JUST ENOUGH R

Learn data analysis with R in a day

Sivakumaran Raman

Includes a sample data-analysis using freely-available health care data! Download the sample programming-code for the book from https://www.dropbox.com/s/tyn5yabn49c1s6x/Just_Enough_R.zip?dl=0

Smashwords Edition

Cover Design: Sivakumaran Raman

The R logo on the cover (https://www.r-project.org/logo/) is the (c-2016) of the R Foundation (https://www.r-project.org/foundation/) and licensed under the Creative Commons Attribution-ShareAlike 4.0 International license (https://creativecommons.org/licenses/by-sa/4.0/).

This book was written using LibreOffice 4.2 on a notebook computer running Ubuntu Linux 14.04.

April-2017 First Edition

Sivakumaran Raman

Email: sraman9757@gmail.com

To my loving aunt Janavi, who introduced me to computer programming, and to Richard Stallman, the man who started the free and open-source software movement.

Contents

Whom is This Book For?

Preface

Preparation to Start

- Computer
- Installation of Java
- Installation of R and associated software and packages
- Text-Editor
- Data-sets
- Sample code and directory-structure

Chapter 1: Introduction

Chapter 2: R syntax

Chapter 3: Variables and Scope

- 3.1: Variables in R
- 3.2: Variable Scope

Chapter 4: Data Structures

- 4.1: Atomic Vector
- 4.2: Array
- 4.3: Matrix
- 4.4: List
- 4.5: Data Frame
- 4.6: Accessing members of a vector, list, or data frame

Chapter 5: Functions

- 5.1: Anonymous Self-Executing Functions
- 5.2: Passing a function all the variables it needs

Chapter 6: Objects

Chapter 7: R packages

Chapter 8: Interactive R through the REPL interface

Chapter 9: Inbuilt help in R

Chapter 10: R programs and scripts

Chapter 11: Reading in Data

Code Listing 11.1 (Initial data-read using data.table)
Code Listing 11.2 (Initial data-read using data.table: no functions version)
Code Listing 11.3 (Initial data-read using readr)
Code Listing 11.4 (Initial data-read from a relational database)

Chapter 12: Data Wrangling/Munging/Manipulation

Code Listing 12.1 (Data wrangling using dplyr)
Exercise 12.1: For the Reader

Chapter 13: Data Quality Checks in R

Code Listing 13.1 (Data quality checks using datacheck)
Exercise 13.1: For the Reader

Chapter 14: Descriptive Statistics and Visualization

Code Listing 14.1 (Descriptive statistics)
Code Listing 14.2 (Descriptive visualizations on the complete data set)
Figure 14.1 (Density Plot)
Figure 14.2 (Density Plot – logarithmic scale)
Figure 14.3 (Bar-chart of provider-count by state)
Figure 14.4 (Interactive Bar-chart of provider-count by state)
Code Listing 14.3 (Descriptive visualizations on the aggregated-by-provider dataset)
Figure 14.5 (Bar-charts of provider-gender faceted by country)
Exercise 14.1: For the Reader

Chapter 15: Interactive Charts and Plots

Code Listing 15.1 (Interactive plotly bar chart of Medicare 2014 payout by state)
Figure 15.1 (Interactive plotly bar-chart of Medicare payout by American state in 2014)
Code Listing 15.2 (Interactive plotly box-plot: Family Practice provider patients)
Figure 15.2 (Interactive plotly box plot: Beneficiaries served per Family Practice Provider in certain mid-Western American states in 2014)
Exercise 15.1: For the Reader

Chapter 16: Geographical Maps and Charts

Code Listing 16.1 (Static choropleth maps)
Figure 16.1 (Static choropleth of Medicare payout by US state)
Figure 16.2 (Static choropleth of Medicare provider-count by US state)
Code Listing 16.2 (Interactive choropleth map using plotly)
Figure 16.3 (Interactive choropleth of Medicare pro-fee payout by US

state)
Code Listing 16.3 (Interactive geographical scatterplot map using plotly)
Figure 16.4 (Interactive geographical scatterplot map of the top Medicare professional services revenue providers by US state)
Code Listing 16.4 (Interactive geographical scatterplot map using plotly)
Figure 16.5 (Interactive geographical scatterplot map of the top individual Medicare professional services revenue providers by US state)
Exercise 16.1: For the Reader
Exercise 16.2: For the Reader
Exercise 16.3: For the Reader

Chapter 17: Regression Modeling & Predictive Models

Code Listing 17.1 (Multiple linear regression model)
Figure 17.1 (Multiple linear regression results pretty-printed using ztable and stargazer)
Code Listing 17.2 (Multiple linear regression predictive model)
Figure 17.2 (Linear Model: Predicted against actual values for Medicare revenue)
Figure 17.3 (Linear Model: Predicted against residual values for Medicare revenue)
Exercise 17.1: For the Reader

Chapter 18: Machine Learning & Predictive Models

Code Listing 18.1 (Gradient Boosted Machine predictive model in H2O)
Figure 18.1 (GBM: Predicted against actual values for Medicare revenue)
Figure 18.2 (GBM: Predicted against residual values for Medicare revenue)
Code Listing 18.2 (Random Forests predictive model in H2O)
Figure 18.3 (Random Forest: Predicted against actual values for Medicare revenue)
Figure 18.4 (Random Forest: Predicted against residual values for Medicare revenue)
Exercise 18.1: For the Reader

Epilogue

Author Information

Whom is This Book For?

If your job involves working with data in any manner, you cannot afford to ignore the R revolution! If your domain is called data analysis, analytics, informatics, data science, reporting, business intelligence, data management, big data, or visualization, you just ***have*** to learn R as this programming language is a game-changing sledgehammer.

However, if you have looked at a standard text on R or read some of the online discussions, you might feel that there is a steep learning curve of six months or more to grok the language. I will debunk this myth through my book by focusing on practical essentials instead of theory.

If you have programmed in some language in the past (whether that language be SAS, SPSS, C, C++, C#, Java, Python, Perl, Visual Basic, Ruby, Scala, shell scripts, or plain old SQL), even if you are rusty, this book will get you up and running with R in a single day, writing programs for data analysis and visualization.

At the end of this book you will be able to:

- write R programs to execute on the 3 major data-analysis phases.

- visualize data in an illustrative and interactive manner

- move on to using R for big data analytics

R you excited? You should be. Let us charge forward!

Preface

R (https://en.wikipedia.org/wiki/R_(programming_language)) is an interpreted, open-source, free, statistical-programming and data-analysis language. It was created by Ross Ihaka and Robert Gentleman. It is a functional language and has all the standard programming features like variables, functions, objects, loops, and data-structures.

R is perfect for data analysis and visualization. Though R can, in theory, be used for tasks like web programming and building software applications, it is not optimized for these purposes and is not preferred for these tasks. R was created in 1993 and has become very popular because of the rapid growth of the domains of big data, data science, visualization, and analytics.

The aim of this book is to teach the elements of R programming in a single day. This book is meant for people who already know how to *program in at least one language* and want to learn R. After completing this book, the reader should be able to write simple R programs for data analysis. Instead of adopting a spoon-feeding approach, I assume that the reader is familiar with standard programming constructs like variables, functions and the like – therefore, I only outline differences in the way R does things. The emphasis is on writing and running programs in R for data analysis and visualization. The book includes a sample data-analysis conducted on freely available CMS-sourced (CMS: Centers for Medicare and Medicaid Services) healthcare data. The book does not aim to teach all the elements of statistics, machine learning or data science – since doing so would expand the scope of the book immensely.

Unlike many standard texts on R, the book teaches the most effective way to accomplish any specific task in R. No effort is made to teach *all* the ways in which a particular task can be completed: No TMTOWTDI (https://en.wikipedia.org/wiki/There's_more_than_one_way_to_do_it)!

All through the text, I provide a lot of Internet links to more information and detail. This is one of the great things about open-source software – it is usually supported by a very active web-based community of users and almost all the answers to questions newbies might have can be found online. The R community is one of the largest and best in this regard. Lastly, instead of laying out all the theory behind R programming (for which there are numerous other sources on the Internet), the emphasis is on *learning by doing* – the code samples provided throughout the book should be read and understood line by line. The reader should make an effort to complete the practice exercises offered at the ends of certain chapters.

Preparation to Start

Computer

Any Windows® or Linux machine can be used. I would recommend at least 8 GB of Random Access Memory be available on the computer.

The R programs used in this book were run on two different computers:

- R version 3.3.2 on a Windows laptop running Windows 10 Pro, Intel(R) Core(TM) i5-2520M CPU @ 2.50 GHz, 8Gb RAM, L3 cache size 3072 KB
- R version 3.3.3 on a Linux laptop running Ubuntu 14.04, Intel(R) Celeron(R) CPU 1007U @ 1.50GHz, 8Gb RAM, L3 cache size 2048 KB

R is available for Mac and other platforms as well – interested readers can use these.

Installation of Java

Some of the R packages we will be using are wrappers around Java-based libraries and thus require the Java Runtime Environment (JRE) to be installed on the computer. Please install the latest version of the Oracle JRE (http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html) if you are on Windows. On Linux, you can install either the OpenJDK (http://openjdk.java.net/) Java Runtime (using apt-get or a similar software installation tool) or the Oracle JRE for Linux.

After installation, ensure that the **java** executable is in the PATH. This can be tested by running the java -version command at the Linux (bash) shell or Windows command line (cmd.exe or powershell.exe) and seeing if the appropriate message appears:

Linux bash shell:

```
radium@aceraspiredelto:~$ java -version
java version "1.7.0_121"
OpenJDK Runtime Environment (IcedTea 2.6.8) (7u121-2.6.8-
1ubuntu0.14.04.3)
OpenJDK 64-Bit Server VM (build 24.121-b00, mixed mode)
```

Windows cmd.exe shell:

```
C:\Users\shiminty\Desktop>java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

If the java executable is not in the PATH, please edit the PATH variable (at the system or user level) and add the path to java (java.exe on Windows) to the PATH variable.

Installation of R and associated software and packages

For Windows, R binaries can be downloaded and installed from the R website (https://www.r-project.org/). After installing R on Windows, please edit the PATH variable (at the system or user level) and add the paths to R.exe, Rscript.exe to the variable.

For installation of R on Linux, it is best to use the software package management tool for your Linux distribution. For Debian and Ubuntu Linux, the tool to use is **apt-get**. Linux installs of R using tools like apt-get mostly add the paths to the R and Rscript executables to the PATH variable. However, if this is not the case, please modify your PATH variable on Linux.

After R has been installed, install the R packages we will need by running the install.packages() command within R with a list of supplied package names. First, start up the R interactive-session (also called a Read-Eval-Print-Loop or REPL) by typing R at the command line. Then run the install.packages() command copied from the text-box below with the full list of packages to be installed. Make sure you are connected to the Internet and choose a CRAN (Comprehensive R Archive Network) package repository mirror close to your geographical location. If R warns you about the fact that it is installing the packages in a user-level local repository (since you are running R on the machine without admin or root privileges), it is not a cause for concern: Respond with a **Yes** to this message, and proceed.

On Linux, the command line session looks like this (list of R packages included):

```
radium@aceraspiredelto:~$ R

R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

 Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

```
> install.packages(c("broom", "choroplethr", "data.table",
 "datacheck", "dplyr", "dtplyr", "ggplot2", "ggvis",
 "h2o", "htmlwidgets", "httr",  "jsonlite", "leaflet",
 "maps", "maptools", "OpenStreetMap", "plotly",
 "randomForest", "R2HTML", "RDSTK", "readr", "rjson",
 "rpart", "RSQLite", "scales", "sqldf", "stargazer",
 "svglite", "tidyr", "tmap", "ztable"));
> q()
Save workspace image? [y/n/c]: n
radium@aceraspiredelto:~$
```

*Note: On Linux, some package and software dependencies might crop up while installing the **svglite** package or other R packages. The **svglite** package depends on **gdtools**. But the installation of **gdtools** first requires the* Cairo *(https://www.cairographics.org/download/) graphics software developer libraries to be installed using apt-get or similar software package tool on Linux.The way to do it on Ubuntu/Debian Linux is:*

sudo apt-get install libcairo2-dev

After this, re-running the install.packages() command for **svglite** within the R REPL should work smoothly:

```
radium@aceraspiredelto:~$ R
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
  Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
> install.packages(c("svglite"));
> q()
```

Re-running the install.packages() command for all of the packages also does not cause any problems as already-present packages are just re-installed.

Text-Editor

A good text editor with features like code-folding, syntax highlighting, and auto-completion is essential to be productive while programming in R. There are a whole host of open-source and free text-editors available but I recommend the following:

- For Windows: Notepad++ (https://notepad-plus-plus.org/)
- For Linux: Geany (https://www.geany.org/)

Note: Notepadqq *(http://notepadqq.altervista.org/s/), which is a Linux-platform clone of Notepad++, is also available for interested users. The only reason I am not recommending Notepadqq as the first-choice text-editor on Linux is because some problems have been reported relating to the version of* Qt *(https://www.qt.io/) installed on the machine.*

Download and install the text-editor of your choice for your computer's platform.

Data-sets

The dataset used in the analysis executed in this book is the freely available Physician and Other Supplier Data Calendar Year 2014 *(https://www.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/Medicare-Provider-Charge-Data/Physician-and-Other-Supplier2014.html)* from the **Centers for Medicare and Medicaid Services** (http://cms.gov). Download the tab-delimited text file (inside the zip file) available on the website here: http://www.cms.gov/apps/ama/license.asp?file=http://download.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/Medicare-Provider-Charge-Data/Downloads/Medicare_Provider_Util_Payment_PUF_CY2014.zip. Also download the PDF document on methodology that explains the file structure, format, and fields from here: https://www.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/Medicare-Provider-Charge-Data/Downloads/Medicare-Physician-and-Other-Supplier-PUF-Methodology.pdf

Sample code and directory-structure

The sample code can be downloaded as a zip file from this Dropbox link I have created: https://www.dropbox.com/s/tyn5yabn49c1s6x/Just_Enough_R.zip?dl=0. Extracting out the zip file to disk creates this tree-structure of folders and files:

```
Just_Enough_R
|--Chapter_003_Code__Variables_and_Scope_in_R/
|--Chapter_011_Code__Reading_Data_into_R/
|--Chapter_012_Code__Data_Wrangling_in_R/
|--Chapter_013_Code__Data_quality_checks_in_R
|--Chapter_014_Code__Descriptive_statistics_and_visualization_in_R/
|--
Chapter_015_Code__Interactive_Charts_and_Plots_using_plotly_in_R
/
|--Chapter_016_Code__Geographical_Maps_in_R/
|--
Chapter_017_Code__Linear_Regression_and_Predictive_Models_in_R
/
|--
Chapter_018_Code__Machine_Learning_and_Predictive_Models_in_
R/
|--data/
|--
LICENSE_FOR_BOOK_CONTENT_EXCLUDING_SOFTWARE_C
ODE.txt
|--LICENSE_FOR_SOFTWARE.txt
```

Download the previously-named dataset from the CMS website. Extract out the ***"Medicare_Provider_Util_Payment_PUF_CY2014.txt"*** file from the dataset's zip-archive file. Move the ***Medicare_Provider_Util_Payment_PUF_CY2014.txt*** file to the ***data*** folder listed above in the tree diagram.

Chapter 1: Introduction

R is an interpreted programming language. A good introduction to how R works is available here: https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

Code written in text-files can be directly run by the R interpreter without the need for a compilation step. While executing a program, R stores data-structures, variables, functions, and other entities created during the course of the program as objects in memory. In fact, even the results of functions and statistical procedures are objects that can be examined, dissected, and pretty-printed. Functions are first-class objects in R and can be directly manipulated within programs in many different ways.

Since R is primarily used for data-analysis (if we ignore the Bioinformatics and Natural Language Processing capabilities of the language for now), it is not an over-simplification to state that all data-analyses involve just three phases:

1. Reading in the data.

2. Data munging or data wrangling (to convert the data to a form or format that makes it easy to analyze).

3. Running the analysis and publishing the results (which might include visualization).

Each one of the three phases can have *more than one* iteration. In some cases, the phases might repeat or cycle. For example to run a different kind of analysis, a different kind of data-wrangling might be needed. But broadly speaking, the *3-phases of data-analysis* simplification holds.

Since R is a functional programming language, *functions* are first-class objects that can be utilized broadly and even passed to other functions as arguments to be acted on (more on this later).

R has all the programming concepts like variables, scope, functions, and loops that are found in modern programming-languages.

R can be run interactively (one statement at a time) using the R shell-interface which can also be described as a REPL (Read-Eval-Print-Loop). This R shell-interface is invoked by typing the command R in the operating system's command shell:

```
radium@aceraspiredelto:~$ R
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
  Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
```

However, though the REPL is useful for exploratory analysis and training sessions, we will be restricting its use to installing packages (as we did earlier in the *Preparation* chapter) and exploring metadata. In most cases, for the purposes of data analysis, we will use a text-editor to write R-programs in files that can then be executed by the R interpreter.

Chapter 2: R syntax

R has an easy-to-learn syntax similar to languages like C, Java, and Perl.

Statements to be executed are usually placed in lines in a program (unless using the interactive Read-Eval-Print-Loop or REPL which we will cover later).

The usual practice is to have one statement per line. e.g.

```
myvar1 <- 45
```

However, multi-line statements can be used if care is taken to let R know that the statement is not complete – this is usually done by ending the line with an operator like +, a comma, <- , etc. This indicates to R that the subsequent line has more code to be read. e.g.

```
mynumber <- 2 + 5 + 5 + 6 + 7 +
 8
```

In this case, mynumber will be equal to 33.

However, this is the incorrect way to write a multi-line statement:

```
mynumber <- 2 + 5 + 5 + 6 + 7
 + 8
```

In this case, mynumber will be equal to 25 since the first line is a complete R statement on its own.

White-space is mostly not significant, except inside strings.

A statement can be ended with a semi-colon (like in Java or C) but this is not mandatory. The semi-colon is useful when placing multiple statements on the same line. Modern R style-guides recommend not using the semi-colon as a statement terminator.

Variables are declared without type information (which is inferred from the *value-side* of the assignment operator <- or =). Variable names have to follow rules: for example, they cannot start with numbers. Access these rules by typing ?make.names in the R REPL. Backticks (`) can be used, to accommodate special cases, to enclose variable names that violate these rules.

Simple variable types include integer, logical, double (numeric), and character.

Complex data-structures (to be covered later) can also be represented by variables.

Strings (which have the type of character in R) can be single-quoted or double-quoted. Characters that have special meanings inside strings include \r, \n, \\, \t, etc.

Single quotes can appear as characters in double-quoted strings and double-quotes as

characters in single-quoted strings. However, to place double-quotes inside double-quoted strings (and vice versa), they have to be escaped with the backslash: \"

Strings can be multi-line as long as the closing quotes are correctly placed and matched.

Single line comments start with a hash (#) sign. The R interpreter ignores these lines for execution purposes.

There is no mechanism for multi-line comments in R. However, one workaround is to create a multi-line string variable holding the comment.

```
mycomment1 <- "This is a multi-line comment created as
 an example"
```

Loops (like for and while loops) and conditional statements (like if-then-else and ifelse) utilize standard syntaxes. Curly braces are commonly used to encapsulate parts of these loops and conditionals.

Chapter 3: Variables and Scope

3.1: Variables in R

A very good introduction to variables is provided here:
https://www.tutorialspoint.com/r/r_variables.htm

As in most modern programming languages, variables help to tag pieces of data and information with names. These named pieces of information can be acted upon and manipulated by R programs. Variables can be initialized with values using the leftward-assignment, rightward-assignment, or the equals operators. Thus, these three statements are synonymous:

```
num1 <- 4;
4 -> num1;
num1 = 4;
```

Variables can represent simple entity/object types like logical, integer, double (numeric), and character. But they can also be used to point to data structures like vectors, lists, arrays, matrices, factors, or other R objects.

In the REPL or within an R program, all the variables declared can be listed using the ls() or objects() commands. The latter works because all entities in R that are pointed to by variables are *objects*. In fact, almost everything in R is an object. The details of the object system in R are beyond the scope of this book but we will be using the class(), mode(), str(), and typeof() functions to look at the characteristics of R objects.

Any named variable/object can be removed from memory using the rm(*variable_name*) command. Garbage collection from memory can be hastened using the gc() function.

Variable names should be chosen carefully. A very good guide to R coding-style, including variable-naming convention, is available on genius R-developer Hadley Wickham's Advanced R Web Site: http://adv-r.had.co.nz/Style.html

3.2: Variable Scope

Variables can be *global* or *local*. To understand these terms, it is necessary to understand a bit about ***environments.*** An environment is considered to contain a collection of objects.

The top-level environment, that exists at the R prompt and also the top level of an R-program (script) is the ***global environment***. Other environments are created within individual functions and similar parts of an R program.

Global variables are variables that can be "seen", accessed, read and modified anywhere in the R program i.e. in all scopes. Global variables are usually defined in the global environment. However, they can also be created or modified inside a local environment (like a function) by using the super-assignment operator <<- instead of <-.

If the regular assignment operator <- is used inside a local environment to assign a value to a global variable, a local variable with the same name (as the global variable) is created instead inside the local environment. Here is an example R-prompt session to help you understand.

```
radium@aceraspiredelto:~$ R
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
  Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
> var1 <- 25
> var2 <- 50
> myfunc1 <- function() {
+ print("Inside function myfunc1");
+ print("Environment is: ");
+ print(environment());
+ print("var2 is:");
+ print(var2);
+ var2 <- 55;
+ print("Local (but not global) var2 is modified to:");
+ print(var2);
```

```
+ }
>
> myfunc2 <- function() {
+ print("Inside function myfunc2");
+ print("Environment is: ");
+ print(environment());
+ var2 <<- 80;
+ var3 <<- 678;
+ var4 <- 444;
+ print("Variables inside the myfunc2 environment:");
+ ls();
+ }
>
> print("In the global environment");
[1] "In the global environment"
> environment()
<environment: R_GlobalEnv>
> myfunc1()
[1] "Inside function myfunc1"
[1] "Environment is: "
<environment: 0x27c1ac8>
[1] "var2 is:"
[1] 50
[1] "Local (but not global) var2 is modified to:"
[1] 55
> print("In the global environment");
[1] "In the global environment"
> print("Global var2:")
[1] "Global var2:"
> print(var2)
[1] 50
>
> myfunc2()
[1] "Inside function myfunc2"
[1] "Environment is: "
<environment: 0x27ca5b8>
[1] "Variables inside the myfunc2 environment:"
[1] "var4"
>
> print("In the global environment");
[1] "In the global environment"
> print("Global var2 has been modified within myfunc2 to:");
[1] "Global var2 has been modified within myfunc2 to:"
> print(var2)
[1] 80
> print("Global var3 has been created within myfunc2:");
```

```
[1] "Global var3 has been created within myfunc2:"
> print(var3)
[1] 678
> print("Variables inside the global environment:");
[1] "Variables inside the global environment:"
> ls();
[1] "myfunc1" "myfunc2" "var1"  "var2"  "var3"
> print("Local var4 created within myfunc2 is NOT seen in the global
environment scope:");
[1] "Local var4 created within myfunc2 is NOT seen in the global
environment scope:"
> print(var4)
Error in print(var4) : object 'var4' not found
```

The session-listing shows the text copied from the command console on a Linux machine. It shows the commands as well as the results of running them.

The text to copy and paste into the R REPL prompt, if you want to run this on your computer's command shell, is below:

```
var1 <- 25
var2 <- 50
myfunc1 <- function() {
print("Inside function myfunc1");
print("Environment is: ");
print(environment());
print("var2 is:");
print(var2);
var2 <- 55;
print("Local (but not global) var2 is modified to:");
print(var2);
}
myfunc2 <- function() {
print("Inside function myfunc2");
print("Environment is: ");
print(environment());
var2 <<- 80;
var3 <<- 678;
var4 <- 444;
print("Variables inside the myfunc2 environment:");
ls();
}
print("In the global environment");
environment()
myfunc1()
print("In the global environment");
print("Global var2:")
```

```
print(var2)
myfunc2()
print("In the global environment");
print("Global var2 has been modified within myfunc2 to:");
print(var2)
print("Global var3 has been created within myfunc2:");
print(var3)
print("Variables inside the global environment:");
ls();
print("Local var4 created within myfunc2 is NOT seen in the global
environment scope:");
print(var4)
```

In the session-listing above, there are three different environments: the global environment, and the two local environments of the functions myfunc1 and myfunc2. The global variables are var1, var2, and var3.

The variable var3 is global because, even though it was created within the function myfunc2, the super-assignment operator <<- was used to define it. The same super-assignment operator was also used to change the value of var2 within myfunc2.

Inside the function myfunc1, the attempt to assign a new value to var2 using the regular assignment operator <- actually creates a new *local* variable var2 (with the same name as the global variable var2) inside the environment of the function myfunc1. The value of the global variable var2 remains unchanged in this case.

The environments form a kind of cascade. An inner environment (local environment) like that created by a function can see all variables from outer environments, including the global environment. However, the outer environment cannot see the variables of the inner environment. This is true even for environments created by functions *within* functions – where there will an inner function environment and an outer function environment.

One other important thing to note about R variable-scope is that there is *nothing* called **"block scope"** in R. In that sense, R is similar to JavaScript but different from languages like Java where variables defined inside a block (which could be delimited by curly braces) are local to that block. In R, only functions create a local environment. Simple curly braces do not create an environment. In the listing below, varConditional is a global variable that is accessible after the end of the curly braces.

```
var6 <- 34;
if (var6 == 34) {
varConditional <- "Good";
} else {
varConditional <- "Bad";
}
print(varConditional);
```

One good method to ensure that global variables do not pollute function-local

environments is to set up a ***self-executing anonymous function*** (or a function called main() which is called as the first statement in the R program). Another good practice to follow is to ensure that functions are passed ***all*** the variables they need to act on. Both these techniques are explained later in the chapter on functions.

Chapter 4: Data Structures

As genius R-developer Hadley Wickham explains in his Advanced R book's chapter on data structures (http://adv-r.had.co.nz/Data-structures.html), the common data structures in R can be classified based on two characteristics: *dimensionality* and *diversity*.

Dimensionality describes whether a data structure is one-dimensional, two-dimensional, or multi-dimensional. Diversity describes whether the contents of a data structure are all of the same type (homogeneous) or of different types (heterogeneous).

The homogeneous data structures are: **atomic vector, matrix,** and **array**.

The heterogeneous data structures are: **list** and **data frame.**

Of the above, the one-dimensional data structures are: **atomic vector** and **list.**

The two-dimensional data structures are: **matrix** and **data frame**.

The multi-dimensional data structure is the **array**. Actually, the **matrix** is just a 2-dimensional **array**.

4.1: Atomic Vector

This contains values that are all of the same type – like integer, character, double, etc...

The c() function is used to create vectors.

E.g.

```
  myCharVector <- c("Jim", "John", 'James', "Jeremy", 'Jonah') # of
class character
  myNumericVector <- c(2.3, 4.5, 6, 8, 9.0) # of class double
  myIntegerVector <- c(2, 3, 4, 5, 6, 8, 9, 0, 1) # of class integer
```

Factors are special vectors used to store categorical data. They are basically integer vectors but have the *class* **"factor"** that changes their behavior to being different from regular integer vectors.

4.2: Array

An array is an atomic vector with a *dimension* attribute – that decides whether the structure is 2-dimensional, 3-dimensional, 4-dimensional, etc.

e.g.

```
# the vector c(2, 4, 8) is the dimension specification
myArray <- array(1:16, c(2, 4, 8))
```

4.3: Matrix

A matrix is a 2-dimensional array.

4.4: List

A list is also unidimensional, but the elements of a list can be of different types and can even be other lists. Thus, lists can be nested data structures.

E.g.

```
myList <- list(1:6, "tiger", c(TRUE, FALSE, TRUE), c(2.3, 5.9),
 list("a", 4, 'nice', 'toy', 7.2))
```

4.5: Data Frame

A **data frame** is a ***list*** of ***equal-length*** vectors. It is a two-dimensional structure like a relational database table. It is the most commonly used data structure in

R-programming for data analysis. Reading in data from text files or other sources usually leads to the creation of a data frame object in memory.

E.g.

```
myDataFrame <- data.frame(x = 1:3, y = c("apple", "boy", "cat"))
```

4.6: Accessing members of a vector, list, or data frame

R has some confusing operators for accessing members/elements of a vector, list, or data frame. This is well explained in this R-bloggers article: https://www.r-bloggers.com/r-accessors-explained/

The three operators are:

1. Single square-brackets: *[]*

2. Double square-brackets: *[[]]*

3. The dollar-sign operator: *$*

The gist is that the single-square brackets return a *subset* of the original object with the same *type* (unless the returned object has a single result – in which case the type might get reduced to a simpler one).

The double-square brackets return a single item or element and the type of the returned object is usually simpler than the original container.

The $ operator is synonymous with the double-square brackets but simplifies syntax by allowing access-by-name of items like data frame column vectors.

The REPL example below clarifies how these access operators work and what they return.

```
# state.abb and state.name are built-in factor-vectors available in R
# Create a data frame with three vectors
> statesUSDataFrame <- data.frame(state.abb, state.name, c(1:50))
> statesUSDataFrame[[1]]
 [1] AL AK AZ AR CA CO CT DE FL GA HI ID IL IN IA KS KY LA
ME MD MA MI MN MS MO
[26] MT NE NV NH NJ NM NY NC ND OH OK OR PA RI SC SD
TN TX UT VT VA WA WV WI WY
50 Levels: AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS
KY LA MA MD ... WY
> statesUSDataFrame[1]
  state.abb
1  AL
2  AK
3  AZ
.
.
.
47  WA
48  WV
```

```
49  WI
50  WY
> class(statesUSDataFrame[1])
[1] "data.frame"
> class(statesUSDataFrame[[1]])
[1] "factor"
> statesUSDataFrame$state.abb
 [1] AL AK AZ AR CA CO CT DE FL GA HI ID IL IN IA KS KY LA
ME MD MA MI MN MS MO
[26] MT NE NV NH NJ NM NY NC ND OH OK OR PA RI SC SD
TN TX UT VT VA WA WV WI WY
50 Levels: AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS
KY LA MA MD ... WY
> class(statesUSDataFrame$state.abb)
[1] "factor"
```

Chapter 5: Functions

Functions are pieces of code that perform specific tasks: they help in promoting the re-usability and modularization of code. This is similar to macros in SAS® but significantly more advanced.

Functions are supported by all good programming languages.

R is special in that it is a *"functional"* programming language – functions are first-class objects.

Therefore, functions can be passed to other functions, returned from other functions, and manipulated within programs!

Functions can be built-in (available as part of the base R software), supplied through installed packages (a topic we will explore later), or user-defined:

1. Built-in functions e.g. *mean()* or *sqrt()*

2. Functions from packages that are installed from the Comprehensive R Archive Network (CRAN) or elsewhere.

3. User-defined functions.

A function performs a task and returns a value.

In R, the return value can be explicitly stated using the return() function at the end of the function OR implicitly be the value of the *last expression evaluated*.

Functions can take parameters a.k.a formals a.k.a arguments.

R allows for named parameters as well as default parameters for functions.

Function scope for variables is active in R (however, as discussed before, R does not have *block scope* for variables).

Function definition is how the function is written with the detail of its name, parameters, and return value. A function that is defined can be called from an R program or script. A function definition has to be available before an R program can call it.

So, it is a good idea to have the function definitions at the top of the file containing the R program.

Basic set up for functions:

```
function.name <- function(arguments or parameters) {
purpose of function
i.e. computations involving the arguments
}
```

```
# Function definition
myfunction1 <- function(x, y) {
  # This is what is returned by the function
  # Could have been explicitly stated as return(x + y)
  x + y;
}


  # Calling the function
sum <- myfunction1 (3, 4);
print(sum);
```

A function can return only a single object. However, the object can be a complex data structure like a list, data-frame, vector, array, etc.

It is a good idea to modularize your R-code into functions of 40-100 lines instead of putting it all in a giant R script of 1000 or more lines! As pugilist Mike Tyson once said (http://articles.sun-sentinel.com/2012-11-09/sports/sfl-mike-tyson-explains-one-of-his-most-famous-quotes-20121109_1_mike-tyson-undisputed-truth-famous-quotes), "Everybody has a plan until they get punched in the mouth." When the *plan* you had for your R program falls apart and it starts doing things that you did not expect, having modularized code in functions will bolster the attempt to fix things and get back on track.

Larry Wall, the creator of the Perl programming language (https://en.wikipedia.org/wiki/Perl) has a humorous yet insightful take on the three virtues of a great programmer: http://threevirtues.com/.

Remembering Larry's words when starting to write R programs will go a long way in helping you write beautiful and effective code that gets the job done.

5.1: Anonymous Self-Executing Functions

An anonymous self-executing function helps encapsulate R code and prevent the problems relating to global variables polluting function spaces. The idea is to put the main part of the script inside an *anonymous self-executing function* of this sort:

```
(function() {
  # All code that used to be in the top-level of the R script goes here
})()
```

This anonymous self-executing function is the first piece of code that runs when an R program is run. However, since it is also a function in its own right, variables created inside it are **not** global and thus **do not** pollute other function spaces. If you find it hard to grok the concept *"anonymous self-executing"*, an easier-to-understand solution is to create a function named main() containing the top-level code (a concept borrowed from languages like C and Java) and execute it as the first statement in your program:

```
main <- function() {
  # All code that used to be in the top-level of the R script goes here
}
main() # Execute the main() function
# Function that increments an integer by 2 and returns it
incrementByTwo <- function(mynumber) {
  mynumber = mynumber + 2

  print("Printing x within the function\n")
  print(x)

  x <- x + 100

  print("Printing x within the function after incrementing by 100\n")
  print(x)

  print(x)

  return(mynumber)
}
# Code encapsulation with a self-executing anonymous function
(function() {
  x <- 2

  print("Printing x outside the function\n")
  print(x)

  x <- x + 3
```

```
  print("Printing x outside the function after incrementing by 3\n")
  print(x)

  y <- incrementByTwo(x)

  print("Printing y\n")
  print(y)
})()
```

In the code block above, the print(x) statement within the function incrementByTwo() causes an error and a halt in execution since the variable x created within the top-level anonymous self-executing function is not seen within the local environment of the function incrementByTwo().

One thing to remember though, is that objects from R packages loaded in *any* function are actually available everywhere - this is because the package environment is inserted in the hierarchy of environments as a parent of the global environment.

5.2: Passing a function all the variables it needs

Another strategy to prevent global variables from bleeding into function environments is to ***pass*** to a function, as arguments, ***all*** the variables it needs for processing. Never make functions act on global variables that are not passed to it as parameters or arguments.

Chapter 6: Objects

Objects are everywhere in R.

Data that are used for analysis are stored as objects in memory.

Functions are objects.

Functions return objects containing data and properties that can be read, manipulated, and pretty-printed.

Even simple integers, strings, and doubles (numbers) are objects.

Use class(), typeof(), mode(), and str() on variables to know more about the objects they represent and the structure of the objects.

Unlike in data-analysis languages like SAS®, in R, the return values or outputs of statistical functions and tests like t-test [t.test()], anova [aov()], linear regression [lm()] are objects containing data and properties that can be read, manipulated, used in calculations, stored, and pretty-printed. This offers immense power in terms of being able to work with the data and the outputs.

Chapter 7: R packages

R packages are collections of functions that perform tasks related to a particular domain or subject area. The concept is similar to modules in SAS® but much broader.

R packages are similar to "libraries" in other programming languages.

More than 9000 packages (https://cran.r-project.org/web/packages/available_packages_by_name.html) are available on CRAN (https://cran.r-project.org/), the Comprehensive R Archive Network. These have been created by contributors in the R community.

The CRAN web-page for a package includes a lot of information, including a PDF that documents the package and its functions. Also, highly-useful are the vignettes (for packages that have them) that show you how to use the package.

Therefore, before thinking of writing your own function for anything, save time and effort by doing a search on CRAN to see if there is an existing R package for the purpose.

Sometimes, it is easier to just do a Google (or some other search-engine) query e.g. "Gradient boosted machines in R".

R packages have to be installed along with and accessible to your core R software before the functions in them are made available to work inside your R programs. This is as simple as using the install.packages() command in R within the REPL. This automatically downloads the package through the Internet from a CRAN mirror-repository and installs it on your computer in a location that makes it available to your R software.

E.g.

```
install.packages("dplyr");
# Multiple R packages can be installed in one go by providing
# their names as a character vector to the
# install.packages() function.
install.packages(c("data.table", "tidyr", "dtplyr", "rpart"));
```

Within an R program or the REPL, the R package has to be loaded using the library() or require() command before the functions in it can be used.

E.g.

```
library("dplyr");
library("readr");
```

One the package has been loaded using the library() command, the functions in the package are available to use. A function can be utilized through the base function name or the fully qualified name that includes the package name and :: as a prefix to the

function name.

E.g.

```
library("dplyr");
dplyr::filter();
filter();
```

In most cases, I would recommend using the fully-qualified name of the function (including the name of the package) in order to adjust for any masking of similarly-named functions in other packages.

Chapter 8: Interactive R through the REPL interface

The R interactive shell is of the type known as a REPL – Read-Eval-Print-Loop.

The REPL shell automatically prints the value of variables and expressions to the screen – this automatic printing does not happen when running R programs as scripts/programs.

The REPL is useful to step through small analyses, to install packages, and to access the built-in help.

Exit the REPL by closing the window or using the quit() or q() command.

When exiting, R will prompt the user about saving the session and the data as an .Rdata file. If you reply “yes or y”, the commands and the data objects used in the REPL session will be available in the next session. If you reply “no or n”, the commands and the data objects used in the REPL session will *not* be saved and will not be available in the next session.

Chapter 9: Inbuilt help in R

The R help mechanism is outlined here: http://www.statmethods.net/interface/help.html

1. If you know the exact name of the object type, function or other entity to access the help on, type a question mark followed by the entity name:

```
> ?mean
OR
> ?mean()
```

In this case, the window shows information on the function *mean()*.

2. If you do not know the exact terms or want to do a deeper, wildcard-type search across all of the documentation, type:

```
> ??mean
```

The window shows information on all the libraries with a function named mean.

3. Packages have to be loaded with the library() command before the REPL interface can be used with a single question mark to look at functions within them.

4. Using *help()*, *apropos()*, *example()*, *RsiteSearch()*, and other help functions is easy and provides a ton of information.

5. Once R is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:

```
help.start()  # general help
help(foo)  # help about function foo
?foo  # same thing
apropos("foo") # list all functions containing string foo
example(foo)  # show an example of function foo
# search for foo in help manuals and archived mailing lists
RSiteSearch("foo")
# get vignettes on using installed packages
vignette()  # show available vingettes
vignette("foo") # show specific vignette
```

6. Sample Datasets: R comes with a number of sample datasets that you can experiment with. Type *data( )* to see the available datasets. The results will depend on which packages you have loaded. Type *help(datasetname)* for details on a sample dataset.

Chapter 10: R programs and scripts

1. R *programs* or scripts are *text files* containing R *commands* that are *executed* by the R *interpreter*.

2. R scripts are created using a text-editor.

It is a good idea to use a text-editor that offers syntax highlighting for R language programming-code, and other features like function folding and automatic indentation.

3. One a program has been written and saved to a text-file, it needs to be interpreted and executed by R.

a. We can use the R GUI REPL (Read-Eval-Print-Loop shell) for this purpose.

b. The first step is to open up the R GUI REPL shell.

c. The next step is to navigate to the folder/directory in which the R script file is stored. The *setwd()* ***(for set working directory)*** command is used to navigate to the directory in which the R script file is stored. The argument to *setwd()* is a directory specification like *C:/Users/my_user_name/Desktop*. Note that *setwd()* uses the front-slash (*/*) as the folder separator, even on Windows, and not the back-slash (\) that Windows uses by default.

d. The *getwd()* command can be used in the REPL to get information on what the current working directory is.

e. In Windows, the shell() command can be used in the R REPL to execute Windows command shell commands. E.g.

f. shell("dir") # Lists the contents of the working directory

g. On Linux, the *system()* command does something similar.

h. Once it is confirmed that the R script file is in the same directory as the working directory for the R REPL, the source() command can be used to run the R script:

source("my_R_program_script.R");

i. The *source()* command can also be used within R scripts to read in and execute R code located in other R script files. This is often used to bring in and execute R function code that might exist in separate files.

4. Syntax for R programs

a. Commands are entered into the file, usually one per line. The semi-colon at the end of each statement-line is optional.

b. Multi-line commands are allowed, but have to include an operator (like a comma, plus-sign, etc.) at the end of each line that indicates that more of the command follows on the

next line.

c. Other R syntax rules have to be followed.

5. R scripts/programs stored in text files can be interpreted and executed by the R software at the command line.

a. The best program to invoke for this purpose, at the Windows or Linux Command shell prompt, is ***Rscript***.

b. To run an R program using ***Rscript*** at the command line:

◦ Navigate, in the command shell (using cd commands), to the directory in which the R script/program is stored. R scripts are usually created with the file extension **.R**.

◦ Execute the R script using this shell-prompt command that instructs Rscript to read, interpret, and execute the program:

```
Rscript nameOfFileContainingRprogram.R arg1 arg2
```

c. The **R** executable can also be used at the command line to run R programs. The syntax is a bit different. And on Windows, it can only be used in cmd.exe and not in powershell.exe.

d.

e. The command when using plain R is (all on one line of the command shell):

```
R –no-save –no-restore –args arg1 arg2 <
nameOfFileContainingRprogram.R > program_output.txt 2>
program_errors.txt
```

Chapter 11: Reading in Data

1. The three main steps in data analysis are:

i. Reading in the data (from text files, databases, binary files, web sources, etc.).

ii. Data munging/wrangling/manipulation and cleaning to convert the data to a format that facilitates analysis. The principles of Tidy Data (https://www.jstatsoft.org/article/view/v059i10) apply.

iii. Running the statistical analysis or machine learning algorithm/function on the manipulated data to get results and inferences.

2. Text data (delimited as well as fixed-width) can be easily read into R objects (like data frames) using the read.table() function. However, newer read-operations like fread() available through the data.table (https://cran.r-project.org/web/packages/data.table/index.html) package and read_delim() (and similar functions) from the readr (https://cran.r-project.org/web/packages/readr/index.html) package are much more efficient.

3. The CRAN pages for the data.table and readr packages have good reference material. We will be using these two packages in real examples of data analysis.

4. R can read data from relational databases by using the **RODBC**, **RJDBC, DBI** and similar packages.

5. Text formats like JSON and XML can be parsed using packages like **jsonlite** and **XML**.

6. Reading binary data is also quite easy but is rarely needed for data analysis tasks. I will not be covering binary file reads and writes in this book.

The first code-listing we will run and analyze is a program to read in the CMS Medicare data tab-separated-value text-file and save it as an R serialized-object to disk.

Code Listing 11.1

(Initial data-read using data.table)

The command to run the program on the command line (all on one line):

```
Rscript 11.1_Medicare_Provider_Util_Payment_initial_dataread.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014.txt
../data/Medicare_Provider_Util_Payment_PUF_CY2014__FULL.rds
# Copyright (C) 2017 Sivakumaran Raman
library("data.table")
library("dplyr")
library("dtplyr")
```

```
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------
# Function that uses the data.table package to read in the data file
readFileIntoDataFrame <- function(CMSPhysicianDataFile) {
  myDataFrame <- data.table::fread(CMSPhysicianDataFile, header =
TRUE,
  colClasses = c(npi = "character", nppes_provider_zip = "character",
  hcpcs_code = "character", line_srvc_cnt = "numeric"),
  data.table = TRUE, verbose = TRUE)
  return(myDataFrame)
}
# Function to manipulate a data frame using dplyr
manipulateDFUsingDplyr <- function(physicianDataDF) {
  # Character vector of names of columns to convert to factors
  varsToConvertToFactors <- c("nppes_credentials",
"nppes_provider_gender",
  "nppes_entity_code", "nppes_provider_state",
"nppes_provider_country",
  "provider_type", "medicare_participation_indicator",
"place_of_service")

  # Use the chained syntax of dplyr to convert certain columns to
factors
  # and to filter out the first row of data with the CPT Header from the
AMA
  physicianDataDF <- physicianDataDF %>%
  dplyr::mutate_at(varsToConvertToFactors, as.factor) %>%
  dplyr::slice(2:nrow(physicianMedicareDataTable))

  return(physicianDataDF)
}
#--------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if data file to read and data file to write to are
not
# provided on the command line as arguments 1 and 2 to the R program
if (length(myArgs) != 2) {
  stop("Error: please provide the name of the file to read and the file
  to write to as the command line arguments")
```

```
}
# Read in the data by calling the function
physicianMedicareDataTable <- readFileIntoDataFrame(myArgs[[1]])
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
# Call the function to manipulate the data frame using the dplyr
package
physicianMedicareDataTable <-
manipulateDFUsingDplyr(physicianMedicareDataTable)
RDSFileToWriteTo <- myArgs[[2]]
# Save the R data-frame+data-table object to a file
saveRDS(physicianMedicareDataTable, file = RDSFileToWriteTo,
compress = TRUE)
# Save the R data-frame+data-table object to a CSV file
data.table::fwrite(physicianMedicareDataTable,
  file = "../data/Medicare_Provider_Util_Payment_PUF_CY2014.csv",
  append = FALSE,
  quote = "auto", col.names = TRUE, row.names = FALSE, na = "",
  nThread = getDTthreads())
sink("physicianMedicareDataTable_info.txt")
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTable)
print("Summary statistics: ")
print(providerMedicareUtilSummaryObj)
sink()
```

The program is run using Rscript and supplied two arguments at the command line: the name of the file to read and the name of the binary R dataset file to save the data to.

The program first loads all the R packages which will be used.

```
library("data.table")
library("dplyr")
library("dtplyr")
```

The program starts off with the statement:

```
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
 traceback(2)
```

```
 stop("Error: stack trace printed above")
})
```

This instructs R to print the stack trace and quit when an error occurs at the time of any function-execution. This helps in debugging.

Next, the program reads in the first argument by calling a function and passing it the first element of the myArgs vector:

```
# Read in the data by calling the function
physicianMedicareDataTable <- readFileIntoDataFrame(myArgs[[1]])
```

The function works like this:

```
# Function that uses the data.table package to read in the data file
readFileIntoDataFrame <- function(CMSPhysicianDataFile) {
  myDataFrame <- data.table::fread(CMSPhysicianDataFile, header =
TRUE,
  colClasses = c(npi = "character", nppes_provider_zip = "character",
  hcpcs_code = "character", line_srvc_cnt = "numeric"),
  data.table = TRUE, verbose = TRUE)
  return(myDataFrame)
}
```

The *fread()* function from the data.table package is used to read the tab-delimited data file with the CMS Medicare Physician data. Using the *colClasses* option, some columns, which look like numbers in the file, are coerced into being read as text/character data. Similarly, some columns that have non-numeric data are coerced into being read as numbers. The *data.table* option set to TRUE ensures that the object created and returned is a data table (and not just a data frame). Also, the program is asked to be verbose in reporting output and errors.

Then, the program prints out information about the class, mode and structure of the data-object created by the data-read:

```
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
```

Next, the function to manipulate the data-object created is called and the object is passed to it as a parameter:

```
# Call the function to manipulate the data frame using the dplyr
package
physicianMedicareDataTable <-
  manipulateDFUsingDplyr(physicianMedicareDataTable)
```

The data-manipulation function uses the dplyr (https://cran.r-

project.org/web/packages/dplyr/index.html) package:

```
# Function to manipulate a data frame using dplyr
manipulateDFUsingDplyr <- function(physicianDataDF) {
  # Character vector of names of columns to convert to factors
  varsToConvertToFactors <- c("nppes_credentials",
"nppes_provider_gender",
  "nppes_entity_code", "nppes_provider_state",
"nppes_provider_country",
  "provider_type", "medicare_participation_indicator",
"place_of_service")

  # Use the chained syntax of dplyr to convert certain columns to
factors
  # and to filter out the first row of data with the CPT Header from the
AMA
  physicianDataDF <- physicianDataDF %>%
  dplyr::mutate_at(varsToConvertToFactors, as.factor) %>%
  dplyr::slice(2:nrow(physicianMedicareDataTable))

  return(physicianDataDF)
}
```

Some variables/columns in the data frame are converted to factors. Also, the first row of data (after the header row), which contains the CPT® copyright line, is discarded using the dplyr slice() function. The modified data object is returned back from the function. The magic of dplyr lies in the chained-method/chained-function syntax implemented through the piping operator %>%. This allows a sequence of functions to be applied to the same data-object (usually a data frame) where the transformed output of one function becomes the input to the next function. The object returned at the end of the chain of functions is the transformed/manipulated object with all of the functions applied to it. We will see more extensive use of dplyr later in the book.

The modified data object is then saved to disk as a binary file (inside the **data** folder) with compression applied:

```
# Save the R data-frame+data-table object to a file
saveRDS(physicianMedicareDataTable, file = RDSFileToWriteTo,
compress = TRUE)
```

The data.table *fwrite()* function is then used to write out the same data object as a CSV file. This works much faster than the traditionally used *write.csv()* function from the utils package as it utilizes multiple threads on the multi-processor computers that are commonly used nowadays. The options also request a header row of column/variable names to be written out as the first row and missing or NA (Not Available) values to be represented by the blank string :

```
# Save the R data-frame+data-table object to a CSV file
```

```
data.table::fwrite(physicianMedicareDataTable,
  file = "../data/Medicare_Provider_Util_Payment_PUF_CY2014.csv",
  append = FALSE,
  quote = "auto", col.names = TRUE, row.names = FALSE, na = "",
  nThread = getDTthreads())
```

Finally, some information about the class, mode, and structure of the data object along with summary statistics are written out to a text file. The sink() function redirects standard-output of the program to whichever file is supplied as its argument. Calling sink() again without a parameter resets the standard output channel to the default:

```
sink("physicianMedicareDataTable_info.txt")
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTable)
print("Summary statistics: ")
print(providerMedicareUtilSummaryObj)
sink()
```

The same program can also be written as one long script (top-to-bottom) without any functions. But that is not the right way to write programs: using functions makes the code modular, re-usable, and easier to debug. However, I am presenting, to bring out the contrasts, the without-functions version of the code listing 11.1 as code listing 11.2.

Code Listing 11.2

(Initial data-read using data.table: no functions version)

To run this (all on one line):

```
Rscript
11.2_Medicare_Provider_Util_Payment_initial_dataread_no_functions.
R ../data/Medicare_Provider_Util_Payment_PUF_CY2014.txt ../data/
Medicare_Provider_Util_Payment_PUF_CY2014__FULL.rds
# Copyright (C) 2017 Sivakumaran Raman
library("data.table")
library("dplyr")
library("dtplyr")
########## ALL FUNCTIONS DEFINED
########## ------------------------------------------------------------------
#-------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
# quit.
options(error = function() {
```

```
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if data file to read and data file to write
# to are not provided on the command line as arguments 1 and 2 to the
R
# program
if (length(myArgs) != 2) {
  stop("Error: please provide the name of the file to read and the file to
  write to as the command line arguments")
}
# Read in the data
physicianMedicareDataTable <- data.table::fread(myArgs[[1]], header
= TRUE,
  colClasses = c(npi = "character", nppes_provider_zip = "character",
  hcpcs_code = "character", line_srvc_cnt = "numeric"), data.table =
TRUE,
  verbose = TRUE)
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
varsToConvertToFactors <- c("nppes_credentials",
"nppes_provider_gender",
  "nppes_entity_code", "nppes_provider_state",
"nppes_provider_country",
  "provider_type", "medicare_participation_indicator",
"place_of_service")
# Use the chained syntax of dplyr to convert certain columns to factors
# and to filter out the first row of data with the CPT Header from the
# AMA
physicianMedicareDataTable <- physicianMedicareDataTable %>%
  dplyr::mutate_at(varsToConvertToFactors, as.factor) %>%
  dplyr::slice(2:nrow(physicianMedicareDataTable))
RDSFileToWriteTo <- myArgs[[2]]
# Save the R data-frame+data-table object to a file
saveRDS(physicianMedicareDataTable, file = RDSFileToWriteTo,
compress = TRUE)
# Save the R data-frame+data-table object to a CSV file
data.table::fwrite(physicianMedicareDataTable,
  file = '../data/Medicare_Provider_Util_Payment_PUF_CY2014.csv',
  append=FALSE,
  quote="auto",
  col.names=TRUE,
```

```
  row.names=FALSE,
  na='',
  nThread = getDTthreads() );
sink("physicianMedicareDataTable_info.txt")
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTable)
print("Summary statistics: ")
print(providerMedicareUtilSummaryObj)
sink()
```

For reading text data into R, another very useful package is readr. The *read_delim()* function in readr can be used to read in delimited-data. The readr package functions are slightly slower at reading files than the fread function in data.table. However, readr offers functions like read_fwf (to read fixed-width-files) while data.table does not contain functions to read fixed-width-files. Code listing 11.3 uses the *read_delim()* function from readr to read in the data. Otherwise, it is pretty much the same as code listing 11.1.

Code Listing 11.3

(Initial data-read using readr)

To run this (all on one line):

```
Rscript
11.3_Medicare_Provider_Util_Payment_initial_dataread_using_readr.
R ../data/Medicare_Provider_Util_Payment_PUF_CY2014.txt ../data/
Medicare_Provider_Util_Payment_PUF_CY2014__FULL_from_readr.
rds
# Copyright (C) 2017 Sivakumaran Raman
library("readr")
library("dplyr")
library("dtplyr")
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------
# Function that uses the readr package to read in the data file
readFileIntoDataFrame <- function(CMSPhysicianDataFile) {
  # The first row contains the header (column names).
  myDataFrame <- readr::read_delim(file = CMSPhysicianDataFile,
  delim = '\t',
  col_names = TRUE);
  return(myDataFrame)
}
# Function to manipulate a data frame using dplyr
```

```
manipulateDFUsingDplyr <- function(physicianDataDF) {
  # Character vector of names of columns to convert to factors
  varsToConvertToFactors <- c("nppes_credentials",
"nppes_provider_gender",
  "nppes_entity_code", "nppes_provider_state",
"nppes_provider_country",
  "provider_type", "medicare_participation_indicator",
"place_of_service")

  # Use the chained syntax of dplyr to convert certain columns to
factors
  # and to filter out the first row of data with the CPT Header from the
  # AMA
  physicianDataDF <- physicianDataDF %>%
  dplyr::mutate_at(varsToConvertToFactors, as.factor) %>%
  dplyr::slice(2:nrow(physicianMedicareDataTable))

  return(physicianDataDF)
}
#--------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
# quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if data file to read and data file to write
# to are not provided on the command line as arguments 1 and 2 to the
R
# program
if (length(myArgs) != 2) {
  stop("Error: please provide the name of the file to read and the file to
  write to as the command line arguments")
}
# Read in the data by calling the function
physicianMedicareDataTable <- readFileIntoDataFrame(myArgs[[1]])
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
# Call the function to manipulate the data frame using the dplyr
# package
physicianMedicareDataTable <-
  manipulateDFUsingDplyr(physicianMedicareDataTable)
```

```
RDSFileToWriteTo <- myArgs[[2]]
# Save the R data-frame+data-table object to a file
saveRDS(physicianMedicareDataTable, file = RDSFileToWriteTo,
compress = TRUE)
# Commented out
# Save the R data-frame+data-table object to a CSV file
# write.csv(physicianMedicareDataTable, file =
# '../data/Medicare_Provider_Util_Payment_PUF_CY2014.csv',
row.names=FALSE,
# na='');
sink("physicianMedicareDataTable_info.txt")
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable)
mode(physicianMedicareDataTable)
str(physicianMedicareDataTable)
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTable)
print("Summary statistics: ")
print(providerMedicareUtilSummaryObj)
sink()
```

R can also read in data from various other sources like relational databases, binary files, URLs, XML files, and various others. Relational databases are an important and commonly used source of data – so we will cover how to get data from a relational database management system (RDBMS). To create a relational database source that R can read from, we will use the SQLite RDBMS: https://www.sqlite.org/

Download and install SQLite for your operating system. Then, open up the **Medicare_Provider_Util_Payment_PUF_CY2014.txt** tab-delimited file in a text-editor, delete the first two rows (the header row with variable names and the second row with the CPT copyright notice from the American Medical Association), and save it as a new file with the name **Medicare_Provider_Util_Payment_PUF_CY2014_NO_HEADER_ROW.txt**. Open up a command shell (bash in Linux, cmd.exe in Windows) and use change-directory (cd) commands to navigate to the same folder as the one containing the tab-delimited text file with the two top rows removed.

Next, type this at the shell command prompt to create a new SQLite database file named **Medicare_Provider_Util_Payment_PUF_CY2014.db**:

```
sqlite3 Medicare_Provider_Util_Payment_PUF_CY2014.db
```

You will now be at the sqlite3 prompt. Run these commands at the sqlite3 prompt to create a database table named physicianMedicareDataTable and load it with data from the tab-delimited CMS Physician Utilization file with the top two rows removed:

```
create table physicianMedicareDataTable (npi text,
```

```
nppes_provider_last_org_name text, nppes_provider_first_name text,
nppes_provider_mi text, nppes_credentials text,
nppes_provider_gender text, nppes_entity_code text,
nppes_provider_street1 text, nppes_provider_street2 text,
nppes_provider_city text, nppes_provider_zip text,
nppes_provider_state text, nppes_provider_country text,
provider_type text, medicare_participation_indicator text,
place_of_service text, hcpcs_code text,
hcpcs_description text, hcpcs_drug_indicator text,
line_srvc_cnt integer, bene_unique_cnt integer,
bene_day_srvc_cnt integer, average_Medicare_allowed_amt real,
average_submitted_chrg_amt real, average_Medicare_payment_amt
real,
average_Medicare_standard_amt real);
.separator "\t"
.import
Medicare_Provider_Util_Payment_PUF_CY2014_NO_HEADER_RO
W.txt physicianMedicareDataTable
.exit
```

The whole session in the shell console looks like this:

```
radium@aceraspiredelto:~/Desktop$ sqlite3
Medicare_Provider_Util_Payment_PUF_CY2014.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table physicianMedicareDataTable (npi text,
   ...> nppes_provider_last_org_name text,
   ...> nppes_provider_first_name text,
   ...> nppes_provider_mi text,
   ...> nppes_credentials text,
   ...> nppes_provider_gender text,
   ...> nppes_entity_code text,
   ...> nppes_provider_street1 text,
   ...> nppes_provider_street2 text,
   ...> nppes_provider_city text,
   ...> nppes_provider_zip text,
   ...> nppes_provider_state text,
   ...> nppes_provider_country text,
   ...> provider_type text,
   ...> medicare_participation_indicator text,
   ...> place_of_service text,
   ...> hcpcs_code text,
   ...> hcpcs_description text,
   ...> hcpcs_drug_indicator text,
   ...> line_srvc_cnt integer,
```

```
  ...> bene_unique_cnt integer,
  ...> bene_day_srvc_cnt integer,
  ...> average_Medicare_allowed_amt real,
  ...> average_submitted_chrg_amt real,
  ...> average_Medicare_payment_amt real,
  ...> average_Medicare_standard_amt real);
sqlite>
sqlite> .separator "\t"
sqlite> .import
Medicare_Provider_Util_Payment_PUF_CY2014_NO_HEADER_RO
W.txt physicianMedicareDataTable
sqlite> .exit
```

Move the SQLite database file **Medicare_Provider_Util_Payment_PUF_CY2014.db** to the **data** folder before running the code.

Code Listing 11.4

(Initial data-read from a relational database)

To run this (all on one line):

```
Rscript
11.4_Medicare_Provider_Util_Payment_initial_dataread_from_SQLite
_RDB.R ../data/Medicare_Provider_Util_Payment_PUF_CY2014.db
physicianMedicareDataTable
../data/Medicare_Provider_Util_Payment_PUF_CY2014__FULL_from
_RDBMS.rds
# Copyright (C) 2017 Sivakumaran Raman
library("DBI") # Load the DBI package in order to use the RSQLite
package
library("dplyr")
library("dtplyr")
########## ALL FUNCTIONS DEFINED
########## ------------------------------------------------------------------
# Function that uses the RSQLite package to read in a database table
# from a SQLite relational database
readDBTableIntoDataFrame <-
  function(CMSPhysicianDataBaseNameSQLite, DBTableName) {
  # Connect to the SQLite relational database
  mydb <- dbConnect(RSQLite::SQLite(),
CMSPhysicianDataBaseNameSQLite);
  # Run the SQL query to read in data from the table into a dataframe
  myDataFrame <- dbGetQuery(mydb, paste('SELECT * FROM ',
DBTableName));
  # Disconnect from the relational database
  dbDisconnect(mydb);
  # Return the dataframe with the data
```

```
  return(myDataFrame)
  }
# Function to manipulate a data frame using dplyr
manipulateDFUsingDplyr <- function(physicianDataDF) {
  # Character vector of names of columns to convert to factors
  varsToConvertToFactors <- c("nppes_credentials",
"nppes_provider_gender",
  "nppes_entity_code", "nppes_provider_state",
"nppes_provider_country",
  "provider_type", "medicare_participation_indicator",
"place_of_service")

  # Use the chained syntax of dplyr to convert certain columns to
factors
  physicianDataDF <- physicianDataDF %>%
  dplyr::mutate_at(varsToConvertToFactors, as.factor);

  return(physicianDataDF)
}
#--------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
# quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if SQLite database name, database table to
read,
# and data file to write to are not provided on the command line as
arguments
# 1, 2, and 3 to the R program
if (length(myArgs) != 3) {
  stop("Error: please provide the name of the SQLite DB file, the DB
table
  to read, and the file to write to as the command line arguments")
}
# Read in the data from the SQLite relational database table by calling
# the function
physicianMedicareDataTableDF <-
  readDBTableIntoDataFrame(myArgs[[1]], myArgs[[2]])
# Print out information about the class, mode and type of object
class(physicianMedicareDataTableDF)
mode(physicianMedicareDataTableDF)
str(physicianMedicareDataTableDF)
```

```
# Call the function to manipulate the data frame using the dplyr
# package
physicianMedicareDataTableDF <-
  manipulateDFUsingDplyr(physicianMedicareDataTableDF)
RDSFileToWriteTo <- myArgs[[3]]
# Save the R data-frame+data-table object to a file
saveRDS(physicianMedicareDataTableDF, file = RDSFileToWriteTo,
compress = TRUE)
# Save the R data-frame+data-table object to a CSV file
# write.csv(physicianMedicareDataTableDF, file =
# '../data/Medicare_Provider_Util_Payment_PUF_CY2014.csv',
row.names=FALSE,
# na='');
sink("physicianMedicareDataTable_info.txt")
# Print out information about the class, mode and type of object
class(physicianMedicareDataTableDF)
mode(physicianMedicareDataTableDF)
str(physicianMedicareDataTableDF)
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTableDF)
print("Summary statistics: ")
print(providerMedicareUtilSummaryObj)
sink()
```

Code listing 11.4 is almost the same as listing 11.1 except for the part where data is read from the SQLite relational database table. R uses the common **DBI** package interface for most relational database systems (including SQLite) while the SQLite-specific driver details are taken care of by the **RSQLite** package. The tasks of connecting to the database, running a query to pull data from a table, and returning the data as a data frame are performed by calling this function:

```
# Function that uses the RSQLite package to read in a database table
# from a SQLite relational database
readDBTableIntoDataFrame <-
  function(CMSPhysicianDataBaseNameSQLite, DBTableName) {
  # Connect to the SQLite relational database
  mydb <- dbConnect(RSQLite::SQLite(),
CMSPhysicianDataBaseNameSQLite);
  # Run the SQL query to read in data from the table into a dataframe
  myDataFrame <- dbGetQuery(mydb, paste('SELECT * FROM ',
DBTableName));
  # Disconnect from the relational database
  dbDisconnect(mydb);
  # Return the dataframe with the data
  return(myDataFrame)
  }
```

Inside the readDBTableIntoDataFrame() function, the dbConnect() function from the DBI package uses the SQLite database driver to connect to the SQLite database. The dbGetQuery() function is used to submit a query to the relational database engine. The results of the query are returned into a data frame. There is an explicit disconnect from the database and the function returns the data frame object. The rest of code listing 11.4 does the same things as code listing 11.1.

Chapter 12: Data Wrangling/Munging/Manipulation

1. The older R ways of manipulating data in structures like data-frames are slow, inefficient, and no longer recommended.

2. Instead, data manipulation is best done through the use of the data.table (https://cran.r-project.org/web/packages/data.table/index.html) , tidyr (https://cran.r-project.org/web/packages/tidyr/index.html) , dplyr (https://cran.r-project.org/web/packages/dplyr/index.html) and sqldf (https://cran.r-project.org/web/packages/sqldf/index.html) packages.

3. Of these, I am going to use **tidyr** and **dplyr** in this book because:

a. The syntax is easy to understand and lends itself to chained methods and pipelines using the **%>%** operator from the magrittr (https://cran.r-project.org/web/packages/magrittr/index.html) package.

b. dplyr-like functions are supported by Hadoop and Spark-related big-data R packages like sparklyr (https://cran.r-project.org/web/packages/sparklyr/index.html). Therefore, learning dplyr makes the transition to data-wrangling on big-data platforms like Hadoop and Spark much easier.

c. The tidyr and dplyr packages provide a variety of functions for cleaning, processing, & manipulating data

- tidyr
    - ¬ gather()
    - ¬ spread()
    - ¬ separate()
    - ¬ unite()
- dplyr
    - ¬ select()
    - ¬ filter()
    - ¬ group_by()
    - ¬ summarize()
    - ¬ arrange()
    - ¬ join()
    - ¬ mutate()

A really good introduction to the use of tidyr and dplyr is available in this web article by Brad Boehmke: https://rstudio-pubs-static.s3.amazonaws.com/58498_dd3b603ba4fb4b469bb1c57b5a951c39.html

As part of understanding how data wrangling works using dplyr, we will summarize the

Medicare data set by ***provider*** (a term used for *any* provider of health care services). The code listing for this is below.

Code Listing 12.1

(Data wrangling using dplyr)

To run this (all on one line):

```
Rscript
12.1_Medicare_Provider_Util_Payment_manipulate_dataset.R ../data/
Medicare_Provider_Util_Payment_PUF_CY2014__FULL.rds ../data/
Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZED.rds
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.csv
# Copyright (C) 2017 Sivakumaran Raman
library("data.table")
library("dplyr")
library("dtplyr")
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------
########## Function to manipulate the data frame using the dplyr
package
manipulateDFUsingDplyr <- function(myDataFrame) {
  # SUMMARIZE THE DATA BY PROVIDER

  myDataFrame <- myDataFrame %>%
  # Create a new variable to represent total allowed amount in the year
for
  # the physician for the HCPCS code
  dplyr::mutate(total_allowed_amt_for_hcpcs_code =
  line_srvc_cnt * average_Medicare_allowed_amt) %>%
  # Group by the variables of interest
  dplyr::group_by(npi, nppes_provider_last_org_name,
  nppes_provider_first_name,
  nppes_provider_mi, nppes_credentials, nppes_provider_gender,
  nppes_entity_code, nppes_provider_street1, nppes_provider_street2,
  nppes_provider_city, nppes_provider_zip, nppes_provider_state,
  nppes_provider_country, provider_type,
  medicare_participation_indicator) %>%
  # Aggregate/summarize
  dplyr::summarize(
  total_medicare_prof_srv_revenue =
sum(total_allowed_amt_for_hcpcs_code,
  na.rm = TRUE),
  total_line_srvc_cnt = sum(line_srvc_cnt, na.rm = TRUE),
  total_bene_unique_cnt = sum(bene_unique_cnt, na.rm = TRUE),
  total_bene_day_srvc_cnt = sum(bene_day_srvc_cnt, na.rm = TRUE))
```

```
%>%
  # Add a random value between 1 and 3 as an extra column
  dplyr::mutate(random_val_between_one_and_three = sample(1:3, 1))

  return(myDataFrame)
}
#------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read, the RDS file
to write
# summarized output to, and the CSV file to write summarized output
to are not
# provided on the command line
if (length(myArgs) != 3) {
  stop("Error: please provide the name of the RDS file to read, the RDS
file
  to write summarized data to, and the CSV file to write summarized
data to as
  the two command line arguments")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Call the function to manipulate the data frame using the dplyr and
tidyr
# packages
summarizedPhysicianMedicareDataTable <-
  manipulateDFUsingDplyr(physicianMedicareDataTable)
# Print out information about the class, mode and type of object
class(summarizedPhysicianMedicareDataTable)
mode(summarizedPhysicianMedicareDataTable)
str(summarizedPhysicianMedicareDataTable)
typeof(summarizedPhysicianMedicareDataTable)
RDSFileNameForSummarizedDataFrame <- myArgs[[2]]
# Save the R data-frame object to a file
saveRDS(summarizedPhysicianMedicareDataTable,
  file = RDSFileNameForSummarizedDataFrame,
  compress = TRUE)
CSVFileNameForSummarizedData <- myArgs[[3]]
# Save the R data-frame+data-table object to a CSV file
```

```
data.table::fwrite(summarizedPhysicianMedicareDataTable,
  file = CSVFileNameForSummarizedData, append = FALSE,
  quote = "auto", col.names = TRUE, row.names = FALSE, na = "",
  nThread = getDTthreads())
```

The initial parts of this R program are similar to what we have seen in previous code listings: loading of R packages, setting debugging options, and reading the command line parameters that the R program is invoked with. But, one big difference is that the data is read directly from the binary R object that was saved to disk after the initial text-file data-read in chapter 12. The data frame that this object on disk is read into is then passed as a parameter to the function *manipulateDFUsingDplyr()* that does all the work (using dplyr functions) of manipulating the data. Here is the *manipulateDFUsingDplyr()* function:

```
########## Function to manipulate the data frame using the dplyr
package
manipulateDFUsingDplyr <- function(myDataFrame) {
 # SUMMARIZE THE DATA BY PROVIDER

 myDataFrame <- myDataFrame %>%
  # Create a new variable to represent total allowed amount in the year
for
  # the physician for the HCPCS code
  dplyr::mutate(total_allowed_amt_for_hcpcs_code =
  line_srvc_cnt * average_Medicare_allowed_amt) %>%
  # Group by the variables that are NOT in the summarize by clause
  dplyr::group_by(npi, nppes_provider_last_org_name,
  nppes_provider_first_name,
  nppes_provider_mi, nppes_credentials, nppes_provider_gender,
  nppes_entity_code, nppes_provider_street1, nppes_provider_street2,
  nppes_provider_city, nppes_provider_zip, nppes_provider_state,
  nppes_provider_country, provider_type,
  medicare_participation_indicator) %>%
  # Aggregate/summarize
  dplyr::summarize(
  total_medicare_prof_srv_revenue =
sum(total_allowed_amt_for_hcpcs_code,
  na.rm = TRUE),
  total_line_srvc_cnt = sum(line_srvc_cnt, na.rm = TRUE),
  total_bene_unique_cnt = sum(bene_unique_cnt, na.rm = TRUE),
  total_bene_day_srvc_cnt = sum(bene_day_srvc_cnt, na.rm = TRUE))
%>%
  # Add a random value between 1 and 3 as an extra column
  dplyr::mutate(random_val_between_one_and_three = sample(1:3, 1))

 return(myDataFrame)
}
```

The mutate() function from the dplyr package is used to create new variables (columns) to add to the dataset represented by the data frame. In the function above, mutate() is used to create a new variable that represents the ***total allowed dollar amount*** for the HCPCS procedure code for the particular provider: this is done by multiplying the number of times the code was used by the provider (***line_srvc_cnt***) with the ***average allowed Medicare dollar amount*** for the code.

The data is then summarized by individual provider using the *group_by()* and *summarize()* functions. For those familiar with SQL queries that are run against relational databases, this will seem very similar to executing **group by** queries with **aggregate functions.** In SQL, from the list of columns being returned by a query, all columns to which an aggregate function is not being applied should appear in the **group by** clause. Something similar is done in the dplyr *group_by()* and *summarize()* functions.

Once the data has been aggregated/summarized, one more *mutate()* statement is issued to create a new random integer variable between 1 and 3 as a new column. This will be used in the statistical and machine learning programs run later in the book.

The chained-function syntax is used all along and the result of each function being applied to the data-object is the input to the next function in the chain or pipe.

The aggregated data is returned by the function. This aggregated/summarized data is then saved to disk as an RDS binary object and also as a CSV file.

Exercise 12.1: For the Reader

Modify code listing 12.1 to summarize the data by provider_type instead of summarizing by individual providers.

Chapter 13: Data Quality Checks in R

A critical part of any data analysis effort is ensuring that the data is clean, well-structured, and of good quality. There are several ways to achieve this in R. One way is to write custom data-quality rules in R that are then run against the data. Another option is to use an existing R package that helps run data-quality checks. We are going to use the latter method. For this purpose, we will use the datacheck (https://cran.r-project.org/web/packages/datacheck/index.html) CRAN package that simplifies the task of creating data-quality checks.

The datacheck package expects the data-quality rules to be presented as a list of assertions. These string-based assertions are read and executed by the R program against the data.

One problem with the datacheck package is that it creates a copy of the whole dataset in memory as part of the *data dictionary profile* object returned as a result of the data quality rules being run. Therefore, running the data quality rules on the whole dataset requires the computer to have 2-3 times the memory of the dataset. Since my machine did not have so much memory, I used (for demonstration purposes) a smaller dataset using the first 100,000 rows of the data-file. However, if you have to run data quality rules against the whole dataset and do not have enough memory on your machine, one method to use is to split the dataset into multiple smaller ones to run the rules against. The data quality results from all the datasets can then be combined at the end.

Also, in the code listing coming up, I use the ztable and stargazer packages to pretty-print the data quality rules-result object. These two packages will be used in subsequent chapters as well in order to pretty-print the output from descriptive, statistical, and machine-learning procedures.

To create a smaller dataset containing only the first 100,000 data-rows from the original dataset on Linux, use the head command in the shell (all on one line):

```
head -n 100002 Medicare_Provider_Util_Payment_PUF_CY2014.txt >
Medicare_Provider_Util_Payment_PUF_CY2014_HEAD100000.txt
```

The reason I use -n 100002 instead of -n 100000 is because I want to adjust for the header row and the CPT® copyright row.

On Windows, the command equivalent to head is the Get-Content command. But it is available only in powershell.exe and not in cmd.exe. The command is (all on one line):

```
Get-Content "Medicare_Provider_Util_Payment_PUF_CY2014.txt"
-TotalCount 100002 | Set-Content
"Medicare_Provider_Util_Payment_PUF_CY2014_HEAD100000.txt"
```

After creating the Medicare_Provider_Util_Payment_PUF_CY2014_HEAD100000.txt file, the R program in code listing 11.1 should be run on it (by changing the arguments)

in order to create an R dataset (with just the first 100,000 rows of data) named Medicare_Provider_Util_Payment_PUF_CY2014_HEAD100000.rds in the data folder. Then, code-listing 13.1 below can be run to execute the data quality rule checks.

Code Listing 13.1

(Data quality checks using datacheck)

To run this (all on one line):

```
Rscript 13.1_Medicare_Provider_Util_Payment_data_checks.R ../data/
Medicare_Provider_Util_Payment_PUF_CY2014_HEAD100000.rds
# Copyright (C) 2017 Sivakumaran Raman
library("data.table");
library("dplyr");
library("dtplyr");
library("datacheck");
library("ztable");
library("stargazer");
# library('broom'); library('tidyr');
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Function to run data-quality-check rules
runDataCheckRules <- function(myDataFrame,
dataqualityRulesCharVector) {
  # Convert the rules vector to a rules data frame
  datacheck_rules_dataframe <-
datacheck::as_rules(dataqualityRulesCharVector)
  # Run the data checks
  rules_results_db <-
  datacheck::datadict_profile(physicianMedicareDataTable,
  datacheck_rules_dataframe)
  return(rules_results_db);
}
# Function to pretty print data-quality rules checks results using ztable
prettyPrintHTMLRulesResultsUsingZtable <-
  function(rulesOutputChecksDF, fileForHTMLOutput) {
  # Rules checks output in pretty HTML
  # Set the options for the ztable printing of pretty tabular output
  options(ztable.type="html", ztable.colnames.bold=TRUE);


  # Use ztable to get pretty-table output about rules checks data-frame
  ztable_object <- ztable::ztable(rulesOutputChecksDF,
  caption="Data-check rules results", caption.bold=TRUE,
  tablewidth=0.1, zebra=1,
  zebra.type=2, zebra.color=5, position="left", show.footer=FALSE,
  hline.after=c(-1:nrow(rulesOutputChecksDF)),
```

```
  wraptable=TRUE, wraptablewidth=6)

  # Add vertical lines
  ztable::vlines(ztable_object, type="all");

  sink(fileForHTMLOutput);
  print(ztable_object);
  sink();

  return(ztable_object);
  }
# Function to pretty print data-quality rules checks results using
stargazer
prettyPrintHTMLRulesResultsUsingStargazer <-
  function(rulesOutputChecksDF, fileForHTMLOutput) {
  stargazer_object <- stargazer::stargazer(rulesOutputChecksDF,
type="html",
  out=fileForHTMLOutput, summary=FALSE,
  title="Data-check rules results")

  return(stargazer_object);
  }
#--------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error= function () {
  traceback(2);
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided
# on the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
command
  line argument");
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]]);
# Print out information about the class, mode and type of object
class(physicianMedicareDataTable);
mode(physicianMedicareDataTable);
str(physicianMedicareDataTable);
```

```
# Create vector of datacheck rules
datacheck_rules_vector = c(
  'sapply(bene_unique_cnt, is.integer) # is right datatype',
  'is_within_range(average_Medicare_standard_amt, 0, 200) # is within
range',
  'sapply(average_Medicare_standard_amt, is.numeric) # is of correct
datatype',
  'sapply(hcpcs_code, is.character) # is of correct datatype'
  );
# Call the function to run the data-quality rules check
rules_results_db <- runDataCheckRules(physicianMedicareDataTable,
  datacheck_rules_vector);
# Is the type of the object a data dictionary profile?
print("Is the type of the datacheck rules-results object
  data-dictionary profile?\n");
datacheck::is_datadict_profile(rules_results_db) == TRUE;
print("Plot rule coverage for db object:\n");
datacheck::rule_coverage(rules_results_db);
print("Plot score summary for db object:\n");
datacheck::score_sum(rules_results_db);
sink("rules_output.txt");
# Print out information about the class, mode and type of data-quality
# rules object
class(rules_results_db);
mode(rules_results_db);
str(rules_results_db);
print("Dump rules object:\n")
print(rules_results_db);
sink();
# File to write out the HTML output from ztable for the datacheck data-
quality
# rules run
outputZTableHTMLFile <- "R_output_ztable_rules_checks.html"
# Call the function to pretty-print data check rules output using the
ztable
# package
myZtableObject <-
  prettyPrintHTMLRulesResultsUsingZtable(rules_results_db$checks,
  outputZTableHTMLFile);
# File to write out the HTML output from stargazer for the datacheck
# data-quality rules run
outputStargazerHTMLFile <- "R_output_stargazer_rules_checks.html"
# Use stargazer to get pretty-table output about rules checks data-frame
myStargazerObject <-
  prettyPrintHTMLRulesResultsUsingStargazer(rules_results_db$check
s,
```

```
  outputStargazerHTMLFile);
```

Parts of the above program are similar to others we have seen earlier in the book. The data (100,000 rows) are read in from the .rds file. Then, the data quality rules checks are set up as a character vector.

```
  # Create vector of datacheck rules
  datacheck_rules_vector = c(
  'sapply(bene_unique_cnt, is.integer) # is right datatype',
  'is_within_range(average_Medicare_standard_amt, 0, 200) # is within
range',
  'sapply(average_Medicare_standard_amt, is.numeric) # is of correct
datatype',
  'sapply(hcpcs_code, is.character) # is of correct datatype'
  );
```

The rules use the *sapply()* function (which applies a particular function to each element of a list) and other helper functions in order to define the rules to be run against each row of a particular, named variable-column. The rules we are applying are:

1. The *beneficiary unique count* or bene_unique_cnt variable value should always be an integer.

2. The average Medicare standardized amount should be within a certain numerical range.

3. The average Medicare standardized amount should be a numeric value.

4. The HCPCS code should be a character/string value.

The rules checks are then executed by calling the runDataCheckRules function.

```
# Call the function to run the data-quality rules check
rules_results_db <- runDataCheckRules(physicianMedicareDataTable,
  datacheck_rules_vector);
```

The *runDataCheckRules()* function converts the character vector of rules passed to it to a *rules data frame* and then runs the rules against the dataset. The results of the rules checks are collected in a ***data dictionary profile*** object.

```
  # Run the data checks
  rules_results_db <-
  datacheck::datadict_profile(physicianMedicareDataTable,
  datacheck_rules_dataframe)
```

The *prettyPrintHTMLRulesResultsUsingZtable()* and *prettyPrintHTMLRulesResultsUsingStargazer()* functions then use the ztable (https://cran.r-project.org/web/packages/ztable/) and stargazer (https://cran.r-project.org/web/packages/stargazer/index.html) packages respectively to pretty-print the rules-results object as HTML tables. We will be using these two packages again later to pretty-print

the output from machine-learning and statistical procedures.

The tabular output lists, in columns, the variable name, the data-type, the rule applied to the variable, the comment that accompanies the rule, the execution status, the number of failed data-values, and an actual listing of data-values that fail the rule. Here are the first two rows of the output from ztable for the rules object.

	Variable	Type	Rule	Comment	Execution	Error.sum	Error.list
1	bene_unique_cnt	integer	sapply(bene_unique_cnt, is.integer)	is right datatype	ok	0	none
2	average_Medicare_standard_amt	numeric	is_within_range(average_Medicare_standard_amt, 0, 200)	is within range	ok	4916	120,121,133,165,195,196, ...

Using the datacheck package, data-quality can be validated or checked and subsequent steps can be taken to clean the data and fix data-quality issues.

Exercise 13.1: For the Reader

Modify the program in code listing 13.1 add an additional data quality check: The NPI number (which is supposed to have numeric characters but is read in as a character string) should consist of only digits and contain no letters or other characters.

Chapter 14: Descriptive Statistics and Visualization

Descriptive statistics are the first step in any data analysis. They help the analyst get familiarized with the characteristics of the data, the patterns, the distributions, and the variables of interest. Visualization, when used as part of initial exploration, is a great tool to further enhance the process of describing the characteristics of the data.

The simplest function available in R for creating summary statistics on a data set is the appropriately-named *summary()* function. Running this on a data set in memory (like a data frame) will generate information on each variable in the data set. The information available in the summary includes the number of observations, missing values, class, mode, and levels (for factors). For numeric variables, the minimum, maximum, mean, median, and quartiles are included in the summary.

The book Practical Data Science with R (by Nina Zumel and John Mount, 2014, Manning Publications: https://www.manning.com/books/practical-data-science-with-r) outlines various visualizations that can be used for different types of data. For a single variable, histograms and bar charts are the commonly-used visualizations. For two variables, line plots, scatter plots, stacked bar charts, and other visualizations can be used.

Code Listing 14.1

(Descriptive statistics)

To run this (all on one line):

```
Rscript 14.1_descriptive_statistics.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014__FULL.rds
# Copyright (C) 2017 Sivakumaran Raman
library("data.table")
library("dplyr")
library("dtplyr")
library("tidyr")
library("ggplot2")
library("stargazer")
library("R2HTML")
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------
# Function to pretty print summary statistics object using R2HTML
prettyPrintHTMLSummaryResultsUsingR2HTML <-
  function(physicianMedicareDataSummaryObj,
HTMLFilenameMinusExtension) {
  R2HTML::HTMLStart(outdir=".",
  file=HTMLFilenameMinusExtension,
  extension="html",
  echo=FALSE,
```

```
  HTMLframe=FALSE)

  R2HTML::HTML.title("HTML Output for Summary Statistics",
HR=1)

  R2HTML::HTML.title("Summary Statistics for the 2014 CMS
Physician data",
  HR=3)

  # Write the summary statistics object out as HTML
  R2HTML::HTML(physicianMedicareDataSummaryObj)

  R2HTML::HTMLhr()

  R2HTML::HTMLStop()
  }
#------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
# provided as the command line argument
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTable)
# Print plain-text summary-statistics output to a file
sink("summary_statistics_all_data.txt")
class(providerMedicareUtilSummaryObj)
mode(providerMedicareUtilSummaryObj)
str(providerMedicareUtilSummaryObj)
print(providerMedicareUtilSummaryObj)
sink()
# Write out pretty-printed HTML format summary statistics using
stargazer
stargazer_summary_stats <-
```

```
stargazer::stargazer(physicianMedicareDataTable,
  type = "html", out = "summary_statistics_using_stargazer.html",
  summary = TRUE, title = "Summary Statistics")
# Call the function to pretty-print the summary object of the data as
HTML
# using the R2HTML package
prettyPrintHTMLSummaryResultsUsingR2HTML(providerMedicareU
tilSummaryObj,
  "summary_statistics_using_R2HTML")
```

Code listing 14.1 aims at producing summary statistics for the read-in ***full*** data set (not the data set which was aggregated by provider).

The traceback options and the reading-in of the binary RDS file from disk are similar to what we have seen in previous code listings. The program then runs the *summary()* function on the data set and reads the results into a summary object.

```
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataTable)
```

The summary is then printed to a plain text file along with information on the summary object itself.

```
# Print plain-text summary-statistics output to a file
sink("summary_statistics_all_data.txt")
class(providerMedicareUtilSummaryObj)
mode(providerMedicareUtilSummaryObj)
str(providerMedicareUtilSummaryObj)
print(providerMedicareUtilSummaryObj)
sink()
```

However, since this plain-text summary-statistics file is not pleasing to the eye, we use the stargazer and R2HTML (https://cran.r-project.org/web/packages/R2HTML/index.html) packages to pretty-print the summary as HTML tables. The stargazer package is simpler to use for this as it has the built-in ability to process and pretty-print a summary-statistics object. However, this stargazer printing only processes the numeric variables and the other variables in the data set do not appear in the table output. To address this, we also use the R2HTML package to print out the summary-statistics object as a nice-looking HTML table.

As the next step in the initial data exploration, we want to look at the data in visual form. For this purpose, we use R's excellent charting packages (ggplot2 and plotly) in code listing 14.2.

Code Listing 14.2

(Descriptive visualizations on the complete data set)

To run this (all on one line):

```
Rscript 14.2_descriptive_visualizations.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014__FULL.rds
# Copyright (C) 2017 Sivakumaran Raman
library("data.table")
library("dplyr")
library("dtplyr")
library("tidyr")
library("ggplot2")
library("scales")
library("plotly")
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------------
#-------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
provided as the
# command line argument
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Ceate a density plot for Average Medicare Allowed amount
ggplot_density_plot <- ggplot(physicianMedicareDataTable) +
  labs(list(title = "Density Plot of Average Medicare Allowed Amount",
  x = "Average Medicare Allowed Amount")) +
  geom_density(aes(x = average_Medicare_allowed_amt)) +
  scale_x_continuous(labels = dollar)
ggplot2::ggsave(filename =
  "Average_Medicare_Allowed_Amount_Density_Plot.jpg",
  plot = ggplot_density_plot,
  dpi = 1200, device = "jpeg")
ggplot2::ggsave(filename =
  "Average_Medicare_Allowed_Amount_Density_Plot.svg",
  plot = ggplot_density_plot,
  device = "svg")
# Ceate a density plot for Average Medicare Allowed amount on a
```

```
# logarithmic scale
ggplot_density_plot <- ggplot(physicianMedicareDataTable) +
  labs(list(title = "Density Plot on Log Scale of Average Medicare
  Allowed Amount",
  x = "Average Medicare Allowed Amount: Log 10 scale")) +
  geom_density(aes(x = average_Medicare_allowed_amt)) +
  # scale_x_continuous(trans='log10', breaks=c(10,
100,1000,10000,100000),
  # expand=c(0.07, 0), labels=dollar) +
  scale_x_log10(breaks = c(10, 100, 1000, 10000, 1e+05),
  expand = c(0.07, 0), labels = dollar) +
  annotation_logticks(sides = "bt")
ggplot2::ggsave(filename =
  "Average_Log_Scale_Medicare_Allowed_Amount_Density_Plot.jpg"
,
  plot = ggplot_density_plot, dpi = 1200, device = "jpeg")
ggplot2::ggsave(filename =
  "Average_Log_Scale_Medicare_Allowed_Amount_Density_Plot.svg
",
  plot = ggplot_density_plot, device = "svg")
# Bar chart for count of providers by state
ggplot_bar_chart <- ggplot(physicianMedicareDataTable) +
  geom_bar(aes(x = nppes_provider_state),
  fill = "green") +
  labs(list(title = "Counts of providers by state", x = "US state",
  y = "Count")) +
  coord_flip() +
  theme(axis.text.y = element_text(size = rel(0.5)))
ggplot2::ggsave(filename =
  "Bar_Chart_of_counts_by_State.jpg",
  plot = ggplot_bar_chart,
  dpi = 1200, device = "jpeg")
ggplot2::ggsave(filename =
  "Bar_Chart_of_counts_by_State.svg",
  plot = ggplot_bar_chart,
  device = "svg")
# Make an interactive plotly chart out of the ggplot2 chart
ggplotly_bar_chart <- ggplotly(ggplot_bar_chart) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
on the modeBar
  plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
  list("sendDataToCloud"))
# Save as an HTML file
htmlwidgets::saveWidget(ggplotly_bar_chart,
  file =
```

```
"ggpplotly_barchart_Medicare_provider_counts_by_US_state_2014.ht
ml",
  selfcontained = TRUE)
```

Code listing 14.2 is quite complex, and uses a lot of packages and concepts that have not been seen before in this book. The dplyr, data.table, and dtplyr (which implements the data table backend for dplyr), and tidyr packages are still being used. However, some new packages we encounter in the library() function calls are ggplot2 (https://cran.r-project.org/web/packages/ggplot2/index.html), scales (https://cran.r-project.org/web/packages/scales/index.html), and plotly (https://cran.r-project.org/web/packages/plotly/index.html).

The ggplot2 library written by genius R-developer Hadley Wickham is now considered the de-facto standard for static charting and plotting in the R world. It has largely replaced the standard plotting functions available in R. The scales library is used to access specialized X and Y axis scales for the ggplot2 charts.

The plotly library, which we will see more of later, is special. It is actually a JavaScript interactive-visualization library which offers R, Python, and MATLAB® wrappers. The library was made open-source and free in 2015 by its creator-company and is great for interactive charting. The charts output from plotly are usually HTML files that are viewed in a browser with full interactivity (hover, click, drag, zoom, etc.).

Code listing 14.2 starts off in the usual way by reading in arguments from the command line. It then reads in the binary RDS file with the full-detail data from disk into a data frame. Next however, we encounter the plotting function *ggplot()* from ggplot2 that creates the density plot for the ***average Medicare allowed amount in dollars*** (because this is a variable of interest that we will analyze further later):

```
# Ceate a density plot for Average Medicare Allowed amount
ggplot_density_plot <- ggplot(physicianMedicareDataTable) +
  labs(list(title = "Density Plot of Average Medicare Allowed Amount",
  x = "Average Medicare Allowed Amount")) +
  geom_density(aes(x = average_Medicare_allowed_amt)) +
  scale_x_continuous(labels = dollar)
```

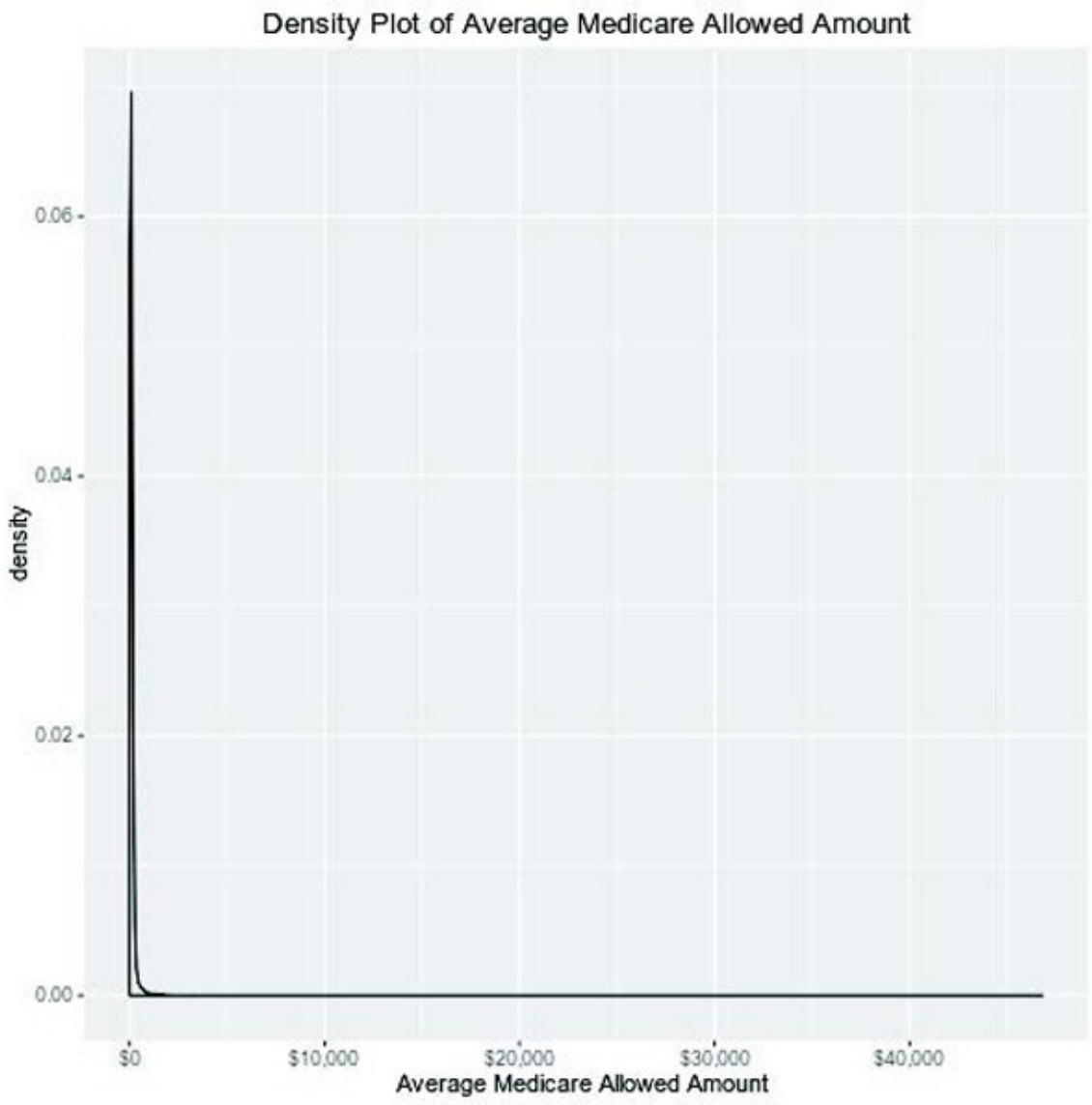
The ggplot2 package uses the same piping mechanism you have seen used by dplyr but instead of using the magrittr-derived %>% piping symbol, ggplot2 uses the + sign. Still, the effect is the same as the chained-function syntax used in dplyr. The chained method syntax allows the output of one function to be fed as input to the next one. In the code segment above, the *ggplot()* function is asked to act on the physicianMedicareDataTable data frame. The *labs()* function applies the label-texts for the title and x axis. The *geom_density()* function is the one that creates the smooth density estimate for display in the plot. The ggplot2 package uses the term aesthetics (http://docs.ggplot2.org/current/vignettes/ggplot2-specs.html) to describe the visual characteristics of the plot. In this case, we do not specify any special visual characteristics like color or line type but simply specify that the x-axis has to display the average_Medicare_allowed_amt variable. The last function in the chain is the

scale_x_continuous() function to apply the dollar-sign notation to the x-axis (the dollar-sign scale actually comes from the scales package).

The density plot is held as an object in memory with the name ggplot_density_plot and this object is then written to disk using the *ggsave()* function as both a JPEG image and an SVG file :

```
ggplot2::ggsave(filename =
  "Average_Medicare_Allowed_Amount_Density_Plot.jpg",
  plot = ggplot_density_plot,
  dpi = 1200, device = "jpeg")
ggplot2::ggsave(filename =
  "Average_Medicare_Allowed_Amount_Density_Plot.svg",
  plot = ggplot_density_plot,
  device = "svg")
```

Figure 14.1 (Density Plot)



We also create a second density plot on a logarithmic scale:

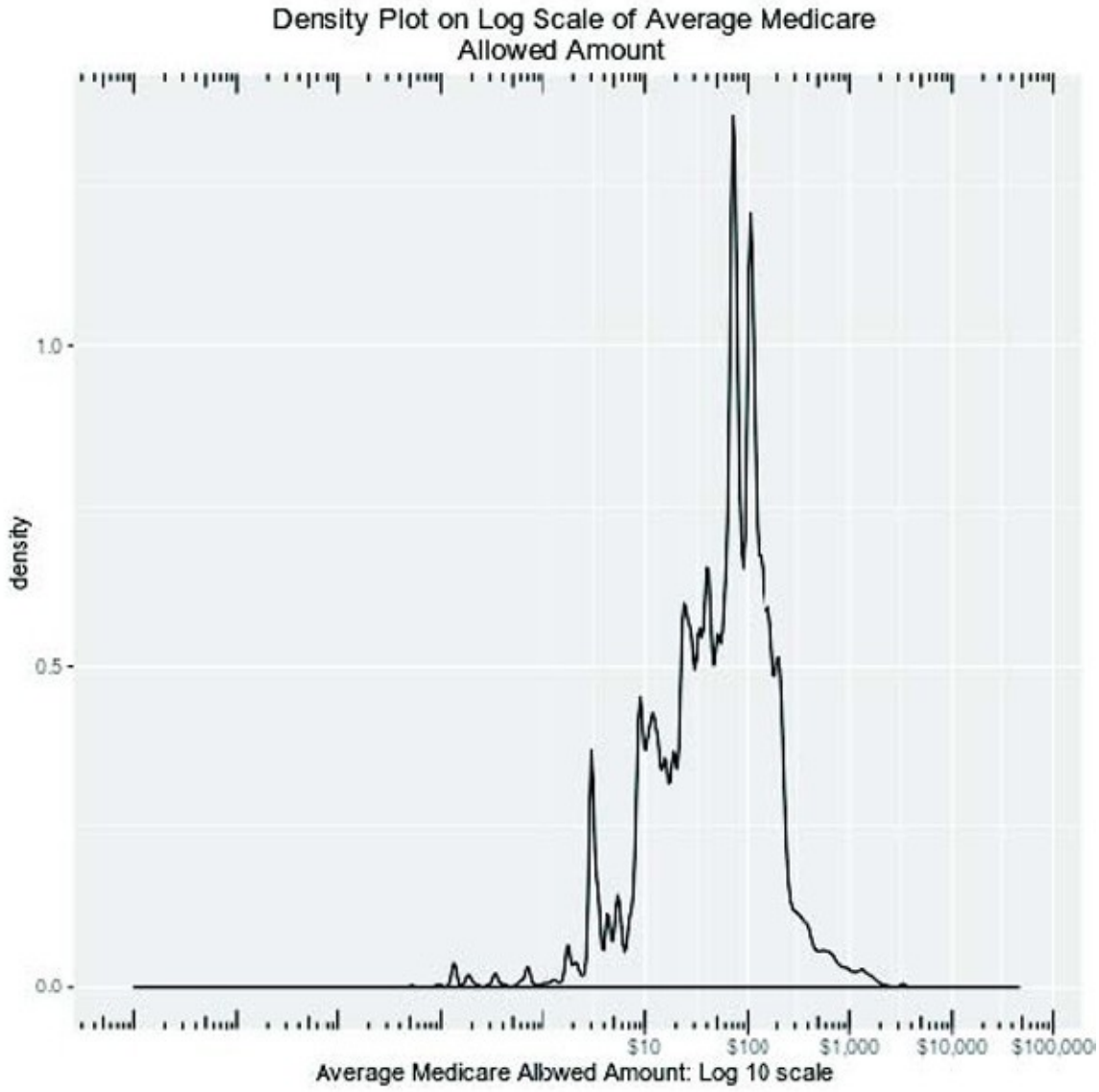
```
# Ceate a density plot for Average Medicare Allowed amount on a
# logarithmic scale
ggplot_density_plot <- ggplot(physicianMedicareDataTable) +
  labs(list(title = "Density Plot on Log Scale of Average Medicare
  Allowed Amount",
  x = "Average Medicare Allowed Amount: Log 10 scale")) +
  geom_density(aes(x = average_Medicare_allowed_amt)) +
  # scale_x_continuous(trans='log10', breaks=c(10,
100,1000,10000,100000),
  # expand=c(0.07, 0), labels=dollar) +
  scale_x_log10(breaks = c(10, 100, 1000, 10000, 1e+05),
  expand = c(0.07, 0), labels = dollar) +
  annotation_logticks(sides = "bt")
```

The two big differences between this and the earlier density plot are the use of the *scale_x_log10()* and *annotation_logticks()* functions.

We use the *scale_x_log10()* function to create the log10 x-axis scale. We use the breaks argument to the function to specify the numeric values to be shown on the axis. The argument expand is (from the CRAN ggplot2 manual): *a numeric vector of length two giving multiplicative and additive expansion constants. These constants ensure that the data is placed some distance away from the axes. The defaults are c(0.05, 0) for continuous variables, and c(0, 0.6) for discrete variables.*

The labels argument is (as before) for the dollar-scale. The *annotation_logticks()* function is used to add the log tick-marks on the x-axis. The argument sides set to the value "bt" requests log tick-marks on both the *bottom* and the *top* of the plot.

Figure 14.2 (Density Plot – logarithmic scale)

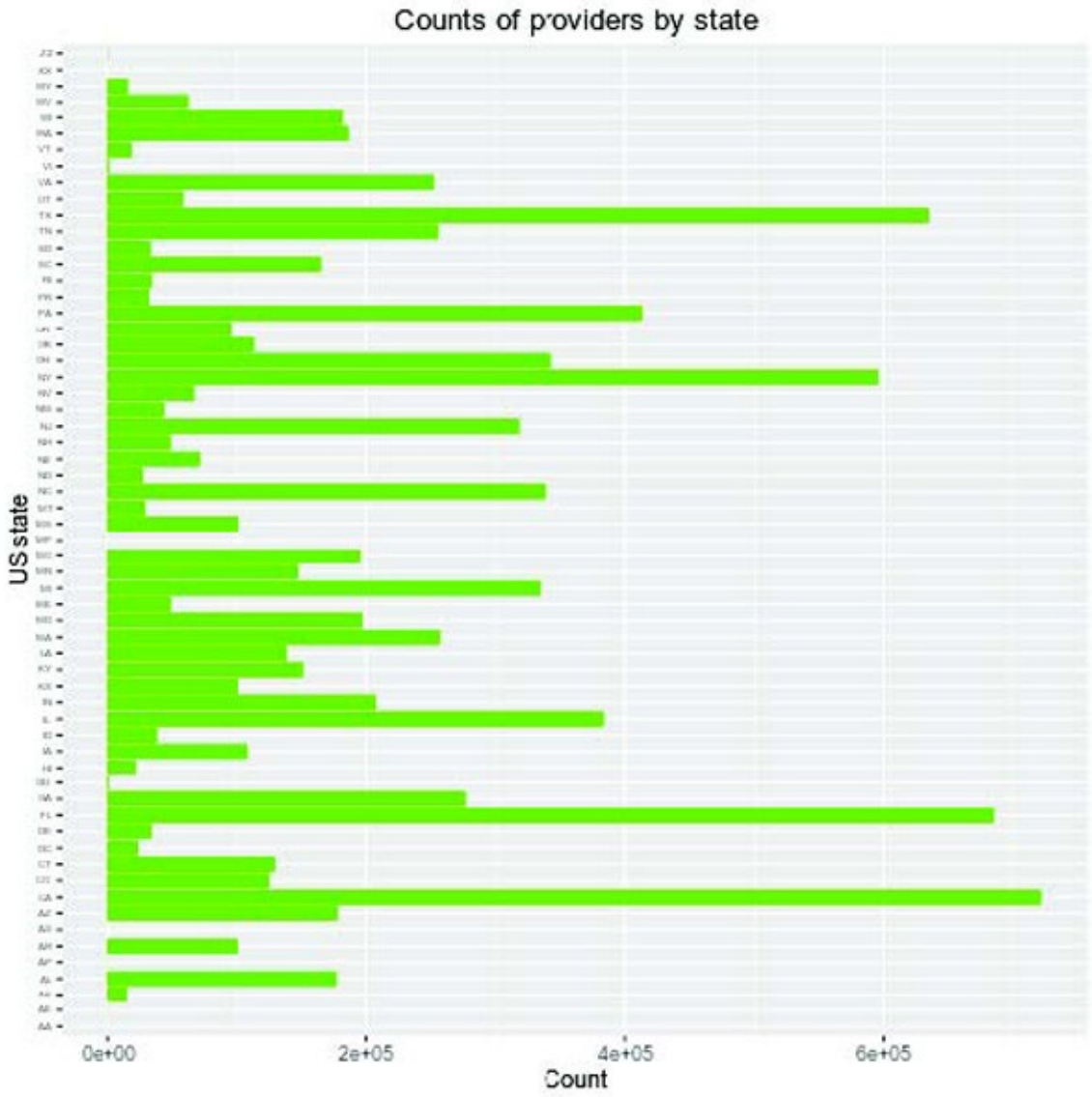


Next, we want to plot a horizontal bar chart of provider-counts by American state.

```
# Bar chart for count of providers by state
ggplot_bar_chart <- ggplot(physicianMedicareDataTable) +
  geom_bar(aes(x = nppes_provider_state),
  fill = "green") +
  labs(list(title = "Counts of providers by state", x = "US state",
  y = "Count")) +
  coord_flip() +
  theme(axis.text.y = element_text(size = rel(0.5)))
```

The code for this uses the same chained method syntax as most ggplot2 plots do. The geom_bar() function creates the bars on the x-axis in the color green. The *labs()* function applies the labels for the title, x-axis and y-axis. The *coord_flip()* function is critical because this flips the Cartesian coordinates so that horizontal becomes vertical, and vertical, horizontal: that is how we get a horizontal bar-chart instead of a vertical one. The *theme()* function sets the tick-labels for the Y-axis (the US state abbreviations) to a relative-size of half (0.5).

Figure 14.3 (Bar-chart of provider-count by state)



The last thing we do in code listing 14.2 is to use the plotly package to create an interactive, HTML-format bar-chart from the ggplot2 bar-chart we created above. We will look at plotly in more detail in subsequent chapters. But for now, we use the convenient plotly function named *ggplotly()* that automatically converts a ggplot2 plot to an interactive plotly plot.

```
# Make an interactive plotly chart out of the ggplot2 chart
ggplotly_bar_chart <- ggplotly(ggplot_bar_chart) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
  # on the modeBar
  plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
  list("sendDataToCloud"))
```

The Plotly (https://en.wikipedia.org/wiki/Plotly) company open-sourced the plotly

JavaScript library and its R API in November-2015. The default plots produced by plotly include the plotly logo and other buttons that reference the Plotly organization. However, we can configure the plotly plot using the displaylogo and modeBarButtonsToRemove options to turn off the default plotly logo and the "Send Data to Cloud" button.

Since a plotly object is also an htmlwidgets (https://cran.r-project.org/web/packages/htmlwidgets/index.html) object, the htmlwidgets function saveWidget() can be used to save the plotly object as an HTML file. The selfcontained option is special. If it is set to FALSE, the HTML file is created separately and the accessory objects (like the plotly JavaScript library .js file and other objects) are placed in a directory with the same base-name as the HTML file plus a "_files" suffix. However, setting the selfcontained option to TRUE causes the JavaScript libraries and other accessory files to be *inlined* within the HTML file: so all that is created is a single HTML file. The caveat is that the selfcontained=TRUE option requires the Pandoc (http://pandoc.org/) software tool to be installed on the computer on which the R code is run.

```
# Save as an HTML file
htmlwidgets::saveWidget(ggplotly_bar_chart,
file =
"ggpplotly_barchart_Medicare_provider_counts_by_US_state_2014.ht
ml",
selfcontained = TRUE)
```

An *interactive* HTML-format plot is best explored in a browser like Mozilla Firefox, Google Chrome or Microsoft Edge. An *interactive* visualization creates a better user-experience than a static plot because the viewer can hover over or click on data points to get more info, zoom in, pan left or right, and carry out many other actions to glean more detail from the chart. One problem with HTML format output (including charts) that it cannot be put into a report-format like PDF and retain its interactivity. However, there are several ways around this. One is to create the HTML output files and zip them up in a zip archive. Another is to use a web-based notebook like Apache Zeppelin (http://zeppelin.apache.org/) to publish results.

This is a static-picture example of how interactivity works in an HTML plot. Hovering over the bar for the state brings up a pop-up that displays the provider-count for the state:

Figure 14.4 (Interactive Bar-chart of provider-count by state)

The last code listing in this chapter, 14.3, aims to create bar-charts faceted by country that show provider gender.

Code Listing 14.3

(Descriptive visualizations on the aggregated-by-provider dataset)

To run this (all on one line):

```
Rscript 14.3_descriptive_visualizations_summarized_data.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds
# Copyright (C) 2017 Sivakumaran Raman
library("data.table")
library("dplyr")
library("dtplyr")
library("tidyr")
library("ggplot2")
library("scales")
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------------
#-------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
provided as the
# command line argument
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataSummaryTable <- readRDS(myArgs[[1]])
# Get summary statistics
providerMedicareUtilSummaryObj <-
summary(physicianMedicareDataSummaryTable)
# Print plain-text summary-statistics output to a file
sink("summary_statistics_aggregated_data.txt")
print(providerMedicareUtilSummaryObj)
sink()
# Create bar charts for provider-gender faceted by country
ggplot_faceted_bar_chart <-
```

```
ggplot(physicianMedicareDataSummaryTable) +
  geom_bar(aes(x = nppes_provider_gender),
  position = "dodge", fill = "darkgray") +
  labs(list(title = "Provider Gender faceted by country",
  y = "count", x = "Provider Gender")) +
  facet_wrap(~nppes_provider_country, scales = "free_y") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
ggplot2::ggsave(filename =
  "Bar_Chart_Medicare_provider_gender_faceted_by_Country.pdf",
  plot = ggplot_faceted_bar_chart, dpi = 1200, device = "pdf")
ggplot2::ggsave(filename =
  "Bar_Chart_Medicare_provider_gender_faceted_by_Country.svg",
  plot = ggplot_faceted_bar_chart, device = "svg")
```

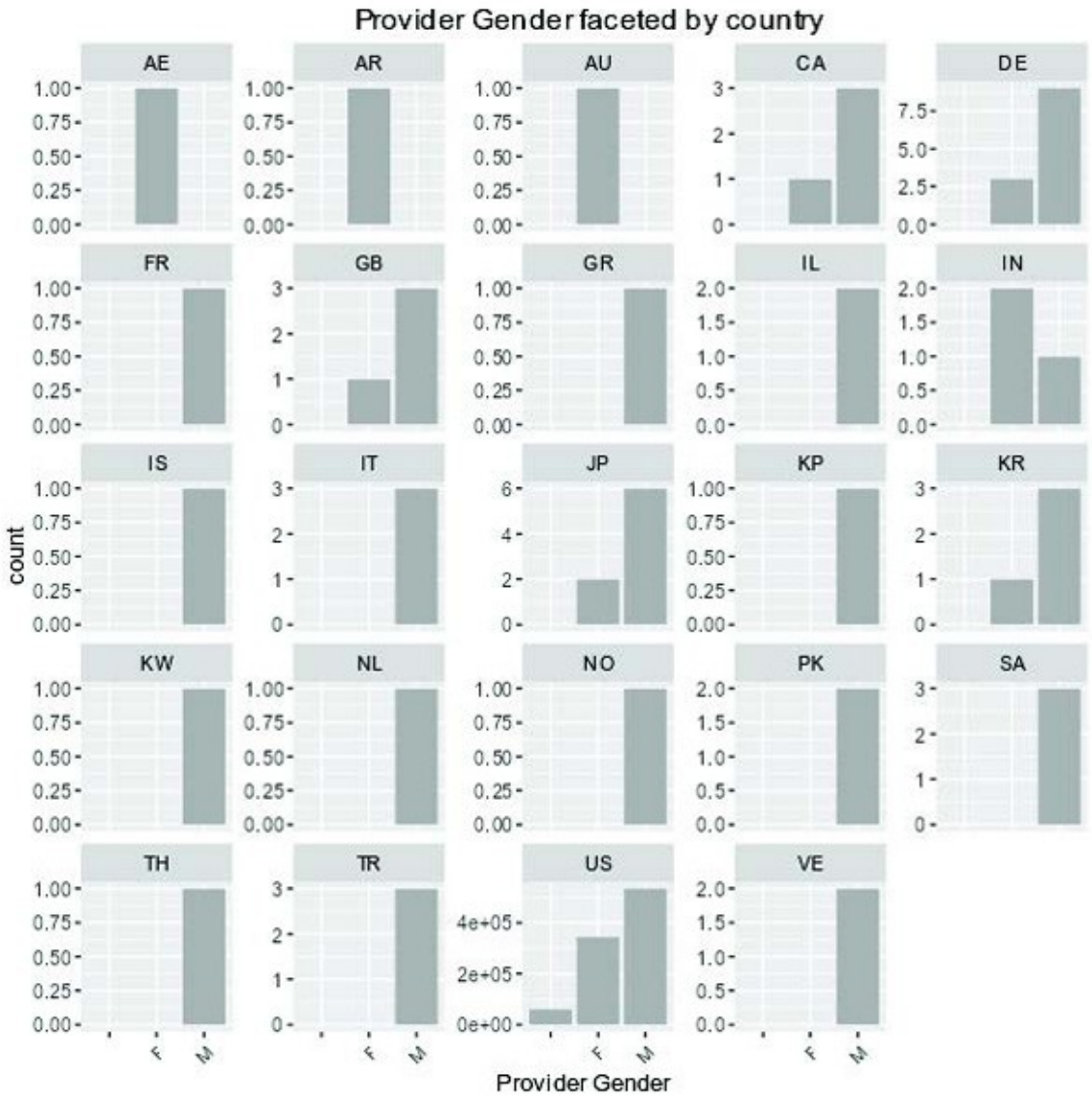
The initial code is pretty much the same as seen before: setting the traceback options, reading in arguments from the command line, and reading in the binary RDS file from disk. The difference is that the **file read in is the summarized dataset where the aggregation has been performed *by provider***.

The code then seeks to create faceted bar-charts in a grid:

```
# Create bar charts for provider-gender faceted by country
ggplot_faceted_bar_chart <-
ggplot(physicianMedicareDataSummaryTable) +
  geom_bar(aes(x = nppes_provider_gender),
  position = "dodge", fill = "darkgray") +
  labs(list(title = "Provider Gender faceted by country",
  y = "count", x = "Provider Gender")) +
  facet_wrap(~nppes_provider_country, scales = "free_y") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

The *geom_bar()* function is similar to the one we saw in code listing 14.2, except for the position argument. This is used (with the value supplied of "dodge") to dodge the bar chart using gender. The labels are applied using the labs() function as in previous code listings. The *facet_wrap()* function is new and it creates the faceting by the country of the provider. This leads to multiple bar-charts being created in a grid – one per country. The scales argument to *facet_wrap()* is set to "free_y" to indicate that the scale for each individual bar-chart is different on the y-axis. The *theme()* function then sets the x-axis category-label text (for gender) to lie right-justified at a 45 degree angle.

Figure 14.5 (Bar-charts of provider-gender faceted by country)



The visualization examples presented in this chapter barely scratch the surface of the capabilities of ggplot2 and plotly. The vast universe of charts, plots, and visualizations available in ggplot2 is too big to explore in this book: the reader is urged to refer to various web-resources and Hadley Wickham's book, ggplot2: Elegant Graphics for Data Analysis (http://www.springer.com/us/book/9780387981413).

Plotly will be explored further later in this book. But it also too vast to cover completely in this text. The reader is pointed towards the plotly website: https://plot.ly/.

Exercise 14.1: For the Reader

Modify code listing 14.3 to create bar-charts of provider-gender faceted by provider-type.

Chapter 15: Interactive Charts and Plots

Plotly is a JavaScript (https://en.wikipedia.org/wiki/JavaScript) library for visualization. It was open-sourced and made free in November-2015. In addition, an R Application Program Interface (API) was made available through an R package that allows the utilization of plotly within R. Plotly helps create beautiful interactive charts and plots. Info on the plotly R package and its use is available at https://plot.ly/r/ and

https://cran.r-project.org/web/packages/plotly/index.html.

In this chapter, we will see a few examples of code listings to highlight the power, beauty, and descriptive capabilities of interactive plotly charts, graphs and plots. Plotly uses other JavaScript libraries like D3 (https://d3js.org/) under the covers.

The first code listing, 15.1, seeks to create a horizontal bar-chart of Medicare 2014 dollar-amount payout by provider.

Code Listing 15.1

(Interactive plotly bar chart of Medicare 2014 payout by state)

To run this (all on one line):

```
Rscript 15.1_plotly_bar_chart_Medicare_pro-
fee_payout_by_US_state.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds
# Copyright (C) 2017 Sivakumaran Raman
# Use the plotly plotting interface
library("plotly")
library("dplyr")
# library('tidyr');
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------
########## Function to manipulate the data frame using the dplyr
package
manipulateDFUsingDplyrAggregateByState <-
function(myDataFrame) {

  # Summarize the data by state
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::summarize(total_medicare_prof_srv_payout =
  sum(total_medicare_prof_srv_revenue,
  na.rm = TRUE), provider_count = n()) %>%
```

```
  dplyr::mutate_at(c("nppes_provider_state"), as.character)

  return(myDataFrame)
}
#--------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
provided
# on the command line
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Call the function to manipulate the data frame using the dplyr
package
# Aggregate Medicare payout for 2014 by American state
summarizedByStateMedicareDataTable <-
  manipulateDFUsingDplyrAggregateByState(physicianMedicareDataT
able)
# Remove the original data-frame object and garbage collect it.
rm(physicianMedicareDataTable)
gc(verbose = FALSE)
# Plot the bar chart of Medicare pro-fee payout by US state
my_plot <- plot_ly(summarizedByStateMedicareDataTable,
  x =
~summarizedByStateMedicareDataTable$total_medicare_prof_srv_pa
yout,
  y = ~summarizedByStateMedicareDataTable$nppes_provider_state,
  color =
~summarizedByStateMedicareDataTable$nppes_provider_state,
  type = "bar", orientation = "h") %>%
  layout(title =
  "Medicare provider professional-fees payout by US state for 2014",
  xaxis = list(title = "Medicare Professional Services Payout"),
  yaxis = list(title = "US state")) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
```

```
  # on the modeBar
  plotly::config(displaylogo = FALSE,
  modeBarButtonsToRemove =
  list("sendDataToCloud"))
# Save as an HTML file
htmlwidgets::saveWidget(my_plot, file =
  "plotly_barchart_Medicare_pro-fee_payout_by_US_state_2014.html",
  selfcontained = FALSE)
```

By now, the first few parts of the program must be familiar to readers. The program reads in the summarized-data RDS binary file from disk and aggregates (using dplyr) the Medicare payout dollar-amounts for 2014 by American state using the *manipulateDFUsingDplyrAggregateByState()* function. The plotly interactive barchart is produced using this code segment:

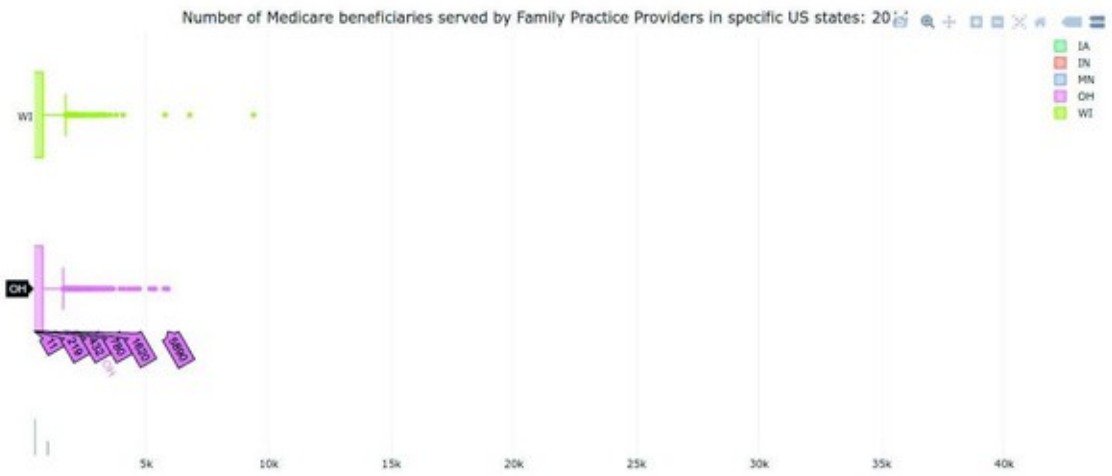
```
# Plot the bar chart of Medicare pro-fee payout by US state
my_plot <- plot_ly(summarizedByStateMedicareDataTable,
  x =
~summarizedByStateMedicareDataTable$total_medicare_prof_srv_pa
yout,
  y = ~summarizedByStateMedicareDataTable$nppes_provider_state,
  color =
~summarizedByStateMedicareDataTable$nppes_provider_state,
  type = "bar", orientation = "h") %>%
  layout(title =
  "Medicare provider professional-fees payout by US state for 2014",
  xaxis = list(title = "Medicare Professional Services Payout"),
  yaxis = list(title = "US state")) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
  # on the modeBar
  plotly::config(displaylogo = FALSE,
  modeBarButtonsToRemove =
  list("sendDataToCloud"))
```

Plotly also uses the chained-method syntax like dplyr using the %>% pipe operator. The *plot_ly()* function is called to act on the summarized-by-state data frame and sets the x-axis parameter, the y-axis parameter, the color-palette to use for the 50 different American states (plotly automatically chooses the colors in this case), type of the chart ("bar"), and the orientation (horizontal).

The *layout()* function called next in the chain sets the title, and the x-axis and y-axis labels.

The *config()* function from the plotly package is then called to ***turn off*** the publishing of the Plotly logo and the "Send to Cloud" buttons to the chart. The *saveWidget()* function from the htmlwidgets package is supplied the selfcontained = FALSE option this time, which leads to multiple files being created for one chart instead of a single HTML file.

Figure 15.1 (Interactive plotly bar-chart of Medicare payout by American state in 2014)



Code listing 15.2 below seeks to create a box-plot of the number of beneficiaries served per Family Practice Provider within a small list of American states.

Code Listing 15.2

(Interactive plotly box-plot: Family Practice provider patients)

To run this (all on one line):

```
Rscript
15.2_plotly_boxplots_provider_FamPrac_Medicare_beneficiaries_serv
ed.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds
# Copyright (C) 2017 Sivakumaran Raman
# Use the plotly plotting interface
library("plotly")
library("dplyr")
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
#-------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
provided
```

```
# on the command line
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Filter down to just family practice providers
physicianMedicareDataTable <- physicianMedicareDataTable %>%
  dplyr::mutate_at(c("nppes_provider_state"), as.character) %>%
  dplyr::filter(provider_type == "Family Practice" &
  nppes_provider_state %in% c("MN", "WI", "IN", "IA", "OH"))
# Plot the box plot for Medicare beneficiaries served per Family
Practice
# provider in specific US states
my_plot <- plot_ly(physicianMedicareDataTable,
  x = physicianMedicareDataTable$total_bene_unique_cnt,
  y = physicianMedicareDataTable$nppes_provider_state,
  color = physicianMedicareDataTable$nppes_provider_state,
  type = "box") %>%
  layout(title =
  paste("Number of Medicare beneficiaries served by Family Practice ",
  "Providers in specific US states: 2014", sep = "")) %>%
  # Edit configuration to turn off the plotly logo along with the
  # 'Produced with Plotly' message and the sendDataToCloud button
  # on the modeBar
  plotly::config(displaylogo = FALSE,
  modeBarButtonsToRemove = list("sendDataToCloud"));
# Save as an HTML file
htmlwidgets::saveWidget(my_plot,
  file = paste("plotly_boxplots_Medicare_beneficiaries_served_",
  "by_FamPrac_provider_2014.html", sep = ""),
  selfcontained = FALSE)
```

Program 15.2 reads in the aggregated-by-provider RDS dataset from disk as before. It then uses dplyr functions to filter the data down to just “Family Practice Providers” in 5 mid-Western US states: Minnesota, Wisconsin, Indiana, Iowa, and Ohio.

The code to request the box-plot is quite similar to the code to create the bar chart in listing 15.1: the one difference is that the type of plot requested within the *plot_ly()* function is “box”. The *layout()* and *config()* functions work quite the same as in listing 15.1.

Like in listing 15.1, the *saveWidget()* function from the htmlwidgets package is supplied the selfcontained = FALSE option, which leads to multiple files being created for one chart instead of a single HTML file.

Figure 15.2 (Interactive plotly box plot: Beneficiaries served per Family Practice

Provider in certain mid-Western American states in 2014)

Exercise 15.1: For the Reader

Modify code-listing 15.2 to create box-plots of number of beneficiaries served by provider for the provider-types of “General Surgery”, “Family Practice”, “Internal Medicine”, “Endocrinology”, and “Emergency Medicine”.

Chapter 16: Geographical Maps and Charts

Geographical maps can be of various kinds: choropleths, map sub-plots, maps with lines, scatter-plots on maps, bubble maps, etc.

Also, maps can either be rendered statically (as PDF, JPG, PNG, and similar outputs) or interactively (as HTML documents with JavaScript providing the interactivity).

Static map creation packages include using maps, choroplethr, maptools, tmap, etc.

Interactive map creation packages include plotly, leaflet, etc.

If there is a need to plot geographical points on a map projection, we need to make use of geocoding tools. Geocoding (https://en.wikipedia.org/wiki/Geocoding) is the process of using a geographical address to assign a latitude and longitude to the data-point, thus giving it spatial representation. Geocoding allows points to be plotted on a map. There are various free, non-free, and restricted-use geocoding tools available.

Gisgraphy (http://www.gisgraphy.com/) is a free and open-source geocoder but it is not usable in R. To use R program with Gisgraphy , we will have to run the address data-points through it beforehand to geocode them.

For our purposes, since the number of points we want to geocode is small, we use the geocoder available within the R Data Science Toolkit (https://cran.r-project.org/web/packages/RDSTK/index.html) CRAN package. This is directly usable within an R program and does geocoding-on-the-fly using a web-service call (which is slow but adequate for our purposes) to the Data Science Toolkit (http://www.datasciencetoolkit.org/about).

Other geocoding options include the Google Geocoding API which is accessible through the ggmap (https://cran.r-project.org/web/packages/ggmap/index.html) CRAN package – but Google places restrictions on the number of points that can be geocoded by a single user within a 24 hour period. The ggmap package also allows access to the geocoding web-service of the Data Science Toolkit (http://www.datasciencetoolkit.org/about).

Still other geocoding options available are Texas A&M Geoservices (http://geoservices.tamu.edu/Services/Geocode/), geocodio (https://geocod.io/), the US Census Bureau Geocoding API (https://geocoding.geo.census.gov/), ArcGIS Pro Geocoding (http://pro.arcgis.com/en/pro-app/help/data/geocoding/what-is-geocoding-.htm), and many others.

Our first set of maps are going to be static choropleths (https://en.wikipedia.org/wiki/Choropleth_map) showing the Medicare Fee For Service non-institutional claims payout for 2014 and provider-counts by US state. A choropleth uses density-of-shading proportional to the values of the variable in the regions being represented in the map.

For this program, we will use the choroplethr (https://cran.r-project.org/web/packages/choroplethr/index.html) and choroplethrMaps (https://cran.r-project.org/web/packages/choroplethrMaps/index.html) CRAN packages in addition to dplyr and ggplot2. The code is presented in listing 16.1.

Code Listing 16.1

(Static choropleth maps)

To run this (all on one line):

```
Rscript
16.1_Medicare_Provider_Util_Payment_choroplethr_geo_maps.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds
# Copyright (C) 2017 Sivakumaran Raman
# Use the choropleth maps interface to create geographical choropleth
maps of
# Medicare professional-fees payout and provider-count in 2014 by US
state
library("choroplethr")
library("choroplethrMaps")
library("ggplot2")
library("dplyr")
# library('tidyr');
########## ALL FUNCTIONS DEFINED
########## ------------------------------------------------------------------
########## Function to manipulate the data frame using the dplyr
package
manipulateDFUsingDplyrForGeoMaps <- function(myDataFrame,
state_regions) {
  # Summarize the data by state
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::summarize(total_medicare_prof_srv_payout =
  sum(total_medicare_prof_srv_revenue, na.rm = TRUE),
  provider_count = n()) %>%
  dplyr::left_join(state_regions, by = c(nppes_provider_state = "abb"))
%>%
  dplyr::rename(value = total_medicare_prof_srv_payout) %>%
  dplyr::filter(complete.cases(.))
  return(myDataFrame)
}
#--------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
```

```
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
provided on the
# command line
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
command
  line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Load the state.regions data frame from the choroplethrMaps package
data(state.regions)
# Call the function to manipulate the data frame using the dplyr
package
summarizedStateMedicareDataTable <-
  manipulateDFUsingDplyrForGeoMaps(physicianMedicareDataTable,
state.regions)
sink("dataframes_info.txt")
# Print out information about the objects
str(summarizedStateMedicareDataTable)
print(summarizedStateMedicareDataTable)
# New data frame to allow creation of choropleth map of provider
counts by state
summarizedPhysicianCountMedicareDataTable <-
  summarizedStateMedicareDataTable %>%
  dplyr::rename(total_medicare_prof_srv_payout = value, value =
provider_count)
# Print out information about the objects
str(summarizedPhysicianCountMedicareDataTable)
print(summarizedPhysicianCountMedicareDataTable)
sink()
# Remove the original data-frame object and garbage collect it.
rm(physicianMedicareDataTable)
gc(verbose = FALSE)
# Create the choropleth maps and save them to PNG files
my_state_choropleth_profee_payout_map <-
  choroplethr::state_choropleth(summarizedStateMedicareDataTable,
  title = "2014 Medicare Pro-Fees payout by US State",
  legend = "US Dollars")
my_state_choropleth_Medicare_providers_map <-
  choroplethr::state_choropleth(summarizedPhysicianCountMedicareDa
```

```
taTable,
  title = "2014 Medicare Providers by US State",
  legend = "Count of Providers")
ggplot2::ggsave(filename =
  "ChoroplethMap_US_States_Medicare_Pro_fee_payout_2014.png",
  plot = my_state_choropleth_profee_payout_map,
  dpi = 600,
  device = "png")
ggplot2::ggsave(filename =
  "ChoroplethMap_US_States_Medicare_Providers_2014.png",
  plot = my_state_choropleth_Medicare_providers_map,
  dpi = 600,
  device = "png")
```

The program starts by reading in the aggregated-by-provider RDS binary data file from disk. Then, the state.regions data frame from the choroplethrMaps package is loaded:

```
# Load the state.regions data frame from the choroplethrMaps package
data(state.regions)
```

This data frame contains column vectors representing info about the 50 US states:

- the state-name (region) as a lower-case string-based

- the state 2-letter abbreviation

- the FIPS numeric code for the state

- the FIPS character code for the state

The read-in data set along with the state.regions data frame are passed as parameters to the *manipulateDFUsingDplyrForGeoMaps()* function that uses dplyr to manipulate the data and perform a join-operation on the two data frames.

```
# Call the function to manipulate the data frame using the dplyr
package
summarizedStateMedicareDataTable <-
  manipulateDFUsingDplyrForGeoMaps(physicianMedicareDataTable,
state.regions)
```

The function works like this:

```
########## Function to manipulate the data frame using the dplyr
package
manipulateDFUsingDplyrForGeoMaps <- function(myDataFrame,
state_regions) {
  # Summarize the data by state
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
```

```
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::summarize(total_medicare_prof_srv_payout =
  sum(total_medicare_prof_srv_revenue, na.rm = TRUE),
  provider_count = n()) %>%
  dplyr::left_join(state_regions, by = c(nppes_provider_state = "abb"))
%>%
  dplyr::rename(value = total_medicare_prof_srv_payout) %>%
  dplyr::filter(complete.cases(.))
  return(myDataFrame)
}
```

The function starts by applying the dplyr *filter()* function to restrict the data-set to US providers. Next, we see the *group_by()* and *summarize()* functions used like before in order to aggregate the dollar pay-out data by US state and create a variable for provider-count by US state.

The dplyr package offers the ability to do SQL-like joins between data frames (https://cran.r-project.org/web/packages/dplyr/vignettes/two-table.html). We see this functionality in the use of the left_join() function that left-joins the Medicare pay-out dataset with the state.regions dataset.

To draw a US state-level choropleth map, the *state_choropleth()* function in the choroplethr package expects, as one of the parameters passed to it, a data frame with a column named "region" and a column named "value". Elements in the "region" column must exactly match how regions (states) are named in the "region" column in the state.regions dataset available from the choroplethrMaps package.

The *rename()* function is used in the code above to change the name of the *total Medicare professional services payout* variable to value. The variable named region is already available in the dataset because of the *left-join* that was performed with the state.regions dataset.

Finally, the *filter()* function is applied to retain only *complete cases* i.e. the rows with all variables present.

A second data-frame with the provider-counts variable renamed to value is created to allow the creation of the provider-counts-by-state choropleth:

```
# New data frame to allow creation of choropleth map of provider
# counts by state
summarizedPhysicianCountMedicareDataTable <-
  summarizedStateMedicareDataTable %>%
  dplyr::rename(total_medicare_prof_srv_payout = value,
  value = provider_count)
```

The actual calls to the *state_choropleth()* function are very simple. Since the objects returned by the state_choropleth() functions are ggplot2 objects, the *ggplot2::ggsave()* function can be used to save them as PNG image files.

```
# Create the choropleth maps and save them to PNG files
my_state_choropleth_profee_payout_map <-
  choroplethr::state_choropleth(summarizedStateMedicareDataTable,
  title = "2014 Medicare Pro-Fees payout by US State",
  legend = "US Dollars")
my_state_choropleth_Medicare_providers_map <-
  choroplethr::state_choropleth(summarizedPhysicianCountMedicareDa
taTable,
  title = "2014 Medicare Providers by US State",
  legend = "Count of Providers")
ggplot2::ggsave(filename =
  "ChoroplethMap_US_States_Medicare_Pro_fee_payout_2014.png",
  plot = my_state_choropleth_profee_payout_map,
  dpi = 600,
  device = "png")
ggplot2::ggsave(filename =
  "ChoroplethMap_US_States_Medicare_Providers_2014.png",
  plot = my_state_choropleth_Medicare_providers_map,
  dpi = 600,
  device = "png")
```

Figure 16.1 (Static choropleth of Medicare payout by US state)

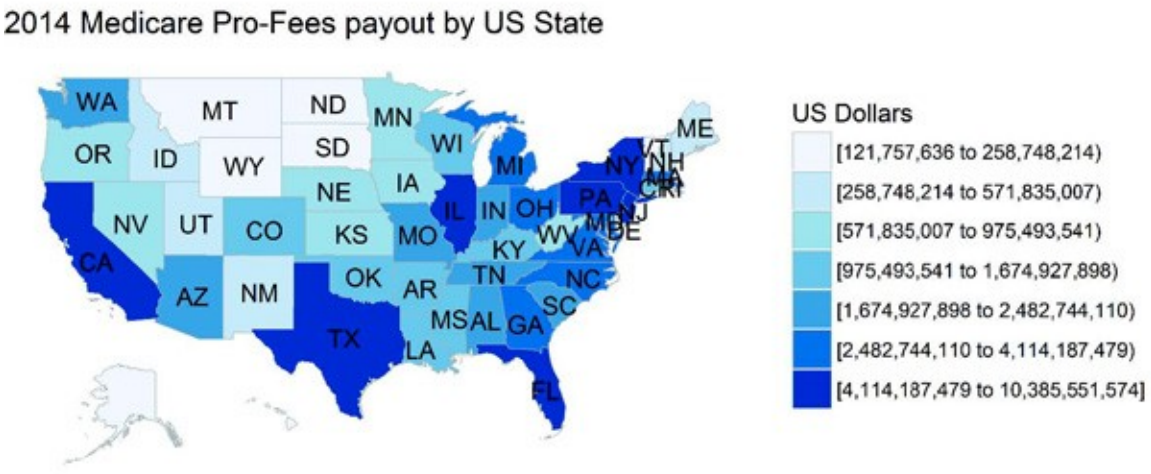


Figure 16.2 (Static choropleth of Medicare provider-count by US state)

The next map we will create is also a choropleth but an interactive one. We will use the previously encountered plotly package to create interactive, HTML-format choropleths. The interactivity of the plotly choropleth map allows more information to be presented in a single map. We will be able to present the information from both choroplethr choropleths created previously in code listing 16.1 in a single plotly choropleth. The code is presented in listing 16.2.

Code Listing 16.2

(Interactive choropleth map using plotly)

To run this (all on one line):

```
Rscript
16.2_Medicare_Provider_Util_Payment_plotly_choropleth_geo_maps.
R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds
# Copyright (C) 2017 Sivakumaran Raman
# Use the plotly maps interface to create a geographical choropleth map
of
# Medicare professional-fees payout in 2014 by US state that also
shows info
# on per-state provider count
library("plotly")
library("dplyr")
# library('tidyr');
########## ALL FUNCTIONS DEFINED
########## ----------------------------------------------------------------
########## Function to manipulate the data frame using the dplyr
package
manipulateDFUsingDplyrForGeoMaps <- function(myDataFrame) {
```

```
  # Summarize the data by state
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::summarize(total_medicare_prof_srv_payout =
  sum(total_medicare_prof_srv_revenue, na.rm = TRUE),
provider_count = n())
  return(myDataFrame)
}
#------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error = function() {
  traceback(2)
  stop("Error: stack trace printed above")
})
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly = TRUE)
# Exit with error message if RDS data-object file to read is not
provided on the
# command line
if (length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
command
  line argument")
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]])
# Call the function to manipulate the data frame using the dplyr
package
summarizedPhysicianMedicareDataTable <-
  manipulateDFUsingDplyrForGeoMaps(physicianMedicareDataTable)
# Remove the original data-frame object and garbage collect it.
rm(physicianMedicareDataTable)
gc(verbose = FALSE)
# Set up the hover pop-up on the map
summarizedPhysicianMedicareDataTable$hover <-
  with(summarizedPhysicianMedicareDataTable,
  paste(nppes_provider_state, "<br>", "No. of providers:",
provider_count))

# give state boundaries a white border
# l <- list(color = toRGB("white"), width = 2)
# specify some map projection/options
g <- list(scope = "usa", projection = list(type = "albers usa"),
```

```
  showlakes = TRUE,
  lakecolor = toRGB("white"))
plotly_geo_choropleth_map <- plotly::plot_geo(data =
  summarizedPhysicianMedicareDataTable,
  locationmode = "USA-states") %>%
  add_trace(z = ~total_medicare_prof_srv_payout, text = ~hover,
locations =
  ~nppes_provider_state,
  color = ~total_medicare_prof_srv_payout, colors = "Purples") %>%
  colorbar(title = "Medicare pro-fee payout 2014 USD") %>%
  layout(title =
  "2014 US Medicare Pro-fee Payout by State<br>(Hover for provider
count)",
  geo = g) %>%
  # Edit configuration to turn off the plotly logo along with the
  # 'Produced with Plotly' message and the sendDataToCloud button on
the modeBar
  plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
  list("sendDataToCloud"))
htmlwidgets::saveWidget(plotly_geo_choropleth_map,
  file = "plotly_geographical_map_Medicare_pro-
fee_payout_2014.html",
  selfcontained = FALSE)
```

Like in code listing 16.1, the program starts by reading in the aggregated-by-provider RDS binary data file, and calling a function to manipulate the data using dplyr. The function starts, as in listing 16.1, by applying the dplyr *filter()* function to restrict the data-set to US providers. Next, we see the *group_by()* and *summarize()* functions used like before in order to aggregate the dollar pay-out data by US state and create a variable for provider-count by US state. Unlike in code listing 16.1, we do not use the state.regions dataset.

For the information we will display when the viewer hovers over a particular state on the map, we set up an additional column named hover in the data frame with the value being the state-abbreviation and the number of providers in the state separated by an HTML line-break:

```
# Set up the hover pop-up on the map
summarizedPhysicianMedicareDataTable$hover <-
  with(summarizedPhysicianMedicareDataTable,
  paste(nppes_provider_state, "<br>", "No. of providers:",
provider_count))
```

We then set up some map projection options. Many standard map projections are available within plotly. Specifically, we are restricting the area shown on the map to the US, using the projection type Albers USA (https://bl.ocks.org/mbostock/5545680). We want water bodies like lakes to be shown, and request that lake-color be white.

```
# specify some map projection/options
g <- list(scope = "usa", projection = list(type = "albers usa"),
  showlakes = TRUE,
  lakecolor = toRGB("white"))
```

The actual creation of the interactive choropleth happens in the *plotly plot_geo()* function:

```
plotly_geo_choropleth_map <- plotly::plot_geo(data =
  summarizedPhysicianMedicareDataTable,
  locationmode = "USA-states") %>%
  add_trace(z = ~total_medicare_prof_srv_payout, text = ~hover,
  locations = ~nppes_provider_state,
  color = ~total_medicare_prof_srv_payout, colors = "Purples") %>%
  colorbar(title = "Medicare pro-fee payout 2014 USD") %>%
  layout(title =
  "2014 US Medicare Pro-fee Payout by State<br>(Hover for provider
count)",
  geo = g) %>%
  # Edit configuration to turn off the plotly logo along with the
  # 'Produced with Plotly' message and the sendDataToCloud button on
the
  # modeBar
  plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
  list("sendDataToCloud"))
```

Plotly too uses the chained-method syntax through the %>% operator. The data frame is passed to the *plot_geo()* function and the location-mode is set to US states.

The next function called in the chain is *add_trace()* which is supplied the numeric matrix of the *total Medicare professional services payout* (the variable that the choropleth map will represent) through the parameter z. The hover text is set and the function is asked to find locations information from the nppes_provider_state variable in the data frame. The color gradient is set to be based on the *total Medicare professional services payout* and the color-scheme for the map is set to shades of purple.

The next function called in the chain is *colorbar()* which sets the title for the color-legend bar that shows the shades of purple.

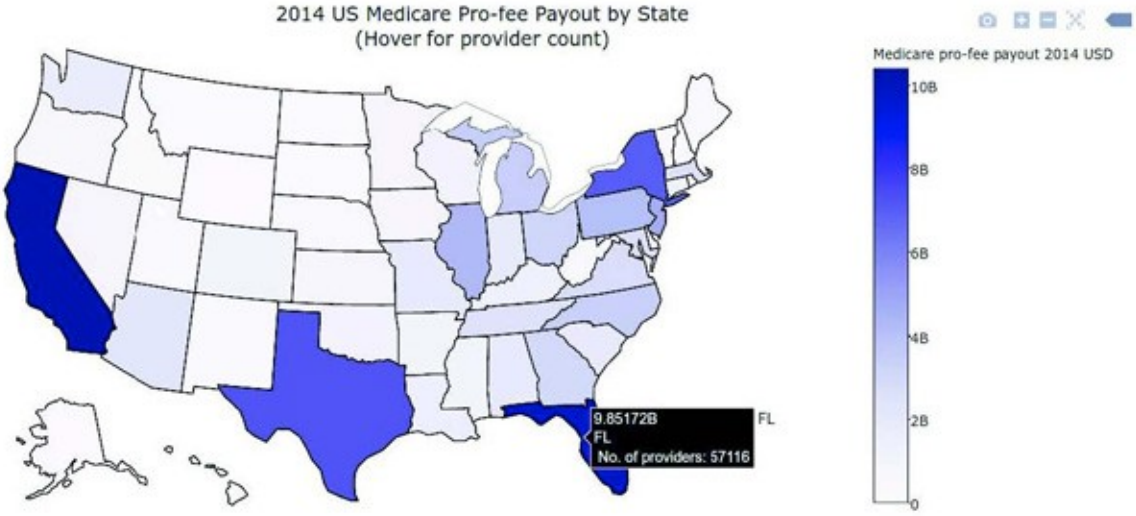
The *layout()* function then sets the title for the plot and assigns the previously-created g object with geographic map-properties to the attribute geo.

The last part of the plot is what we have seen previously – the *config()* function being used to turn off the default plotly logo and some buttons.

Finally, the plot is saved as an HTML file using the *saveWidget()* function from the htmlwidgets package. Figure 16.3 shows (in a static manner) how the interactive plot looks and works: hovering over any state displays a pop-up showing both the Medicare payout amount and the provider-count for the state. The density of the purple color for

any state is correlated with the Medicare-payout for the state. Thus, this interactive choropleth map is able to display the information from both previously created static choropleth maps and adds exploration capabilities because of the interactivity. For experiencing the interactive features of the map, please open the HTML file in a browser.

Figure 16.3 (Interactive choropleth of Medicare pro-fee payout by US state)



Next, we will create an interactive geographical scatterplot map with plotly. We will be identifying the *top Medicare Fee-For-Service Professional Services Revenue-Earning Provider* in each US state and plotting the data on an interactive map. This will require geocoding in order to convert the address of the provider to a latitude+longitude value. For geocoding, we will be using the R Data Science Toolkit package (RDSTK: https://cran.r-project.org/web/packages/RDSTK/index.html), which is a wrapper around the Data Science Toolkit: http://www.datasciencetoolkit.org/

Some packages like httr and rjson are required to use RDSTK.

The data manipulation will still be done using dplyr.

Let us dive right into the code listing.

Code Listing 16.3

(Interactive geographical scatterplot map using plotly)

To run this (all on one line):

Rscript
16.3_Medicare_Provider_Util_Payment_plotly_scatterplot_geo_maps_top_providers.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZED.rds
Copyright (C) 2017 Sivakumaran Raman

```
# Use the plotly maps interface to create a geographical scatterplot map
of
# the top Medicare revenue providers in 2014 by US state
library("plotly");
library("RDSTK");
library("httr");
library("rjson");
library("dplyr");
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Get latitude by geocoding the address
geo.dsk.lat <- function(addr){
  # single address geocode with data sciences toolkit to return latitude
  url  <- "http://www.datasciencetoolkit.org/maps/api/geocode/json";
  response <- GET(url,query=list(sensor="FALSE",address=addr));
  json <- fromJSON(content(response, type="text", encoding="UTF-
8"));
  loc  <- json['results'][[1]][[1]]$geometry$location;
  return(loc$lat);
}
# Get longitude by geocoding the address
geo.dsk.long <- function(addr){
  # single address geocode with data sciences toolkit to return longitude
  url  <- "http://www.datasciencetoolkit.org/maps/api/geocode/json";
  response <- GET(url,query=list(sensor="FALSE",address=addr));
  json <- fromJSON(content(response, type="text", encoding="UTF-
8"));
  loc  <- json['results'][[1]][[1]]$geometry$location;
  return(loc$lng);
}
# Function to manipulate the data frame using the dplyr and tidyr
packages
manipulateDFUsingDplyrForGeoMaps <- function(myDataFrame) {
  # Summarize the data by selecting only the top-earning provider by
  # US state (if there is more than one provider in a state with the same
  # min-revenue, choose one of them at random)
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::mutate(the_rank = rank(-total_medicare_prof_srv_revenue,
  ties.method="random")) %>%
  dplyr::filter(the_rank == 1) %>%
  dplyr::select(-the_rank) %>%
  # Filter to only those rows with a valid zip code
  dplyr::filter(nchar(nppes_provider_zip) >= 5) %>%
  # Has to be a US State
```

```
  dplyr::filter(nppes_provider_state %in% state.abb)  %>%
  dplyr::mutate(fulladdress = paste(nppes_provider_street1,
  nppes_provider_city,
  nppes_provider_state,
  substr(nppes_provider_zip, 1, 5),
  nppes_provider_country, sep=", "),
  fullname = paste(nppes_provider_first_name,
  nppes_provider_mi,
  nppes_provider_last_org_name,
  sep=" "));

  # Add in latitude, longitude as columns to the data frame.
  myDataFrame <- myDataFrame %>%
  dplyr::mutate(latitude = geo.dsk.lat(fulladdress),
  longitude = geo.dsk.long(fulladdress));

  write.csv(myDataFrame, file="topRevenueProvidersByUSState.csv");

  return(myDataFrame);
}
#------------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error= function () {
  traceback(2);
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided
# on the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument");
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]]);
# Call the function to manipulate the data frame using the dplyr
package
summarizedPhysicianMedicareDataTable <-
  manipulateDFUsingDplyrForGeoMaps(physicianMedicareDataTable)
;
# Remove the original data-frame object and garbage collect it.
rm(physicianMedicareDataTable);
```

```
gc(verbose=FALSE);
# geo styling
geo_info <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  showland = TRUE,
  landcolor = toRGB("gray95"),
  subunitcolor = toRGB("gray85"),
  countrycolor = toRGB("gray85"),
  countrywidth = 0.5,
  subunitwidth = 0.5
)
# Plot the map showing top-Medicare-revenue providers by US state
plotly_geo_scatterplot_map <-
plot_geo(summarizedPhysicianMedicareDataTable,
  lat = ~latitude, lon = ~longitude) %>%
  add_markers(text = ~paste(fullname,
  nppes_provider_state,
  paste("Total Medicare Revenue 2014 in USD: ",
  total_medicare_prof_srv_revenue),
  sep = "<br />"),
  color = ~total_medicare_prof_srv_revenue,
  symbol = I("square"),
  size = I(8),
  hoverinfo = "text") %>%
  colorbar(title = "Medicare Pro-Fee Revenue in USD<br />2014") %>
%
  layout(title = paste('Top 2014 Medicare-Revenue Providers by US
State',
  '<br />(Hover for provider info)'),
  geo = geo_info) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
on the
  # modeBar
  plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
  list("sendDataToCloud"));
# Save the map to HTML
htmlwidgets::saveWidget(plotly_geo_scatterplot_map,
  file=
  "plotly_geo_map_Medicare_pro-
fee_top_revenue_providers_by_state_2014.html",
  selfcontained=FALSE);
```

The program reads in the summarized-by-provider RDS binary-file dataset from disk. It then calls the manipulateDFUsingDplyrForGeoMaps() function to manipulate the data using dplyr.

```
# Function to manipulate the data frame using the dplyr and tidyr
packages
manipulateDFUsingDplyrForGeoMaps <- function(myDataFrame) {
  # Summarize the data by selecting only the top-earning provider by
  # US state (if there is more than one provider in a state with the same
  # max-revenue, choose one of them at random)
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::mutate(the_rank = rank(-total_medicare_prof_srv_revenue,
  ties.method="random")) %>%
  dplyr::filter(the_rank == 1) %>%
  dplyr::select(-the_rank) %>%
  # Filter to only those rows with a valid zip code
  dplyr::filter(nchar(nppes_provider_zip) >= 5) %>%
  # Has to be a US State
  dplyr::filter(nppes_provider_state %in% state.abb)  %>%
  dplyr::mutate(fulladdress = paste(nppes_provider_street1,
  nppes_provider_city,
  nppes_provider_state,
  substr(nppes_provider_zip, 1, 5),
  nppes_provider_country, sep=", "),
  fullname = paste(nppes_provider_first_name,
  nppes_provider_mi,
  nppes_provider_last_org_name,
  sep=" "));

  # Add in latitude, longitude as columns to the data frame.
  myDataFrame <- myDataFrame %>%
  dplyr::mutate(latitude = geo.dsk.lat(fulladdress),
  longitude = geo.dsk.long(fulladdress));

  write.csv(myDataFrame, file="topRevenueProvidersByUSState.csv");

  return(myDataFrame);
}
```

The dplyr function-chain first restricts the data to US providers. It then groups and summarizes by US state. The *rank()* function applied to the total_medicare_prof_srv_revenue variable (with a negative sign) helps to identify the top-revenue provider in each state. Ties are broken using random selection. The data is filtered down to just those rows that have a valid 5-or-more-digit zip codes. The nppes_provider_state variable is then checked against the state.abb dataset of two-character state-abbreviations to ensure that the data is only for valid US states. String-concatenation is used through the paste() function to create new variables fullname and fulladdress.

Next, the data is geocoded. This is done by calling the *geo.dsk.lat()* and *geo.dsk.long()* functions for each row of the data. These two functions use the web-services provided by the Data Science Toolkit to assign latitude and longitude to each address data-point.

The function also saves this top-revenue-provider-by-state data as a CSV file using the *write.csv()* function.

Next, we move on to the actual plotting. The geographical map criteria are set up as a list using this segment of code:

```
# geo styling
geo_info <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  showland = TRUE,
  landcolor = toRGB("gray95"),
  subunitcolor = toRGB("gray85"),
  countrycolor = toRGB("gray85"),
  countrywidth = 0.5,
  subunitwidth = 0.5
)
```

As in the interactive choropleth map, we are using the Albers USA map projection.

Next, the scatterplot map is drawn:

```
# Plot the map showing top-Medicare-revenue providers by US state
plotly_geo_scatterplot_map <-
plot_geo(summarizedPhysicianMedicareDataTable,
  lat = ~latitude, lon = ~longitude) %>%
  add_markers(text = ~paste(fullname,
  nppes_provider_state,
  paste("Total Medicare Revenue 2014 in USD: ",
  total_medicare_prof_srv_revenue),
  sep = "<br />"),
  color = ~total_medicare_prof_srv_revenue,
  symbol = I("square"),
  size = I(8),
  hoverinfo = "text") %>%
  colorbar(title = "Medicare Pro-Fee Revenue in USD<br />2014") %>
%
  layout(title = paste('Top 2014 Medicare-Revenue Providers by US
State',
  '<br />(Hover for provider info)'),
  geo = geo_info) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
on the
```

```
# modeBar
plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
list("sendDataToCloud"));
```

The *plot_geo()* function sets up the dataset to work with, and the latitude and longitude variables to use from within the dataset.

The *add_markers()* function then sets up the text-string to display when hovering over a plotted data-point, the variable (total_medicare_prof_srv_revenue) to be used to create the color-gradient for the plotted points, the symbol (square) to be used for the plotted point, and the size of the plotted point.

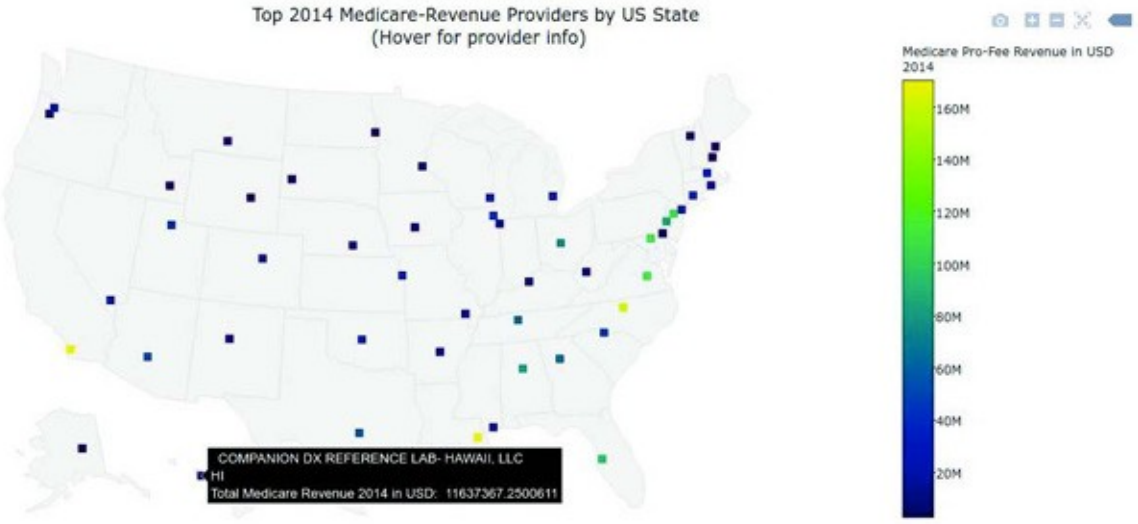
The *colorbar()* function sets up the title for the color-legend bar. The *layout()* function sets the title for the plot and sets the geo parameter (the geographical info for the layout) to the geo_info list created earlier.

Lastly, like before, the *config()* function is used to turn off the plotly logo and some buttons.

The HTML file containing the plot is saved using the *htmlwidgets::saveWidget()* function.

The interactivity in the HTML plot is experienced by hovering over any plotted provider data-point: doing so displays a pop-up showing the name of the provider, the state, and Medicare FFS professional services revenue in 2014.

Figure 16.4 (Interactive geographical scatterplot map of the top Medicare professional services revenue providers by US state)



interactive geographical scatterplot map of the top ***individual*** Medicare professional services revenue providers by US state. The things that code listing 16.4 does differently from code listing 16.3 are;

- In the data-manipulation function, the data is filtered by nppes_entity_code to include only individual providers.

- The *htmlwidgets::saveWidget()* function to save the HTML plot-file uses the selfcontained=TRUE option to create a single HTML file as the output with all the JavaScript and other accessory libraries inlined. This option requires the Pandoc software library to be installed and available on the computer.

Code Listing 16.4

(Interactive geographical scatterplot map using plotly)

To run this (all on one line):

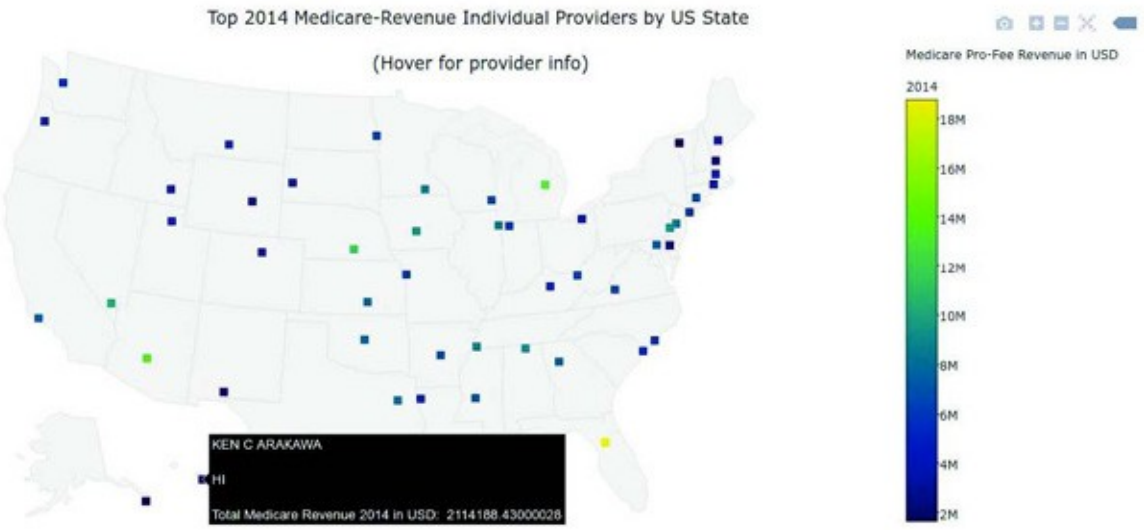
```
Rscript
16.4_Medicare_Provider_Util_Payment_plotly_scatterplot_geo_maps_
top_indiv_providers.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds
# Copyright (C) 2017 Sivakumaran Raman
# Use the plotly maps interface to create a geographical scatterplot map
of
# the top Medicare revenue individual-providers in 2014 by US state
library("plotly");
library("RDSTK");
library("httr");
library("rjson");
library("dplyr");
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Get latitude by geocoding the address
geo.dsk.lat <- function(addr){
  # single address geocode with data sciences toolkit to return latitude
  url  <- "http://www.datasciencetoolkit.org/maps/api/geocode/json";
  response <- GET(url,query=list(sensor="FALSE",address=addr));
  json <- fromJSON(content(response, type="text", encoding="UTF-
8"));
  loc  <- json['results'][[1]][[1]]$geometry$location;
  return(loc$lat);
}
# Get longitude by geocoding the address
geo.dsk.long <- function(addr){
  # single address geocode with data sciences toolkit to return longitude
  url  <- "http://www.datasciencetoolkit.org/maps/api/geocode/json";
  response <- GET(url,query=list(sensor="FALSE",address=addr));
  json <- fromJSON(content(response, type="text", encoding="UTF-
8"));
  loc  <- json['results'][[1]][[1]]$geometry$location;
```

```
  return(loc$lng);
}
# Function to manipulate the data frame using the dplyr and tidyr
packages
manipulateDFUsingDplyrForGeoMaps <- function(myDataFrame) {
  # Summarize the data by selecting only the top-earning provider by
US state
  # (if there is more than one provider in a state with the same max-
revenue,
  # choose one of them at random)
  myDataFrame <- myDataFrame %>%
  dplyr::filter(nppes_provider_country == "US") %>%
  dplyr::filter(nppes_entity_code == "I") %>%
  dplyr::group_by(nppes_provider_state) %>%
  dplyr::mutate(the_rank = rank(-total_medicare_prof_srv_revenue,
  ties.method="random")) %>%
  dplyr::filter(the_rank == 1) %>%
  dplyr::select(-the_rank) %>%
  # Filter to only those rows with a valid zip code
  dplyr::filter(nchar(nppes_provider_zip) >= 5) %>%
  # Has to be a US State
  dplyr::filter(nppes_provider_state %in% state.abb)  %>%
  dplyr::mutate(fulladdress = paste(nppes_provider_street1,
  nppes_provider_city,
  nppes_provider_state,
  substr(nppes_provider_zip, 1, 5), nppes_provider_country, sep=", "),
  fullname  = paste(nppes_provider_first_name,
  nppes_provider_mi,
  nppes_provider_last_org_name,
  sep=" "));

  # Add in latitude, longitude as columns to the data frame.
  myDataFrame <- myDataFrame %>%
  dplyr::mutate(latitude = geo.dsk.lat(fulladdress),
  longitude = geo.dsk.long(fulladdress));

  write.csv(myDataFrame,
file="topRevenueIndivProvidersByUSState.csv");

  return(myDataFrame);
}
#--------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error= function () {
  traceback(2);
```

```
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided
# on the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
command
  line argument");
}
# Read in the R data-object from disk by calling the function readRDS
physicianMedicareDataTable <- readRDS(myArgs[[1]]);
# Call the function to manipulate the data frame using the dplyr
package
summarizedPhysicianMedicareDataTable <-
  manipulateDFUsingDplyrForGeoMaps(physicianMedicareDataTable)
;
# Remove the original data-frame object and garbage collect it.
rm(physicianMedicareDataTable);
gc(verbose=FALSE);
# geo styling
geo_info <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  showland = TRUE,
  landcolor = toRGB("gray95"),
  subunitcolor = toRGB("gray85"),
  countrycolor = toRGB("gray85"),
  countrywidth = 0.5,
  subunitwidth = 0.5
)
# Plot the map showing top-Medicare-revenue providers by US state
plotly_geo_scatterplot_map <-
  plot_geo(summarizedPhysicianMedicareDataTable,
  lat = ~latitude, lon = ~longitude) %>%
  add_markers(text =
  ~paste(fullname,
  nppes_provider_state,
  paste("Total Medicare Revenue 2014 in USD: ",
  total_medicare_prof_srv_revenue),
  sep = "<br />"),
  color = ~total_medicare_prof_srv_revenue,
  symbol = I("square"),
```

```
  size = I(8),
  hoverinfo = "text"
  ) %>%
  colorbar(title = "Medicare Pro-Fee Revenue in USD<br />2014") %>
%
  layout(yz =
  paste('Top 2014 Medicare-Revenue Individual Providers by US
State',
  '<br />(Hover for provider info)'),
  geo = geo_info) %>%
  # Edit configuration to turn off the plotly logo along with the
  # "Produced with Plotly" message and the sendDataToCloud button
on
  # the modeBar
  plotly::config(displaylogo = FALSE, modeBarButtonsToRemove =
  list("sendDataToCloud"));
# Save the map to HTML
htmlwidgets::saveWidget(plotly_geo_scatterplot_map,
  file=
  paste("plotly_geo_map_Medicare_pro-fee_top_revenue",
  "_individual_providers_by_state_2014.html", sep = ""),
  selfcontained=TRUE);
```

Figure 16.5 (Interactive geographical scatterplot map of the top individual Medicare professional services revenue providers by US state)



The *static* and *interactive* charting/visualization capabilities of R are very extensive and need book-level treatments to do them justice. The reader is encouraged to explore ggplot2, plotly, and various other charting and visualization packages in R: http://www.computerworld.com/article/2921176/business-intelligence/great-r-packages-for-data-import-wrangling-visualization.html.

Exercise 16.1: For the Reader

Modify code listing 16.4 and create a scatterplot map that shows the highest-revenue-earning and the lowest-revenue-earning individual provider in each state.

Exercise 16.2: For the Reader

Modify code listing 16.4 and create a scatterplot map that shows the highest-revenue-earning male and the highest-revenue-earning female individual provider in each state.

Exercise 16.3: For the Reader

Modify code listing 16.4 and create a scatterplot map that shows the highest-revenue-earning individual provider for each provider_type.

Chapter 17: Regression Modeling & Predictive Models

Regression analysis/modeling is commonly used in data analysis. From the Wikipedia page on Regression Analysis (https://en.wikipedia.org/wiki/Regression_analysis): *In statistical modeling, regression analysis is a statistical process for estimating the relationships among variables. It includes many techniques for modeling and analyzing several variables, when the focus is on the relationship between a dependent variable and one or more independent variables (or 'predictors').*

Regression analysis is also used to create predictive models and forecast outcomes. Predictive Regression models can be developed by training a model on a particular data set and using it to make predictions on a different but similar data set. We will see this in in action.

For our analysis, we are working with the summarized-by-provider Medicare non-institutional medical claims for 2014 that we created from the full Public Use File dataset, which was unsummarized. Our dependent variable is total_medicare_prof_srv_revenue (Total Medical Fee-for-Service Professional Services Revenue for the provider in 2014). We will run a multiple linear regression to develop a model that uses a set of explanatory variables to predict the total Medicare professional services revenue for a provider. Then, we will use the model to make predictions and see if our predictions are close to reality or not.

We will also be using the ggplot2 package to create plots that visualize the accuracy of the predictive model.

Code listing 17.1 runs a multiple linear regression model with total_medicare_prof_srv_revenue as the dependent variable and utilizes a whole lot of explanatory variables.

Code Listing 17.1

(Multiple linear regression model)

To run this (all on one line):

```
Rscript
17.1_Medicare_Provider_Util_Payment_linear_regression_model.R ../
data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZE
D.rds > LM_output.txt 2> LM_errors.txt
# Copyright (C) 2017 Sivakumaran Raman
library("data.table");
library("dplyr");
library("dtplyr");
library("ggplot2");
library("broom");
```

```
library("ztable");
library("stargazer")
# library("tidyr");
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Function to pretty print linear regression model info using ztable
prettyPrintHTMLRulesResultsUsingZtableStargazer <-
  function(linearRegressionModelTidyDF, linearRegressionModel,
  fileForHTMLOutput) {

  # Rules checks output in pretty HTML
  # Set the options for the ztable printing of pretty tabular output
  options(ztable.type="html", ztable.colnames.bold=TRUE);

  # Use ztable to get pretty-table output about Linear Regression Model
  # data-frame
  ztableObject <- ztable::ztable(linearRegressionModelTidyDF,
  caption="Linear Regression Model Info (using ztable)",
caption.bold=TRUE,
  tablewidth=0.1, zebra=1,
  zebra.type=2, zebra.color=5, position="left", show.footer=FALSE,
  hline.after=c(-1:nrow(linearRegressionModelTidyDF)),
  wraptable=TRUE, wraptablewidth=6);

  sink(fileForHTMLOutput);
  print(ztableObject);

  # Use stargazer to get pretty-table output about the
  # Linear Regression Model data-frame
  stargazer(linearRegressionModel,
  type = "html",
  title = "Linear Regression Model Info (using stargazer)",
  nobs = TRUE,
  single.row = TRUE)

  sink();

  return(ztableObject);
}
# Function to calculate the R-Squared value for a linear regression
model
rsquared_value <- function(y,f) {
  return(1 - sum((y-f)^2)/sum((y-mean(y))^2));
}
#--------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
```

```
quit.
options(error= function () {
  traceback(2);
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided on
# the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument");
}
# Read in the R data-object from disk by calling the function readRDS
summarizedPhysicianMedicareDataTable <- readRDS(myArgs[[1]]);
print("Minimum value for actual
total_medicare_prof_srv_revenue:\n");
min(summarizedPhysicianMedicareDataTable$total_medicare_prof_sr
v_revenue);
print("Maximum value for actual
total_medicare_prof_srv_revenue:\n");
max(summarizedPhysicianMedicareDataTable$total_medicare_prof_sr
v_revenue);
# Develop the linear regression model on the data
linearRegressionModel <-
  lm(total_medicare_prof_srv_revenue ~
  nppes_provider_gender +
  nppes_entity_code +
  provider_type +
  medicare_participation_indicator +
  total_line_srvc_cnt +
  total_bene_unique_cnt +
  total_bene_day_srvc_cnt +
  nppes_provider_country +
  nppes_provider_state,
  data=summarizedPhysicianMedicareDataTable);
print("Summary for linear regression model:");
summary(linearRegressionModel);
# Get a tidy data-frame representation of the linear regression model
object
linearRegressionModelTidyBroomDataFrame <-
broom::tidy(linearRegressionModel);
# File to write out the HTML output from ztable for the linear
regression
```

```
# model object
outputZTableHTMLFile <- "R_output_linear_regression.html";
# Call the function to pretty-print the linear regression model info using
# the ztable and stargazer packages
myZtableObject <-
  prettyPrintHTMLRulesResultsUsingZtableStargazer(
  linearRegressionModelTidyBroomDataFrame,
linearRegressionModel,
  outputZTableHTMLFile);
```

Code listing 17.1 sets up the multiple linear regression model. The summarized-by-provider dataset is read in from the binary RDS file. The linear regression model is set up using specific model-notation in R:

```
# Develop the linear regression model on the data
linearRegressionModel <-
  lm(total_medicare_prof_srv_revenue ~
  nppes_provider_gender +
  nppes_entity_code +
  provider_type +
  medicare_participation_indicator +
  total_line_srvc_cnt +
  total_bene_unique_cnt +
  total_bene_day_srvc_cnt +
  nppes_provider_country +
  nppes_provider_state,
  data=summarizedPhysicianMedicareDataTable);
```

Here, *lm()* is the function that runs the multiple linear regression model. The results of the model are read into the linearRegressionModel object. The notation used to set up the regression model inside the call to the *lm()* function is special and specific to R. The dependent variable (total_medicare_prof_srv_revenue) is named first. Then comes the tilde (~) symbol. The complete list of explanatory variables comes after the tilde and they are separated by + signs. The data parameter to the *lm()* function specifies the dataset to act on – the variables specified in the regression model are all columns of this data frame.

After the model has completed running, the results are read into the linearRegressionModel object named in the code. Since the results are in a data object, they can be manipulated and pretty-printed. The *tidy()* function from the broom package is used to tidy-up the results-object before it is pretty-printed as an HTML table using the ztable package. The results-object is again pretty-printed as an HTML table using the stargazer package – however, this does not require any tidying up of the results-object.

Partial-views of the results tables created by ztable and stargazer are available in Figure 17.1.

Figure 17.1 (Multiple linear regression results pretty-printed using ztable and stargazer)

Linear Regression Model Info (using ztable)

	term	estimate	std.error	statistic	p.value
1	(Intercept)	42326.45	373747.75	0.11	0.91
2	nppes_provider_genderF	-122218.66	24766.60	-4.93	0.00
3	nppes_provider_genderM	-91375.23	24763.93	-3.69	0.00
4	provider_typeAllergy/Immunology	-10079.50	35089.79	-0.29	0.77
5	provider_typeAll Other Suppliers	80539.10	44878.13	1.79	0.07
6	provider_typeAmbulance Service Supplier	373577.99	42581.34	8.77	0.00
7	provider_typeAmbulatory Surgical Center	495727.55	42786.44	11.59	0.00
8	provider_typeAnesthesiologist Assistants	-42073.62	36158.61	-1.16	0.24
9	provider_typeAnesthesiology	-22277.84	34530.61	-0.65	0.52

Linear Regression Model Info (using stargazer)

	Dependent variable:
	total_medicare_prof_srv_revenue
nppes_provider_genderF	-122,218.700*** (24,766.600)
nppes_provider_genderM	-91,375.240*** (24,763.930)
nppes_entity_codeO	
provider_typeAllergy/Immunology	-10,079.500 (35,089.790)
provider_typeAll Other Suppliers	80,539.100* (44,878.130)
provider_typeAmbulance Service Supplier	373,578.000*** (42,581.340)
provider_typeAmbulatory Surgical Center	495,727.600*** (42,786.440)
provider_typeAnesthesiologist Assistants	-42,073.620 (36,158.610)
provider_typeAnesthesiology	-22,277.840 (34,530.610)

Looking at the results of the multiple linear regression model, we pick out the explanatory variables that meet our threshold level of significance as defined by a p value of < 0.01. For variables that are factors with multiple levels, we consider the variable significant if any one of its factor-levels has a p value < 0.01. In the next code listing, 17.2, we re-run the multiple linear regression model (on a random two-thirds of the data that we call the ***training*** dataset) but, this time, we include only the significant explanatory variables from the model we created in code listing 17.1. We then use this regression model object to make predictions on the left-over one-third of the data (which we call the ***test*** dataset).

Code Listing 17.2

(Multiple linear regression predictive model)

To run this (all on one line):

```
Rscript
17.2_ALLDATA_Medicare_Provider_Util_Payment_linear_regression
_with_prediction.R
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds > LM_and_predictions_output.txt 2>
LM_and_predictions_errors.txt
# Copyright (C) 2017 Sivakumaran Raman
library("data.table");
library("dplyr");
library("dtplyr");
library("ggplot2");
library("broom");
library("ztable");
library("stargazer")
# library("tidyr");
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Function to pretty print linear regression model info using ztable
prettyPrintHTMLRulesResultsUsingZtableStargazer <-
  function(linearRegressionModelTidyDF, linearRegressionModel,
  fileForHTMLOutput) {

  # Rules checks output in pretty HTML
  # Set the options for the ztable printing of pretty tabular output
  options(ztable.type="html", ztable.colnames.bold=TRUE);


  # Use ztable to get pretty-table output about Linear Regression Model
  # data-frame
  ztableObject <- ztable::ztable(linearRegressionModelTidyDF,
  caption="Linear Regression Model Info - Sign. variables (using
ztable)",
  caption.bold=TRUE,
  tablewidth=0.1, zebra=1,
  zebra.type=2, zebra.color=5, position="left", show.footer=FALSE,
  hline.after=c(-1:nrow(linearRegressionModelTidyDF)),
  wraptable=TRUE, wraptablewidth=6);


  sink(fileForHTMLOutput);
  print(ztableObject);


  # Use stargazer to get pretty-table output about the
  # Linear Regression Model data-frame
  stargazer(linearRegressionModel,
```

```
  type = "html",
  title = "Linear Regression Model Info - Sign. variables (using
stargazer)",
  nobs = TRUE,
  single.row = TRUE)

  sink();

  return(ztableObject);
}
# Function to calculate the R-Squared value for a linear regression
model
rsquared_value <- function(y,f) {
  return(1 – sum((y – f)^2)/sum((y - mean(y))^2));
}
#-------------------------------------------------------------------------
# Set option to print the stack trace at the time of any error and then
quit.
options(error= function () {
  traceback(2);
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided on
# the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument");
}
# Read in the R data-object from disk by calling the function readRDS
summarizedPhysicianMedicareDataTable <- readRDS(myArgs[[1]]);
print("Minimum value for actual
total_medicare_prof_srv_revenue:\n");
min(summarizedPhysicianMedicareDataTable$total_medicare_prof_sr
v_revenue);
print("Maximum value for actual
total_medicare_prof_srv_revenue:\n");
max(summarizedPhysicianMedicareDataTable$total_medicare_prof_sr
v_revenue);
# Create a training data set
trainingDataFrame <- summarizedPhysicianMedicareDataTable %>%
  dplyr::filter(random_val_between_one_and_three == 1 |
  random_val_between_one_and_three == 2);
```

```
# Create a test data set
testDataFrame <- summarizedPhysicianMedicareDataTable %>%
  dplyr::filter(random_val_between_one_and_three == 3);
# Develop the linear regression model on the training data
linearRegressionModel <-
  lm(total_medicare_prof_srv_revenue ~
  nppes_provider_gender +
  # nppes_entity_code +
  provider_type +
  medicare_participation_indicator +
  total_line_srvc_cnt +
  total_bene_unique_cnt +
  total_bene_day_srvc_cnt  ,
  data=trainingDataFrame);
print("Summary for linear regression model:");
summary(linearRegressionModel);
# Get a tidy data-frame representation of the linear regression model
object
linearRegressionModelTidyBroomDataFrame <-
broom::tidy(linearRegressionModel);
# File to write out the HTML output from ztable for the linear
regression
# model object
outputZTableHTMLFile <-
"R_output_linear_regression_significant_variables.html";
# Call the function to pretty-print the linear regression model info using
# the ztable and stargazer packages
myZtableObject <-
  prettyPrintHTMLRulesResultsUsingZtableStargazer(
  linearRegressionModelTidyBroomDataFrame,
linearRegressionModel,
  outputZTableHTMLFile);
# Predict values using the linear regression model and store them in an
# additional column in the test data frame
testDataFrame$predicted_value_total_medicare_prof_srv_revenue =
  predict(linearRegressionModel, newdata=testDataFrame);
# Predict values using the linear regression model and store them in an
# additional column in the training data frame
trainingDataFrame$predicted_value_total_medicare_prof_srv_revenue
=
  predict(linearRegressionModel, newdata=trainingDataFrame);
print("Minimum value for predicted total_medicare_prof_srv_revenue
in
  training data set:\n");
min(trainingDataFrame$predicted_value_total_medicare_prof_srv_rev
enue);
```

```
print("Maximum value for predicted total_medicare_prof_srv_revenue
in
  training data set:\n");
max(trainingDataFrame$predicted_value_total_medicare_prof_srv_rev
enue);
print("Minimum value for predicted total_medicare_prof_srv_revenue
in
  test data set:\n");
min(testDataFrame$predicted_value_total_medicare_prof_srv_revenue
);
print("Maximum value for predicted total_medicare_prof_srv_revenue
in
  test data set:\n");
max(testDataFrame$predicted_value_total_medicare_prof_srv_revenu
e);
# Quick plot using the ggplot2 library: Plotting physician revenue as a
# function of predicted physician revenue for the test data
ggplotTestPredAgainstActualsQuick <-
ggplot2::qplot(data=testDataFrame,
  x=predicted_value_total_medicare_prof_srv_revenue,
  y=total_medicare_prof_srv_revenue,
  geom=c("point", "smooth")) +
  labs(list(title = "Predicted versus Actual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Actual total Medicare Professional Fees Revenue"));

# Save the plot as a JPG file
ggplot2::ggsave(filename="TestPredAgainstActualsQuickPlot.jpg",
  plot=ggplotTestPredAgainstActualsQuick, dpi=1200, device="jpeg");
# Plotting physician revenue as a function of predicted physician
revenue for
# the test data
ggplotTestLMPredictionsAgainstActualsObject <-
  ggplot2::ggplot(data = testDataFrame,
  aes(x=predicted_value_total_medicare_prof_srv_revenue,
  y=total_medicare_prof_srv_revenue)) +
  geom_point(alpha=0.2,color="black") +
  geom_smooth(aes(x=predicted_value_total_medicare_prof_srv_reven
ue,
  y=total_medicare_prof_srv_revenue), color = "black") +
  geom_line(aes(x=total_medicare_prof_srv_revenue,
  y=total_medicare_prof_srv_revenue), color = "blue", linetype = 2) +
  scale_x_continuous(limits = c(-0, 5000000)) +
  scale_y_continuous(limits = c(-0, 5000000)) +
  labs(list(title = "Predicted versus Actual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
```

```
  y = "Actual total Medicare Professional Fees Revenue"));
# Save the plot as a JPG file
ggplot2::ggsave(filename="TestLMPredictionsAgainstActuals.jpg",
  plot=ggplotTestLMPredictionsAgainstActualsObject, dpi=1200,
device="jpeg");
#-------------------------------------------------------------------------------
-----------------------------------------------------
# Plotting residuals income as a function of predicted income for the
test data
ggplotTestLMPredictionsAgainstResidualsObject <-
  ggplot2::ggplot(data=testDataFrame,
  aes(x=predicted_value_total_medicare_prof_srv_revenue,
  y=predicted_value_total_medicare_prof_srv_revenue -
  total_medicare_prof_srv_revenue)) +
  geom_point(alpha = 0.2,color = "black") +
  geom_smooth(aes(x =
predicted_value_total_medicare_prof_srv_revenue,
  y = predicted_value_total_medicare_prof_srv_revenue -
  total_medicare_prof_srv_revenue),
  color="black") +
  scale_x_continuous(limits = c(-0, 5000000)) +
  labs(list(title = "Predicted versus Residual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Residuals"));
# Save the plot as a JPG file
ggplot2::ggsave(filename="TestLMPredictionsAgainstResiduals.jpg",
  plot=ggplotTestLMPredictionsAgainstResidualsObject,
  dpi=1200, device="jpeg");
# R-squared value for the Test Data
rSqredValueForTestData <-
  rsquared_value(testDataFrame$total_medicare_prof_srv_revenue,
  testDataFrame$predicted_value_total_medicare_prof_srv_revenue
  );
print("R-squared value for the Test Data:\n");
print(rSqredValueForTestData);
# R-squared value for the Training Data
rSqredValueForTrainingData <-
  rsquared_value(trainingDataFrame$total_medicare_prof_srv_revenue,
  trainingDataFrame$predicted_value_total_medicare_prof_srv_revenu
e
  );
print("R-squared value for the Training Data:\n");
print(rSqredValueForTrainingData);
# Save the R data-frame+data-table object to a CSV file
data.table::fwrite(testDataFrame,
  file =
```

```
'LinearModel_Predictions_Medicare_Prov_Util_2014_testDataFrame.c
sv',
  append=FALSE,
  quote="auto",
  col.names=TRUE,
  row.names=FALSE,
  na='',
  nThread = getDTthreads() );
```

Code listing 17.2 starts, like 17.1, by reading in the summarized-by-provider dataset and running a multiple linear regression on it. However, there are some critical differences from listing 17.1:

- The random_val_between_one_and_three variable in the dataset, which was created in code listing 12.1, is used to create a training and a test dataset. All rows where the random_val_between_one_and_three variable was assigned a value of 1 or 2 go into the training dataset. The rows where the random_val_between_one_and_three variable was assigned a value of 3 go into the test dataset.

- The multiple linear regression model is run ***only*** on the training dataset. Also, only the *significant* explanatory variables from the previously-run regression model (run in code listing 17.1) are included in this model-run in 17.2.

Using the regression model object held in memory in the variable linearRegressionModel, predictions are made (using the *predict()* function) for the values of "total Medicare professional services revenue for the provider" in both the training and test datasets.

```
# Predict values using the linear regression model and store them in an
# additional column in the test data frame
testDataFrame$predicted_value_total_medicare_prof_srv_revenue =
  predict(linearRegressionModel, newdata=testDataFrame);
# Predict values using the linear regression model and store them in an
# additional column in the training data frame
trainingDataFrame$predicted_value_total_medicare_prof_srv_revenue
=
  predict(linearRegressionModel, newdata=trainingDataFrame);
```

Two different kinds of plots are created using the test dataset:

- a plot of the predicted values for total-Medicare-professional-services-revenue against the actual values

- a plot of the predicted values for total-Medicare-professional-services-revenue against the *residual* (predicted minus actual) values

All these plots are created using the ggplot2 package. However, the first plot of the predicted against the actual values is created in *two* different ways for the purposes of illustration:

- using the *qplot()* (quick-plot: http://docs.ggplot2.org/dev/vignettes/qplot.html) function from the ggplot2 package, which allows a user to create simple plots without being burdened by all the options and parameters that a full-featured ggplot2 plot can contain

- using the full-featured *ggplot()* function from the ggplot2 package

The *qplot()* function-use is self-explanatory. So, we will concentrate on the code for the *ggplot()* function used to create the predicted-versus-actuals plot:

```
# Plotting physician revenue as a function of predicted physician
revenue for
# the test data
ggplotTestLMPredictionsAgainstActualsObject <-
  ggplot2::ggplot(data = testDataFrame,
  aes(x=predicted_value_total_medicare_prof_srv_revenue,
  y=total_medicare_prof_srv_revenue)) +
  geom_point(alpha=0.2,color="black") +
  geom_smooth(aes(x=predicted_value_total_medicare_prof_srv_reven
ue,
  y=total_medicare_prof_srv_revenue), color = "black") +
  geom_line(aes(x=total_medicare_prof_srv_revenue,
  y=total_medicare_prof_srv_revenue), color = "blue", linetype = 2) +
  scale_x_continuous(limits = c(-0, 5000000)) +
  scale_y_continuous(limits = c(-0, 5000000)) +
  labs(list(title = "Predicted versus Actual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Actual total Medicare Professional Fees Revenue"));
# Save the plot as a JPG file
ggplot2::ggsave(filename="TestLMPredictionsAgainstActuals.jpg",
  plot=ggplotTestLMPredictionsAgainstActualsObject, dpi=1200,
device="jpeg");
```

The data parameter given to the *ggplot()* function is the test dataset. The aesthetic parameters (aes) set are the variables to be plotted on the x and y axes.

Next, the *geom_point()* function is called in the chain (remember: the + sign is the function-chaining operator in ggplot2) to set the color of the plotted points to black and the transparency to 80% (alpha=0.2). Transparent plotted data-points are very useful to visualize large-dataset plots. The *qplot()* function used earlier did not use the alpha parameter for plotted-point-transparency.

The *geom_smooth()* function is then called to create a best-fit-line or smoothing curve in black.

This is then followed by the *geom_line()* function used to create what can be called a ***faultless-prediction line*** in blue. This represents the case where the predictions are 100% accurate and all the predicted values are equal to the actual values. The closer the best-fit-line is to the faultless-prediction line, the better the predictive model is performing. The

linetype for this line is set to the value 2 to denote a dashed-line.

The *scale_x_continuous()* and *scale_y_continuous()* functions are then used to set up the x axis and y axis scales respectively. The *labs()* function is then used to set up the title for the plot, and the x and y axis labels.

The plot is then saved as a JPEG image file with 1200 dots per inch resolution.

The plotting of the predicted values against the residuals is also done similarly and another image is saved.

R-squared values are calculated for the predictions on both the training and the test data. R-squared value is one of indicators of the quality of a predictive model. R-squared values greater than 0.7 usually indicate a model with good predictive abilities when applied to new data.

In our case, the regression model's R-squared value for our training dataset is **0.6040586** while the R-squared value for the test dataset is **0.6597319**. This is a bit of a weirdly-surprising result that we are getting – usually, one expects the R-squared value for the training data to be higher than that for the test data. However, others seem to have seen the same situation: http://stats.stackexchange.com/questions/86314/higher-r-squared-value-on-test-data-than-training-data. The explanation for why this happens sometimes is best supplied by a statistician.

The reason the training dataset is named so is because it actually *trains* the model to be able to predict. We also save the predictions data from the program as a CSV file.

If you see weird errors like “provider_type has a new level” when running the prediction program, it might mean that your randomly chosen training dataset did not reflect all the factor levels for a variable that were found in the test data. In this case, you might have to run code listing 12.1 again to this time (*hopefully ☺)* create both training and test datasets with *all* of the levels for all of the factors represented.

Figure 17.2 (Linear Model: Predicted against actual values for Medicare revenue)

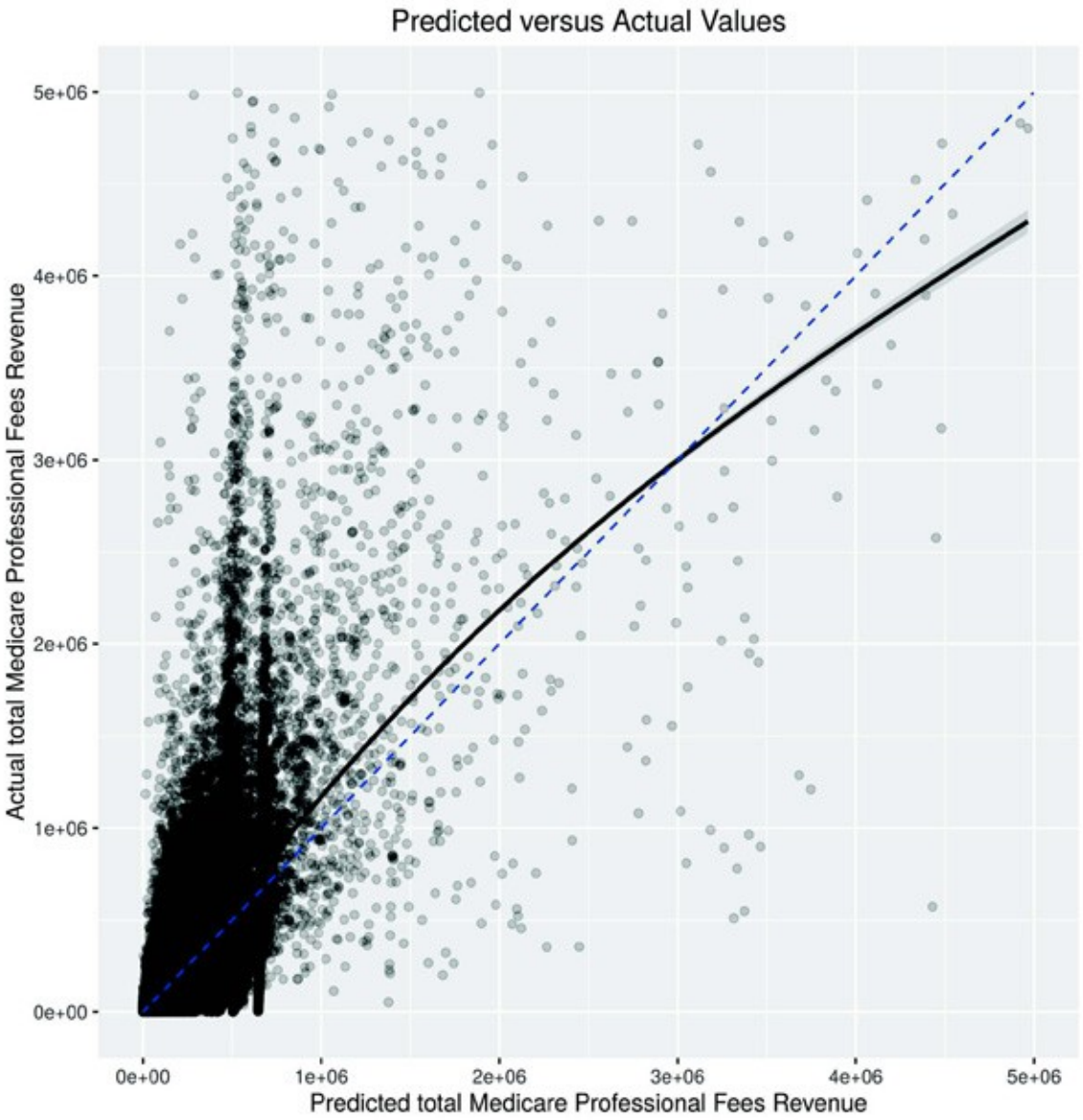
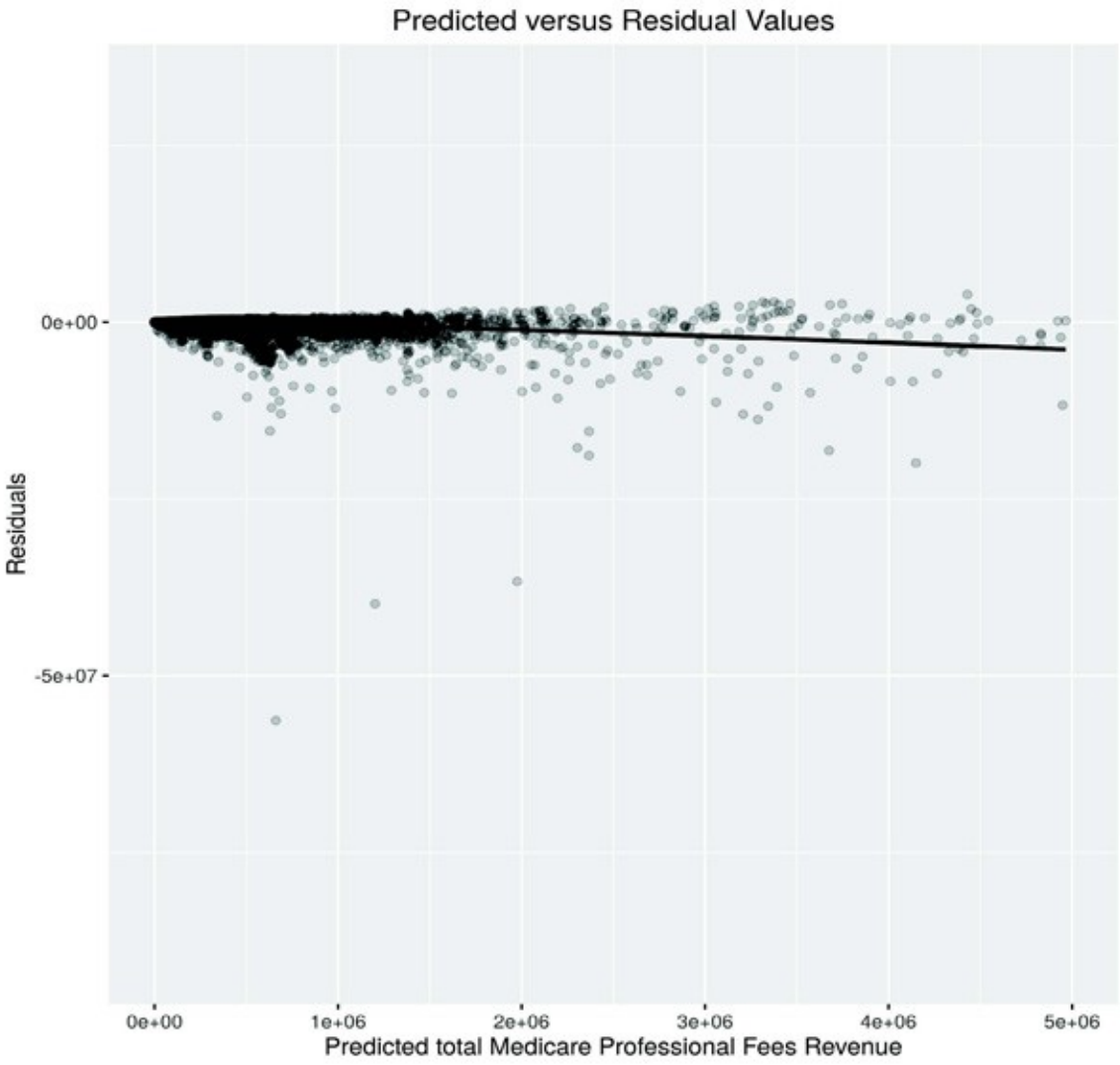


Figure 17.3 (Linear Model: Predicted against residual values for Medicare revenue)



Statistical models (like those utilizing linear regression or logistic regression) are one category of predictive-models encountered. R has a plethora of packages and functions that can run statistical procedures and make predictions.

In the next chapter, we will take a look at machine-learning packages in R that can also be used for predictive models.

Exercise 17.1: For the Reader

Modify the code in listing 17.1 and run a multiple linear regression model where the dependent variable is total_line_srvc_cnt (the total number of service lines that the provider has billed for in calendar year 2014).

Chapter 18: Machine Learning & Predictive Models

A hugely popular part of data analytics, data mining, and data science is the use of machine learning (and statistical learning) algorithms and tools to examine, detail out, explore, analyze, and find patterns in data. R has a lot of packages in this machine and statistical learning (https://cran.r-project.org/web/views/MachineLearning.html) space.

However, the R package we will be utilizing in this chapter, h2o (https://cran.r-project.org/web/packages/h2o/index.html), is actually a wrapper around the Java-based, open source H2O (http://www.h2o.ai/) machine learning and predictive analytics library.

The reason I chose to use the h2o R-package instead of any of the other native-R machine learning packages is because:

- The Java H2O library is extremely efficient and able to accommodate big-data size datasets with no degradation in performance. As an example, the *h2o.randomForest()* function I used in my code was easily able to accommodate our Medicare dataset. On the other hand, the CRAN randomForest (https://cran.r-project.org/web/packages/randomForest/index.html) R package was not able to complete the building of the random forest model using our dataset on my notebook computer.

- The sparklyr (https://cran.r-project.org/web/packages/sparklyr/index.html) package, which is an R interface into the Spark big-data platform, has a new extension-package called rsparkling (https://cran.r-project.org/web/packages/rsparkling/index.html) that offers an R interface to H2O.

Thus, there are exciting times ahead in terms of the potential to run R on top of big-data platforms like Spark, Hadoop, and H2O.

Since H2O is written in Java, we need the Java runtime to be installed on the machine before we run our code. The R code in this case merely provides an interface to the H2O engine that does the heavy lifting of modeling and predictive analytics.

We are going to use two different machine learning algorithms available within H2O to carry out the same prediction that we saw in the previous chapter: predicting the "total Medicare professional services revenue for the provider". The two H2O algorithms we will use are Gradient Boosted Machines (https://en.wikipedia.org/wiki/Gradient_boosting) and Random Forests (https://en.wikipedia.org/wiki/Random_forest).

An introductory tutorial on how to use the R h2o package is available: http://h2o-release.s3.amazonaws.com/h2o/rel-lambert/5/docs-website/Ruser/rtutorial.html

The programs in listings 18.1 and 18.2 use these concepts to create the predictive models and make the predictions. One thing to note is that Rscript is not able to run the code that uses the h2o package – errors are seen when running the h2o package code with Rscript. So, we run the code in this chapter in the command shell using the R executable (which

requires a slightly different syntax than Rscript). Also, on Microsoft Windows, R cannot be run in powershell.exe because the < character (which is used in the command syntax to run the program with R) is a reserved one in powershell.exe. On Windows, the programs from this chapter should be run using R in cmd.exe.

Code Listing 18.1

(Gradient Boosted Machine predictive model in H2O)

To run this (all on one line):

```
R --no-save --no-restore --args
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds <
18.1_Medicare_Provider_Util_Payment_h2o_GBM_with_prediction.R
> outputH2O_GBM.txt 2> errorsH2O_GBM.txt
# Copyright (C) 2017 Sivakumaran Raman
library("dplyr");
library("data.table");
library("dtplyr");
library("ggplot2");
library("broom");
library("ztable");
library("h2o");
# library("tidyr");
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Function to calculate the R-Squared value
rsquared_value <- function(y,f) {
  return(1 - sum((y - f)^2)/sum((y - mean(y))^2));
}
#-------------------------------------------------------------------------------
# Send textual output to file
sink("GradientBoostedMachine_H2O_regression_output.txt");
# Set option to print the stack trace at the time of any error and then
quit.
options(error= function () {
  traceback(2);
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided on
# the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
```

```
  command line argument");
}
# Read in the R data-object from disk by calling the function readRDS
summarizedPhysicianMedicareDataTable <- readRDS(myArgs[[1]]);
# Initialize the h2o engine
h2o.init(
  # -1: asks the program to use all available threads
  nthreads = -1,
  # Specifing the memory size for the h2o cloud
  max_mem_size = "2G"
);
# Clean up everything - just in case the cluster was already running
h2o.removeAll()
# Convert the provider_type variable to a character from a factor
# summarizedPhysicianMedicareDataTable <-
#  summarizedPhysicianMedicareDataTable %>%
#  dplyr::mutate(provider_type = as.character(provider_type)) %>%
#  dplyr::filter(!is.na(provider_type));
# Create a training data set
trainingDataFrame <- summarizedPhysicianMedicareDataTable %>%
  dplyr::filter(random_val_between_one_and_three == 1 ||
  random_val_between_one_and_three == 2);
# Convert the R training data frame to an H2O data frame
trainingDataFrameH2O <- as.h2o(trainingDataFrame,
  destination_frame = "trainH2O");
# Create a test data set
testDataFrame <- summarizedPhysicianMedicareDataTable %>%
  dplyr::filter(random_val_between_one_and_three == 3);
# Convert the R test data frame to an H2O data frame
testDataFrameH2O <- as.h2o(testDataFrame, destination_frame =
"testH2O");
# Assign the first result to the R variable train and the H2O name
train.h2o
train <- h2o.assign(trainingDataFrameH2O, "train.h2o");
# R variable test and the H2O name test.h2o
test <- h2o.assign(testDataFrameH2O, "test.h2o")
# Remove objects and release memory
rm(trainingDataFrame, trainingDataFrameH2O, testDataFrameH2O);
gc();
# Predictor and response variables
predictor_variables = c("nppes_provider_gender",
  # "nppes_entity_code",
  "medicare_participation_indicator",
  "total_line_srvc_cnt",
  "total_bene_unique_cnt",
  "provider_type",
```

```
  "total_bene_day_srvc_cnt");
response_variable = "total_medicare_prof_srv_revenue";
# Run the first predictive model
gbm1 <- h2o.gbm(
  # THE H2O GRADIENT BOOSTED MACHINE FUNCTION

  # The H2O frame for training the machine
  training_frame = train,

  # The H2O frame for validation
  validation_frame = test,

  # The predictor columns/variables, by column index
  x=predictor_variables,

  # The target index or dependent variable (what we are predicting)
  y=response_variable,

  distribution = "AUTO",

  # Naming the model in H2O - this is not required, but helps use Flow.
  model_id = "gbm_Medicare_v1",

  # Use a maximum of 50 trees to create the Gradient Boosted Machine
model
  ntrees = 50,

  # Increase the learning rate
  learn_rate = 0.3,


  max_depth = 10,

  # Use a random 70% of the rows to fit each tree
  sample_rate = 0.7,

  # Use a random 70% of the columns to fit each tree
  col_sample_rate = 0.7,

  stopping_tolerance = 0.01,

  # Stop fitting new trees when the 2-tree average is within 0.001
(default) of
  # the prior two 2-tree averages. Can be thought of as a convergence
setting.
  stopping_rounds = 2,
```

```
  # Predict against training and validation for each tree. The default will
skip
  # several trees.
  score_each_iteration = T,

  # Set the random seed so that this can be reproduced.
  seed = 2000000
);
# View information about the Gradient Boosted Machine model. The
Keys to examine
# are validation performance and variable importance
summary(gbm1);
# This is a more direct way to access the validation metrics.
Performance
# metrics depend on the type of model being built. With a multinomial
# classification, we will primarily look at the confusion matrix,
# and overall accuracy via hit_ratio @ k=1.
gbm1@model$validation_metrics;
# Make predictions on the test data
gbmpredictions <- h2o.predict(gbm1, newdata=test);
# Get Gradient Boosted Machine Regression Model performance for
the
# predictions on the test data
gbmPerf <- h2o.performance(gbm1, newdata=test);
print("Mean squared error for the Gradient Boosted Machine
Performance
  Model object on the test data\n");
# Get mean squared error for the Gradient Boosted Machine
# Performance Model object on the test data
h2o.mse(gbmPerf);
# Convert the H2Oframe to an R dataframe
testPredictionsDF <- as.data.frame(gbmpredictions);
# Add the predicted values to the test data frame
testDataFrame$predicted_value_total_medicare_prof_srv_revenue =
  testPredictionsDF$predict;
# R-squared value for the Test Data
rSquredValueForTestData <-
  rsquared_value(testDataFrame$total_medicare_prof_srv_revenue,
  testDataFrame$predicted_value_total_medicare_prof_srv_revenue
  );
print("R-squared value for the Test Data:\n");
print(rSquredValueForTestData);
sink();
# Plotting physician revenue as a function of predicted physician
revenue
```

```
# for the test data
ggplotTestPredictionsAgainstActualsObject <-
  ggplot2::ggplot(data = testDataFrame,
  aes(x = predicted_value_total_medicare_prof_srv_revenue,
  y = total_medicare_prof_srv_revenue)) +
  geom_point(alpha = 0.2, color = "black") +
  geom_smooth(aes(x =
predicted_value_total_medicare_prof_srv_revenue,
  y = total_medicare_prof_srv_revenue), color = "black") +
  geom_line(aes(x = total_medicare_prof_srv_revenue,
  y = total_medicare_prof_srv_revenue), color = "blue", linetype = 2) +
  scale_x_continuous(limits = c(-0, 5e+06)) +
  scale_y_continuous(limits = c(-0, 5e+06)) +
  labs(list(title = "Predicted versus Actual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Actual total Medicare Professional Fees Revenue"))

# Save the plot as a JPG file
ggplot2::ggsave(filename = "TestGBMPredictionsAgainstActuals.jpg",
  plot = ggplotTestPredictionsAgainstActualsObject,
  dpi = 1200, device = "jpeg")
# Plotting residuals income as a function of predicted income for the
test data
ggplotTestPredictionsAgainstResidualsObject <-
  ggplot2::ggplot(data = testDataFrame,
  aes(x = predicted_value_total_medicare_prof_srv_revenue,
  y = predicted_value_total_medicare_prof_srv_revenue -
  total_medicare_prof_srv_revenue)) +
  geom_point(alpha = 0.2, color = "black") +
  geom_smooth(aes(x =
predicted_value_total_medicare_prof_srv_revenue,
  y = predicted_value_total_medicare_prof_srv_revenue -
  total_medicare_prof_srv_revenue), color = "black") +
  scale_x_continuous(limits = c(-0, 5e+06)) +
  labs(list(title = "Predicted versus Residual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Residuals"))
# Save the plot as a JPG file
ggplot2::ggsave(filename =
"TestGBMPredictionsAgainstResiduals.jpg",
  plot = ggplotTestPredictionsAgainstResidualsObject,
  dpi = 1200, device = "jpeg")
# Save the R data-frame+data-table object to a CSV file
data.table::fwrite(testDataFrame,
  file =
"GBM_Predictions_Medicare_Prov_Util_2014_testDataFrame.csv",
```

```
  append = FALSE, quote = "auto", col.names = TRUE, row.names =
FALSE, na = "",
  nThread = getDTthreads())
```

The program starts by reading in the summarized-by-provider dataset RDS file from disk into a data frame.

The Java H2O engine is then initialized from within R, setting the options for the number of threads to use, and the maximum memory-size of the H2O cloud that can be created:

```
# Initialize the h2o engine
h2o.init(
  # -1: asks the program to use all available threads
  nthreads = -1,
  # Specifing the memory size for the h2o cloud
  max_mem_size = "2G"
);
```

A clean-up statement is issued in order to remove any previously running Java H2O clusters:

```
# Clean up everything - just in case the cluster was already running
h2o.removeAll()
```

Training and test datasets are created as in code listing 17.2. However, the additional step required when using H2O is to convert the R training and test data frames to H2O data frames. To convert the R training data frame to an H2O data frame:

```
# Convert the R training data frame to an H2O data frame
trainingDataFrameH2O <- as.h2o(trainingDataFrame,
  destination_frame = "trainH2O");
```

And the same for the test data frame:

```
# Convert the R test data frame to an H2O data frame
testDataFrameH2O <- as.h2o(testDataFrame, destination_frame =
"testH2O");
```

H2O names are then assigned for these:

```
# Assign the first result to the R variable train and the H2O name
train.h2o
train <- h2o.assign(trainingDataFrameH2O, "train.h2o");
# R variable test and the H2O name test.h2o
test <- h2o.assign(testDataFrameH2O, "test.h2o")
```

The predictor (explanatory) variables and the response (dependent) variable are then set up:

```
# Predictor and response variables
```

```
predictor_variables = c("nppes_provider_gender",
  # "nppes_entity_code",
  "medicare_participation_indicator",
  "total_line_srvc_cnt",
  "total_bene_unique_cnt",
  "provider_type",
  "total_bene_day_srvc_cnt");
response_variable = "total_medicare_prof_srv_revenue";
```

The H2O Gradient Boosted Machine (GBM) predictive model is then set-up and trained using the training dataset. A number of options are provided for the h2o.gbm() function:

```
# Run the first predictive model
gbm1 <- h2o.gbm(
  # THE H2O GRADIENT BOOSTED MACHINE FUNCTION

  # The H2O frame for training the machine
  training_frame = train,

  # The H2O frame for validation
  validation_frame = test,

  # The predictor columns/variables, by column index
  x=predictor_variables,

  # The target index or dependent variable (what we are predicting)
  y=response_variable,

  distribution = "AUTO",

  # Naming the model in H2O - this is not required, but helps use Flow.
  model_id = "gbm_Medicare_v1",

  # Use a maximum of 50 trees to create the Gradient Boosted Machine
model
  ntrees = 50,

  # Increase the learning rate
  learn_rate = 0.3,


  max_depth = 10,

  # Use a random 70% of the rows to fit each tree
  sample_rate = 0.7,

  # Use a random 70% of the columns to fit each tree
```

```
  col_sample_rate = 0.7,

  stopping_tolerance = 0.01,

  # Stop fitting new trees when the 2-tree average is within 0.001
(default) of
  # the prior two 2-tree averages. Can be thought of as a convergence
setting.
  stopping_rounds = 2,

  # Predict against training and validation for each tree. The default will
skip
  # several trees.
  score_each_iteration = T,

  # Set the random seed so that this can be reproduced.
  seed = 2000000
);
```

The summary from the GBM model run is printed out. Validation metrics are also printed.

```
# View information about the Gradient Boosted Machine model.
# The Keys to examine are validation performance and variable
importance
summary(gbm1);
# This is a more direct way to access the validation metrics.
Performance
# metrics depend on the type of model being built. With a multinomial
# classification, we will primarily look at the confusion matrix,
# and overall accuracy via hit_ratio @ k=1.
gbm1@model$validation_metrics;
```

Predictions are then made on the test data using the trained GBM model and the performance metrics for the predictions are collected:

```
# Make predictions on the test data
gbmpredictions <- h2o.predict(gbm1, newdata=test);
# Get Gradient Boosted Machine Regression Model performance for
the
# predictions on the test data
gbmPerf <- h2o.performance(gbm1, newdata=test);
```

The mean square error value is printed for the predictions made on the test data.

The H2O data frame with the predictions made on the test data is then converted to an R data frame and the predict variable added as a column to the original test R dataset.

```
# Convert the H2Oframe to an R dataframe
testPredictionsDF <- as.data.frame(gbmpredictions);
# Add the predicted values to the test data frame
testDataFrame$predicted_value_total_medicare_prof_srv_revenue =
  testPredictionsDF$predict;
```

The R-squared value for the predictions made on the test data is calculated like was done for the multiple linear regression model in code listing 17.2. Both the *predictions against actuals* and the *predictions against residuals* plots are created using the ggplot2 package, like was done in code listing 17.2. The test data with the predictions column included is saved to a CSV file.

The H2O GBM model does significantly better at predicting "Total Medicare Professional Services Revenue for a Provider" than the multiple linear regression model seen earlier. The R-squared value for the GBM model predictions on the test data comes out to be **0.7507081**. The parameters for running the GBM model can be tweaked to improve its predictive capabilities. This is left as an exercise for the reader.

Figure 18.1 (GBM: Predicted against actual values for Medicare revenue)

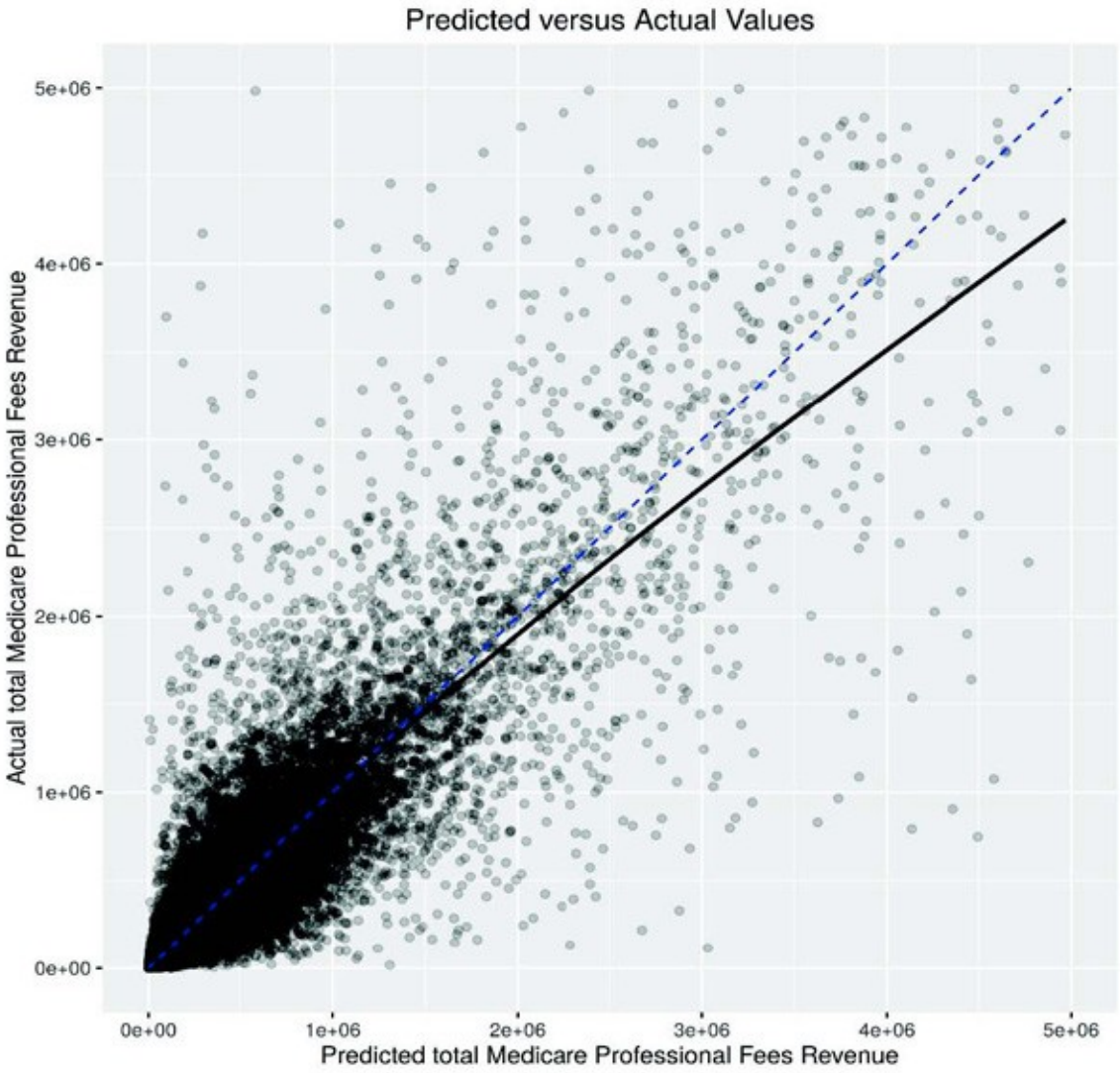
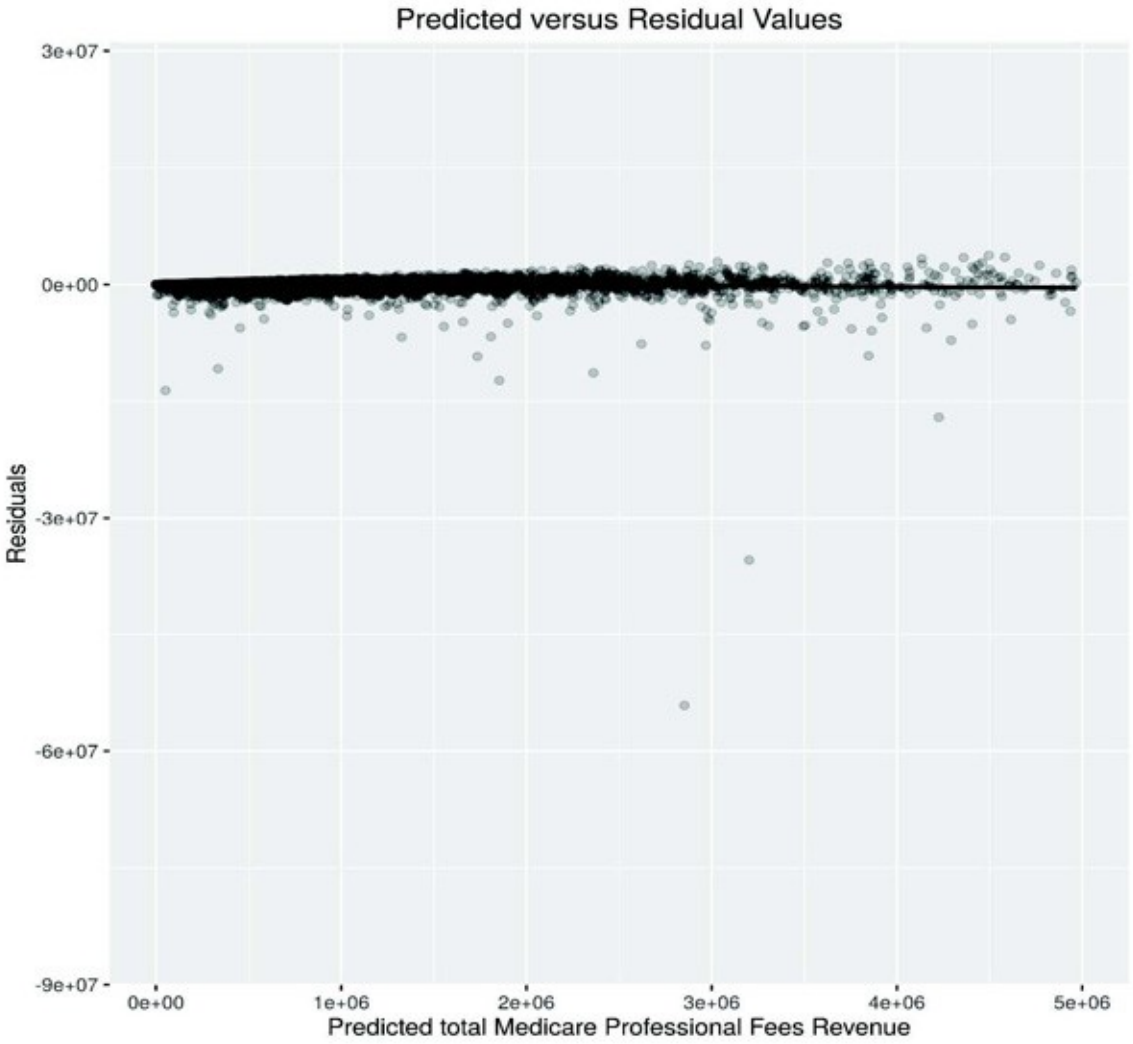


Figure 18.2 (GBM: Predicted against residual values for Medicare revenue)



Random Forest model to make the same predictions that we made in code listing 18.1 using the Gradient Boosted Machine.

Code Listing 18.2

(Random Forests predictive model in H2O)

To run this (all on one line):

```
R --no-save --no-restore --args
../data/Medicare_Provider_Util_Payment_PUF_CY2014_SUMMARIZ
ED.rds <
18.2_Medicare_Provider_Util_Payment_h2o_randomForest_with_pred
iction.R > outputH2O_RandomForest.txt 2>
errorsH2O_RandomForest.txt
# Copyright (C) 2017 Sivakumaran Raman
library("dplyr");
library("data.table");
```

```
library("dtplyr");
library("ggplot2");
library("broom");
library("ztable");
library("h2o");
# library("tidyr");
########## ALL FUNCTIONS DEFINED
-----------------------------------------------
# Function to calculate the R-Squared value
rsquared_value <- function(y,f) {
  return(1 - sum((y - f)^2)/sum((y - mean(y))^2));
}
#-------------------------------------------------------------------------------
# Send textual output to file
sink("randomForest_H2O_regression_output.txt");
# Set option to print the stack trace at the time of any error and then
quit.
options(error= function () {
  traceback(2);
  stop("Error: stack trace printed above");
  }
);
# Read in the command line arguments into a character vector
myArgs <- commandArgs(trailingOnly=TRUE);
# Exit with error message if RDS data-object file to read is not
provided on the command line
if(length(myArgs) != 1) {
  stop("Error: please provide the name of the RDS file to read as the
  command line argument");
}
# Read in the R data-object from disk by calling the function readRDS
summarizedPhysicianMedicareDataTable <- readRDS(myArgs[[1]]);
# Initialize the h2o engine
h2o.init(
  # -1 means to use all available threads
  nthreads = -1,

  # The memory size for the H2O cloud
  max_mem_size = "2G");
# Fresh start - just in case the cluster was already running
h2o.removeAll()
# Convert the provider_type variable to a character from a factor
# summarizedPhysicianMedicareDataTable <-
#  summarizedPhysicianMedicareDataTable %>%
#  dplyr::mutate(provider_type = as.character(provider_type)) %>%
#  dplyr::filter(!is.na(provider_type));
```

```
# Create a training data set
trainingDataFrame <- summarizedPhysicianMedicareDataTable %>%
  dplyr::filter(random_val_between_one_and_three == 1 ||
  random_val_between_one_and_three == 2);
# Convert to an H2O data frame
trainingDataFrameH2O <- as.h2o(trainingDataFrame,
  destination_frame = "trainH2O");
# Create a test data set
testDataFrame <- summarizedPhysicianMedicareDataTable %>%
  dplyr::filter(random_val_between_one_and_three == 3);
# Convert to an H2O data frame
testDataFrameH2O <- as.h2o(testDataFrame, destination_frame =
"testH2O");
# Give the first result the R variable name train and the H2O name
train.h2o
train <- h2o.assign(trainingDataFrameH2O, "train.h2o");
# Give the test result the R variable name test and the H2O name
test.h2o
test <- h2o.assign(testDataFrameH2O, "test.h2o")
# Remove objects and release memory
rm(trainingDataFrame, trainingDataFrameH2O, testDataFrameH2O);
gc();
# Predictor and response variables
predictor_variables = c("nppes_provider_gender",
  # "nppes_entity_code",
  "medicare_participation_indicator",
  "total_line_srvc_cnt",
  "total_bene_unique_cnt",
  "provider_type",
  "total_bene_day_srvc_cnt");
response_variable = "total_medicare_prof_srv_revenue";
## run our first predictive model
rf1 <- h2o.randomForest(
  # THE H2O RANDOM FOREST FUNCTION

  # The H2O frame for training
  training_frame = train,

  # the H2O frame for validation
  validation_frame = test,

  # The predictor columns/variables, by column index
  x = predictor_variables,

  ## The variable being predicted (target index)
  y=response_variable,
```

```
  # Give the model a name in H2O - not required, but helps use Flow.
  model_id = "rf_Medicare_v1",
  # Use a maximum of 200 trees to create the random forest model.
  # The default is 50. We have increased it because we will let
  # the early stopping criteria decide when the random forest is
sufficiently
  # accurate.
  ntrees = 200,

  # Increase depth, from the default of 20
  max_depth = 30,

  # Set mtries to number_of_predictors/2 instead of the default of
  # number_of_predictors/3 for regression
  mtries=as.integer(length(predictor_variables)/2),

  # Stop fitting new trees when the 2-tree average is within 0.001
(default) of
  # the prior two 2-tree averages. This can be thought of as a
convergence
  # setting.
  stopping_rounds = 2,

  # Predict against training and validation for each tree. Default will
skip
  # several trees.
  score_each_iteration = T,

  # Set the random seed so that this can be reproduced.
  seed = 3000000
);
# View information about the random forest model. The Keys to
examine are
# validation performance and variable importance
summary(rf1);
# This is a more direct way to access the validation metrics.
Performance
# metrics depend on the type of model being built. With a multinomial
# classification, we will primarily look at the confusion matrix,
# and overall accuracy via hit_ratio @ k=1.
rf1@model$validation_metrics;
# Make predictions on the test data
rfpredictions <- h2o.predict(rf1, newdata=test);
# Get Random Forest Regression Model performance for the
predictions on
```

```
# the test data
rfPerf <- h2o.performance(rf1, newdata=test);
print("Mean squared error for the Random Forest Performance Model
object on
  the test data\n");
# Get mean squared error for the Random Forest Performance Model
object
# on the test data
h2o.mse(rfPerf);
# Convert the H2Oframe to an R dataframe
testPredictionsDF <- as.data.frame(rfpredictions);
# Add the predicted values to the test data frame
testDataFrame$predicted_value_total_medicare_prof_srv_revenue =
  testPredictionsDF$predict;
# R-squared value for the Test Data
rSquredValueForTestData <-
  rsquared_value(testDataFrame$total_medicare_prof_srv_revenue,
  testDataFrame$predicted_value_total_medicare_prof_srv_revenue
  );
print("R-squared value for the Test Data:\n");
print(rSquredValueForTestData);
sink();
# Plotting physician revenue as a function of predicted physician
revenue
# for the test data
ggplotTestPredictionsAgainstActualsObject <-
  ggplot2::ggplot(data = testDataFrame,
  aes(x = predicted_value_total_medicare_prof_srv_revenue,
  y = total_medicare_prof_srv_revenue)) +
  geom_point(alpha = 0.2, color = "black") +
  geom_smooth(aes(x =
predicted_value_total_medicare_prof_srv_revenue,
  y = total_medicare_prof_srv_revenue), color = "black") +
  geom_line(aes(x = total_medicare_prof_srv_revenue,
  y = total_medicare_prof_srv_revenue), color = "blue", linetype = 2) +
  scale_x_continuous(limits = c(-0, 5e+06)) +
  scale_y_continuous(limits = c(-0, 5e+06)) +
  labs(list(title = "Predicted versus Actual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Actual total Medicare Professional Fees Revenue"))

# Save the plot as a JPG file
ggplot2::ggsave(filename = "TestRFPredictionsAgainstActuals.jpg",
  plot = ggplotTestPredictionsAgainstActualsObject,
  dpi = 1200, device = "jpeg")
# Plotting residuals income as a function of predicted income for the
```

```
test data
ggplotTestPredictionsAgainstResidualsObject <-
  ggplot2::ggplot(data = testDataFrame,
  aes(x = predicted_value_total_medicare_prof_srv_revenue,
  y = predicted_value_total_medicare_prof_srv_revenue -
  total_medicare_prof_srv_revenue)) +
  geom_point(alpha = 0.2, color = "black") +
  geom_smooth(aes(x =
predicted_value_total_medicare_prof_srv_revenue,
  y = predicted_value_total_medicare_prof_srv_revenue -
  total_medicare_prof_srv_revenue), color = "black") +
  scale_x_continuous(limits = c(-0, 5e+06)) +
  labs(list(title = "Predicted versus Residual Values",
  x = "Predicted total Medicare Professional Fees Revenue",
  y = "Residuals"))
# Save the plot as a JPG file
ggplot2::ggsave(filename = "TestRFPredictionsAgainstResiduals.jpg",
  plot = ggplotTestPredictionsAgainstResidualsObject,
  dpi = 1200, device = "jpeg")
# Save the R data-frame+data-table object to a CSV file
data.table::fwrite(testDataFrame,
  file =
"RandomForest_Predictions_Medicare_Prov_Util_2014_testDataFram
e.csv",
  append = FALSE, quote = "auto", col.names = TRUE, row.names =
FALSE, na = "",
  nThread = getDTthreads())
```

Code listing 18.2 for the H2O Random Forest model is almost exactly the same as code listing 18.1 for the H2O Gradient Boosted Machine model - except for the slight differences in the options passed to the *h2o.randomForest()* function.

```
## run our first predictive model
rf1 <- h2o.randomForest(
  # THE H2O RANDOM FOREST FUNCTION

  # The H2O frame for training
  training_frame = train,

  # the H2O frame for validation
  validation_frame = test,

  # The predictor columns/variables, by column index
  x = predictor_variables,

  ## The variable being predicted (target index)
  y=response_variable,
```

```
  # Give the model a name in H2O - not required, but helps use Flow.
  model_id = "rf_Medicare_v1",
  # Use a maximum of 200 trees to create the random forest model.
  # The default is 50. We have increased it because we will let
  # the early stopping criteria decide when the random forest is
sufficiently
  # accurate.
  ntrees = 200,

  # Increase depth, from the default of 20
  max_depth = 30,

  # Set mtries to number_of_predictors/2 instead of the default of
  # number_of_predictors/3 for regression
  mtries=as.integer(length(predictor_variables)/2),

  # Stop fitting new trees when the 2-tree average is within 0.001
(default)
  # of the prior two 2-tree averages. This can be thought of as a
convergence
  # setting.
  stopping_rounds = 2,

  # Predict against training and validation for each tree. Default will
skip
  # several trees.
  score_each_iteration = T,

  # Set the random seed so that this can be reproduced.
  Seed = 3000000
);
```

The H2O Random Forest model does only slightly better at predicting “Total Medicare Professional Services Revenue for a Provider” than the H2O Gradient Boosted Machine model seen earlier. The R-squared value for the Random Forest model predictions on the test data comes out to be **0.7527851**. The parameters for running the Random Forest model can be tweaked to improve its predictive capabilities. This is left as an exercise for the reader.

Figure 18.3 (Random Forest: Predicted against actual values for Medicare revenue)

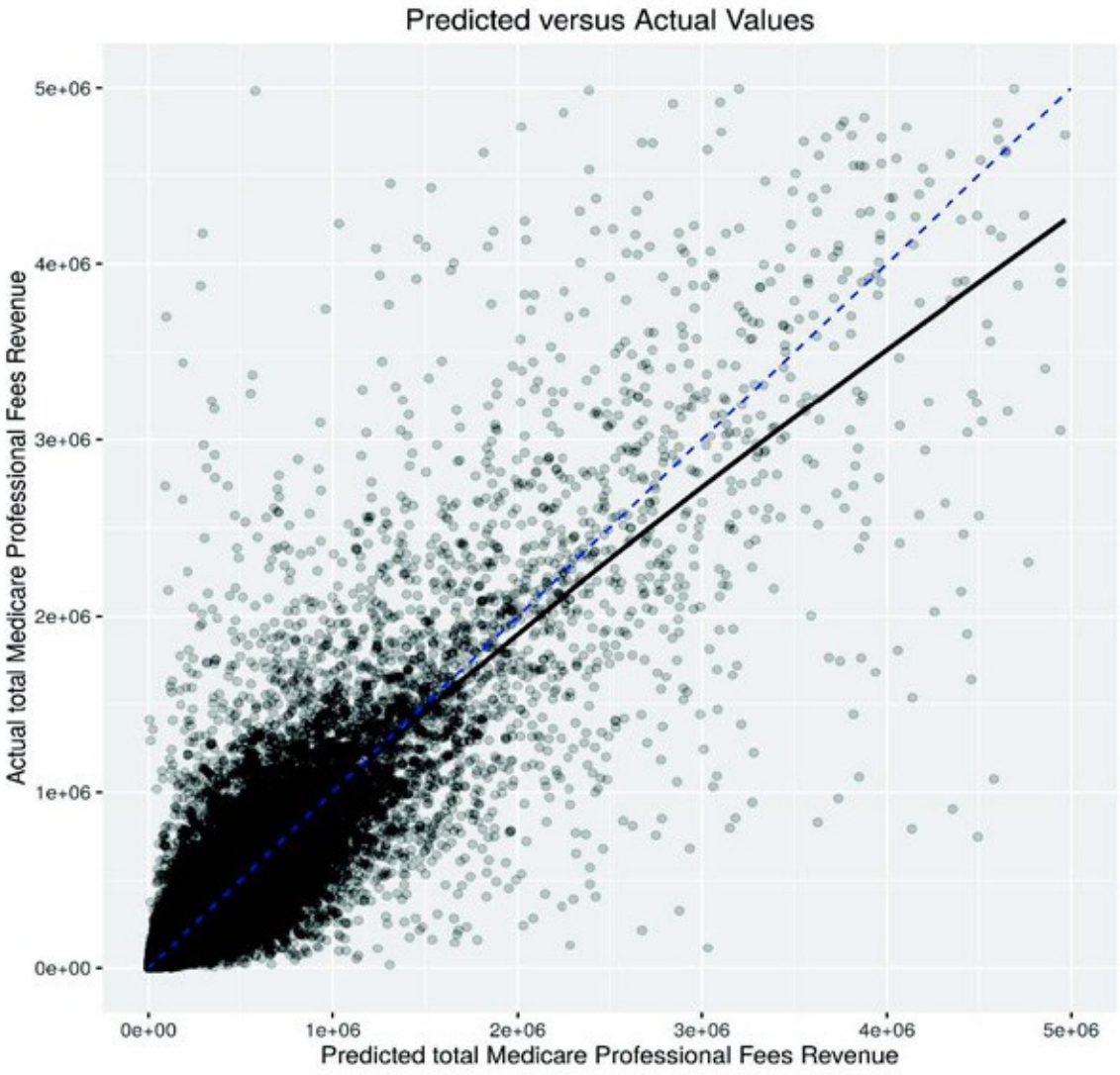
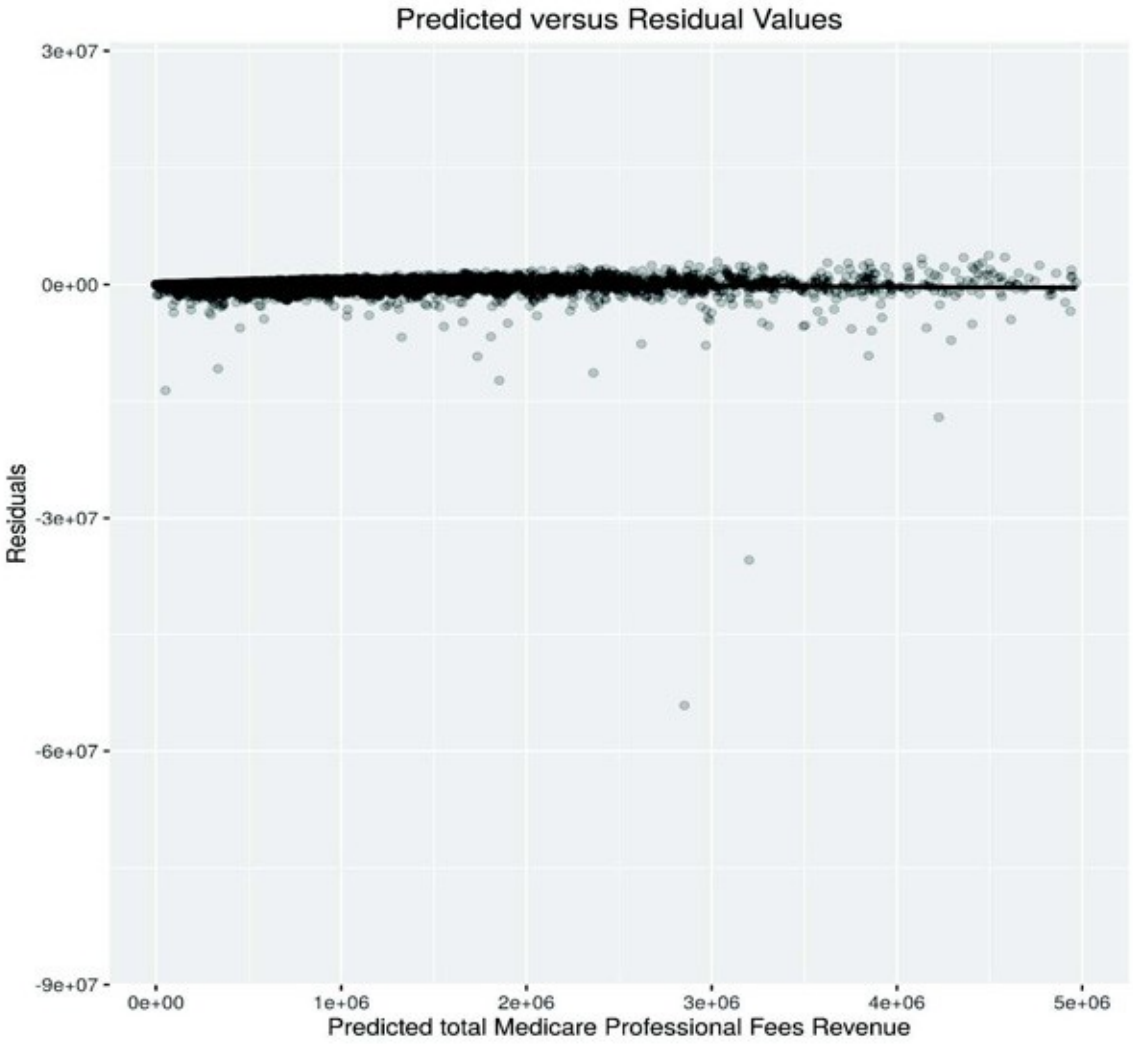


Figure 18.4 (Random Forest: Predicted against residual values for Medicare revenue)



This chapter gives the reader an introduction into the use of machine learning libraries and algorithms in R. There is a whole lot more to explore in this realm as the capabilities of R with regard to machine learning and big data analytics are getting better and more exciting every day.

Exercise 18.1: For the Reader

Use a Random Forest predictive model from the H2O package to predict the total_line_srvc_cnt (the total number of service lines that the provider has billed for in calendar year 2014). Refer to Reader Exercise 17.1 for the linking details.

Epilogue

The light-speed introduction to data analysis using R presented in this book should be able to get the reader started on a journey of self-exploration and self-improvement in order to understand and utilize the full set of capabilities of R. There are a whole lot of resources available on the Internet and through other sources for interested readers:

- CRAN (https://cran.r-project.org/)
- Google and other search engines to find answers and solutions
- Stack Overflow (http://stackoverflow.com/)
- Stat Methods (http://www.statmethods.net/)
- R Pubs (https://rpubs.com/)
- R Bloggers (https://www.r-bloggers.com/)
- Courses on coursera.org and other learning sites
- A plethora of free books on R available on the Internet

I hope I have whetted your appetite to acquire more knowledge about R and have a blast along the way, exploring and visualizing data while programming!

Author Information

Sivakumaran Raman is a physician who has spent most of his career in Medical Informatics and Analytics. With the experience of leadership positions at several large US health insurance and information technology firms, he has extensive expertise working with medical claims and clinical data using big-data platforms like Hadoop and Spark. He counts R among his favorite programming languages along with Scala and Perl. Contact him at sraman9757@gmail.com.