figure 3-6: A histogram of fare amounts in the sampled NYC Taxi dataset that is loaded to the SQL Server.

Creating a ggmap plot

You also can generate a plot object by using the SQL Server instance as the compute context and then return the plot object to the R-IDE for rendering. It is important to note that for security reasons the SQL Server compute context typically does not have the ability to connect to the Internet and download the map

representation. So, to create these plots, you'll first generate the map representation in the R-IDE by calling an online map service, and then pass the map representation to the SQL context to overlay the points on the map that are stored as attributes (pickup latitudes and longitudes) in the `nyctaxi_sample` table.

> **Note** Many production database servers completely block Internet access. So, this is a pattern that you might find useful when developing your own applications.

The following example has three steps that you can run in an R-IDE:

1. Define the function that creates the plot object.

   The custom R function, `mapPlot`, creates a scatter plot of taxi pickup locations on a ggmap object, as shown in the following (note that it requires the `ggplot2` and `ggmap` packages, which you should have already installed and loaded):

```
mapPlot <- function(inDataSource, googMap){
    library(ggmap)
    library(mapproj)
    ds <- rxImport(inDataSource)
    p <- ggmap(googMap)+
    geom_point(aes(x = pickup_longitude, y
=pickup_latitude ), data=ds, alpha =.5,
color="darkred", size = 1.5)
```

```
    return(list(myplot=p))
}
```

The function `mapPlot` takes the following arguments and returns a ggmap plot object:

- An existing data object, which you defined earlier by using `RxSqlServerData`. This object has pickup latitudes and longitudes that are used to generate points on the two-dimensional map.
- The map representation—that is, the ggmap object—passed from the R-IDE.

2. Create the map object, as follows:

```
library(ggmap)
library(mapproj)
gc <- geocode("Times Square", source = "google")
googMap <- get_googlemap(center =
as.numeric(gc), zoom = 12, maptype = 'roadmap',
color = 'color');
```

Note We make repeated calls to the libraries `ggmap` and `mapproj` because the previous function definition (mapPlot) ran in the server context and the libraries were never loaded locally in the R-IDE; now you are bringing the plotting operation back to the R-IDE, which might be on a client.

The `gc` variable stores a set of coordinates for Times Square, NY.

3. Run the plotting function and render the results in your R-IDE. To do this, wrap the plotting function in `rxExec`.

```
myplots <- rxExec(mapPlot, inDataSource,
googMap, timesToRun = 1)
plot(myplots[[1]][["myplot"]]);
```

The plot with the rendered data points is serialized back to the local R environment that you can view in the plot window of the R-IDE or in its graphic output. The `rxExec` function is included in the `RevoScaleR` package; it supports execution of arbitrary R functions in the remote compute context. The output plot, with the pickup locations on the map marked with red dots, is shown in Figure 3-7.

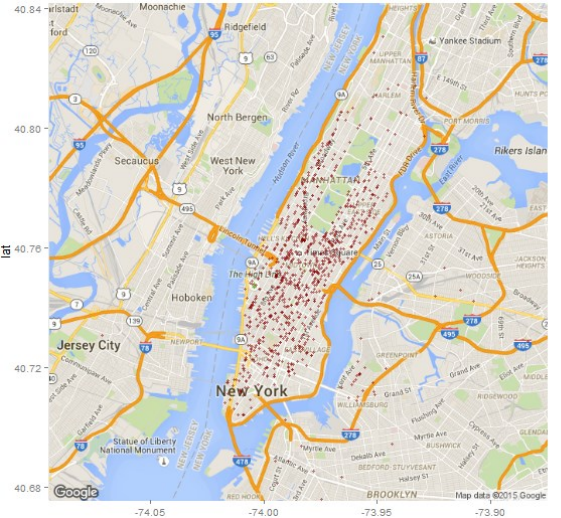


Figure 3-7: A plot showing pickup locations in **ggmap**. X-axis = longitude; Y-axis = latitude.

Creating a new feature (feature engineering)

You might not have all the features you need for your model, or they might be in multiple columns that need to be combined, or perhaps there are other data transformation tasks that you need for the proper columns to act as the features. *Feature engineering* is the process of generating transformed or new features from

existing ones; it is an important step before you use the data for building models.

For this task, rather than using the raw latitude and longitude values of the pickup and drop-off locations, you would like to derive the direct or linear distance in miles between the two locations. You can compute this by using the *haversine* formula. You can use two different methods for creating the feature:

Using R functions

The following R code defines a function, `ComputeDist`, that takes in two pairs of latitude and longitude values and then calculates the linear distance between them.

```
env <- new.env()
env$ComputeDist <- function(pickup_long, pickup_lat,
dropoff_long, dropoff_lat){
  R <- 6371/1.609344 #radius in mile
  delta_lat <- dropoff_lat - pickup_lat
  delta_long <- dropoff_long - pickup_long
  degrees_to_radians = pi/180.0
  a1 <- sin(delta_lat/2*degrees_to_radians)
  a2 <- as.numeric(a1)^2
  a3 <- cos(pickup_lat*degrees_to_radians)
  a4 <- cos(dropoff_lat*degrees_to_radians)
  a5 <- sin(delta_long/2*degrees_to_radians)
  a6 <- as.numeric(a5)^2
  a <- a2+a3*a4*a6
  c <- 2*atan2(sqrt(a),sqrt(1-a))
  d <- R*c
  return (d)
}
```

After you've defined the function, you can apply it to the data source to create the new feature, `direct_distance`. You can create the output feature data source, as follows:

```
featuretable = paste0("NYCTaxiDirectDistFeatures")
featureDataSource = RxSqlServerData(table =
featuretable,
colClasses = c(pickup_longitude = "numeric",
pickup_latitude = "numeric", dropoff_longitude =
"numeric",
dropoff_latitude = "numeric", passenger_count  =
"numeric",
trip_distance  = "numeric", trip_time_in_secs  =
"numeric",
direct_distance  = "numeric"), connectionString =
connStr)
```

You can then apply this function to the input data by using the `rxDataStep` function, provided in the `RevoScaleR` package:

```
# Create feature (direct distance) by calling
rxDataStep() function, which calls
# the env$ComputeDist function to process records
and output it along with other
# variables as features to the featureDataSource.
This will be the feature set
# for training machine learning models

rxDataStep(inData = inDataSource, outFile =
featureDataSource,  overwrite = TRUE,
           varsToKeep=c("tipped", "fare_amount",
"passenger_count", "trip_time_in_secs",
                "trip_distance", "pickup_datetime",
"dropoff_datetime",
                "pickup_longitude",
"pickup_latitude", "dropoff_longitude",
                "dropoff_latitude"),
transforms = list(direct_distance =
ComputeDist(pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude)),
```

```
transformEnvir = env, rowsPerRead = 500,
reportProgress = 3)
```

Note The `rxDataStep` function can modify data in place. The arguments include a character vector of columns to pass through (`varsToKeep`), and a list that defines transformations. Any columns that are transformed are automatically output and therefore do not need to be included in the `varsToKeep` argument. Alternatively, you can specify that all columns in the source be included except the specified variables, using `varsToDrop`.

The `rxDataStep` call in the preceding example will create a table called `NYCTaxiDirectDistFeatures` in the database. You can use this afterward for getting the input features for training models.

Finally, you can use `rxGetVarInfo` to inspect the schema of the new data source:

```
> rxGetVarInfo(data = featureDataSource)
```

```
Output:

Var 1: tipped, Type: integer
Var 2: tip_amount, Type: numeric
Var 3: fare_amount, Type: numeric
Var 4: passenger_count, Type: numeric
Var 5: trip_time_in_secs, Type: numeric
Var 6: trip_distance, Type: numeric
Var 7: pickup_datetime, Type: character
Var 8: dropoff_datetime, Type: character
```

```
Var 9: pickup_longitude, Type: numeric
Var 10: pickup_latitude, Type: numeric
Var 11: dropoff_longitude, Type: numeric
Var 12: dropoff_latitude, Type: numeric
Var 13: payment_type, Type: character
Var 14: direct_distance, Type: numeric
```

Using a SQL function

The code for this SQL user-defined function was provided as part of the Windows PowerShell script that you ran to create and configure the database. If you ran the Windows PowerShell script setup, this function should already exist in your database:

```
-- Create function for calculating distance
CREATE FUNCTION [dbo].[fnCalculateDistance] (@Lat1
float, @Long1 float, @Lat2 float, @Long2 float)
-- User-defined function calculate the direct
distance between two geographical
-- coordinates.
RETURNS float
AS
BEGIN
  DECLARE @distance decimal(28, 10)
  -- Convert to radians
  SET @Lat1 = @Lat1 / 57.2958
  SET @Long1 = @Long1 / 57.2958
  SET @Lat2 = @Lat2 / 57.2958
  SET @Long2 = @Long2 / 57.2958
  -- Calculate distance
  SET @distance = (SIN(@Lat1) * SIN(@Lat2)) +
(COS(@Lat1) * COS(@Lat2) * COS(@Long2 - @Long1))
  --Convert to miles
  IF @distance <> 0
  BEGIN
    SET @distance = 3958.75 * ATAN(SQRT(1 -
POWER(@distance, 2)) / @distance);
  END
  RETURN @distance
END
```

You can run this function from any application that supports T-SQL. For example, using SSMS you can run the following statement to generate the new `direct_distance` feature:

```
SELECT
tipped, fare_amount,
passenger_count,trip_time_in_secs,trip_distance,
pickup_datetime, dropoff_datetime,
dbo.fnCalculateDistance(cast(pickup_latitude as
float), cast(pickup_longitude as float),
cast(dropoff_latitude as float),
cast(dropoff_longitude as float)) as
direct_distance,
pickup_latitude, pickup_longitude, dropoff_latitude,
dropoff_longitude, cast(dropoff_longitude as float)
```

```
as a
FROM nyctaxi_sample
```

You also can use the function `fnCalculateDistance` in a SQL statement from the R-IDE to generate the new feature in SQL Server compute context. In the code that follows, the query `featureEngineeringQuery` uses the function `fnCalculateDistance` to compute direct distance. The query is used to define the `RxSqlServerData` object `featureDataSource`, which is used as input data for a creating a binary classification model, as shown in the next step:

```
# Do feature engineering through a SQL Query in R,
and save the new feature in a
# SQL Server data object, featureDataSource
ptypeColInfo <- list(
  payment_type = list(
    type = "factor",
    levels = c("CSH", "CRD", "DIS", "NOC", "UNK"),
    newLevels= c("CSH", "CRD", "DIS", "NOC", "UNK")
  )
)

# Alternatively, use a user defined function in SQL
to create features
# Sometimes, feature engineering in SQL might be
faster than R
# You need to choose the most efficient way based on
real situation
# Here, featureEngineeringQuery is just a reference
to the result from a SQL
# query.
# NOTE: 1% of sampled data is used to create the
featureDataSource. This can be
# increased to increase the amount of data for model
construction.
```

```
featureEngineeringQuery = "SELECT tipped,
fare_amount, passenger_count,
                                trip_time_in_secs,
trip_distance, pickup_datetime,
      dropoff_datetime,
      dbo.fnCalculateDistance(
      cast(pickup_latitude as float),
      cast(pickup_longitude as float),
      cast(dropoff_latitude as float),
      cast(dropoff_longitude as float)) as
direct_distance,
      pickup_latitude, pickup_longitude,
dropoff_latitude,
      dropoff_longitude, payment_type
   FROM nyctaxi_sample
      tablesample (1 percent) repeatable (98052)"

featureDataSource = RxSqlServerData(sqlQuery =
featureEngineeringQuery,
colInfo = ptypeColInfo, colClasses =
c(pickup_longitude = "numeric",
pickup_latitude = "numeric", dropoff_longitude =
"numeric",
dropoff_latitude = "numeric",
passenger_count  = "numeric",
trip_distance  = "numeric",
trip_time_in_secs  = "numeric",
direct_distance  = "numeric",
fare_amount="numeric"),
connectionString = connStr)
```

Creating and saving models

Now, we're ready to build and save a logistic regression model using the `rxLogit` function from the `RevoScaleR` library. `RevoScaleR` also contains additional functions (such as `rxBTrees` for boosted trees and `rxDForest` for random

forests) to create models for binary classification problems.

Using an R environment

The following code builds a logistic regression model using `tipped` (0/1) as the destination variable, and `passenger_count`, `trip_distance`, `trip_time_in_secs`, and `direct_distance` (an engineered feature) as predictor variables:

```
logitObj <- rxLogit(tipped ~ passenger_count +
trip_distance + trip_time_in_secs + direct_distance,
data = featureDataSource)
```

After you build the model, you can inspect it by using the `summary` function, as shown here:

```
> summary(logitObj)

Output:

Logistic Regression Results for: tipped ~
passenger_count + trip_distance + trip_time_in_secs
+ direct_distance

Data: featureDataSource (RxSqlServerData Data
Source)

Dependent variable(s): tipped

Total independent variables: 5

Number of valid observations: 17068

Number of missing observations: 0
-2*LogLikelihood: 23540.0602 (Residual deviance on
17063 degrees of freedom)

Coefficients:
```

```
Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.509e-03 3.223e-02 -0.078 0.93793
passenger_count -5.753e-02 1.088e-02 -5.289 1.23e-07
***
trip_distance -3.896e-02 1.466e-02 -2.658 0.00786 **
trip_time_in_secs 2.115e-04 4.336e-05 4.878 1.07e-06
***
direct_distance 6.156e-02 2.076e-02 2.966 0.00302 **
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '. '
0.1 ' ' 1
Condition number of final variance-covariance
matrix: 48.3933

Number of iterations: 4
```

You can serialize the trained model into a hexadecimal string and save it in the SQL Server for future use by other processes. Let's take a look at the code:

```
# Serialize a model
modelbin <- serialize(logitObj, NULL)
modelbinstr=paste(modelbin, collapse="")


-- Create a stored procedure for saving model in the
nyc_taxi_models table
CREATE PROCEDURE [dbo].[PersistModel]
@m nvarchar(max)
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result
sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    insert into nyc_taxi_models (model) values
(convert(varbinary(max),@m,2))
END

# Persist model by calling a stored procedure from
SQL
library(RODBC)
conn <- odbcDriverConnect(connStr )
```

```
q <- paste("EXEC PersistModel @m='",
modelbinstr,"'", sep="")
sqlQuery (conn, q)
```

Here's what it does:

- Creates a serialized object from the trained model.
- Creates a stored procedure, `PersistModel`, that inserts a `varbinary(max)` string into the table `nyc_taxi_models`.
- Calls the stored procedure with the serialized object to insert a row in the table named `nyc_taxi_models` in the SQL server.

Using T-SQL

Using T-SQL, you can create a stored procedure to build a model using the R Services in SQL Server. The stored procedure that follows defines the input data and creates a logistic regression model. All calls to the R runtime are done by using the system stored procedure, `sp_execute_external_script`.

```
-- Create a stored procedure for training and saving
model in nyc_taxi_models
-- table.
-- NOTE: Model is created from 1000 observations of
1% of sampled data in the
-- nyctaxi_sample table. This can be increased to
increase the amount of data
-- for model construction.
```

```
CREATE PROCEDURE [dbo].[TrainTipPredictionModel]


AS
BEGIN
  DECLARE @inquery nvarchar(max) = N'
    select top 1000 tipped, fare_amount,
passenger_count,trip_time_in_secs,trip_distance,
    pickup_datetime, dropoff_datetime,
    dbo.fnCalculateDistance(cast(pickup_latitude as
float),
 cast(pickup_longitude as float),
 cast(dropoff_latitude as float),
 cast(dropoff_longitude as float) ) as
direct_distance
            from nyctaxi_sample
            tablesample (1 percent) repeatable
(98052)
  -- Insert the trained model into a database table
  INSERT INTO nyc_taxi_models
  EXEC sp_execute_external_script @language = N'R',
                                  @script = N'

## Create model
logitObj <- rxLogit(tipped ~ passenger_count +
trip_distance + trip_time_in_secs + direct_distance,
data = InputDataSet)
summary(logitObj)

## Serialize model and put it in data frame
trained_model <-
data.frame(model=as.raw(serialize(logitObj, NULL)));
',
                                  @input_data_1 =
@inquery,

@output_data_1_name = N'trained_model' ;

END
```

In the preceding code, the stored procedure `TrainTipPredictionModel` performs the following activities as a part of model training:

- The SELECT query uses the custom scalar function `fnCalculateDistance` to calculate the direct distance between the pick-up and drop-off locations. The results of the query are stored in the default R input variable, `InputDataset`.
- The R script calls the `rxLogit` function from the `RevoScaleR` package to create the logistic regression model.
- The trained model, saved in the R variable `logitObj`, is serialized and put in a data frame for output to the SQL Server. That output is inserted into the database table `nyc_taxi_models` so that you can use it for performing predictions.

To create the R model using the stored procedure, simply run it via SSMS:

```
EXEC TrainTipPredictionModel
```

Note In the stored procedure `TrainTipPredictionModel`, the model is created from 1,000 observations of 1 percent of sampled data in the `nyctaxi_sample` table so that it runs quickly as an initial exercise. This can be raised to increase the amount of data for model construction. Typically, you would keep about two-thirds to four-fifths of the data

for training, and the remaining data for testing, which is called the "holdout data."

Model consumption: scoring data with a saved model

Now that the model is built and saved in the SQL Server, you can load and use it to score datasets to predict whether a driver is likely to get a tip on a future trip. For predictions, we can use the `rxPredict` function from the `RevoScaleR` package to create a *score*, and, as always, you can save the scored data back to a table in SQL Server. You can write and save the prediction or scoring functions as stored procedures. The prediction stored procedure can then simply be used from R-IDE using R-scripts, or from SSMS using T-SQL to score more data.

There are two different ways that you can call a model for scoring:

- **Batch scoring mode** This lets you create multiple predictions based on input from a SELECT query.
- **Individual or single scoring mode** This lets you create predictions one at a time, by passing a set of feature values for individual observations to the stored procedure, which

returns a single prediction value as the result.

We'll use the batch scoring mode only. For individual scoring, refer to Deploying or Operationalization.[1,2]

Using an R environment

You can use the model object created earlier, `logitObj`, to score datasets, and save the scored results in SQL Server. For this, you need to define the data object to use for storing the scored results:

```
# Create a SQL server data object to store scored
results
scoredOutput <- RxSqlServerData(
  connectionString = connStr,
  table = "taxiScoreOutput"
)
```

In this example, `taxiScoreOutput` is the table in which scored results are stored. Figure 3-8 show what the table looks like.

Column Name	Data Type	Allow Nulls
Score	float	☑
tipped	int	☑
passenger_count	int	☑
trip_distance	float	☑
trip_time_in_secs	bigint	☑
direct_distance	float	☑

Figure 3-8: The `taxiScoreOutput` table for saving scored results. Here the "Score" column indicates the

scored output (probability of receiving a tip), the rest of the columns are features (based on which the scores are generated), or the feature ("tipped").

> **Note** The schema for this table is not defined when you create it by using `rxSqlServerData`; rather, it is obtained from the `scoredOutput` object output from `rxPredict`. To create the table that stores the predicted values, the SQL sign-in running the `rxSqlServer` data function must have table creation privileges in the database. If the sign-in cannot create tables, the statement will fail.

Next, call the `rxPredict` function from the `RevoScaleR` package to score the input data, `featureDataSource`, and then insert the results into the `taxiScoreOutput` table:

```
# Predict using rxPredict
rxPredict(modelObject = logitObj, data =
featureDataSource, outData = scoredOutput,
                    predVarNames = "Score", type =
"response",
                    writeModelVars = TRUE, overwrite
= TRUE)
```

In another approach to consume the model, you can create a stored procedure (`PredictTipBatchMode`; see the code that follows) and use it for scoring from an R-IDE:

```
-- Create prediction stored procedure
CREATE PROCEDURE [dbo].[PredictTipBatchMode]
@inquery nvarchar(max)
AS
```

```
BEGIN

  DECLARE @lmodel2 varbinary(max) = (SELECT TOP 1
    model
  FROM nyc_taxi_models);
  EXEC sp_execute_external_script @language = N'R',
                                  @script = N'
mod <- unserialize(as.raw(model));
print(summary(mod))
OutputDataSet<-rxPredict(modelObject = mod, data =
InputDataSet, outData = NULL,
          predVarNames = "Score", type = "response",
writeModelVars = FALSE,
  overwrite = TRUE);
str(OutputDataSet)
print(OutputDataSet)',
                                  @input_data_1 =
@inquery,
                                  @params = N'@model
varbinary(max)',
                                  @model = @lmodel2
  WITH RESULT SETS ((Score float));

END
```

This stored procedure performs the following steps:

1. The SELECT statement gets a serialized model from the database and stores the model in the R variable, `mod`, for further processing using R.

2. The new cases to be scored are obtained from the T-SQL query specified in `@inquery`, the first parameter to the stored procedure. As the query data is read, the rows are saved in the default data frame, `InputDataSet`. This data frame is passed to the `rxPredict`

function available from the `RevoScaleR` library, which generates the scores.

3. `rxPredict` scores and returns results data to `OutputDataSet`, the default output data frame. The predicted values are floats representing the probability of a tip (of any amount) being given.

You can run the R script that follows in an R-IDE to query input data and score it by using the batch scoring stored procedure, `PredictTipBatchMode`. First, you need to define the input query:

```
# Define input data using a SQL query. Here we use
the table, NYCTaxiDirectDistFeatures,
# that was created earlier to store the engineered
feature, direct_distance.

input = "N' SELECT passenger_count,
trip_time_in_secs, trip_distance, direct_distance
            FROM NYCTaxiDirectDistFeatures '"
```

Using the input data obtained from the query, you can run the following R script to obtain scored data in the `scoredData` data frame.

```
# The query is sent to be executed with input data,
scored data frame is returned
q <- paste("EXEC PredictTipBatchMode @inquery = ",
input, sep="")
scoredData <- sqlQuery (conn, q)
```

Using T-SQL

With the stored procedure `PredictTipBatchMode`, you can score data by using T-SQL. First, you need to define the query string to pass into the stored procedure. The stored procedure will execute this query, get the data to be scored, and pass the data into the `rxPredict` function in the stored procedure for generating the predictions. The following code shows how to define the query for input data and use the stored procedure for scoring:

```
-- Specify input query
DECLARE @query_string nvarchar(max)
SET @query_string=' SELECT passenger_count,
trip_time_in_secs, trip_distance,
 direct_distance
                FROM NYCTaxiDirectDistFeatures '


-- Call stored procedure for scoring
EXEC [dbo].[PredictTipBatchMode] @inquery =
@query_string;
```

You also can insert the scored output into a table in the SQL Server (refer to the examples of how a serialized, trained model is inserted into the table `nyc_taxi_models`).

Evaluating model accuracy

Next, we need to ensure that the model we've chosen is delivering good results. For a binary classification problem, the accuracy of a model is frequently evaluated by using the *receiver operator curve* (ROC) and *area under the receiver operator curve* (AUC). You can use the `rxRocCurve` function from `RevoScaleR` to plot the ROC. You can run the plot in the SQL Server compute context and then return the plot to your R-IDE for rendering (see Figure 3-9). To do this from the R-IDE, run the following code:

```
# plot ROC curve
rxRocCurve( "tipped", "Score", scoredOutput)
```

You also can use the `rxImport` function from `RevoScaleR` library to import `scoredOutput` to a data frame in your client R environment. You can then call functions in the `ROCR` library to generate the plot (see Figure 3-9) and obtain the AUC.

```
# Import scoredOutput data into a data frame
scoredOutputDF = rxImport(scoredOutput)


# Plot ROC Curve using ROCR library
library('ROCR')
pred <- prediction(scoredOutputDF$Score,
scoredOutputDF$tipped)
perf <- performance( pred, "tpr", "fpr" )
plot( perf )
```

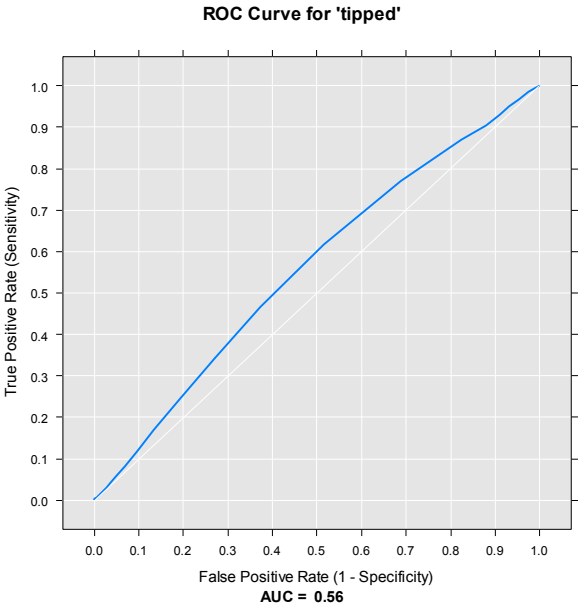


Figure 3-9: ROC plot created by using `rxRocCurve`; the AUC is 0.56.

In this case, the accuracy of the model is not very high (AUC is 0.56), so we probably want to consider exploring and adding other features when we create the model, or perhaps use more data for the training. As discussed in the data science process, some amount of iteration is often necessary before you get a deployable model.

Summary

In this chapter, we explained the data science process and explored how to use SQL Server R Services to perform its various steps. After determining the problem to solve, we moved on to data exploration, feature engineering, creating and training models, saving models, and scoring data from the model output. You learned how to use the R and T-SQL environments in this process—allowing the data scientist to integrate into the production data environment.

In the next few chapters, you'll see more examples, such as customer churn prediction, predictive maintenance, and forecasting—all following the data science process we've demonstrated here.

CHAPTER 4

Building a customer churn solution

One of the most canonical uses for predictive analytics is to predict customer churn. Customer churn is defined as the number of lost customers divided by the number of new customers gained. But, as long as you're gaining new customers faster than you're losing them, that's a good thing, right? No—for multiple reasons.

In this chapter, we show you how to build a customer churn model for telecommunication companies. You

will be able to take advantage of the same ideas and techniques and apply this to proactively managing customer churn for your business.

Overview

The primary reason why it is critical for companies to manage customer churn is that it costs far more to *gain* a new customer—or regain a lost one—than it does to *keep* an existing customer. Over time, too much customer churn can hurt the bottom line of a business. Identifying customer churn and the factors that causes it are essential tasks for a company to stay profitable.

Interestingly, customer churn extrapolates out to other uses, as well. Consider the case for which you want to manage churn for students and reduce the number of dropouts from a course.

In today's commercial environment, customers have many options for whatever they buy, and they can easily switch between service providers or vendors. This puts enormous pressure on organizations to retain their current customers.

Nowhere is this more evident than in the mobile phone market.

Improving customer attrition rates and enhancing a customer's experience are valuable ways to reduce customer acquisition costs and maintain a high-quality service. Many industries, including mobile providers, use churn models to predict which customers are most likely to leave, and to understand which factors cause customers to stop using their service.

Predicting customer churn is a mature exercise in data science, most notably in R. Customer transactional data is often stored in a Relational Database Management System (RDBMS) like Microsoft SQL Server. Many companies have already built applications for supporting their frontline staff. When the business need to imbue intelligence into these applications, it is important that staff members do not need to relearn a new application. In addition, database developers must operationalize a churn model and integrate its existing applications so that frontline staff can make smarter, better decisions. In Microsoft SQL Server 2016, you can now use R for training and operationalizing the churn model. This is difficult to implement using plain vanilla Transact-SQL (T-SQL), but now you can take advantage of the power of SQL Server and R to create intelligent applications.

When using R with SQL Server, all computations take place in the database, avoiding unnecessary data movement. When the R code is deployed to production, SQL Server R Services provides the best of the R and SQL worlds.

After the R script or model is ready for production use, the data scientist or a database developer can embed the code or model in a stored procedure, and developers can invoke that stored procedure from an application. The entire team stays in their familiar environments, and everyone can use the power of each system seamlessly.

In this chapter, we use a mix of R and T-SQL to generate scores from a predictive model in production and then return plots generated by R to present them in an application such as SQL Server Reporting Services or Microsoft Power BI.

Understanding the data

We've stored the data and code for this chapter at https://github.com/weehyong/telcocutomerchurn. Download the file telcoedw.zip from the GitHub repository and unzip the file. Then, restore the SQL Server 2016 backup to a SQL 2016 instance.

This solution uses data from two sources: Call Detail Records (CDR) log files and customer profile information.

Telephone companies have hardware called *call switches* that collect CDR data, which in this case have been exported as files. For each call a mobile user makes, a CDR record is generated and captured by the call switch (see Figure 4-1). A CDR record typically contains the following information: calling number, called number, date and time, duration of the call, completion status, source number, unique identifier for the SIM card of the phone, and often other data.

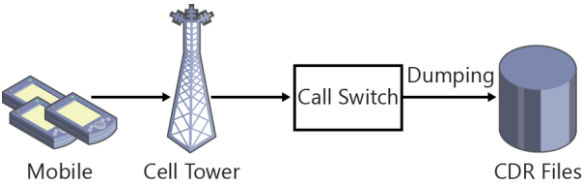


Figure 4-1: End-to-end flow, from placing a mobile call to CDR records.

Figure 4-2 shows a sample of the data in CDR files.

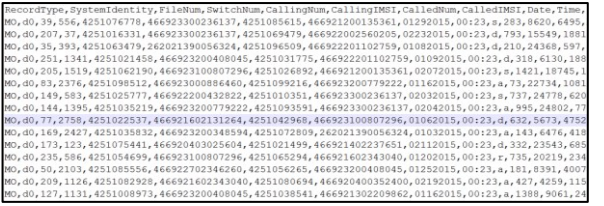


Figure 4-2: A sample of a CDR file.

After the call switch produces the data, the CDR file is loaded into a SQL Server table using a data integration tool such as SQL Server Integration Services (SSIS). After the data is loaded into the database, we can combine it with other customer information as part of the data preparation process.

Let's take a look at the data in the sample telco database by using the following T-SQL.

```
SELECT TOP 10 [age]
      ,[annualincome]
      ,[calldroprate]
      ,[callfailurerate]
      ,[callingnum]
      ,[customerid]
      ,[customersuspended]
      ,[education]
      ,[gender]
      ,[homeowner]
      ,[maritalstatus]
      ,[monthlybilledamount]
      ,[noadditionallines]
      ,[numberofcomplaints]
      ,[numberofmonthunpaid]
      ,[numdayscontractequipmentplanexpiring]
      ,[occupation]
      ,[penaltytoswitch]
```

```
      ,[state]
      ,[totalminsusedinlastmonth]
      ,[unpaidbalance]
      ,[usesinternetservice]
      ,[usesvoiceservice]
      ,[percentagecalloutsidenetwork]
      ,[totalcallduration]
      ,[avgcallduration]
      ,[year]
      ,[month]
      ,[churn]
FROM [telcoedw].[dbo].[edw_cdr]
```

From these results, we've aggregated the mobile customer usage by month and included other customer profile information such as gender, whether the customer is a home owner, marital status, and other features, as depicted in Figure 4-3.

Results | Messages

	age	annualincome	calldroprate	callfailurerate	callingnum	customerid	customersuspended	education	gender	homeowner	maritalstatus	monthlybilledamount
1	12	168147	0.06	0	...	1	Yes	Bachelor or equivalent	Male	Yes	Single	71
2	12	168147	0.06	0	...	1	Yes	Bachelor or equivalent	Male	Yes	Single	71
3	42	29047	0.05	0.01	...	2	Yes	Bachelor or equivalent	Female	Yes	Single	8
4	42	29047	0.05	0.01	...	2	Yes	Bachelor or equivalent	Female	Yes	Single	8
5	58	27076	0.07	0.02	...	3	Yes	Master or equivalent	Female	Yes	Single	16
6	58	27076	0.07	0.02	...	3	Yes	Master or equivalent	Female	Yes	Single	16
7	20	137977	0.05	0.03	...	4	Yes	PhD or equivalent	Male	No	Single	74
8	20	137977	0.05	0.03	...	4	Yes	PhD or equivalent	Male	No	Single	74
9	36	136006	0.07	0	...	5	Yes	High School or below	Male	Yes	Married	81
10	36	136006	0.07	0	...	5	Yes	High School or below	Male	Yes	Married	81

Figure 4-3: Data preparation for CDR and customer profile data

From the preceding T-SQL statement, we also can get the column that provides information about whether the customer has churned. We'll use this data for training the customer churn model in the next section.

Note Getting the data into this shape is often the most difficult part of the data scientist's

role. In fact, getting that label column of "churn" can be nontrivial, because it does not always exist in the source data. In this example, we are including that to demonstrate the actual process of working with a customer churn model, but lots of effort goes into figuring out how to get a label column of "churn" for the data that is used for training.

Building the customer churn model

Let's explore the features we'll need to build the customer churn model. In this sample, we'll use information about the customer and the customer call details. Figure 4-4 shows the features used to determine customer churn.

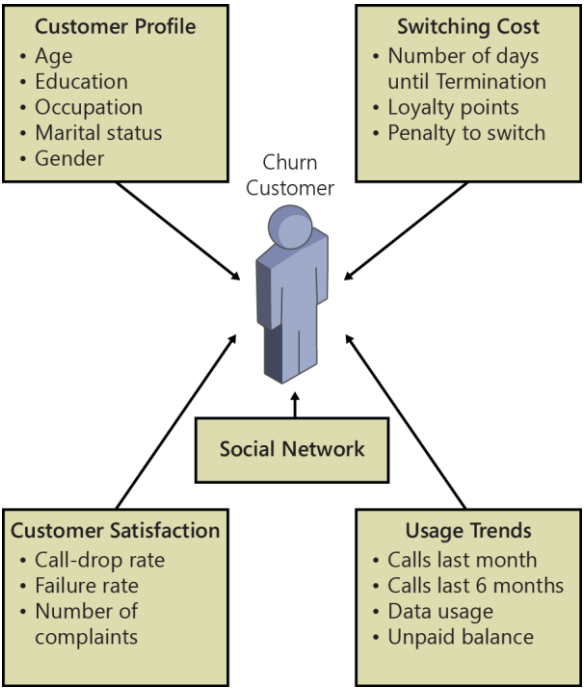


Figure 4-4: Features used to determine whether the customer will be likely to churn.

Note Feature selection is another interesting and important area in data science. that is, knowing which elements in your data are the most predictive. You might not even have the relevant feature information in your data, and you might need to include other data (such as competitor information), or create new columns of data from existing ones to act as new features. To learn more information on feature design, go to https://azure.microsoft.com/documentation/articles/machine-learning-feature-selection-and-engineering/.

Now that we have the data prepared, let's take a look at the steps that are used for training, evaluating, and using the model. Each of the stored procedures combines R and T-SQL.

```
-- Training the customer churn model for telco
exec train_customer_churn_model;

-- Finding the model
select * from cdr_models
where model_name = 'rxDForest'

-- Evaluating the model
exec model_evaluate

-- Predicting customers that will churn
exec predict_customer_churn 'rxDForest';
```

In the next section, we'll dive deeper into each of the stored procedures to gain a better understanding of how to build them.

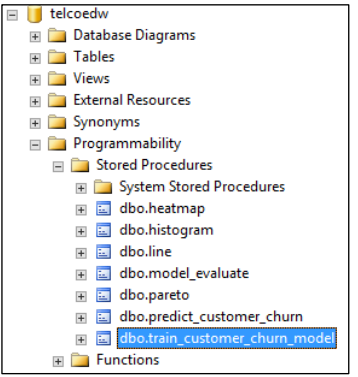
Step-by-step

First, we'll explore the sample database and then move on to building each stored procedure. To begin, restore the sample database, called **telcoedw**.

Training the model

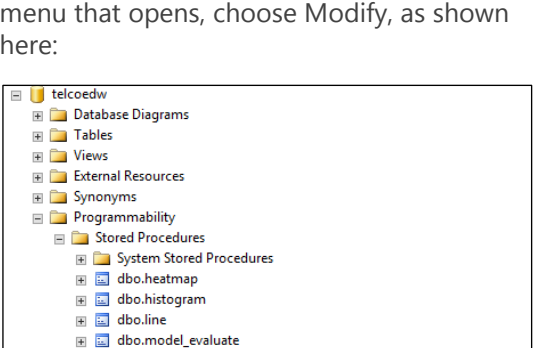
Following is the procedure for training your model:

1. You can use SQL Server Management Studio (SSMS) to manage the tables, and stored procedures in the customer churn solution. In SSMS, expand the node for Programmability | Stored Procedures, as shown in the following screenshot:



Notice that there are several stored procedures for training, evaluating, and prediction. Each of these intelligent stored procedures harnesses the power of R to deliver advanced analytics to the application developers.

2. Let's explore one of the stored procedure: `dbo.train_customer_churn_model`. To do this, right-click it, and then, on the shortcut menu that opens, choose Modify, as shown here:



Notice the T-SQL and R code that uses the `ScaleR` libraries for training a decision forest. We use the `rxDForest()` function in the following example to build a classification

model for determining whether the customer will be likely to churn:

```
USE [telcoedw]
GO

ALTER procedure
[dbo].[train_customer_churn_model]
as
begin
    execute sp_execute_external_script
      @language = N'R'
    , @script = N'
        require("RevoScaleR");

        train_vars <-
rxGetVarNames(edw_cdr_train_SMOTE)
        train_vars <- train_vars[!train_vars
%in% c("churn")]

        temp<-paste(c("churn",paste(train_vars,
collapse="+") ),collapse="~")
        formula<-as.formula(temp)

        forest_model <- rxDForest(formula =
formula,
                                  data =
edw_cdr_train_SMOTE,
                                  nTree = 8,
                                  maxDepth = 32,
                                  mTry = 2,
                                  minBucket=1,
                                  replace = TRUE,
                                  importance = TRUE,
                                  seed=8,

parms=list(loss=c(0,4,1,0)))
        rxDForest_model <- data.frame(payload

        =as.raw(serialize(forest_model,
connection=NULL)));'
    , @input_data_1 = N'select * from
edw_cdr_train_SMOTE'
    , @input_data_1_name =
N'edw_cdr_train_SMOTE'
```

```
        , @output_data_1_name = N'rxDForest_model'
        with result sets ((model varbinary(max)));
    end;
```

We've used the `rxGetVarNames()` function to get the features that will be used for training. In the R code, the following two lines define the label and features that will be used in training:

```
churn ~ <feature 1> + <feature 2> …

temp<-paste(c("churn",paste(train_vars,
collapse="+") ),collapse="~")
formula<-as.formula(temp)
```

The data used for training is stored in the table `edw_cdr_SMOTEtrain`.

Note `rxDForest()` is a parallel external memory decision forest algorithm that is designed to work with very large datasets. It is based on the random forest work done by Leo Breiman and Adele Cutler, and the `randomForest` package of Andy Liaw and Matthew Weiner. You can read more about `rxDForest()` at http://www.rdocumentation.org/packages/RevoScaleR/functions/rxDForest and http://blog.revolutionanalytics.com/2014/01/a-first-look-at-rxdforest.html.

When you execute the stored procedure, you will see that it returns the serialized model returned

as the SQL Server data type `varbinary(max)`, as demonstrated in Figure 4-5.

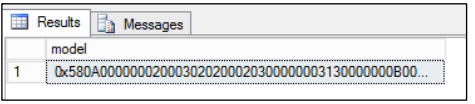


Figure 4-5: Serialized customer churn model.

You can store the serialized model into a SQL Server table, using it to make predictions. To do that, use this code:

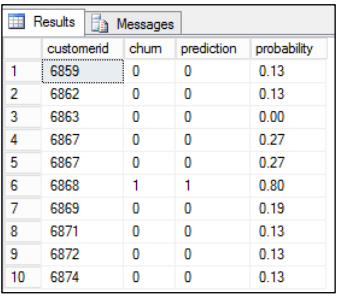
```
DECLARE @t table (churn_model varbinary(max))
INSERT @t (churn_model)
EXEC train_customer_churn_model
SELECT * FROM @t

INSERT into cdr_models
SELECT TOP 1 'rxDForest', churn_model FROM @t
```

Evaluating the model

After you build the model, you should evaluate it by using test data. To do this, we created a table called `edw_cdr_test_pred` that contains the test data. This table contains the actual label value and the predicted churn value. Let's take a look at the data in the table.

```
SELECT TOP 10 [customerid]
      ,[churn]
      ,[prediction]
      ,cast([probability] as decimal(4,2)) as
probability
  FROM [telcoedw].[dbo].[edw_cdr_test_pred]
```

This returns the results shown in Figure 4-6.



Results | Messages

	customerid	churn	prediction	probability
1	6859	0	0	0.13
2	6862	0	0	0.13
3	6863	0	0	0.00
4	6867	0	0	0.27
5	6867	0	0	0.27
6	6868	1	1	0.80
7	6869	0	0	0.19
8	6871	0	0	0.13
9	6872	0	0	0.13
10	6874	0	0	0.13

Figure 4-6: Test data preparation.

After you have prepared the test data, you can run the stored procedure `evaluate_customer_churn_model` to evaluate the quality of the model.

To see the details for the stored procedure, right-click Evaluate_Customer_Churn_Model, and then choose Modify, as illustrated in Figure 4-7.

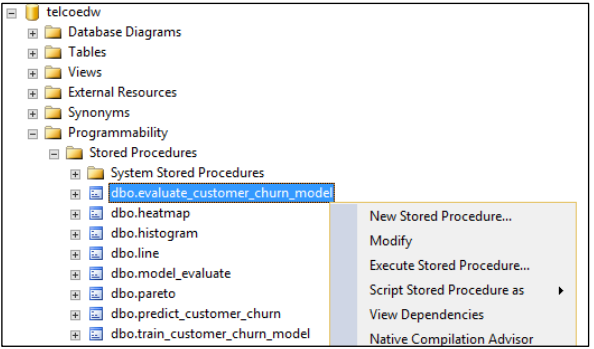


Figure 4-7: Evaluating the customer churn model.

```
USE [telcoedw]
GO
ALTER procedure
[dbo].[evaluate_customer_churn_model]
as
begin
    execute sp_execute_external_script
      @language = N'R'
    , @script = N'
  evaluate_model <- function(data, observed,
predicted) {
  confusion <- table(data[[observed]],
data[[predicted]])
  print(confusion)
  tp <- confusion[2, 2]
  fn <- confusion[2, 1]
  fp <- confusion[1, 2]
  tn <- confusion[1, 1]
  accuracy <- (tp + tn) / (tp + fn + fp + tn)
  precision <- tp / (tp + fp)
  recall <- tp / (tp + fn)
  fscore <- 2 * (precision * recall) / (precision +
recall)
  metrics <- c("Accuracy" = accuracy,
               "Precision" = precision,
               "Recall" = recall,
               "F-Score" = fscore)
```

```
  return(metrics)}

metrics <- evaluate_model(data = edw_cdr_test_pred,
                                  observed =
"churn",
                                  predicted =
"prediction")
print(metrics)
metrics<-matrix(metrics,ncol=4)
metrics<-as.data.frame(metrics);
'
    , @input_data_1 = N'
    select * from edw_cdr_test_pred'
    , @input_data_1_name = N'edw_cdr_test_pred'
    , @output_data_1_name = N'metrics'
    with result sets ( ("Accuracy" float,
"Precision" float, "Recall" float, "F-Score" float)
                );
end;
```

After you execute the stored procedure, you can view the key metrics for the model by running `exec evaluate_customer_churn_model`. This returns the results displayed in Figure 4-8.

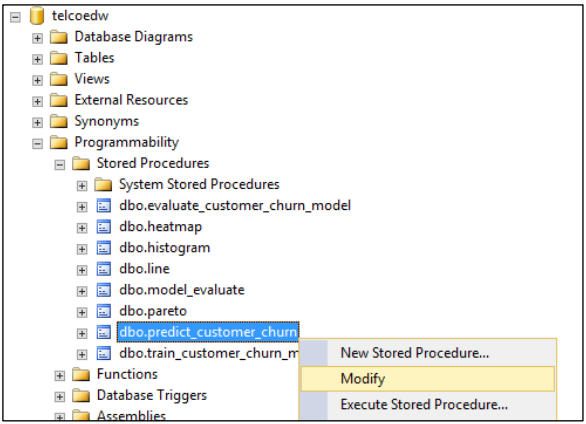
Results | Messages

	Accuracy	Precision	Recall	F-Score
1	0.960430463576159	0.955555555555556	0.572243346007605	0.715814506539833

Figure 4-8: Evaluating the model.

Predicting customers that will churn

Next, let's take a look at how we can create a stored procedure that you can use for scoring the data. To do this, right-click the stored procedure Predict_Customer_Churn, and then choose Modify, as demonstrated here:



This shows the T-SQL and R code that is used for making the prediction:

```
USE [telcoedw]
GO

ALTER procedure [dbo].[predict_customer_churn]
(@model varchar(100))
as
begin
    declare @rx_model varbinary(max) = (select model
from cdr_rx_models
                                where model_name =
```

```
@model);

    -- Use the selected model for prediction
    exec sp_execute_external_script
          @language = N'R'
        , @script = N'

require("RevoScaleR");
cdr_model<-unserialize(rx_model);
predictions <- rxPredict(modelObject = cdr_model,
                         data = edw_cdr_new,
                type="prob",
                         overwrite = TRUE)
print(head(predictions))

threshold <- 0.5
predictions$X0_prob <- NULL
predictions$churn_Pred <- NULL
names(predictions) <- c("probability")

predictions$prediction <-
ifelse(predictions$probability > threshold, 1, 0)
predictions$prediction<-
factor(predictions$prediction, levels = c(1, 0))

edw_cdr_new_pred  <-
cbind(edw_cdr_new[,c("customerid")],predictions)
print(head(edw_cdr_new_pred ))
edw_cdr_new_pred <-as.data.frame(edw_cdr_new_pred);
'
    , @input_data_1 = N'select * from edw_cdr_new'
    , @input_data_1_name = N'edw_cdr_new'
    , @output_data_1_name=N'edw_cdr_new_pred'
    , @params = N'@rx_model varbinary(max)'
    , @rx_model = @rx_model
    with result sets (("customerid" int,
"probability " float, "prediction" float));
end;
```

You can specify how the `rxPredict()` function is used to predict the customers that are likely to churn. The input data (containing new and existing customers) is stored in the table

edw_cdr_new. You can run the stored procedure by using the following T-SQL code:

```
-- Predicting customers that will churn
exec predict_customer_churn 'rxDForest';
```

The stored procedure uses the trained model to predict the customers that are likely to churn, provides the score probability, and the customer ID, as illustrated in Figure 4-9.

Results | Messages

	customerid	probability	prediction
1	1	0.25	0
2	2	0.0113706804506295	0
3	3	0	0
4	6	0.157320752432872	0
5	6	0.157320752432872	0
6	7	0.117526188063285	0
7	9	0.00223948244074958	0
8	14	0.0941806402909003	0
9	15	0.158395447166248	0
10	17	0	0

Figure 4-9: Running the stored procedure for predicting customer churn.

Summary

Many industries, including mobile providers, use churn models to predict which customers are most likely to leave, and to understand which factors cause customers to stop using their service. In many of these industries, the customer and transactional data are stored in a database. The ability to run the churn model where the

data gravity is makes it possible for you to do data science without having to bring data outside of the database (e.g., to your client tool). This also give you the ability to use the powerful capabilities available in the database for speeding up aggregation, filtering, and joining data.

In this chapter, you learned how to explore the telecommunication customer data stored in a database, built a customer churn model, and evaluated the model. You then learned how to operationalize the model by using T-SQL stored procedures. Using SQL Server R Services, you can easily blend both R and T-SQL code and make them available as stored procedures that you can invoke in your applications that work with T-SQL.

Tell us what you think!

Is this book useful?
Does it meet your expectations?
Is there room for improvement?

Let us know at
http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!

CHAPTER 5

Predictive maintenance and the Internet of Things

With the emergence of devices connected to the web, or the "Internet of Things" and an increasing amount of data being collected about devices and machines, many companies are looking to take advantage of this data to improve production and maintenance efficiency. In this chapter, we discuss example use cases and different ways to

formulate a predictive maintenance problem into a machine learning model. We then show you how to use Microsoft SQL Server R Services to fit these models to example data and predict failures before a problem occurs, walking through the data science process that was used to build the solution.

What is the Internet of Things?

The Internet of Things (IoT) is no longer just an emerging trend but something that has already proven valuable for many businesses that are harnessing connected devices to drive new innovative scenarios. Consulting firm McKinsey & Company predicts that within the next 10 years, the potential economic impact of the IoT will be up to $11 trillion *per year*.[1] But it isn't just about the cheap hardware and pervasive connectivity; it's about the possibilities when you combine analytics and the right data to solve a particular business need.

The IoT is defined by four main aspects:

- **Things** Physical assets such as machines, devices, or sensors.
- **Connectivity** The "things" are connected directly to the Internet, to one another, and/or another device that is able to understand information from the "thing."

[1] McKinsey Global Institute. "The Internet of Things: Mapping the Value beyond the hype," aka.ms/cite10

- **Data** The "things" are able to collect and communicate information, such as an asset's state or environment.
- **Analytics** Analytics on top of the data from the connected "things" make it possible for people or machines to take action.

The IoT facilitates insight and agility, giving access to information in near real time. Companies that take advantage of the IoT also have potential to gain a competitive edge through identifying new business opportunities, such as the potential to redefine customer service.

Use cases made possible through the IoT span across industries, from healthcare (monitoring device data making it possible to adjust dosage), distribution (tracking products and authentication to prevent fraud), to retail (creating personalized offers through connected devices and location information), and operations (proactively fix equipment before it fails). In this chapter, we concentrate on predictive maintenance using SQL Server R Services.

Predictive maintenance in the era of the IoT

Predicting when you should perform maintenance is not a new concept. In fact, many maintenance operations already have a planned maintenance schedule. Maintenance ensures the reliability of the machine in the future and prevents unwanted issues or disruptions. If you think about a car, for example, the manufacturer might suggest doing an oil change every 6 months or 5,000 miles. They are, in fact, predicting when you should do maintenance, yet it is a very simple model based solely on time and one indicator of usage.

With the increasing amount of data being collected about devices, you can build more informative models about when maintenance should be performed—it's not necessary to rely on a schedule alone. Using this new technology, you can improve production and maintenance efficiency. For instance, cars don't need service every *X* miles; instead, you can perform the maintenance depending on the driving and environment conditions. This concept is called *Condition-Based Monitoring* (CBM), and it uses the condition of the asset to decide what maintenance needs to be done, acting on it

based on indicators of decreasing performance or potential fault.

Building on these ideas and using the data available about the assets of interest rather than solely identifying faults or failures after they occur or monitoring to understand current performance of the asset, using machine learning techniques, we can also forecast into the future to predict when a problem is likely to occur. This might be done through a data-driven model that learns from the historical relationship between the relevant data points such as the sensor readings from the assets and when problems have occurred in the past. An example of the benefits of applying predictive maintenance techniques within the aerospace industry is shown in Figure 5-1.

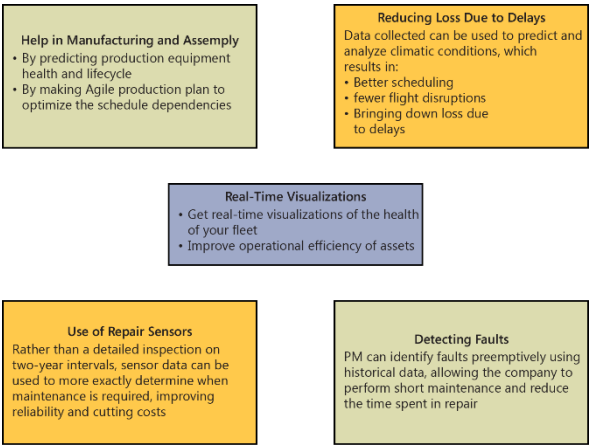


Figure 5-1: The benefits of using predictive maintenance within aerospace industry.[2]

Although predictive maintenance can be considered to be forecasting when to do maintenance in order to prevent an unwanted failure, it also encompasses many other related problems such as maintenance action recommendations, root-cause prediction, and fault detection, for example. In the section that follows, we describe example use cases across a variety of industries.

[2] See the entire infographic at aka.ms/cite11.

Example predictive maintenance use cases

Although it's particularly popular within manufacturing, the concept of predictive maintenance is useful across many industries. The list that follows presents some examples of problems that can be solved by using the concepts that follow in this chapter:[3]

- Aerospace
  - What is the likelihood of delay due to mechanical issues?
  - When is this aircraft component likely to fail next?

[3] See more example use cases here: Fidan Boylu Uz, Cortana Intelligence Solution Template Playbook for predictive maintenance in aerospace and other businesses, aka.ms/cite2

- Utilities
  - When is a solar panel or wind turbine going to fail next?
  - Which circuit breakers in a system are likely to fail in the next month?
  - Is this ATM going to dispense the next five notes without failing?
- Manufacturing
  - Will the component pass the next stage of testing on the factory floor or does it need to be reworked?
  - What is the root cause of the test failure?
- Transportation and logistics
  - Should the brake rotors in a car be replaced or can it wait for another month?
  - What maintenance tasks should be performed on an elevator?

These types of problems can often be formulated in many ways, and it is important to consider what formulation provides the most value. In addition, it might be that solving a related problem rather predicting failure or

maintenance needs directly actually provides more value. Consider an oil tank that fills over time and very rarely is overfilled, which would result in a failure (i.e., a spill). In this case, forecasting the tank volume itself provides more value than predicting when a spill might occur, because knowing the future tank volume can both help prevent spills as well as improve other operations such as when to extract and move the liquid.

Before beginning a predictive maintenance project

Before you begin a predictive-maintenance project using a data-driven approach, there are several important requirements to consider:[4]

- **Is the question "sharp"?** The question should be formulated in a way that can be answered with either a category (e.g., yes/no, reason A/B/C) or a number (seven days). "Is this circuit breaker likely to fail in the next month?" and "When is my solar panel going to experience a fault?" are examples of sharp questions.

[4] Brandon Rohrer, "What can data science do for me?" aka.ms/cite1.

Note In addition, it is important to consider whether the problem is actually predictive in nature (e.g., machines that have degradation patterns) or whether it is an unpredictable phenomenon that results in maintenance requirements (e.g., human vandalism can be very difficult to predict). In any case, there should also be a *clear path of action* if the answer to the sharp question is known. For example, if the question reflects whether a failure will happen in the next seven days but maintenance needs to be scheduled a month in advance, the resulting model will not be as helpful to business operations.

- **Does the data measure what you care about?** The data needs to have identifiers and information at the level that is being predicted. For example, if you're trying to predict door-related problems within a vehicle, information should be collected about the doors and the doors should be uniquely identifiable (versus having only information at the level of the vehicle).
- **Is the data accurate?** The failures recorded in the historical data should be actual failures, not simply potential failures, and the data should reflect what actually happened. For example, if a "failure" is defined as when unplanned maintenance is

required, but that unplanned maintenance is done based on monthly checks to the system, the data doesn't accurately reflect the exact timing of failure; rather, it reflects just the month in which the failure occurred. Maintenance data should be collected in a structured manner when possible, such as categories of operations completed rather than free-form text, which might or might not include all of the necessary information and can differ across who supplied the information.

- **Is the data connected?** The amount of missing data should be limited and the different data sources should be linkable, such as machine information and usage information. If the data sources cannot be joined together through identifiers, they will not provide as much (if any) value. Usually, it is not just the "IoT data" or machine usage information that is important, but also other information such as previous maintenance and repair history, and this must be linkable together.
- **Is there enough data?** For predicting time left to failure, there should be failures or some proxy recorded. In fact, to predict failures, there should be many (e.g., hundreds or thousands) of examples of

failure in the past. If there are very few failures, the outcome should be reconsidered (e.g., predicting a fault or something that often leads to a failure rather than the failure itself).

The data science process using SQL Server R Services

Let's look at a sample predictive maintenance problem using SQL Server R Services.[5] We'll develop and deploy an end-to-end solution using publicly available data from NASA on simulated aircraft-engine run-to-failure events,[6]

---

[5] This example builds on the SQL Server R Services template available through the Cortana Intelligence Gallery, aka.ms/cite3. You can find other example templates on this site, as well.

[6] Saxena, A. and Goebel, K. (2008). "PHM08 Challenge Data Set," NASA Ames Prognostics Data Repository (http://ti.arc.nasa.gov/project/prognostic-data-repository), NASA Ames Research Center, Moffett Field, CA.

and then walk through the data science process used.[7]

In the example that follows, the model is developed by running the code within an R integrated development environment (R-IDE) with the compute context generally set to SQL Server, such as RStudio or R Tools for Visual Studio (see Figure 5-2). The models are then operationalized through SQL stored procedures, which can be run within a SQL environment (such as Microsoft SQL Management Studio) or called by applications to make predictions. Windows PowerShell scripts are provided to invoke the steps end to end.

[7] Ehrlinger, J and Dogan B. "A linear method for non-linear work: Our data science process," aka.ms/cite4.

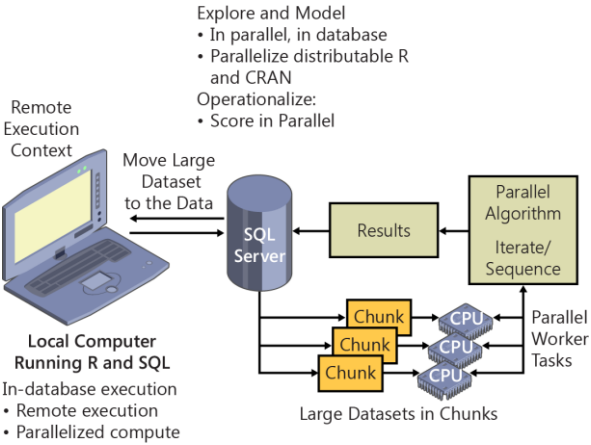


Figure 5-2: Model development in R using remote execution context with parallelized compute.

The example code within this chapter makes heavy use of the `ScaleR` functions to handle large datasets that do not fit in memory; for example, running a regression model that can scale out with the dataset size. However, barring memory limitations, you could also apply the concepts outlined in a moment by using standard R libraries within SQL Server.

More info You can find the code described in this chapter and additional code at http://aka.ms/cite3. The directory structure includes a folder for the raw data, a folder for

the R development code, and a folder for the stored SQL procedures.

An overview of the data science process taken in this chapter is outlined in Figure 5-3.

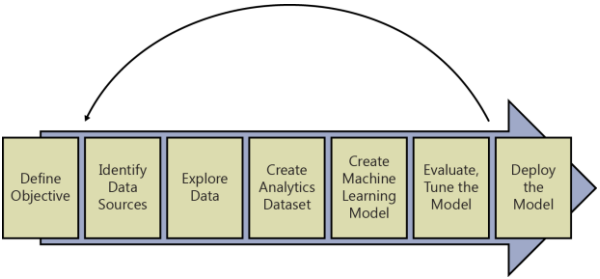


Figure 5-3: The data science process.

Although the process outlined in Figure 5-3 appears very linear because it is drawn as a series of steps, in practice it is very iterative, often requiring revisiting a previously completed step. For example, a model that doesn't perform as well as needed might require revisiting the creation of the analytics dataset to include more informative features from which the model can learn. In the sections that follow, more detail is given about what is expected during each step of the process, especially for predictive maintenance problems, as illustrated through the NASA data.

Define objective

As predictive maintenance use case examples demonstrate, there are many ways to formulate the problem. In the example in this chapter, we formulate the problem in three ways:[8]

- Remaining Useful Life (RUL) of an asset, which is solved by using *regression* techniques.
- Whether an asset will fail within a certain time frame, such as whether it will fail in the next seven days, which is solved using *binary classification* techniques.
- Whether an asset will fail within certain time windows, such as 0–7 days, 8–15 days, or longer than 15 days, which is solved by using *multiclass classification* techniques. Although not shown here, you can also apply this method for root-cause analysis, where the cause is a categorical value and the model aims to predict whether the model will fail due to cause A, B, C, …, or not fail during a certain time window.

[8] See also the Azure Machine Learning template for predictive maintenance at aka.ms/cite5.

An implicit assumption is made in the steps that follow that the engines have a progressive degradation pattern reflected in their sensor measurements. The machine learning model learns from the historical relationship between the sensor readings and the historical failures to predict when a failure will happen in the future.

Identify data sources

There are usually many data sources that contain the relevant information needed to build a predictive maintenance model. Although not all of these sources might be necessary, the more relevant information that we can find on the problem, the more likely the solution created will have value and predictive power.

Data sources that should be considered include, but are not limited to the following:

- **Failure history** When each machine or component within the machine failed in the past, preferably with categorical labels on the type of failure.
- **Repair history** Previous maintenance records, components replaced, and maintenance activities performed.

- **Machine conditions** Data regarding operational conditions, sensor readings, settings of the machine, collected over time.
- **Machine features** Features of the machine as well as component-level information such as production date and technical specifications.
- **Operating conditions** Environmental characteristics that might influence a machine's performance, such as temperature, location.
- **Operator attributes** Attributes of the operator of the machine, such as the driver of a car or the person running a piece of machinery. This can be a one-to-many relationship if, for example, many individuals are using one machine.

In the example in this chapter, we use publicly available data from NASA with respect to run-to-failure events for aircraft-engines. There are three datasets in total in this example, and the data is provided in a format that is already separated into training, testing, and *ground truth* (more on this in a moment).

Dataset	Description
Training data ("PM_Train")	This is historical run-to-failure data that includes

	settings of the engines and sensor readings over time until failure occurred. The data will be used to train the machine learning model.
Testing data ("PM_Test")	The settings of the engines and sensor readings for machines in operation that had not yet failed.
Ground truth data ("PM_Truth")	Ground truth of when the machines in the testing data actually failed.

Note In practice, data won't be already separated neatly into training and testing datasets like we have here—just like in the previous examples, the real work is often in finding, vetting, and conditioning the data to make it ready for modeling.

The reason data is separated, and not all of the historical data is fed into the machine learning model, is that we want to create a model that is generalizable and evaluate how well the model is performing in a realistic manner. For example, creating a model that is able to perfectly predict in the past is not useful if it doesn't generalize to predict failures in the

future. One way to do this is to train based on data until a certain point in time (e.g., using data from the first part of the year January to September), and then validate how well the model is doing by evaluating it for historical data after that point in time (e.g., October to December of the year). Also separating by machine identifier so that some machines are in the training dataset and a distinct set of machines is in the testing set is another good method. These can be combined together for the strictest validation.[9]

Explore data

The model is developed by using remote execution that employs an R-IDE such as R Tools for Visual Studio. We'll develop the code locally but set the compute context—which defines the remote connection, where the processing is done—to the SQL Server to run there, using its higher compute and memory resources.

Important Using remote execution allows for often faster computation, larger dataset capacity, fewer security concerns, and no data movement or copies are required. This mitigates the R memory and scalability issues.

[9] See more detail at the source from footnote 3.

However, it also might be useful in certain contexts to move data to the local context, especially when doing quick, iterative tasks on small data. An example of this is shown in the evaluation section of this chapter.

```
## Compute context ##
connection_string <- "Driver=SQL Server;
                      Server=[Insert Server];
                      Database=[Insert Database];
                      UID=[Insert user id];
                      PWD==[Insert password]"
sql_share_directory <- paste(" c:\\AllShare\\",
Sys.getenv("USERNAME"), sep = "")
dir.create(sql_share_directory, recursive = TRUE,
showWarnings = FALSE)
sql <- RxInSqlServer(connectionString =
connection_string,
                     shareDir = sql_share_directory)
local <- RxLocalSeq()
```

After the compute context is defined, you can explore and visualize the data through the R IDE. You can find the code at http://aka.ms/cite3, and then upload the data to your own database or use the sample database along with this chapter; the rest of the code within this chapter assumes the three aforementioned raw datasets are already loaded in your database. When the data is loaded, you can begin exploring it and creating visualizations. In the code example that follows, some example data exploration visualizations are created, and Figure 5-4 shows an example view of what running this might look like with R Tools for Visual Studio:

```
### Data exploration examples
rxSummary( ~ ., train_data_table)
rxHistogram(~s11,train_data_table)
rxHistogram( ~ s11 | F(id), type = "p", data =
train_data_table)
rxLinePlot(s11~cycle|id,train_data_table)
```

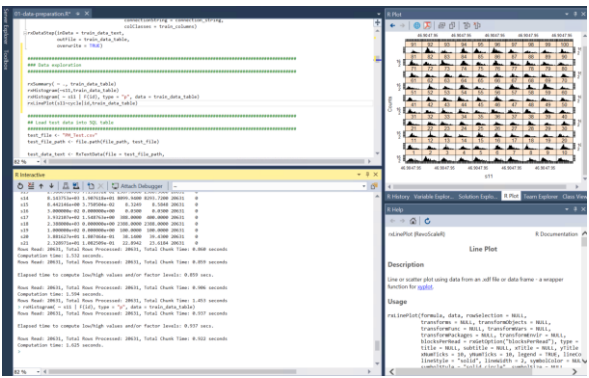


Figure 5-4: Data exploration using R Tools for Visual Studio.

Create analytics dataset

In this stage of the process, the raw data sources must be combined together and manipulated in such a way as to create the analytics dataset that will feed into the machine learning model. Both data labeling—where the target outcome of the machine learning model is created—as well as feature engineering—where extra columns of

data are created that inform the machine learning model—are completed in this step.[10]

In the dataset in this chapter, time is represented by the column "cycle"; however, this can be generalized to be a different measure of time, such as days, weeks, or months.

Note In the example in this chapter, there is no aggregation that is done to the data before the data labeling and feature engineering is conducted; in other words, the analytics dataset has the same number of rows before and after these steps (although it has additional columns). In practice, there is often a layer of aggregation that is done before this.

For example, a sensor could be recording measurements every second, but that level of detail is not informative and in fact would likely add too much noise to the model. In this case, you can aggregate the data first (such as average measures by hour) and then conduct

[10] For another example walkthrough of how you can use R for predictive maintenance and the process of how the analytics data set was created (in this case for manufacturing use case and root cause prediction), see the Jupyter R notebook example created by Fidan Boylu Uz at aka.ms/cite6.

feature engineering and data labeling. Similar to other steps of the process, this is often an iterative process to understand the level of aggregation that is necessary.

Data labeling

Because the problem is formulated in three different ways, three sets of labels are generated for the model to learn, as illustrated in Figure 5-5. In the training data, we know when the engine actually failed due to the fact that the data is run-to-failure data, so the last time period for each engine in the training data represents the failure point.

To model the remaining useful life of the engine, we can label the data by subtracting the time period from the maximum time period for each engine for each row of data. For example, let's assume that engine id 1 runs for 192 cycles. We thus know that at cycle 1, it has 191 cycles left, and the column "RUL" which represents remaining useful life (the target outcome in this case) is labeled to be 191 for the row that represents the data at cycle 1. See Figure 5-6 for an example of what the dataset looks like after data labeling is complete.

For the formulation of whether the engine is going to fail in the next *X* periods, the label of

"1" is created for all data points from the failure point back *X* time periods; otherwise, it is labeled "0." For the formulation of whether the engine is going to fail in the window [1, w0] cycles or to fail within the window [w0+1, w1] cycles, or it will not fail within w1 cycles, a label is created for each of the categories and applied to the rows according to how far back from the failure point the row is.

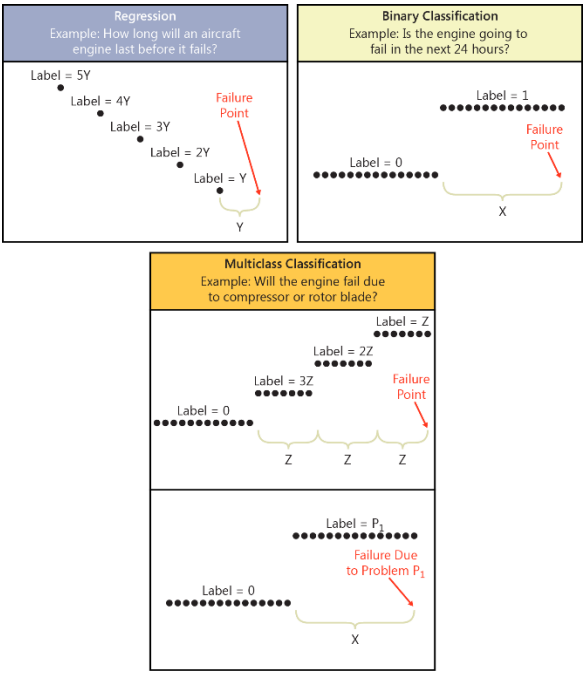
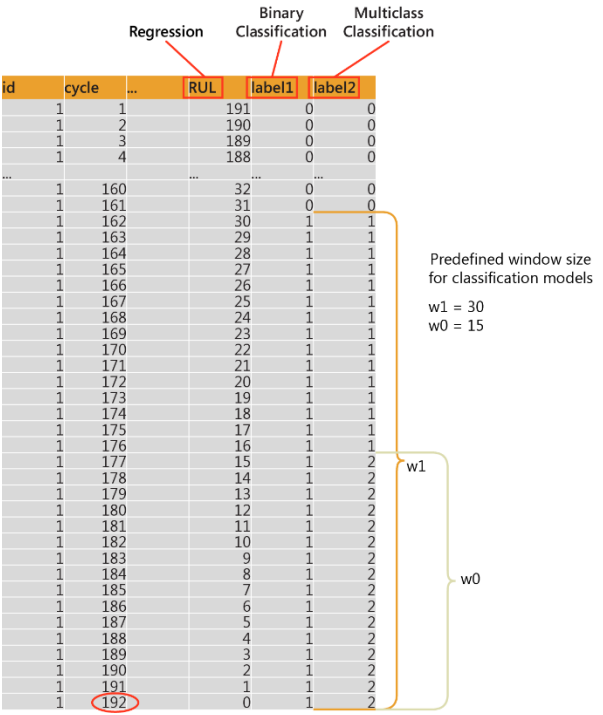


Figure 5-5: Three formulations of data labeling.



id	cycle	...	RUL	label1	label2
1	1		191	0	0
1	2		190	0	0
1	3		189	0	0
1	4		188	0	0
...			...	...	...
1	160		32	0	0
1	161		31	0	0
1	162		30	1	1
1	163		29	1	1
1	164		28	1	1
1	165		27	1	1
1	166		26	1	1
1	167		25	1	1
1	168		24	1	1
1	169		23	1	1
1	170		22	1	1
1	171		21	1	1
1	172		20	1	1
1	173		19	1	1
1	174		18	1	1
1	175		17	1	1
1	176		16	1	1
1	177		15	1	2
1	178		14	1	2
1	179		13	1	2
1	180		12	1	2
1	181		11	1	2
1	182		10	1	2
1	183		9	1	2
1	184		8	1	2
1	185		7	1	2
1	186		6	1	2
1	187		5	1	2
1	188		4	1	2
1	189		3	1	2
1	190		2	1	2
1	191		1	1	2
1	192		0	1	2

Figure 5-6: A dataset after data labeling is complete (where the target outcome for the machine learning model is created), for three example formulations of the problem.

Note These are simple ways to label the data to enable the building of a machine learning

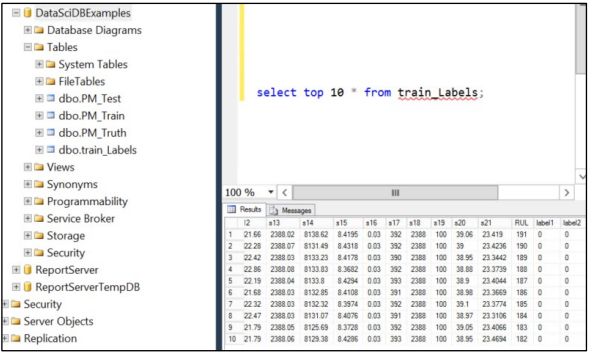
model. But these models might violate standard statistical assumptions, such as independence of observations, and you should evaluate them for robustness through techniques discussed in the evaluation section of this chapter. Additionally, you can use more advanced statistical and machine learning models.

In the code example that follows, we create a function that will label the data as described above, and then apply that function to the training data:

```
## Data labeling
library(plyr)
data_label <- function(data) {
    data <- as.data.frame(data)
    max_cycle <- plyr::ddply(data, "id",
plyr::summarise, max = max(cycle))
    if (!is.null(truth)) {
        max_cycle <- plyr::join(max_cycle, truth, by
= "id")
        max_cycle$max <- max_cycle$max +
max_cycle$RUL
        max_cycle$RUL <- NULL
    }
    data <- plyr::join(data, max_cycle, by = "id")
    # Label for regression
    data$RUL <- data$max - data$cycle
    # Label for binary/multi-class classification
    data$label1 <- ifelse(data$RUL <= 30, 1, 0)
    # Label for multi-class classification
    data$label2 <- ifelse(data$RUL <= 15, 2,
data$label1)
    data$max <- NULL

    return(data)
}
```

```
## Add data labels for train data
tagged_table_name <- "train_Labels"
truth_df <- NULL
tagged_table_train = RxSqlServerData(table =
tagged_table_name,

colClasses = train_columns,

connectionString = connection_string)
inDataSource <- RxSqlServerData(table =
train_table_name,
                                 connectionString =
connection_string,
                                 colClasses =
train_columns,
                                 rowsPerRead = 30000)
rxDataStep(inData = inDataSource,
           outFile = tagged_table_train,
           overwrite = TRUE,
           transformObjects = list(truth =
truth_df),
           transformFunc = data_label,
           rowsPerRead = -1,
           reportProgress = 3)
```

After you run the data labeling code for the training data, there should be a new table called "train_Labels" in the database that contains three new columns: RUL, label1, and label2.
You can verify the new table exists through the R-IDE by using a command such as `rxSqlServerTableExists("train_Labels",connection_string)` or by running a SQL query against the database, as demonstrated in Figure 5-7.



	i2	s13	s14	s15	s16	s17	s18	s19	s20	s21	RUL	label1	label2
1	21.66	2388.02	8138.62	8.4195	0.03	392	2388	100	39.06	23.419	191	0	0
2	22.28	2388.07	8131.49	8.4318	0.03	392	2388	100	39	23.4236	190	0	0
3	22.42	2388.03	8133.23	8.4178	0.03	390	2388	100	38.95	23.3442	189	0	0
4	22.86	2388.08	8133.83	8.3682	0.03	392	2388	100	38.88	23.3739	188	0	0
5	22.19	2388.04	8133.8	8.4294	0.03	393	2388	100	38.9	23.4044	187	0	0
6	21.68	2388.03	8132.85	8.4108	0.03	391	2388	100	38.98	23.3669	186	0	0
7	22.32	2388.03	8132.32	8.3974	0.03	392	2388	100	39.1	23.3774	185	0	0
8	22.47	2388.03	8131.07	8.4076	0.03	391	2388	100	38.97	23.3106	184	0	0
9	21.79	2388.05	8125.69	8.3728	0.03	392	2388	100	39.05	23.4066	183	0	0
10	21.79	2388.06	8129.38	8.4286	0.03	393	2388	100	38.95	23.4694	182	0	0

Figure 5-7: View of database tables using Microsoft SQL Server Management Studio.

Although the testing data will not be used to train the model, it also needs to be labeled in order to evaluate how well the model performs on data it has not yet seen. Follow the code at aka.ms/cite3 to also add labels to the testing data using the "ground truth" available in the third dataset.

Note In practice, the data for these types of problems might not be "run-to-failure," and the data labeling process would be different in that case, although the concepts would be similar. For example, the timing of failures might be recorded in a separate dataset rather than implied from the last point in time for each machine. In this case, the failure data would need to be joined with the machine usage

information and then labels created by using the aforementioned concepts.

Feature engineering

Feature engineering is the process of creating features (i.e., extra columns in the dataset) that are inputs that aim to provide better or additional predictive power to the learning algorithm. This is a very iterative process, that should be informed by domain knowledge. In the case of predictive maintenance, it is important to create features that capture the degradation pattern over time. As one example, for each labelled record of an asset, pick a rolling window of size *w* and then compute the rolling aggregate features such as averages and standard deviations for the periods before the labelling date and time of that record (see Figure 5-8). Other potential features include but are not limited to tumbling window aggregations, changes from initial or known safe value, velocity of change, and frequency count over a predefined threshold.

Although R is used to create the features in this example, it is also possible to use SQL directly to create the additional features. Additionally, it might be useful to do basic aggregations in SQL and then use R for more advanced functionality.

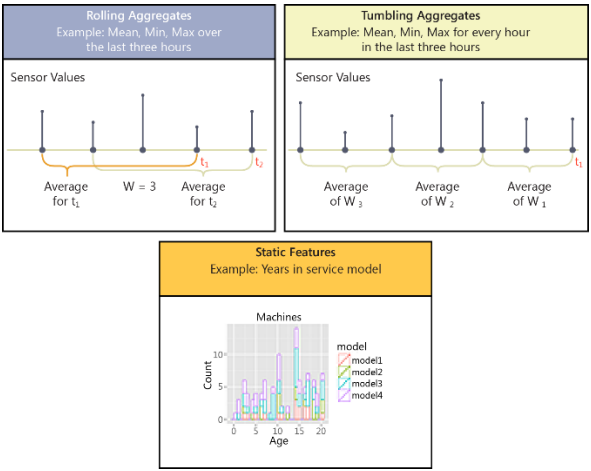


Figure 5-8: An example of features to be included in the model.[11]

The following code snippet creates a function that generates features from the raw data:

```
## Create features from the raw data by computing
the rolling means.
create_features <- function(data) {
    create_rolling_stats <- function(data) {
        data <- data[, sensor]
        rolling_mean <- zoo::rollapply(data = data,
                                       width = window,
                                       FUN = mean,
                                       align = "right",
                                       partial = 1)
```

[11] Fidan Boylu Uz. "Data Science of Predictive Maintenance: A Modelling Guide Using Azure Notebooks." Azure Webinar, 7/5/2016.

```
        rolling_mean <- as.data.frame(rolling_mean)
        names(rolling_mean) <- gsub("s", "a",
names(rolling_mean))
        rolling_sd <- zoo::rollapply(data = data,
                                     width = window,
                                     FUN = sd,
                                     align = "right",
                                     partial = 1)
        rolling_sd <- as.data.frame(rolling_sd)
        rolling_sd[is.na(rolling_sd)] <- 0
        names(rolling_sd) <- gsub("s", "sd",
names(rolling_sd))
        rolling_stats <- cbind(rolling_mean,
rolling_sd)
        return(rolling_stats)
    }

    data <- as.data.frame(data)
    window <- ifelse(window < nrow(data), window,
nrow(data))
    features <- plyr::ddply(data, "id",
create_rolling_stats)
    features$id <- NULL
    data <- cbind(data, features)

    if (!identical(data_type, "train")) {
        max_cycle <- plyr::ddply(data, "id",
plyr::summarise, cycle = max(cycle))
        data <- plyr::join(max_cycle, data, by =
c("id", "cycle"))
    }

    return(data)
}
```

Note The function here is defined so that only the last cycle is saved for prediction. However, because of the feature engineering, the last cycle contains features that are informed by values in the past, such as the rolling average over the last five cycles.

After this function is defined, it can be applied to the training and testing datasets, as shown here:

```
## Create features for train dataset and save into
SQL table
window_size <- 5
train_vars <-
names(rxGetVarInfo(tagged_table_train))
sensor_vars <- train_vars[grep("s[[:digit:]]",
train_vars)]
rxSetComputeContext(sql)

train_table <- RxSqlServerData(table =
"train_Features",
                               connectionString =
connection_string,
                               colClasses =
train_columns)

rxDataStep(inData = tagged_table_train,
           outFile = train_table,
           overwrite = TRUE,
           transformObjects = list(window =
window_size,
                                   sensor =
sensor_vars,
                                   data_type =
"train"),
           transformFunc = create_features,
           rowsPerRead = -1,
           reportProgress = 3)
```

The resulting training dataset "train_Features" should now have 71 columns and be ready to be fed into a machine learning model, as shown in Figure 5-9.

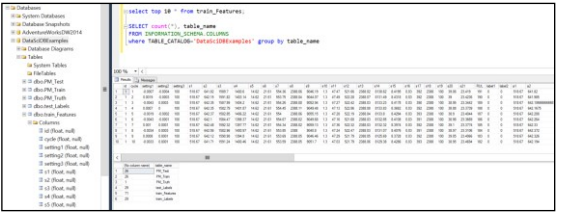


Figure 5-9: Database view after the "create analytics dataset" step is complete and both data labeling and feature engineering are done.

Following is the SQL code shown in Figure 5-9 that is run within Microsoft SQL Server Management Studio:

```
select top 10 * from train_Features;

SELECT count(*), table_name
FROM INFORMATION_SCHEMA.COLUMNS
where TABLE_CATALOG='DataSciDBExamples' group by
table_name
```

You can follow along with the code at http://aka.ms/cite3 to apply the function to the testing data, and to complete feature normalization (putting the features on the same scale so that the model is not biased by scale). The training data is normalized and then the same transformation is applied to the testing data.

Create machine learning model

In this step of the process, several machine learning models are applied to the different labels that were created in the section "Data labeling" earlier. Different models are applied and investigated to understand how well they perform. The optimal model is then chosen based on the evaluation criteria, as discussed in the next section. In predictive-maintenance problems, although not shown in depth here, often cost-sensitive learning and sampling methodologies are applicable, due to the common imbalanced data problem that results from only a small percentage of the data constituting failures.

In the code that follows, a simple feature selection method selects a subset of the features for modeling (removing those that are not very well correlated with the outcome). Then, several binary classification models are applied to predict the "label1" column, which represents whether the machine will fail in the next seven days.

```
## Drop variables and make label a factor in train
table
rxSetComputeContext(sql)
train_table_name <- "train_Features"
train_table <- RxSqlServerData(table =
train_table_name,
                               connectionString =
connection_string,
```

```
                                colInfo = list(label1
= list(type = "factor", levels = c("0", "1"))))

## Find top 35 variables most correlated with label1
rxSetComputeContext(sql)
train_vars <- rxGetVarNames(train_table)
train_vars <- train_vars[!train_vars %in% c("RUL",
"label2", "id", "cycle_orig")]
formula <- as.formula(paste("~", paste(train_vars,
collapse = "+")))
correlation <- rxCor(formula = formula,
                     data = train_table,
                     transforms = list(label1 =
as.numeric(label1)))
correlation <- correlation[, "label1"]
correlation <- abs(correlation)
correlation <- correlation[order(correlation,
decreasing = TRUE)]
correlation <- correlation[-1]
correlation <- correlation[1:35]
formula <- as.formula(paste(paste("label1~"),

paste(names(correlation), collapse = "+")))

## Decision forest modeling
forest_model <- rxDForest(formula = formula,
                          data = train_table,
                          nTree = 8,
                          maxDepth = 32,
                          mTry = 35,
                          seed = 5)
## Boosted tree modeling
boosted_model <- rxBTrees(formula = formula,
                          data = train_table,
                          learningRate = 0.2,
                          minSplit = 10,
                          minBucket = 10,
                          nTree = 100,
                          seed = 5,
                          lossFunction =
"bernoulli")
## Logistic regression modeling
logistic_model <- rxLogit(formula = formula,
                          data = train_table)

## Neural network regression modeling - example with
```

```
local context
library(nnet)
rxSetComputeContext(local)
train_df <- rxImport(inData = train_table)
nodes <- 10
weights <- nodes * (35 + 1) + nodes + 1
nnet_model <- nnet(formula = formula,
                   data = train_df,
                   Wts = rep(0.1, weights),
                   size = nodes,
                   decay = 0.005,
                   MaxNWts = weights)
```

You can find code examples for regression and multiclass classification models at http://aka.ms/cite3.

Evaluate, tune the model

In the evaluation stage, the model is applied to a separate set of data ("testing data," here) in order to understand how well it generalizes to new data and would apply in practice. In the data in this chapter, we need to add labels to the testing data using the "ground truth" dataset because the testing data is *not* run-to-failure data and thus does not have the failure time implied. After we add the labels to the testing data of when the failures really happened, we can evaluate the model by comparing the ground truth to the predicted failure time.

When evaluating the regression model, important metrics include the *mean absolute error*, *root mean squared error*, *relative absolute*

error, *relative squared error*, and *coefficient of determination*. These are standard metrics that have formal definitions.[12] As an example, a model that has a coefficient of determination of 0.41 (the scale is from 0–1) can explain 41 percent of the variance in the remaining useful life. As another example, if the model had a mean absolute error of 7.1 and the RUL was measured in days, this would imply the model is on average 7.1 days off in predicting RUL. Deciding what the target metric should be for a "good" model is highly dependent on the business scenario. The following example code creates data labels for the testing data:

```
## Add data labels for test data
truth_df <- rxImport(truth_data_table)
#add index to the original truth table
truth_df$id <- 1:nrow(truth_df)
tagged_table_name <- "test_Labels"
tagged_table_test = RxSqlServerData(table =
tagged_table_name,

colClasses = train_columns,

connectionString = connection_string)
inDataSource <- RxSqlServerData(table =
test_table_name,
                                connectionString =
connection_string,
                                colClasses =
train_columns,
```

[12] See the posts by Shaheen Gauher and Said Bleik at aka.ms/cite7, aka.ms/cite8, and aka.ms/cite9.

```
                                    rowsPerRead = 30000)
rxDataStep(inData = inDataSource,
           outFile = tagged_table_test,
           overwrite = TRUE,
           transformObjects = list(truth =
truth_df),
           transformFunc = data_label,
           rowsPerRead = -1,
           reportProgress = 3)
```

For binary classification, in which the model is predicting whether the asset will fail in the next *X* days, standard metrics in use include *accuracy*, *precision*, *recall*, and *f-score*. It is important not to examine one metric in isolation, especially when the data is imbalanced.

For example, if only 1 percent of the data shows failures, a model could be built with 99 percent accuracy by simply predicting everything the model sees as a nonfailure. However, this model would not be of any use. For a more detailed discussion of this topic as well as more advanced code that calculates not only these metrics but also baseline metrics against which one can compare (such as what the accuracy, recall, and precision would be based on the distribution of failures), see footnote 12.

Note Although not shown here, it is very common to iterate through the previous steps (such as creating more features or revisiting the problem formulation) after evaluating the

model and investigating how the model can be improved.

Finally, after you create the functions for evaluation, the model is applied to the testing data and evaluated by using the ground truth dataset.

The following code example predicts the RUL in the testing data by using a decision forest model and evaluating how well the model performs using the `evaluate_model` function defined for the regression model we saw earlier:

```
## Import test into data frame for faster prediction
and model evaluation
test_table_name <- "test_Features"
test_table <- RxSqlServerData(table =
test_table_name,
                                connectionString =
connection_string)
prediction_df <- rxImport(inData = test_table)
rxSetComputeContext(local)
forest_prediction <- rxPredict(modelObject =
forest_model,
                               data = prediction_df,
                               predVarNames =
"Forest_Prediction",
                               overwrite = TRUE)

forest_metrics <- evaluate_model(observed =
prediction_df$RUL,
                                 predicted =
forest_prediction$Forest_Prediction)
```

The full code at http://aka.ms/cite3 applies all of the models to the testing data and uses the evaluation functions to evaluate them, because

in practice more than one model will usually be applied. The models are compared to determine the best one for the prediction.

Deploy the model

Figure 5-10 shows the overall work flow, which is fully automated by using a Windows PowerShell script, which you can find at http://aka.ms/cite12. The light-green blocks represent each step of the process. Each step interacts with SQL server, either by performing SQL table operations or invoking R through stored procedures.

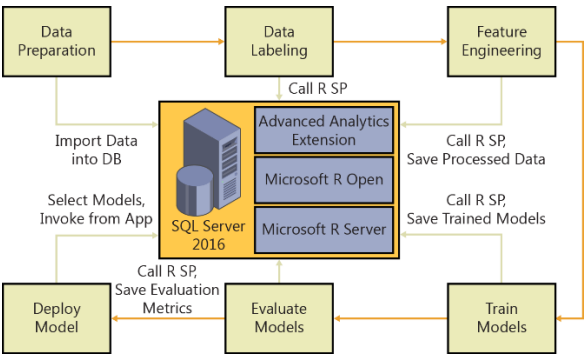


Figure 5-10: Workflow automation.[13]

Although the entire end-to-end process is automated in the code found at

[13] You can see more details at aka.ms/cite12.

http://aka.ms/cite12—from uploading the data to training the models to deploying the models—in production scenarios, you do not need to automate all of these steps.

For example, you can follow the steps in the previous sections in training the models in a local R-IDE, and you might want to deploy the model by simply using a stored procedure.

When you deploy the model, you do not need to "label" the data, because the truth of when the unit is going to fail is not known and is simply predicted using the model. However, the data processing done through the feature engineering step is still critical and necessary to use the trained model to get predictions. The stored procedure must do the feature engineering and then use the model to predict the outcome.

For deploying-to-production-scoring scenarios, use the following steps:

1. Create a raw dataset that needs predictions in a SQL table.

   The code associated with this chapter assumes this data is in a table called `PM_Score` in the SQL Server. For demonstration purposes, the data used for

scoring is taken from testing a dataset with engine id as 2 and 3.

2. Call the feature engineering SQL script.

   You should use DataProcessing\feature_engineering_scoring.sql as well as the results in the SQL table `score_Features_Normalized`, which then contains the data with new features and normalized.

3. Call the model SQL script.

   a. Regression model: Regression\score_regression_model.sql

   b. Binary classification model: BinaryClassification\score_binaryclass_model.sql

   c. Multiclass classification model: MultiClassification\score_multiclass_model.sql

The results contain the predictions; for example, SQL table `Regression score [model_name]`, scoring result for regression model.

As an example, here is the binary classification SQL stored procedure:

```
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

drop procedure if exists score_binaryclass_model;
go

CREATE PROCEDURE [score_binaryclass_model]
@modelname varchar(20),

@connectionString varchar(300)

AS
BEGIN
  DECLARE @inquery NVARCHAR(max) = N'SELECT * FROM
score_Features_Normalized';
  declare @model varbinary(max) = (select model from
[PM_Models] where model_name = @modelname);

  EXEC sp_execute_external_script @language = N'R',
                                  @script = N'
########################################################
########################################
## Get score table data for prediction
########################################################
########################################
prediction_df <- InputDataSet

########################################################
########################################
## Binary classification prediction
########################################################
########################################
model <- unserialize(model)
prediction <- rxPredict(modelObject = model,
                        data = prediction_df,
                        type = "prob",
                        overwrite = TRUE)
threshold <- 0.5
names(prediction) <- c("Probability")
```

```
prediction$Prediction <-
ifelse(prediction$Probability > threshold, 1, 0)

##################################################
##########################################
## Write score results to SQL
##################################################
##########################################
predictions <- cbind(prediction_df$id,
prediction_df$cycle_orig, prediction)
colnames(predictions)[1] <- "id"
colnames(predictions)[2] <- "cycle"
prediction_table <- RxSqlServerData(table =
paste("Binaryclass_score", modelname, sep = "_"),
                                    connectionString
= connection_string)
rxDataStep(inData = predictions,
           outFile = prediction_table,
           overwrite = TRUE)'
, @input_data_1 = @inquery
, @params = N'@model varbinary(max), @modelname
varchar(20), @connection_string varchar(300)'
, @model = @model
, @modelname = @modelname
, @connection_string = @connectionString

END

GO
```

The full code for this scenario is at http://aka.ms/cite12. You can use these stored procedures from other code to get predictions whenever new data is available and you want a prediction.

Summary

In this chapter, we gave you an overview of predictive maintenance use cases, with examples

of several different ways of formulating the business problem into a machine learning model. We covered the data science process of how to create the end-to-end solution using SQL R Services. We also took you on a step-by-step walk-through leading up to the deployment of the solution through SQL stored procedures. You can find the full code for both R and SQL for these scenarios at http://aka.ms/cite3.

CHAPTER 6

Forecasting

"Forecasting is the art of saying what will happen, and then explaining why it didn't!"
—Anonymous

Forecasting is used widely in many applications and critical business decisions that depend on having as accurate a picture of the future as possible. Meteorologists use it to generate weather predictions, CFOs use it to generate revenue forecasts, Wall Street analysts use it to predict stock prices, and inventory managers use it to

forecast demand and supply of materials. Many businesses use qualitative judgement–based forecasting methods and typically manage their forecasts in Microsoft Excel. Organizations face significant challenges with this approach as the amount and availability of relevant data has grown exponentially. Using Microsoft SQL Server R Services, it is possible to create statistically reliable forecasts in an automated fashion giving organizations greater confidence and business responsiveness. This chapter gives an overview of using SQL Server R Services for forecasting, including basic

principles of forecasting, some common forecasting scenarios, and step-by-step instructions on creating statistical forecasts using Microsoft SQL Server 2016 and R.

Introduction to forecasting

Forecasting is the process of making predictions based on historical data. This data includes trends, seasonal patterns, exogenous factors, and any other available pertinent data. It's essential to many important business decisions. Let's take a look at some common forecast scenarios used in organizations today.

Financial forecasting

Financial forecasting is the process of forecasting key financial metrics such as revenue, costs, and profit. Financial forecasting can also forecast key performance indicators (KPIs) of revenue, costs and profit, such as new customers or average selling price. The purpose of financial forecasting

is to manage an organization's financial performance versus its financial plan. Companies must monitor sales and costs to ensure that the organization is on track to deliver the financials required in the plan to fund ongoing operations such as manufacturing, investments in new products, and hiring. Financial forecasting is an essential function in the financial performance management process.

Demand forecasting

Demand forecasting is a critical step in the supply-chain management of an organization. The purpose of demand forecasting is to help inform how many units of a product (sometimes referred to as a *SKU*) will be in demand by its customers in the future so that the company can plan for the amount of raw materials it needs to purchase, by when, and where the manufacturing should occur. Demand forecasting is a critical element to ensure that the right amount is manufactured in order to minimize inventory costs and stock-outs.

Demand forecasting is often unique to an industry sector. For example, forecasting demand in the energy industry involves estimating the amount of energy customers will likely use in the future. Demand forecasting in the manufacturing industry is focused on supply

chain and manufacturing optimization. In the healthcare industry, it might involve forecasting the number of patients who will need healthcare services to plan on how many hospital beds to make available.

Supply forecasting

Supply forecasting is used to forecast the supply of inputs needed to create an output. It is often used in close conjunction with demand forecasting. For example, a municipal water organization will forecast not only the demand for water but also how much water will be available to supply that customer demand. A healthcare organization might forecast the supply of skilled healthcare professionals that will be available to meet the demand for future healthcare services.

Forecasting accuracy

A publicly held corporation must provide forward-looking guidance to its investors. The cost of inaccurate forecasts can be significant. A missed forecast can negatively affect the stock price and eliminate billions of dollars of market capitalization. Overoptimistic forecasting can result in manufacturing too many items that will need to be disposed of, which can negatively

affect margins and profitability. A low forecast can result in stock-outs and missed revenue opportunities.

Most organizations produce forecasts in Excel or, less frequently, with software packages that rely on basic statistical analysis. Manual Excel-based forecasting relies on substantial human judgement and error-prone manual consolidation and collaboration. This manual work also makes forecasting a time-consuming process and renders it impractical to integrate new data streams as they become available. This prevents organizations from being agile, causing them to be less able to adapt to rapidly changing business environments.

An intelligent and automated forecasting environment can help organizations manage increasing volumes of data, generate more accurate forecasts and increase the frequency of generating forecasts giving organizations greater agility to respond to dynamically changing business conditions. The benefit of such a solution is increased forecast accuracy, timeliness, and efficiency versus current approaches.

Forecasting tools

There are several tools available today for forecasting. Following are three examples.

Excel

Excel is one of the most popular tools for forecasting. It is usually used for rules-based forecasting. For example, a rules-based forecast might begin with an assumed revenue growth rate, and spread the revenue over four quarters based on a rule (20 percent in Q1, 20 percent in Q2, 25 percent in Q3, and 35 percent in Q4).

Forecasting software

There are several software packages available that perform statistical forecasting. These packages typically use time–series methods for forecasting. They also might include some end-user reporting. A couple of examples are ForecastPro and ForecastX.

R

The R statistical language provides significant flexibility in creating forecast models. R enjoys a large collection of packages that can make forecasting easier. One of the most popular of them is the *forecast package for R* by Rob

Hyndman which includes several of the most popular time–series forecasting methods.

Statistical models for forecasting

Now that we've examined some of the use cases and tools for forecasting, let's take a look at the various models that we can use for the process.

Time–series analysis

There are two important characteristics of time series data. The first is that data is ordered by…you guessed it: *time*. In many other datasets discussed in this book, the order of the rows is not necessarily important. However, for time–series data, order *does* matter and the order is defined by the column that stores the time sequence.

Another important characteristic of time–series data is that intervals between time stamps are equally spaced. A dataset of annual sales that has weekly data for five months and then monthly data for the remaining seven months will cause problems for forecasting.

Some types of time–series analysis can reveal two types of relationships: *trend* and *seasonality*.

Other methods allow for decomposition of time series–data into three components: *trend*, *seasonality*, and *residual noise*.

Let's take a closer look at trend and seasonality to see what we can learn from them.

Trend

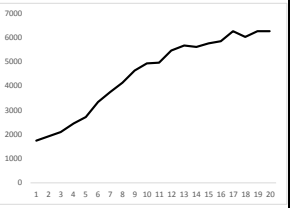
Time–series analysis can reveal if there is a long-term trend, either declining or increasing. A chart of the population of the United States over time would show a long-term increasing trend, whereas a chart of the percent of adults in that same population who smoke would show a decreasing trend over time.

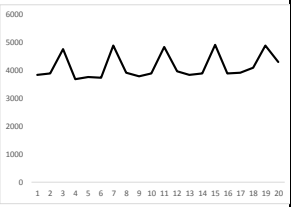
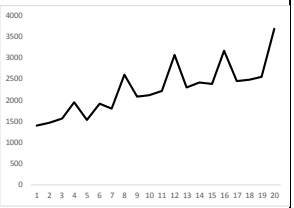
Seasonality

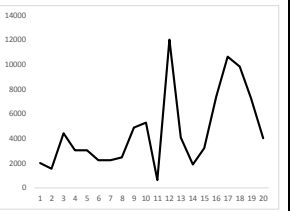
Time–series analysis can reveal if there are seasonal patterns in the data. For example, for certain consumer products, Black Friday sales might always spike, suggesting that this year's Week 4 of November sales is correlated with last year's and Week 4 November sales for the previous years.

Table 6-1 illustrates the characteristics of several time–series datasets that reveal trend and seasonal patterns.

Table 6-1: Examples of time series datasets illustrating trend and seasonal characteristics

Trend	Seasonality	Example	
✓	✗		The time–series data shows a long-term year-over-year *increasing* trend. Example: smart phone sales.

✗	✓		The time–series data shows an evenly spaced *seasonal* pattern with no trend. Example: a company's sales that spike every quarter.
✓	✓		The time–series data shows both a long-term increasin

			g *trend* and a regularly spaced *seasonal* pattern. Example: ice cream sales.
✘	✘		The time–series data shows neither a clear trend nor a seasonal pattern. Note: time series methods are not suitable

			for this data set.

Time–series forecasting

Forecasting distinctly deals with predicting a number. The number to forecast can represent various measurements. Let's take a look at some those measurements.

Amount in a currency

Examples include revenue in US dollars, people costs in Euros, or the foreign exchange rate between two currencies.

Percentages

Examples include interest rates, product return rate, percent of warranty claims filed, and annual rate of unemployment.

Units

Examples include the number of new customers, number of smart phones, and gallons of water demanded.

Forecasting almost always involves time–series data. Examples of this data include weekly actual sales of smartphones from 2013 to 2016, or daily exchange rates between the Euro and US dollar.

In both cases the dataset has two columns: time and value.

Finally, one more list—here are the three types of time–series models you'll see most often:

- Autoregressive Integrated Moving Average (ARIMA)
- Seasonal decomposition of Time–Series by Loess (STL)
- Exponential Smoothing (ETS)

As with most statistical analysis, correlation is a necessary condition for modeling. With time–series analysis there is a special kind of correlation: the correlation of a specific time-stamped value with a previous time-stamped value. In other words, the algorithm seeks to understand if a given period's values are correlated with a previous period's values. For example, time–series analysis can determine if today's sales are related to the sales from yesterday or the sales from two days ago. This correlation has a special name: *autocorrelation*. ARIMA models explicitly learn this autocorrelation, or autoregression. These models also learn the relationship between the current time point and *previous unobserved error* terms, also known as a *moving average*.

ETS and STL decompose a time–series dataset into its constituent parts and makes it easier to forecast the constituent parts and reaggregate the parts into a total forecast. There are models that sum the parts (called *additive* models), models that multiply the parts (called *multiplicative* models), and several other variations. Figure 6-1 illustrates how a time–series dataset at the top has been decomposed into the three parts. After it is decomposed, the ETS and STL time–series analysis forecasts the trend and seasonal components. For the trend component, it is clear that the next forecasted point will be somewhat higher. For the seasonal component, it seems clear that the next period will be lower. When the time series model has forecast the next period values for the trend and seasonal components, the algorithm aggregates the forecasted points to get a new forecasted value.

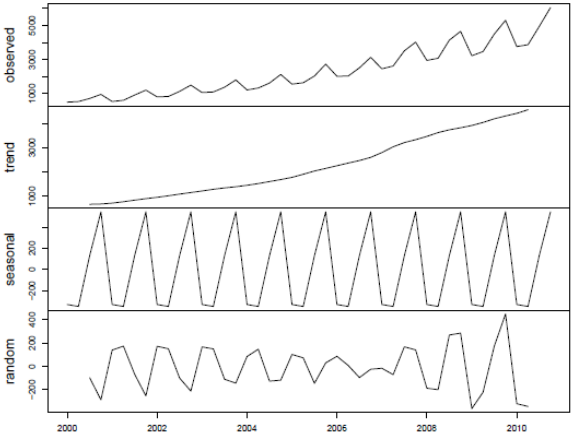


Figure 6-1: Decomposition of additive time series showing four line charts stacked on top of each other.

In Figure 6-1, the line chart at the top is the actual historical time–series data; the line chart below that shows the elements that represent the trend component of the actual line chart; the next chart shows the elements that represent the seasonal components of the actual line chart, and the fourth line chart shows the remaining random noise components.

Forecasting by using SQL Server R Services

Let's take a look at a sample forecasting solution built by using SQL Server R Services.[24] We 'll train and evaluate a forecasting model using a single time series from the publicly available M3 competition dataset. Our sample forecasting solution comprises the following steps:

4. Upload data to the SQL Server
5. Split the data into training and testing
6. Train and score time–series forecasting models
7. Generate accuracy metrics

Upload data to SQL Server

First, we'll get the N1725 time–series data from the publicly available M3 competition dataset (https://forecasters.org/resources/time-series-

[24] This sample is similar to the Time–Series Forecasting experiment in the Cortana Intelligence Gallery; http://gallery.cortanaintelligence.com/Experiment/Time-Series-Forecasting-8

data/), which is available in the M3Month tab in the M3 dataset, and upload that data to SQL Server using SQL Server Integration Services or a simple query from a text file. This dataset has 126 rows and two columns, **`time`** and **`value`**.

Figure 6-2 shows a snapshot of the data.

	A	B
1	time	value
2	1	3740
3	2	2880
4	3	2620
5	4	2860
6	5	2640
7	6	2740
8	7	2940
9	8	4040
10	9	3800
11	10	3640

Figure 6-2: A screenshot showing 2 columns and the first 10 rows. Column 1 is the ordered time stamp (from 1–10) and column two are the first 10 actual historical values.

We've created a database called *demodb* in SQL Server 2016. Next, we've uploaded the time–series dataset to the *demodb* database by using the Import Data function in SQL Server

Management Studio (https://msdn.microsoft.com/library/ms140052.aspx). This data is stored in the `TimeSeriesForecastingData` table in *demodb* database, as shown in Figure 6-3.

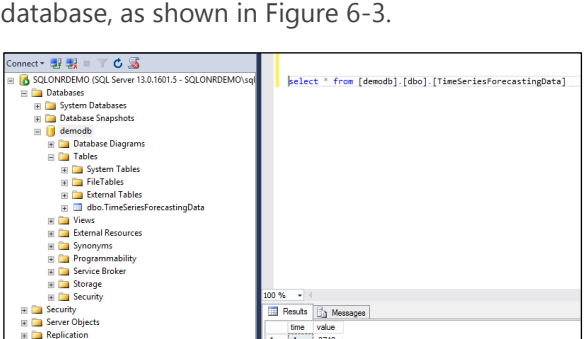


Figure 6-3: SQL Management Studio with the query and expected output after the the query has been run.

Splitting data into training and testing

After the data is in the SQL Server database, we are ready to build forecasting models using R. We decide to split the data into training and testing sets. Let's pick the first 108 time points as the training set and forecast the future 18 time points. We will evaluate the accuracy of our

forecasting models on these 18 points. The SQL query that follows encapsulates an R script that divides the data into these training and testing sets. We will extend this R script to train and score forecasting models in the next section.

```
execute sp_execute_external_script
         @language = N'R'
        , @script = N'
#training dataset
                  dataset1 <-
InputDataSet[InputDataSet$time <= 108,]

                  #testing dataset
                  dataset2 <-
InputDataSet[InputDataSet$time > 108,]
 '
        , @input_data_1 = N'select * from
TimeSeriesForecastingData'
        , @input_data_1_name = N'InputDataSet'
        , @output_data_1_name=N'dataset1'
        with result sets (("time" int,"value"
float));
```

Training and scoring time–series forecasting models

We will use the *forecast25 package for R* to train and score our time–series forecasting models. We'll need to install the forecast package into

---

[25] Hyndman R. J. 2016. forecast: Forecasting functions for time series and linear models. R package version 7.1; http://github.com/robjhyndman/forecast.

our R library on the SQL Server instance. To do so, open the RGui console in administrator mode, and then, on the server, install the forecast package and its dependencies by using the following commands:

```
lib.sql <- "C:/Program Files/Microsoft SQL
Server/MSSQL13.MSSQLSERVER/R_SERVICES/library/"
install.packages("forecast",lib=lib.sql)
install.packages("quadprog",lib=lib.sql)
install.packages("zoo",lib=lib.sql)
install.packages("timeDate",lib=lib.sql)
install.packages("tseries",lib=lib.sql)
install.packages("fracdiff",lib=lib.sql)
install.packages("Rcpp",lib=lib.sql)
install.packages("colorspace",lib=lib.sql)
install.packages("RcppArmadillo",lib=lib.sql)
```

After the R packages are installed, you can now proceed to writing the model training and scoring the R script. Because the patterns in our data repeat after every 12 time points, we set the frequency parameter to 12 when constructing the time series object. We use the ETS function available in the forecast package. The `ets()` function call returns a trained ETS model. We'll use this model in the forecast function to generate the forecast, as shown the following SQL query:

```
execute sp_execute_external_script
     @language = N'R'
    , @script = N'
        require("forecast");

        seasonality<-12

        #training dataset
        dataset1 <- InputDataSet[InputDataSet$time
```

```
<= 108,]

        #testing dataset
        dataset2 <- InputDataSet[InputDataSet$time >
108,]

        #Create time series object
        labels <- as.numeric(dataset1$value)
        timeseries <-
ts(labels,frequency=seasonality)

        #Train an ETS model
model <- ets(timeseries)

#Compute the forecasting horizon
        model <- ets(timeseries)
        numPeriodsToForecast <-
ceiling(max(dataset2$time)) -
ceiling(max(dataset1$time))
        numPeriodsToForecast <-
max(numPeriodsToForecast, 0)

        #Use the trained ETS model to generate the
forecats
        forecastedData <- forecast(model,
h=numPeriodsToForecast)
        forecastedData <-
data.frame(as.numeric(forecastedData$mean))
        forecastDataDf <-
data.frame(time=dataset2$time,forecast=forecastedDat
a)

        #Merge the observed data with the forecasts
        output <- merge(x = InputDataSet, y =
forecastDataDf, by = "time", all.x = TRUE)
        colnames(output) <-
c("time","value","forecast")

 '
    , @input_data_1 = N'select * from
TimeSeriesForecastingData'
    , @input_data_1_name = N'InputDataSet'
    , @output_data_1_name=N'output'
    with result sets (("time" int,"value"
float,"forecast" float));
```

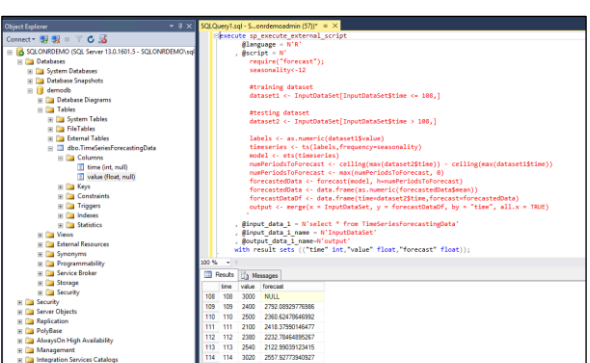


Figure 6-6: SQL Management Studio with the query and expected output after the query has been run. Now three columns are displayed: time, value, and forecast.

Generate accuracy metrics

After the forecasts are generated, we use Mean Absolute Percentage Error (MAPE) to compute the accuracy of our model. MAPE is defined as follows:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|A_i - F_i| * 100}{A_i}$$

Where A_i is the actual value at time *i*, and F_i is the forecast at the same time *i*. We include the following R script to our existing SQL query to generate the MAPE metric.

```
        output$ape <- NA
        output$ape[which(!is.na(output$forecast))]
<-
abs((output[which(!is.na(output$forecast)),c("value"
)]) -
(output[which(!is.na(output$forecast)),c("forecast")
]))/(output[which(!is.na(output$forecast)),c("value"
)])*100;
        mape <- data.frame(mean(output$ape,
na.rm=TRUE))
```

Summary

In this chapter, we provided an overview of time–series forecasting using SQL R Services. In addition, we described some common forecasting scenarios in enterprises today, introduced some popular statistical methods for time series forecasting, and followed-up with detailed steps on how to generate forecasts using SQL R Services.

About the Authors

Wee-Hyong Tok is a senior data scientist lead at Microsoft in the Algorithms and Data Science group. Wee-Hyong has decades of database systems experience, spanning academia and industry, including deep experience driving and shipping products and services that include distributed engineering teams from Asia and the United States. Before joining Microsoft, Tok worked on in-database analytics, demonstrating how association rule mining can be integrated into a relational database management system, Predator-Miner, which makes it possible for users to express data-mining operations using SQL queries and provides opportunities for better query optimization and processing.

Tok is instrumental in driving data-mining boot camps in Asia and was honored as a Microsoft SQL Server Most Valuable Professional for several consecutive years because of his active contributions to the database community throughout Asia. He has coauthored several

books, including the first book on Azure machine learning, *Predictive Analytics with Microsoft Azure Machine Learning*, and has also published more than 20 peer-reviewed academic papers and journals. He has a Ph.D. in computer science from the National University of Singapore.

Buck Woody works on the Microsoft Machine Learning and Data Science Team, using data and technology to educate others on solving business and science problems. With more than 30 years of professional and practical experience in computer data technologies, he is also a popular speaker at many conferences around the world. Buck is the author of more than 650 articles and 7 books on databases and machine learning technologies. In addition, he teaches database courses and sits on the Data Science Board at the University of Washington, and specializes in data analysis techniques.

Debraj GuhaThakurta is a senior data Scientist at Microsoft in the Algorithms and Data Science group. His effort focuses on the use of different platforms and toolkits such as Microsoft's Cortana Intelligence suite, Microsoft R Server, SQL Server, Hadoop, and Spark for creating scalable and operationalized analytical processes for business problems. Debraj has extensive industry experience in biopharma and financial forecasting domains. He has a Ph.D. in chemistry and biophysics, and post-doctoral research experience in machine learning applications in bio-informatics. He has published more than 25 peer-reviewed papers, book chapters, and patents

Danielle Dean is a senior data scientist lead at Microsoft in the Algorithms and Data Science group. She leads a team of data scientists and engineers on end-to-end analytics projects that use Microsoft's Cortana Intelligence Suite for applications ranging from automating the ingestion of data to analyzing and implementing algorithms, creating web services of these implementations, and integrating them into customer solutions or building end-user dashboards and visualizations. Danielle holds a Ph.D. in quantitative psychology from the University of North Carolina at Chapel Hill, where she studied the application of multilevel event history models to understand the timing and processes leading to events between dyads within social networks.

Gagan Bansal is a data scientist leading the development of financial forecasting capabilities in Cortana Analytics at Microsoft. Gagan joined Microsoft from Yahoo Labs, where he was a lead engineer building and deploying large-scale user modeling and scoring pipelines on both grid (Hadoop) and stream scoring systems for display-ad targeting applications. Prior to Yahoo!, he worked on social targeting in online advertising at 33Across. Before that, he worked for another startup where he was involved in the development of real-time video processing algorithms for advertising in sports broadcasts. Gagan obtained his masters in computer science from Johns Hopkins University, where he worked on pedestrian detection in videos for his thesis. Before that, he graduated with a Bachelors in Computer Science degree from Indian Institute of Technology, Delhi. Gagan enjoys working on problems related to machine learning, large-scale data processing, computer systems, and image processing.

Matt Conners is a senior data sciences program manager in Microsoft's Algorithms and Data Sciences group. He is focused on the forecasting domain, working with customers, partners, and data scientists to operationalize machine learning financial forecasting solutions. He has extensive business operations and industry domain experience, with more than 20 years' of financial technology experience across sales, marketing, business operations, securities, and banking. He has an undergraduate degree in economics, and master's degrees in finance and statistics.

From technical overviews to drilldowns on special topics, get free ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in three formats:

- PDF
- EPUB
- Mobi for Kindle

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.

Microsoft Press