The Analytics Edge: R Manual

Dimitris Bertsimas
Allison K. O'Hair
William R. Pulleyblank

Dynamic Ideas LLC
Belmont, Massachusetts

This R manual is designed as a companion to the textbook *The Analytics Edge*. It starts by giving a brief introduction to R, and then explains how to implement the methods discussed in Chapter 21 of the book in the software R. All of the datasets used in this manual can be found online, and we encourage you to follow along as you learn R.

1 Getting Started in R and Data Analysis

In this section, we introduce the software R and cover the basics for getting started. R is a software environment for data analysis, statistical computing, and graphics. It is also a programming language, which is natural to use and enables complete data analyses in just a few lines. The first version of R was developed by Robert Gentleman and Ross Ihaka at the University of Auckland in the mid-1990s.

R is free and is an open-sourced project, so it is available to everyone. For this reason and many others, R is widely used, with over two million users around the world. New features and packages are being developed all the time, and there are a lot of community resources available online.

If you get stuck, want more information on a topic, or just want to learn more about R in general, there are a lot of great resources online. Here are a few helpful websites:

- Official Page: `www.r-project.org`
- Download Page: `www.cran.r-project.org`
- Quick-R: `www.statmethods.net`
- R Search Page: `www.rseek.org`
- R Resources: `www.ats.ucla.edu/stat/r/`

The best way to learn R is through trial and error, so let's get started.

Downloading R

You can download and install R for free from the Download Page listed above. At the top of the page, select your operating system (Linux, Windows, or Mac) and then follow the instructions. For most Windows users, you will select "install R for the first time" and then select the top link for the latest version of R. For most Mac users, you will also want to download the latest version (unless you have an older operating system). Once you have downloaded and installed R, start the application, and then test your installation by going through the following exercise.

When you start R, you should see a window titled "R Console." In this window, there is some text, and then at the bottom there should be

a > symbol (greater-than symbol), followed by a blinking cursor. At the cursor, type the following:

```
> sd(c(5,8,12))
```

and then hit the enter key. You should see `[1]` followed by the number 3.511885 (this is the standard deviation of the numbers 5, 8, and 12). If you don't get this answer, try reading through the R installation steps again, or look at the R FAQ page (`http://cran.r-project.org/doc/FAQ/R-FAQ.html`) for help. If you did get this answer, you are ready to start working in R.

Basic Calculations

In your R Console, you will always type commands after the arrow, or greater-than sign. You can type commands directly in the console, or you can use a script file. We highly recommend using script files to save your work. Using script files to save and run commands is discussed in more detail later in this section. You can find script files with the commands used in each section of this manual in the Online Companion for this book.

Let's start by using R for basic calculations. Try typing `8*6` in your R console after the > symbol, hit enter, and then type `2^16` after the > symbol, and hit enter again. You should see the following in your R console:

```
> 8*6
[1] 48
> 2^16
[1] 65536
```

The first line computed the product of 8 and 6 (48), and the second line computed 2 to the power 16 (65536). The `[1]` is just R's way of labeling the output, and you can safely ignore it.

If you type a command but do not finish it properly, R will display a plus sign. For example, in the output below we just typed `2^` and hit enter.

```
> 2^
+
```

The plus sign is R's way of telling you that the command is not finished. You can either complete the command (in this case by typing a number) or you can hit the escape key to get back to the greater-than symbol.

R has a nice shortcut to scroll through previous commands. Just hit the up arrow to scroll through the previous commands you ran in your current session. You can also use the down arrow to get back to where you started in the list of commands. If you find one that you want to re-run, just hit enter (or you can edit the command first, and then hit enter).

Functions and Variables

R mostly works in terms of functions and variables. A *function* can take in several arguments or inputs, and returns an output variable. If you are familiar with Excel, functions in R are very similar to functions in Excel. Examples are the square root function (`sqrt`) and the absolute value function (`abs`). The following output shows how these functions can be used:

```
> sqrt(2)
[1] 1.414214
> abs(-65)
[1] 65
```

There are thousands of functions built into R, and we will add packages in later sections to access even more functions.

You can get help on any function by typing a question mark, and then the name of the function in your R console (for example, `?sqrt`). The help pages in R can take a little getting used to, but remember that you can also try searching online for more information about a function if needed.

We will often want to save the output of a function so that we can use it later. We can do this by assigning the output to a *variable*. The following R code assigns the output of the `sqrt` function to a variable named `SquareRoot2`:

```
> SquareRoot2 = sqrt(2)
```

You should notice that you don't see the output of the `sqrt` function when you assign it to a variable name. You can check that everything went okay by viewing the value of the variable - just type the name of the variable in your R console and hit enter:

```
> SquareRoot2
[1] 1.414214
```

We used the equals sign (=) here to assign the output of a function to a variable name. You could instead use the notation `<-` for assignment. For example, `SquareRoot2 <- sqrt(2)`. We will use the equals sign in this manual because we feel that it is more natural to use. However, you can use either notation, and if you look online for R help, you will probably encounter both options.

The name of the variable is completely up to you, but there are some basic variable naming rules: do not use spaces in variable names, and do not start variable names with a number. Also, keep in mind that variable names in R are case sensitive (capital and lowercase letters are not equivalent).

If you want to see a list of all of the variables you have created in your current R session, you can type `ls()` in your R console and hit enter (this can be very useful if you forget what name you used for a variable).

Vectors and Data Frames

So far, we have only created a variable that equals a single number. We will also work with *vectors* (a series of numbers stored as the same object) and *data frames* (similar to a matrix, and looks like a spreadsheet in Excel).

You can create a vector by using the `c` function. For example, we can create a vector of country names:

```
> CountryName = c("Brazil", "China", "India", "Switzerland",
"USA")
> CountryName
[1] "Brazil"    "China"    "India"    "Switzerland"    "USA"
```

Our vector is called `CountryName`, and it contains five elements. When the output of `CountryName` is displayed, you might notice that the names are in quotes, just like they were when we created the vector. This tells us that R recognized that this is a *character* vector, as opposed to a *numeric* vector.

We can also create a vector that contains the life expectancy of each country:

```
> LifeExpectancy = c(74, 75, 66, 83, 79)
> LifeExpectancy
[1] 74 75 66 83 79
```

We know that this one is a numeric vector, because the output is not in quotes. Be careful to not mix characters and numbers in one vector, since R will convert all of the values to characters, and you will no longer be able to do any numeric calculations with the numbers, like compute the mean. We will use *data frames* to have character vectors and numeric vectors in the same object.

A single element of a vector can be displayed using square brackets:

```
> CountryName[1]
[1] "Brazil"
> LifeExpectancy[3]
[1] 66
```

This shows us that the first element of `CountryName` is "Brazil" and the third element of `LifeExpectancy` is 66.

Another nice function for creating vectors is the `seq` function, or the sequence function. This function takes three arguments, and creates a sequence of numbers. The sequence starts at the first argument, ends at the second argument, and jumps in increments defined by the third argument. As an example, let's create a sequence from 0 to 50, in increments of 5:

```
> seq(0,50,5)
[1] 0 5 10 15 20 25 30 35 40 45 50
```

This can be very useful if you want to create a column of identifiers. For example, if you have 1000 data points, and you want to number them from 1 to 1000, you can use `seq(1,1000,1)`.

To store multiple vectors as one object, they can be combined into a *data frame*. When we work with data in this manual, we will read it in to a data frame from a CSV (Comma Separated Values) file, so we will be working with data frames extensively. You can think of a data frame as a bunch of vectors stored as the same object.

Let's create our first data frame by using the `data.frame` function to combine `CountryName` and `LifeExpectancy` into a data frame called `CountryData`:

```
> CountryData = data.frame(CountryName, LifeExpectancy)
> CountryData
     CountryName  LifeExpectancy
 1        Brazil              74
 2         China              75
 3         India              66
 4   Switzerland              83
 5           USA              79
```

The data frame `CountryData` contains two columns (`CountryName` and `LifeExpectancy`) and five rows, or observations (one for each country). We can learn more about the structure of our data frame by using the `str` function:

```
> str(CountryData)
'data.frame':  5 obs.  of 2 variables:
 $ CountryName     :Factor w/ 5 levels "Brazil","China",..:1 2 3 4 5
 $ LifeExpectancy  :num 74 75 66 83 79
```

The first row tells us that `CountryData` is a data frame with 5 observations (or rows) and 2 variables (or columns). Then each variable is listed after a dollar sign. The dollar sign is used to denote a vector in a data frame, and we will see how we can use it to access a particular vector of a data frame shortly.

After each variable name, the type of the variable is listed (in this case factor or numeric) and then a sample of the values. A factor variable can be a little tricky, but you can think of it as a categorical variable, or a variable with several different categories. Any character vector will by default be stored in a data frame as a factor variable. The "levels" of a factor variable are the number of different possible values in the variable. In the case of `CountryName`, there are five levels because there are five countries. The numbers at the end of the `CountryName` row are just assigning a numerical value to each factor level for R to use internally – we generally do not need to worry about knowing these numbers.

If we want to add another vector to our data frame (let's say we want to add the population of each country, in thousands), we can do this easily by using the "dollar sign notation." We just need to type the name of the data frame, a dollar sign, then the name of the new variable we want to create, followed by the values:

```
> CountryData$Population = c(199000, 1390000, 1240000, 7997,
318000)
> str(CountryData)
'data.frame':  5 obs.  of 3 variables:
 $ CountryName     :Factor w/ 5 levels "Brazil","China",..:1 2 3 4 5
 $ LifeExpectancy  :num 74 75 66 83 79
 $ Population      :num 199000 1390000 1240000 7997 318000
```

If you instead want to add a new observation (a new row, or new country in this case) you can use the `rbind` function. We will not go into detail here, but you can learn more by typing `?rbind` in your R console.

Loading CSV Files

Most of the time, you will not create your data set from scratch in R. You will instead read the data in from a file. In this manual, we will mostly use CSV files, but you can read in any type of delimited file. (For more information, see `?read.table`.) A CSV file can easily be opened and modified in Excel, or in another spreadsheet software.

The first thing you need to do to read in a CSV file is to navigate to the directory on your computer where the CSV file is stored. You can do this on a Mac by going to the "Misc" menu in R, and then selecting "Change Working Directory." Then you should navigate to the folder on your computer containing the data file you want to load into R, and click "Open." On a Windows machine, you should go to the "File" menu, select "Change dir ...", and then navigate to the directory on your computer containing the file you want to load. (You can also use the `setwd` function to set your working directory in R. For more information, see `?setwd`.)

After navigating to the correct directory in R, nothing should have happened in your R console, but if you type `getwd()`, you should see the folder you selected at the end of the output (this is called the path to the file).

Let's read in the CSV file "WHO.csv." You can find this file in the Online Companion for this book. Download the file, and save it to a location that you will remember. Then navigate to the location of the file using the instructions above. Once you are in the folder containing "WHO.csv", you can read in the data and look at its structure by using the `read.csv` and `str` functions:

```
> WHO = read.csv("WHO.csv")
> str(WHO)
'data.frame':  194 obs.  of 10 variables:
 $ Country              :Factor w/ 194 levels "Afghanistan",..:1 2
 3 4 5 6 7 8 9 10 ...
 $ Region               :Factor w/ 6 levels "Africa", "Americas",
 ..:  3 4 1 4 1 2 ...
 $ Population           :int 29825 3162 38482 78 20821 89 41087
 2969 23050 8464 ...
 $ Under15              :num 47.4 21.3 27.4 15.2 47.6 ...
 $ Over60               :num 3.82 14.93 7.17 22.86 3.84 ...
 $ FertilityRate        :num 5.4 1.75 2.83 NA 6.1 2.12 2.2 1.74
 1.89 1.44 ...
 $ LifeExpectancy       :int 60 74 73 82 51 75 76 71 82 81 ...
 $ CellularSubscribers  :num 54.3 96.4 99 75.5 48.4 ...
 $ LiteracyRate         :num NA NA NA NA 70.1 99 97.8 99.6 NA
 NA ...
 $ GNI                  :num 1140 8820 8310 NA 5230 ...
```

This dataset contains recent statistics about 194 countries from the World Health Organization (WHO). The variables are: the name of the country (`Country`), the region the country is in (`Region`), the population of the country in thousands (`Population`), the percentage of the population under 15 years of age (`Under15`), the percentage of the population over 60 years of age (`Over60`), the average number of children per woman (`FertilityRate`), the life expectancy in years (`LifeExpectancy`), the number of cellular subscribers per 100 population (`CellularSubscribers`), the literacy rate among adults at least 15 years of age (`LiteracyRate`), and the gross national income per capita (`GNI`).

Another useful function for summarizing a data frame is the `summary` function. If you type `summary(WHO)` in your R console, you should see a statistical summary of each of the variables. For factor variables like `Country` and `Region`, the output counts the number of observations belonging to each of the possible values. For numeric variables, the output shows the minimum, first quartile (the value for which 25% of the data is less than the value), median, mean, third quartile (the value for which 75% of the data is less than the value), maximum, and number of missing values (NAs) of the variable.

Subsetting Data

It is often useful to create a subset of a data frame to be used in analysis or for building models. This can be done with the `subset` function. For example, suppose we wanted to subset our data frame `WHO` to only contain the countries in the Europe region. We can do this with the following command:

```
> WHOEurope = subset(WHO, Region == "Europe")
```

This creates a new data frame called `WHOEurope`, which contains the observations from the data frame `WHO` for which the `Region` variable has value "Europe." Note that a double equals sign (`==`) is used here to test for equality, versus the single equals sign, which is used for assignment. If you take a look at the structure of `WHOEurope` using the `str` function, you should see that it has the same 10 variables that the original data frame `WHO` has, but only 53 observations, corresponding to the 53 countries in the Europe region.

Basic Data Analysis

Once we have loaded our data set into R, we can quickly compute some basic statistical properties of the data. The following commands compute the mean, standard deviation, and statistical summary of the variable `Under15`:

```
> mean(WHO$Under15)
[1] 28.73242
> sd(WHO$Under15)
[1] 10.53457
> summary(WHO$Under15)
   Min.   1st Qu.   Median    Mean   3rd Qu.    Max.
  13.12     18.72    28.65   28.73     37.75   49.99
```

Note that we reference a variable in a data frame by using the dollar sign. If we just typed `mean(Under15)` we would get an error message. R will only recognize a variable name in a data frame if you link it to the proper data frame with the dollar sign. We will see an exception to this later when we start building models, but in general, you will need to type the data frame name, then a dollar sign, and then the name of the variable (for example, `WHO$Under15`).

The `which.min` and `which.max` functions are also very useful. Let's look at an example:

```
> which.min(WHO$Under15)
[1] 86
> WHO$Country[86]
[1] Japan
```

The `which.min` function returned the index of the observation with the minimum value of the variable `Under15`. By looking at the country name for the 86th observation, we can see that the country is Japan. So in our dataset, Japan has the lowest percentage of the population under 15 years of age. The `which.max` function works similarly:

```
> which.max(WHO$Under15)
[1] 124
> WHO$Country[124]
[1] Niger
```

This tells us that Niger is the country with the largest percentage of its population under the age of 15.

Plots

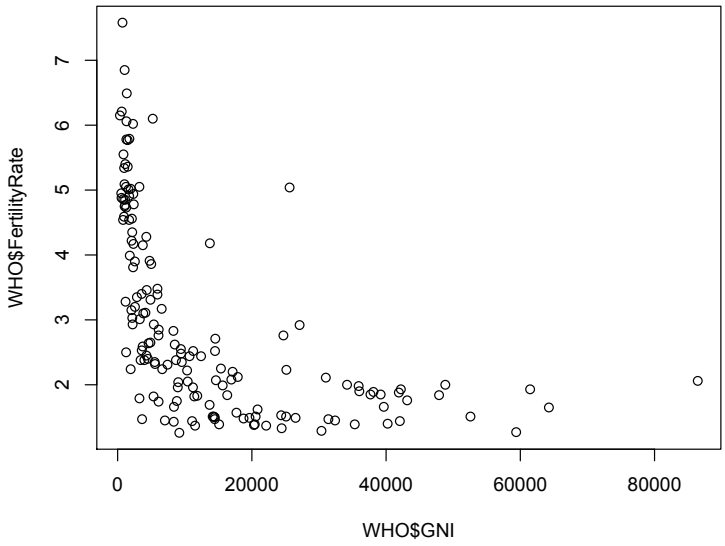
We can easily generate basic plots in R. Later in this manual, we will see how we can generate more sophisticated and visually appealing plots using `ggplot2`, a great visualization package.

Let's start with a basic scatterplot of `GNI` versus `FertilityRate`:

```
> plot(WHO$GNI, WHO$FertilityRate)
```

The `plot` function generates a scatterplot. The first argument will go along the x-axis, and the second argument will go along the y-axis. If you run this command in your R Console, you should see a plot like the one in Figure 1.

Figure 1: Scatterplot generated with the R command `plot(WHO$GNI, WHO$FertilityRate)`.



By looking at this plot, we can see that there are a few abnormal points with an unusually high income given how high the fertility rate is. Let's investigate these points. First, we will use the subset function:

```
> Outliers = subset(WHO, GNI > 10000 & FertilityRate > 2.5)
```

This takes the observations from `WHO` that have both a `GNI` greater than 10000 and a `FertilityRate` greater than 2.5, and stores them in a new data frame called `Outliers`. The `&` symbol indicates that we want the first condition to be true, *and* we want the second condition to be true (if instead we wanted the first condition to be true *or* the second condition to be true, we could have used the `|` symbol). After running this command, if you type `nrow(Outliers)` in your R console, you should get the output 7. The `nrow` function counts the number of rows in the data frame. So there are seven countries in our data set that have a GNI greater than 10000 and a fertility rate greater than 2.5. If you want to see which countries these are, you can take a look at the country names, by typing `Outliers$Country` in your R console.

Other simple plots to make in R include a histogram and a box plot. The following commands generate a histogram of the variable `CellularSubscribers`, and a box plot of the variable `LifeExpectancy` (with the observations sorted by the variable `Region`):

```
> hist(WHO$CellularSubscribers)
> boxplot(WHO$LifeExpectancy ~ WHO$Region)
```

The histogram is shown in Figure 2. The value of `CellularSubscribers` is shown along the x-axis, and the frequency, or count, is shown on the y-axis. The bars denote the frequency of the different "buckets" of possible values. The bucket sizes can be changed by adding an additional parameter (for more information, see `?hist`). A histogram is useful for understanding the distribution of a variable. This histogram shows us that the most frequent value of `CellularSubscribers` is around 100.

The box plot is shown in Figure 3. A box plot is useful for understanding the statistical range of a variable. This box plot shows how the life expectancy in countries varies according to the region they are in (the regions are listed along the x-axis, and the values for `LifeExpectancy` are listed along the y-axis). The box for each region shows the range of values between the first and third quartiles, with the middle line marking the median value. The dashed lines, or "whiskers," show the range of values, excluding any outliers, which are plotted as circles. Outliers are defined by first computing the difference between the first and third quartiles, or the height of the box. This number is called the inter-quartile range (IQR). Any point that is greater than the third quartile plus the IQR, or any point that is less than the first quartile minus the IQR is considered an outlier.

To add a nice title and axis labels to the plot, we can add a few additional arguments. For example, let's add the y-axis label "Life Expectancy," and the title "Life Expectancy of Countries by Region" to the box plot:

```
> boxplot(WHO$LifeExpectancy ~ WHO$Region, ylab = "Life
Expectancy", main = "Life Expectancy of Countries by Region")
```

Figure 2: Histogram generated with the R command `hist(WHO$CellularSubscribers)`.

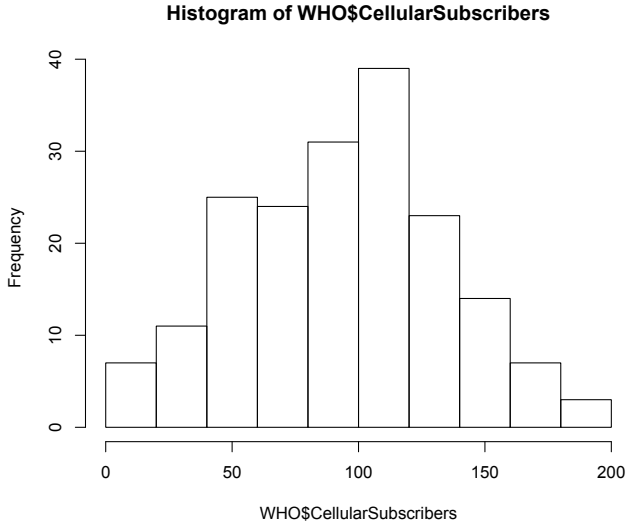
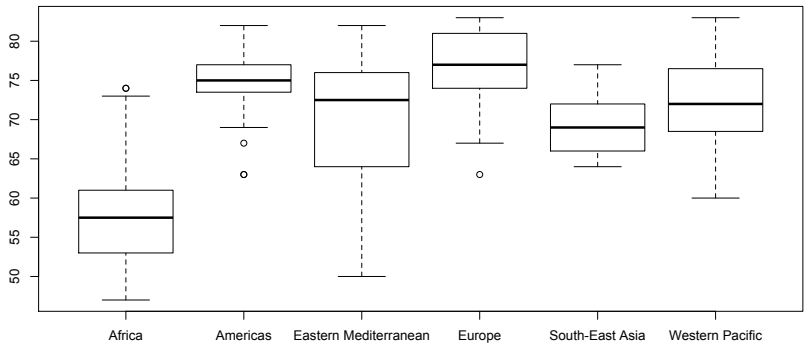


Figure 3: Box plot generated with the R command `boxplot(WHO$LifeExpectancy ~ WHO$Region)`.



We could add similar arguments to the `plot` command or to the `hist` command. For more information about the different plotting functions and

the different options available, type `?plot`, `?hist`, or `?boxplot` in your R console.

Summary Tables

The `table` and `tapply` functions are very useful in R for generating summary tables, which can be helpful for understanding trends in the data. The `table` function is particularly helpful for understanding factor variables, or numeric variables that take a small number of values. Let's start by creating a table of the variable `Region`:

```
> table(WHO$Region)
          Africa         Americas  Eastern Mediterranean   Europe
              46               35                     22       53
 South-East Asia  Western Pacific
              11               27
```

The table function counts the number of observations that take each of the possible values of the variable. So this output shows us that there are 46 countries in the region Africa, 35 countries in the region Americas, and so on.

We can also use logical statements to count observations. For example, let's see how many countries have a life expectancy greater than 75:

```
> table(WHO$LifeExpectancy > 75)
 FALSE   TRUE
   134     60
```

This output tells us that there are 134 countries for which the statement is false, or 134 countries for which the life expectancy is less than or equal to 75. There are 60 countries for which the statement is true, meaning that there are 60 countries for which the life expectancy is greater than 75.

We can create a two-dimensional table if we want to compare the values of two variables. For example, let's see how the life expectancy varies according to the region the country is in:

```
> table(WHO$Region, WHO$LifeExpectancy > 75)
                        FALSE  TRUE
 Africa                    46     0
 Americas                  23    12
 Eastern Mediterranean     16     6
 Europe                    20    33
 South-East Asia           10     1
 Western Pacific           19     8
```

The possible values of the first variable we put in the `table` command are listed on the left, and the possible values of the second variable (in this case a logical statement) that we put in the `table` command are listed on the top. Each number counts the number of observations (or countries in

this case) that belong to the region labeling the row, and have either a life expectancy less than or equal to 75 (the `FALSE` column) or a life expectancy greater than 75 (the `TRUE` column).

This table provides us with some interesting information that we have not learned yet. The life expectancy in countries varies significantly depending on which region they are in. Note that we have not created any predictive models yet, but we are still able to learn some valuable insights about our data by just using descriptive analytics.

Another useful function for creating summary tables is the `tapply` function (if you are familiar with Excel, the `tapply` function is like a pivot table). It can be a little tricky to understand, but let's go ahead and just try out the function:

```
> tapply(WHO$Over60, WHO$Region, mean)
    Africa         Americas   Eastern Mediterranean
  5.220652        10.943714                5.620000
    Europe   South-East Asia         Western Pacific
 19.774906         8.769091               10.162963
```

The `tapply` function takes three arguments. The first argument is the variable that we want to compute something about, the second argument is the variable we want to sort our observations by, and the third argument is what we want to compute. In this case, we want to compute the mean of `WHO$Over60`, with the observations sorted by `WHO$Region`. The output tells us that in Africa, the mean percentage of the population over 60 is 5.22%, while in Europe, the mean percentage of the population over 60 is 19.77%.

Let's try another example. This time, we want to compute the minimum value of `LiteracyRate`, again sorted by `Region`:

```
> tapply(WHO$LiteracyRate, WHO$Region, min)
          Africa         Americas   Eastern Mediterranean   Europe
              NA               NA                      NA       NA
 South-East Asia   Western Pacific
              NA               NA
```

Unfortunately, it didn't work this time because the variable `LiteracyRate` has missing values. To tell the `tapply` function to do the computation ignoring missing values, we just need to add an extra argument, which will compute the minimum value by ignoring all of the countries with a missing value for the `LiteracyRate` variable:

```
> tapply(WHO$LiteracyRate, WHO$Region, min, na.rm=TRUE)
          Africa         Americas   Eastern Mediterranean   Europe
            31.1             75.2                    63.9    95.2
 South-East Asia   Western Pacific
            56.8             60.6
```

This tells us that the smallest literacy rate of any country in the Americas (for which we have data) is 75.2, while the smallest literacy rate of any country in South-East Asia (for which we have data) is 56.8.

Saving Your Work with Scripts

While working in R, you often want to save your work so that you can easily re-run commands and re-build models. There are several ways of doing this, but the method we recommend is using a script file.

You can open a new script file on a Mac by going to the "File" menu, and selecting "New Document," and on a Windows machine by again going to the "File" menu, and selecting "New script." You should edit the script file just like any text file. When you include commands in the script file, do not include the `>` symbol or any of the output. To include a comment, just start the line with a pound symbol (#).

If you re-open your script file in R, it is easy to re-run lines of code. Just highlight the line, and then hit the keys Command-Enter on a Mac, or Control-r on a Windows machine.

When you quit R, it will ask you if you want to save your workspace. If you have everything you want in a script file, then you do not need to worry about saving your workspace. Additionally, script files can easily be shared with others, or used on multiple computers. You can find a script file containing all of the commands used in this section in the Online Companion.

2 Linear Regression in R

In this section, we will see how to create linear regression models in R. Our example will be Ashenfelter's linear regression model, which was described in Chapter 1 of *The Analytics Edge*. The dataset "Wine.csv" is provided in the Online Companion. Let's go ahead and read the data into R, and then take a look at it using the `str` function. Do not forget to navigate to the directory on your computer containing the file "Wine.csv" before running these commands (see Section 1 for more information on how to do this).

```
> Wine = read.csv("Wine.csv")
> str(Wine)
'data.frame':  25 obs.  of 7 variables:
 $ Year          :int 1952 1953 1955 1957 1958 1959 1960 1961 1962...
 $ Price         :num 7.5 8.04 7.69 6.98 6.78 ...
 $ WinterRain    :int 600 690 502 420 582 485 763 830 697 608 ...
 $ AGST          :num 17.1 16.7 17.1 16.1 16.4 ...
 $ HarvestRain   :int 160 80 130 110 187 187 290 38 52 155 ...
 $ Age           :int 31 30 28 26 25 24 23 22 21 20 ...
 $ FrancePop     :num 43184 43495 44218 45152 45654 ...
```

This dataset has seven variables, which are described in Table 1. We will build a linear regression model to predict `Price`, using `WinterRain`, `AGST`, `HarvestRain`, `Age`, and `FrancePop` as independent variables.

Table 1: Explanation of the Variables in the Wine dataset.

Variable	Description
Year	The year the wine was produced.
Price	A measurement of wine quality computed by Ashenfelter.
WinterRain	The amount of winter rain the year the wine was produced, measured in millimeters.
AGST	The average growing season temperature the year the wine was produced, measured in degrees Celsius.
HarvestRain	The amount of rain during harvest season (August and September) the year the wine was produced, measured in millimeters.
Age	The age of the wine, relative to 1983.
FrancePop	The population of France the year the wine was produced.

We can do this in R using the `lm` function, which stands for *linear model*:

```
> WineReg = lm(Price ~ WinterRain + AGST + HarvestRain + Age +
FrancePop, data=Wine)
```

The `lm` function gives as output a linear regression model, which we saved to the variable named `WineReg`. The first argument of the `lm` function is the formula for the model we want to build. The formula starts with the dependent variable, or the variable we want to predict, then is followed by a tilde symbol (~), and then each of the independent variables, separated by plus signs. The second argument of the function is the data set we want to use to build our model. Note that here we do not need to use the dollar sign notation to refer to our variables, because we have the `data` argument telling R which data set to use. (In fact, you should be careful to not use the dollar sign notation when building models, because it will prevent this model from being used on any other dataset.)

To look at the output of the model, we can use the `summary` command:

```
> summary(WineReg)
Call:
lm(formula = Price ~ WinterRain + AGST + HarvestRain + Age +
  FrancePop, data = Wine)

Residuals:
      Min         1Q     Median       3Q      Max
 -0.48179  -0.24662  -0.00726   0.2212  0.51987

Coefficients:
                Estimate  Std.  Error  t value  Pr(>|t|)
 (Intercept)     -0.4504        10.19   -0.044  0.965202
 WinterRain     0.001043    0.0005310    1.963  0.064416 .
 AGST             0.6012       0.1030    5.836  1.27e-05 ***
 HarvestRain   -0.003958    0.0008751   -4.523  0.000233 ***
 Age           0.0005847      0.07900    0.007  0.994172
 FrancePop    -4.953e-05    1.667e-04   -0.297  0.769578
---
Signif.codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'  0.1 ' ' 1

Residual standard error:  0.3019 on 19 degrees of freedom
Multiple R-squared:  0.8294, Adjusted R-squared:  0.7845
F-statistic:  18.47 on 5 and 19 DF, p-value:  1.044e-06
```

The output starts by describing the "Call", or the function that was used to generate this output. Below the Call is a description of the residuals, giving the minimum, first quartile, median, third quartile, and maximum values of the residuals.

Below the residual section is a table describing the coefficients. The first column of this table gives a name to each coefficient - either "Intercept", or one of the independent variable names. The "Estimate" column gives the coefficients for the intercept and for each of the independent values, or the b_0, b_1, b_2, $\ldots b_k$ values. The other three columns ("Std. Error", "t value", and "Pr($>|$t$|$)") help us determine if a variable should be included in the model, or if its coefficient is significantly different from zero, according to this dataset. For more information about the meaning of these columns, see the linear regression section of Chapter 21 of *The Analytics Edge*.

These columns give a lot of information, but R actually makes it easy for us to quickly see which variables are significant. At the end of each row of this table, there is either nothing, a period, or one to three stars (or asterisks). In this particular output, we can see a period after the `WinterRain` row, and three stars after both the `AGST` row and the `HarvestRain` row. Three stars denote the most significant variables, and are variables that are really critical to our model. Two stars are slightly less significant (but still important), and one star is significant, but even less so. A period means that the variable is "borderline" significant, and

nothing means that the variable is not significant at all.

At the bottom of the output are some statistics describing the regression equation, such as the "Multiple R-squared" value. For more information about regression in R and the `lm` function, use R help by typing `?lm` into your R console.

Refining the Model

If you have determined that you should remove one or more insignificant independent variables, it is easy to adjust the model in R. Keep in mind that due to potential multicollinearity issues, you should remove independent variables one at a time.

In our `WineReg` model, we have two variables that are not significant: `Age` and `FrancePop`. Let's try removing `FrancePop` first, since it is the variable that makes the least intuitive sense in the model. We can do this by re-running the model, and just leaving out `FrancePop` in the independent variable sum:

```
> WineReg = lm(Price ~ WinterRain + AGST + HarvestRain + Age,
data=Wine)
```

We can look at the summary output again to see how this changed the model:

```
> summary(WineReg)
Call:
lm(formula = Price ~ WinterRain + AGST + HarvestRain + Age, data =
Wine)

Residuals:
      Min         1Q     Median        3Q       Max
 -0.45470   -0.24273   -0.00752   0.19773   0.53637

Coefficients:
               Estimate   Std.  Error   t value   Pr(>|t|)
 (Intercept)   -3.4299802    1.7658975    -1.942   0.066311 .
 WinterRain     0.0010755    0.0005073     2.120   0.046694 *
 AGST           0.6072093    0.0987022     6.152    5.2e-06 ***
 HarvestRain   -0.0039715    0.0008538    -4.652   0.000154 ***
 Age            0.0239308    0.0080969     2.956   0.007819 **
---
Signif.codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'  0.1 ' ' 1

Residual standard error:  0.295 on 20 degrees of freedom
Multiple R-squared:  0.8286, Adjusted R-squared:  0.7943
F-statistic:  24.17 on 4 and 20 DF, p-value:  2.036e-07
```

Now, all of our independent variables are significant! This is a clear sign that our model was suffering from multicollinearity. The variables `Age` and `FrancePop` are highly correlated, with a correlation of -0.99. (You can

check the correlation of two variables in R by using the `cor` function: type `cor(Wine$Age, Wine$FrancePop)` in your R console.) Also note that by removing an independent variable, all of our coefficient estimates adjusted slightly.

It is also important to note that our Multiple R-squared only slightly decreased. This is another clear sign that removing `FrancePop` created a better model than the one we had before. If we had removed `Age` and `FrancePop` at the same time (remember that they were both insignificant in our original model), our Multiple R-squared would have decreased to 0.7537. This is a significant loss in terms of R^2, which we were able to avoid by removing just one variable at a time.

Making Predictions

You can use your linear regression model to make predictions on new observations by using the `predict` function. We will make predictions for the dataset "WineTest.csv," which you can find in the Online Companion. Go ahead and read this data file into R using the `read.csv` function, and call it `WineTest`. Then you can make predictions using the following command:

```
> WinePredictions = predict(WineReg, newdata=WineTest)
```

If you take a look at `WinePredictions`, you can see that our prediction for the first data point is 6.768925, and our prediction for the second data point is 6.684910. How does this compare to the actual values? Let's compute our test set R^2. We can do this in R with just a few calculations:

```
> SSE = sum((WineTest$Price - WinePredictions)^2)
> SST = sum((WineTest$Price - mean(Wine$Price))^2)
> 1 - SSE/SST
[1] 0.7944278
```

The first line computes the sum of squared errors for our predictions on the test set, and the second line computes the total sum of squares on the test set. Note that we use the mean of `Price` in the *training set* to calculate SST. Our test set R^2 is 0.79.

3 Logistic Regression in R

To show how to build a logistic regression model in R, we will use a subset of the data from Chapter 1 in *The Analytics Edge* on predicting the quality of health care. The dataset is called "Quality.csv," and can be found in the Online Companion. Our goal will be to predict whether or not a patient was classified as receiving poor quality care, based on some information that could be extracted from the patient's medical claims history.

Before building a logistic regression model, let's take a look at our data. Read the dataset "Quality.csv" into R and take a look at the structure of the data frame. This dataset has 131 observations and 8 different variables. The variables are described in Table 2. The first variable is just a unique identifier, the next six variables will be the independent variables in our model, and the last variable is the outcome, or dependent variable of our model.

Table 2: Explanation of the Variables in the Quality dataset.

Variable	**Description**
MemberID	A unique identifier for each observation.
ERVisits	The number of times the patient visited the emergency room.
OfficeVisits	The number of times the patient visited any doctor's office.
Narcotics	The number of prescriptions the patient had for narcotics.
ProviderCount	The number of providers that saw or treated the patient.
NumberClaims	The total number of medical claims the patient had.
StartedOnCombination	Whether or not the patient was started on a combination of drugs to treat their diabetes.
PoorCare	Whether or not the patient received poor care (1 if the patient had poor care, 0 otherwise).

Before building a predictive model, we want to split our dataset into a *training dataset*, which we will use to build the model, and a *testing dataset*, which we will use to test the model's out-of-sample accuracy. In the previous section on linear regression, the data was already split into two pieces for us, according to the year of the observations. This time, we just have one dataset, and there is no chronological order to the observations. The standard way to split the data in this situation is to randomly assign observations to the training set or testing set.

Creating Training and Testing Sets

The following approach can be used to split a dataset into a training and testing set for any classification problem.

The first step is to install and load a new package in R, called `caTools`. In your R console, first enter the command `install.packages("caTools")`. You will probably be asked to select a CRAN Mirror. When this window comes up, just pick a location near you. After the package has finished installing, it can be loaded into your current R session by typing `library(caTools)`. When you want to use this package in the future, you will not need to re-install it, but you will need to load it with the `library` function.

Now that the `caTools` package is installed and loaded, we are ready to split our data. We will do this using the `sample.split` function, which takes two arguments: the dependent variable, and the fraction of the data that you want in your training set. Since this function is randomly splitting our data, we can set the random seed first with the `set.seed` function to make sure that we can replicate our results later. We will set the seed to the number 88 here, but this could be any number you want. Keep in mind that if you pick a different number than we do here, or if you do not set the seed, you might get slightly different results than those reported in this section, due to the random assignment of observations to the training set or testing set.

```
> set.seed(88)
> spl = sample.split(Quality$PoorCare, SplitRatio = 0.75)
```

The `sample.split` function produces a TRUE/FALSE vector that will help us randomly split the data into two pieces according to the selected `SplitRatio` value. Here, we selected a `SplitRatio` of 0.75, which means that we want to put 75% of our data in the training set, and 25% of our data in the testing set. So `spl` should have 99 TRUE values (or 75% of the original 131 observations) and 32 FALSE values (or 25% of the original 131 observations). You can check using the `table` function.

Additionally, `sample.split` splits the data strategically, to make sure that the dependent variable is balanced in the two resulting datasets. If 90% of the observations have one outcome, and 10% have the other outcome, this ratio will be maintained in both resulting datasets. This is especially important for very unbalanced datasets, to make sure that your training set and your testing set are representative of the entire dataset.

Now we are ready to actually split our data into two pieces, according to the values of `spl`. If `spl` is equal to TRUE, we will put the corresponding observation in the training set, and if `spl` is equal to FALSE, we will put the corresponding observation in the testing set:

```
> QualityTrain = subset(Quality, spl==TRUE)
> QualityTest = subset(Quality, spl==FALSE)
```

If you take a look at the structure of `QualityTrain` and `QualityTest`, you should see that they have the same eight variables as `Quality`, but they have 99 and 32 observations, respectively.

Building a Logistic Regression Model

The process of creating a logistic regression model in R is very similar to creating a linear regression model in R, but this time we will use the `glm` function, which stands for *generalized linear model*:

```
> QualityModel = glm(PoorCare ~ ERVisits + OfficeVisits + Narcotics
+ ProviderCount + NumberClaims + StartedOnCombination, data =
QualityTrain, family=binomial)
```

The first argument to the `glm` function lists the dependent variable, followed by a list of the independent variables, separated by plus signs. This is the same format that we used for the `lm` function. Then we specify the data set we want to use to build our model (`QualityTrain`), and lastly we have to add the argument `family=binomial`. The `glm` function can be used for many different types of models, and the `family=binomial` argument indicates that we want to build a logistic regression model. We can look at our model using the `summary` function:

```
> summary(QualityModel)
Call:
glm(formula = PoorCare ~ ERVisits + OfficeVisits + Narcotics +
  ProviderCount + NumberClaims + StartedOnCombination,
  family=binomial, data = QualityTrain)

Deviance Residuals:
      Min         1Q     Median         3Q       Max
 -1.90550   -0.60126   -0.49550   -0.03788   2.21568

Coefficients:
                     Estimate   Std.  Error   z value   Pr(>|z|)
 (Intercept)         -2.887727      0.737894    -3.913   0.000091 ***
 ERVisits             0.017754      0.131474     0.135     0.8926
 OfficeVisits         0.079668      0.038430     2.073     0.0382 *
 Narcotics            0.076416      0.033741     2.265     0.0235 *
 ProviderCount        0.016655      0.027430     0.607     0.5437
 NumberClaims        -0.005524      0.014204    -0.389     0.6974
 StartedCombination   1.919179      1.368662     1.402     0.1608
---
Signif.codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’  0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance:  111.888 on 98 degrees of freedom
Residual deviance:  86.214 on 92 degrees of freedom
AIC: 100.21

Number of Fisher Scoring iterations:  5
```

This output looks similar to the linear regression model output. The coefficient estimate gives the β values of the model, and the remaining

columns of the coefficients table allow you assess the significance of an independent variable. We will not go into details about what the different columns mean here, but the probabilities and stars at the end of each row can be used to measure significance in the same way as for a linear regression model. The AIC, or Akaike Information Criterion, is a measure of the quality of the model and can be found near the bottom of the output.

Evaluating the Model

To evaluate the logistic regression model, let's start by computing the accuracy of the model on the training set with a threshold of 0.5:

```
> PredictTrain = predict(QualityModel, type="response")
> table(QualityTrain$PoorCare, PredictTrain > 0.5)

    FALSE  TRUE
 0     71     3
 1     14    11
```

The first line creates a vector of predictions for the training set, and the second line creates what is called a *classification matrix* or *confusion matrix* for a threshold of 0.5. The actual classes are listed on the left of the table, and the predictions are listed along the top. We selected a threshold of 0.5, so the prediction is FALSE if the probability is less than 0.5, and TRUE if the probability is greater than 0.5. The table tells us that we classify 81 observations correctly (71 of class 0, and 10 of class 1), for an overall accuracy rate of 81/99, or 81.8%. The table also tells us about the types of errors we make. We end up making three false positive errors, in which we predict that the patient is receiving poor care but they are actually receiving good care, and 15 false negative errors, in which we predict that the patient is receiving good care but they are actually receiving poor care. You can change the value of 0.5 in the table command to see what the classification matrix looks like with different threshold values.

Now, let's generate an ROC curve, and compute the AUC of the model. You will first need to install and load the `ROCR` package if you have not already, by using the `install.packages` and `library` functions. Then we can use the following commands to generate an ROC curve:

```
> ROCRpred = prediction(PredictTrain, QualityTrain$PoorCare)
> ROCCurve = performance(ROCRpred, "tpr", "fpr")
> plot(ROCCurve)
```

The first line generates predictions that the `ROCR` package can understand. The second line generates the information needed to create the ROC curve, where `"tpr"` stands for "true positive rate," and `"fpr"` stands for "false positive rate." The third line plots the ROC curve.

Unfortunately, it is a little hard to pick a threshold value using this ROC curve, because we don't know where a specific threshold is on the

curve. This can easily be fixed by adding threshold labels and a color legend to the plot with the following command:

```
> plot(ROCCurve, colorize=TRUE, print.cutoffs.at=seq(0,1,0.1),
text.adj=c(-0.2,0.7))
```

Now, let's compute the AUC of our model on the training set. We can output the AUC value using the following command:

```
> as.numeric(performance(ROCRpred, "auc")@y.values)
[1] 0.7945946
```

Lastly, let's make predictions for our test set. We just need to use the `predict` function again, but this time, we will pass in the argument `newdata`:

```
> PredictTest = predict(QualityModel, type="response",
newdata=QualityTest)
> table(QualityTest$PoorCare, PredictTest > 0.5)

     FALSE  TRUE
  0     23     1
  1      3     5
```

We can see from this classification matrix that the overall accuracy of our model on the test set is 28/32, or 87.5%, and we make three false positive predictions, and one true positive prediction. You can also generate the ROC curve and compute the AUC for the test set using the commands given above, but for the predictions `PredictTest`. More about how this model can be used and the interpretation of the model can be found in Chapter 1.

4 Trees in R

To show how to build a CART model and a random forest model in R with a categorical outcome, we will use data from Chapter 1 in *The Analytics Edge* on predicting decisions made by the Supreme Court. Specifically, we will build a model to predict the decisions of Justice Stevens, one of the Supreme Court Justices in 2002. The dataset is called "Stevens.csv," and can be found in the Online Companion. Our goal will be to predict whether or not Justice Stevens will reverse a case, using some basic properties of the case.

If you read this dataset into R and take a look at the structure of the data frame, you should see that there are 566 observations (Supreme Court cases) and 9 different variables. The variables are described in Table 3. The first variable is just a unique identifier, the second variable is the year of the case, the next six variables will be the independent variables in our model, and the last variable is the outcome, or dependent variable of our model.

Table 3: Variables in the dataset "Stevens.csv"

Variable	Description
Docket	A unique identifier for each case.
Term	The year of the case.
Circuit	The circuit court of origin of the case. One of $1^{st} - 11^{th}$, Federal, or D.C.
Issue	The issue area of the case. Examples are criminal procedure, civil rights, privacy, etc.
Petitioner	The type of petitioner in the case. Examples are an employer, an employee, the United States, etc.
Respondent	The type of respondent in the case. Examples are an employer, an employee, the United States, etc.
LowerCourt	The ideological direction of the lower court ruling, either liberal or conservative.
Unconst	Whether or not the petitioner argued that a law or practice is unconstitutional.
Reverse	Whether or not Justice Stevens voted to reverse the case.

Later in this section, to show how to build a CART model and a random forest model with a continuous outcome, we will use the data on wine that was used in Section 2. The file "Wine.csv" can be found in the Online Companion. The variables are described in Table 1.

Before we can build a CART or a random forest model, we need to install and load three new packages: `rpart`, for building CART models, `rpart.plot`, for plotting CART models, and `randomForest` for building random forest models. Go ahead and install and load these packages using the following commands:

```
> install.packages("rpart")
> library(rpart)
> install.packages("rpart.plot")
> library(rpart.plot)
> install.packages("randomForest")
> library(randomForest)
```

Remember to pick a location near you when the CRAN Mirror window appears. When you want to use these packages in the future, you will not need to re-install them, but you will need to load them with the `library` function.

Categorical Outcomes

Let's start by building models to predict a categorical outcome. Load the dataset "Stevens.csv" into R and call the resulting data frame `Stevens`. Then split the dataset into a training set, called `StevensTrain`, and a testing set, called `StevensTest`, using the method covered in the "Creating Training and Tests Sets" part of Section 3. Put 70% of the data in the training set, and use the outcome variable `Stevens$Reverse` as the first argument to the `sample.split` function. (To replicate the models built in this section, set the random seed to 100 using the `set.seed` function before creating the split.)

CART

To build a CART model using the data set `StevensTrain`, we can use the `rpart` function:

```
> StevensTree = rpart(Reverse ~ Circuit + Issue + Petitioner
+ Respondent + LowerCourt + Unconst, method="class", data =
StevensTrain, minbucket=25)
```

The first argument should look similar to the first argument for the `lm` or `glm` function. It gives the "formula" of the model, starting with the dependent variable, followed by a tilde symbol (~), and then a list of independent variables separated by plus signs. The next argument, `method="class"`, is necessary to indicate that we have a classification problem, or one for which the dependent variable is a categorical variable. The third argument specifies the data set that should be used to build the model. The last argument is how we will prevent our tree from over fitting, and specifies the minimum number of observations that should be in each bucket of the final tree. Here we set the `minbucket` value to 25, but we will see later in this section how we can use a validation set to pick a better parameter choice.

The first thing we should do after building a CART model is plot the tree. This can be done with the `prp` function:

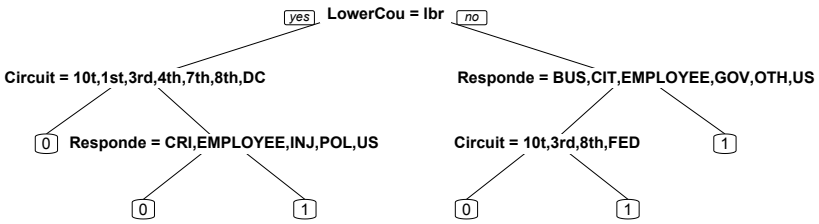
```
> prp(StevensTree)
```

The output is shown in Figure 4.

The first split of the tree is whether or not the lower court decision is liberal. If it is, then we move to the left in the tree, and check the circuit court of origin. If it is the 10th, 1st, 3rd, 4th, 7th, 8th, or DC circuit court, predict affirm. If the circuit court is not one of these, we move on to the next split, which checks the respondent. If the respondent is a criminal defendant, an employee, an injured person, a politician, or the US, we predict 0, or affirm. Otherwise, we predict reverse.

We can repeat the same process to read the splits on the other side of the tree. If at any point you need to see what the codes for the variable

Figure 4: The CART model to predict the decision of Justice Stevens, using a `minbucket` value of 25.



values represent, just table that variable in your console. For example, if you have no idea what the `Respondent` codes are, type in your console `table(Stevens$Respondent)`. The `prp` function will shorten the names of variable values so that the text fits on the tree, but by looking at a table of all of the possible values, you can map the abbreviations to the full descriptions.

There are a few important things to note regarding trees constructed with the `prp` function:

- If the split is true, always move to the left in the tree. This is denoted by a "yes" response on the image of the tree.

- The splits should always be read from top to bottom, and the observations in the buckets should be described by all of the splits that happened before that bucket was reached. For example, the bucket on the far right in Figure 4 contains observations for which the lower court decision was conservative, and the respondent was *not* a business, city, employee, government official, the United States, or other.

- The prediction for each bucket is denoted by a 0 or 1 in a circle. This is the majority outcome for each bucket.

Before we see how well our model does on the test set, let's validate our selection of the `minbucket` parameter. We first need to split the training set into two pieces, one to build the model, and the other to test the accuracy of the model. We can do this by just using the `sample.split` function again:

```
> set.seed(100)
> spl = sample.split(StevensTrain$Reverse, SplitRatio = 0.5)
> StevensValidateTrain = subset(StevensTrain, spl == TRUE)
> StevensValidateTest = subset(StevensTrain, spl == FALSE)
```

Now, let's build three models, each with a different value of `minbucket`. We will test `minbucket` values of 5, 15, and 25. All of the models should be built using the `StevensValidateTrain` data set:

```
> StevensTree1 = rpart(Reverse ~ Circuit + Issue + Petitioner
+ Respondent + LowerCourt + Unconst, method="class", data =
StevensValidateTrain, minbucket=5)
> StevensTree2 = rpart(Reverse ~ Circuit + Issue + Petitioner
+ Respondent + LowerCourt + Unconst, method="class", data =
StevensValidateTrain, minbucket=15)
> StevensTree3 = rpart(Reverse ~ Circuit + Issue + Petitioner
+ Respondent + LowerCourt + Unconst, method="class", data =
StevensValidateTrain, minbucket=25)
```

Now, let's use each of these models to make predictions on the second piece of the validation set, `StevensValidateTest`. This can be done using the `predict` function, just like we have done for the other methods. When making predictions using a CART model, we need to include the `type = "class"` argument to get predictions that are the majority outcome for each bucket:

```
> StevensPredict1 = predict(StevensTree1, newdata =
StevensValidateTest, type="class")
> StevensPredict2 = predict(StevensTree2, newdata =
StevensValidateTest, type="class")
> StevensPredict3 = predict(StevensTree3, newdata =
StevensValidateTest, type="class")
```

Lastly, we want to compute the accuracy of each of the models by building classification matrices:

```
> table(StevensValidateTest$Reverse, StevensPredict1)
StevensPredict1
     0    1
 0   52   38
 1   47   61
> table(StevensValidateTest$Reverse, StevensPredict2)
StevensPredict2
     0    1
 0   38   52
 1   28   80
> table(StevensValidateTest$Reverse, StevensPredict3)
StevensPredict3
     0    1
 0   64   26
 1   45   63
```

The first model correctly classifies 52+61, or 113 observations, out of the 198 in the data set `StevensValiateTest`. The second model correctly classifies 38+80, or 118 observations, and the third model correctly classifies 64+63, or 127 observations. Using these results, we will use a minbucket parameter of 25 for our final model, since we are trying to maximize our accuracy.

This demonstrates how a validation split of the training set can be used to select the minbucket parameter value. To more throughly test the parameter setting, you should try more parameter values, or use the method of cross-validation. To learn more about cross-validation, see the references at the end of Chapter 21 in *The Analytics Edge*.

Now let's re-build our model using a minbucket value of 25 and the entire training set, and then compute our accuracy on the test set:

```
> StevensTreeFinal = rpart(Reverse ~ Circuit + Issue + Petitioner
+ Respondent + LowerCourt + Unconst, method="class", data =
StevensTrain, minbucket=25)
> StevensPredictTest = predict(StevensTreeFinal, newdata =
StevensTest, type="class")
> table(StevensTest$Reverse, StevensPredictTest)
StevensPredictTest
      0    1
  0   36   41
  1   17   76
```

We end up correctly predicting 112 out of the 170 cases in our test set, for an accuracy of 65.9%.

Random Forest

Now, let's build a random forest model using the `randomForest` function. To indicate that we have a classification problem with the `rpart` function, we added the argument `method="class"`. Unfortunately, the `randomForest` function does not have this option. We instead need to make sure that our outcome variable is stored as a factor before building our model:

```
> StevensTrain$Reverse = as.factor(StevensTrain$Reverse)
> StevensTest$Reverse = as.factor(StevensTest$Reverse)
```

If you try building the model before doing this, you should get an error message asking if you do indeed want to do regression. This is a warning that you probably want to convert your dependent variable to a factor with the above method.

Now we are ready to build our random forest model:

```
> StevensForest = randomForest(Reverse ~ Circuit + Issue +
Petitioner + Respondent + LowerCourt + Unconst, data = StevensTrain,
ntree=200, nodesize=15)
```

The first two arguments are exactly like the ones used for the `rpart` function: the formula, and the data set to used to build the model. The last two arguments are the parameter settings. The number of trees to build is set with the `ntree` parameter, and the minimum number of observations that should be in each bucket is set with the `nodesize` parameter. (The `nodesize` parameter corresponds to the `minbucket` parameter in a CART model.)

As mentioned previously in this section, random forest models are typically robust to the parameter settings, so we will not use a validation set to pick the parameters here. However, you can use the exact same method shown above for the `minbucket` parameter to validate the `ntree` and `nodesize` parameters.

Let's see how our random forest model does at predicting the observations in the test set. We will first use the `predict` function, and then build a classification matrix. Note that the `predict` function does not need a `type` argument when it is used to get class predictions for a random forest model.

```
> StevensPredictForest = predict(StevensForest, newdata =
StevensTest)
> table(StevensTest$Reverse, StevensPredictForest)
StevensPredictForest
     0   1
 0  39  38
 1  15  78
```

Our random forest model correctly predicts 117 out of the 170 observations in our test set, for a slight improvement in accuracy over the CART model. Keep in mind that random forests typically perform better when there are many independent variables, so the improvement in accuracy from a random forest model over a CART model could be bigger or smaller than this.

In this section, we built a model to predict an outcome with two classes. To build models with more than two classes, you can follow the exact same steps as we did here. An example is given in the "Letter Recognition" exercise of Chapter 22 in *The Analytics Edge*.

Continuous Outcomes

CART and random forest models can easily be extended to the case where the outcome variable is continuous. We will demonstrate this using the "Wine.csv" data from Section 2. Read this dataset into R, and call the resulting data frame `WineTrain`. Then read in the data file "WineTest.csv", and call the resulting data frame `WineTest`.

CART

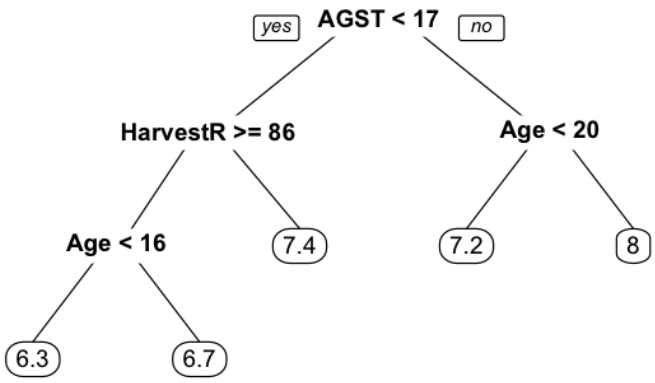
To build a CART tree with a continuous outcome, we can just leave out the `method="class"` argument:

```
> WineTree = rpart(Price ~ WinterRain + AGST + HarvestRain + Age,
data = WineTrain, minbucket=3)
```

Note that we selected a much smaller `minbucket` parameter here than in the Supreme Court model, because we have significantly less data (this parameter selection should be selected through validation to make sure you are building well-parameterized model).

If you plot the tree with `prp(WineTree)`, you should see the tree given in Figure 5. The difference between this tree and one with a categorical outcome is the values at the leaves, or buckets of the tree. For a continuous outcome, the values in each bucket give the average price for all training set observations that fell into that bucket. So, for example, the average price is 7.4 for all observations with an average growing season temperature less than 17 degrees Celsius and harvest rain of less than 86 milliliters. This is our prediction for all training set observations in this bucket, and will be our prediction for all test set observations that fall into this bucket.

Figure 5: The CART model to predict the price of wine.



Let's now make predictions on the test set. We can do this with the `predict` function, just like we did before, but leaving out the `type="class"` argument. Then to compute the accuracy on the test set, we want to compute the R^2, like we did for linear regression:

```
> WinePredictTree = predict(WineTree, newdata = WineTest)
> SSE = sum((WineTest$Price - WinePredictTree)^2)
> SST = sum((WineTest$Price - mean(Wine$Price))^2)
> 1 - SSE/SST
[1] -3.022451
```

Our R^2 is negative! This can happen sometimes when computing the out-of-sample R^2 for a model, and means that the model is doing significantly worse than just predicting the average value for all observations. We should be careful drawing any conclusions for this particular problem though, since our test set only consists of two observations.

Random Forest

How about a random forest model? None of the arguments change for the `randomForest` function with a continuous outcome. We just do not convert our dependent variable to a factor before building the model. We can also make predictions in the same way as before:

```
> WineForest = randomForest(Price ~ WinterRain + AGST + HarvestRain
+ Age, data = WineTrain, ntree=200, nodesize=3)
> WinePredictForest = predict(WineForest, newdata = WineTest)
> SSE = sum((WineTest$Price - WinePredictForest)^2)
> SST = sum((WineTest$Price - mean(Wine$Price))^2)
> 1 - SSE/SST
[1] 0.4405377
```

While our R-squared here is not as good as the one we got with our linear regression model, this is a significant improvement over the CART model.

For more practice building tree models, see the exercises in Chapter 22 of *The Analytics Edge*.

5 Clustering in R

To show how Hierarchical Clustering and K-Means Clustering models can be constructed in R, we will use the "WHO.csv" data file containing information on countries that was used in Section 1.

First, navigate to the directory on your computer containing the file WHO.csv, and use the `read.csv` function to read the datafile into R and call it `WHO`. We will be using the numerical variables in this dataset that are not missing values to cluster the countries: `Population`, `Under15`, `Over60`, and `LifeExpectancy`. So the first thing we need to do is create a new data frame with only these variables, which we will call `WHOCluster`:

```
> WHOCluster = WHO[c("Population", "Under15", "Over60",
"LifeExpectancy")]
```

If you look at the structure of this new data frame, you should see that it has the same 194 observations, but only the four variables that we listed when creating it.

Normalization

Since we have variables in our data that are on vastly different scales (`Population` is, on average, significantly larger than the other variables) we want to normalize the variables. We will do this using the `caret` package, which we will need to install and load (if you have already installed this package, you do not need to run the first line below). Then we can create a normalized dataset with the functions `preProcess` and `predict`:

```
> install.packages("caret")
> library(caret)
> preproc = preProcess(WHOCluster)
> WHOClusterNorm = predict(preproc, WHOCluster)
```

If you take a look at the summary of `WHOClusterNorm`, you should see that all variables now have mean zero (and by using the `sd` function, you can check that all variables have standard deviation 1).

Hierarchical Clustering

We are now ready to cluster our data. Let's start with hierarchical clustering, for which we need to compute all distances using the `dist` function, and then run the hierarchical clustering algorithm using the `hclust` function:

```
> distances = dist(WHOClusterNorm, method="euclidean")
> HierClustering = hclust(distances, method="ward")
```

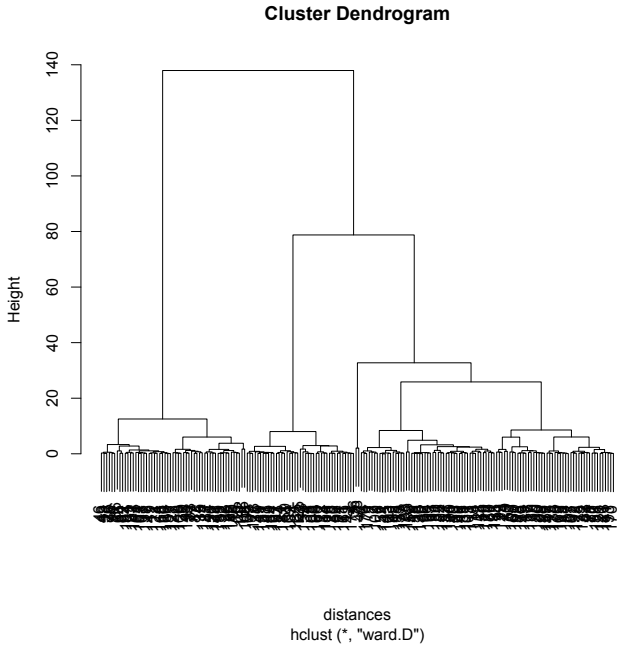
Here we computed the point distances using the Euclidean distance metric, and we performed the hierarchical clustering using Ward's method, which aims to find compact and spherical clusters. For more options of methods you can use, see the help pages for the `dist` and `hclust` functions.

You can plot the dendrogram (shown in Figure 6) using the command `plot(HierClustering)`.

The dendrogram tells us that the countries very easily cluster into two or three clusters. Five clusters also seems reasonable here if we are looking for more clusters. For more about selecting the number of clusters using a dendrogram, see Chapter 21 of *The Analytics Edge*.

Let's proceed with three clusters. We can assign each data point to a cluster using the `cutree` function, we can count the number of countries in each cluster using the `table` function, and then we can take a look at the centroids of the clusters using the `tapply` function:

Figure 6: The dendrogram produced from running the hierarchical clustering algorithm on the dataset WHO.csv.




```
> clusterGroups = cutree(HierClustering, k = 3)
> table(clusterGroups)
clusterGroups
  1   2   3
 41  98  55
> tapply(WHOCluster$Population, clusterGroups, mean)
        1         2         3
 20380.80  49721.88  24463.24
> tapply(WHOCluster$Under15, clusterGroups, mean)
        1         2         3
 43.62488  29.27765  16.65927
> tapply(WHOCluster$Over60, clusterGroups, mean)
        1         2         3
 4.625854  8.176735  21.359455
> tapply(WHOCluster$LifeExpectancy, clusterGroups, mean)
        1         2         3
 55.65854  71.20408  78.58182
```

Note that here we are using the `tapply` function on the *original* dataset WHOCluster, not the normalized dataset. We can do this because we never changed the order of the observations (so the cluster assignments can be

matched with the observations in the original dataset), and it makes the centroids more intuitive. However, you could instead look at the centroids with the normalized dataset if you prefer.

Looking at the centroids, we can see that cluster 1 stands out as the cluster containing the countries with a high percentage of the population under 15, and a low life expectancy. Cluster 2 is distinguishable as the countries with a large population, and cluster 3 contains the countries with a high percentage of the population over 60 and a long life expectancy.

K-Means Clustering

Let's now instead use the k-means clustering algorithm to cluster the countries. Since the k-means algorithm is random in its assignment of points to initial clusters, we want to start by setting the random seed so that we can replicate our results (we arbitrarily selected the number 100 here – it could be any number you want). We can then use the `kmeans` function to perform the clustering, picking the number of clusters equal to three based on the knowledge we got from running the hierarchical clustering algorithm:

```
> set.seed(100)
> KmeansClustering = kmeans(WHOClusterNorm, centers = 3)
```

Note that we did not need to compute the distances first with the k-means algorithm. Now we can extract the cluster assignments and analyze the centroids of the k-means clusters just like we did with the hierarchical clusters. The `kmeans` function makes this a little easier for us though – the cluster assignments are in the vector `KmeansClustering$cluster`:

```
> table(KmeansClustering$cluster)
  1   2   3
 57  54  83
```

We can see here that the k-means algorithm definitely found different clusters, since there is a different number of countries in each cluster. But they could still be very similar to the hierarchical clusters. To complete the analysis, you should take a look at the centroids of the k-means clusters too. The normalized centroids are already computed, and can be seen by typing `KmeansClustering$centers` in your R console. Or, you can find the centroids using the `tapply` function just like we did for the hierarchical clusters.

You can also run different types of clustering in R, including a model based approach with the `mclust` package, and spectral clustering with the `kernlab` package. We refer you to the R page on cluster analysis for more information about the packages and functions available for clustering: http://www.cran.r-project.org/web/views/Cluster.html.

6 Visualization

The ggplot2 package was created by Hadley Wickham, and was designed to improve the basic plotting functionality in R and to make the process of creating powerful and complex graphics much easier. Plotting using the ggplot2 package is done by adding layers to a plot: a data layer, an aesthetic layer, and a geometric layer. We will see several examples throughout the rest of this section.

Scatterplots

First, let's use the ggplot2 package to make some scatterplots with the "WHO.csv" dataset that we used in Sections 1 and 5. For more information about the data and the variables included, see Section 1.

The first thing we need to do is install and load the ggplot2 package:

```
> install.packages("ggplot2")
> library(ggplot2)
```

Now, let's re-create the scatterplot that we made in Section 1, but this time using ggplot. We can do this with the following command:

```
> ggplot(WHO, aes(x = GNI, y = FertilityRate)) + geom_point()
```

The first part of this command defines the data frame and aesthetic mapping that should be used to create the plot. Then, we add the layer geom_point, which will create points on the plot. The resulting plot is shown in Figure 7.

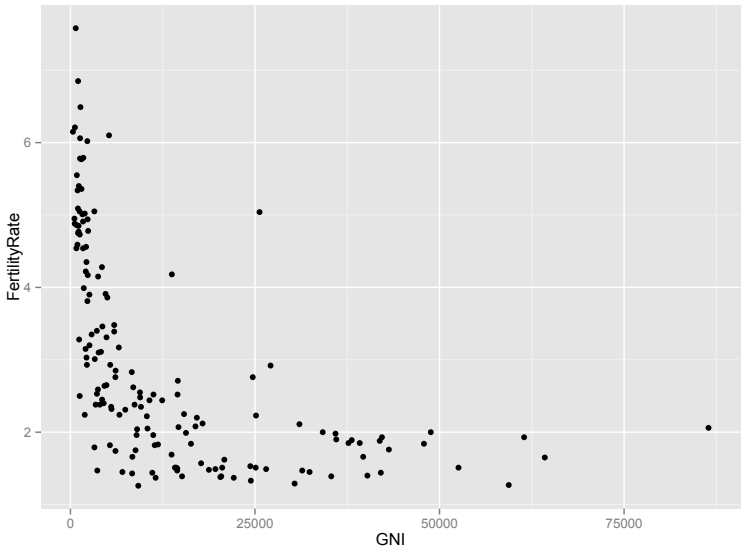
If you compare this plot to the one created in Section 1, you should see that we are creating the same plot, but with some nice visual improvements. (You can re-create the original plot with the command plot(WHOGNI, WHOFertilityRate) in R.) We can easily add some other features, like specifying a color, size, and shape for the points, and adding a title and new axis labels to the plot:

```
> ggplot(WHO, aes(x = GNI, y = FertilityRate)) +
geom_point(color="blue", size = 3, shape = 17) + ggtitle("Fertility
Rate vs.  Gross National Income") + xlab("Gross National Income") +
ylab("Fertility Rate")
```

If you create this plot in R, you should see that the points are now large blue triangles. There are many different colors and shapes that you can specify. To see all of the available colors, type colors() in your R console. To see all of the available shapes, we direct you to: www.cookbook-r.com/Graphs/Shapes_and_line_types/.

Let's now make the coloring of the points more advanced, by coloring the points by Region. We can do this by adding a color option to the aesthetic:

Figure 7: Scatterplot of gross national income versus fertility rate in the WHO dataset, using ggplot2.




```
> ggplot(WHO, aes(x = GNI, y = FertilityRate, color=Region)) +
geom_point()
```

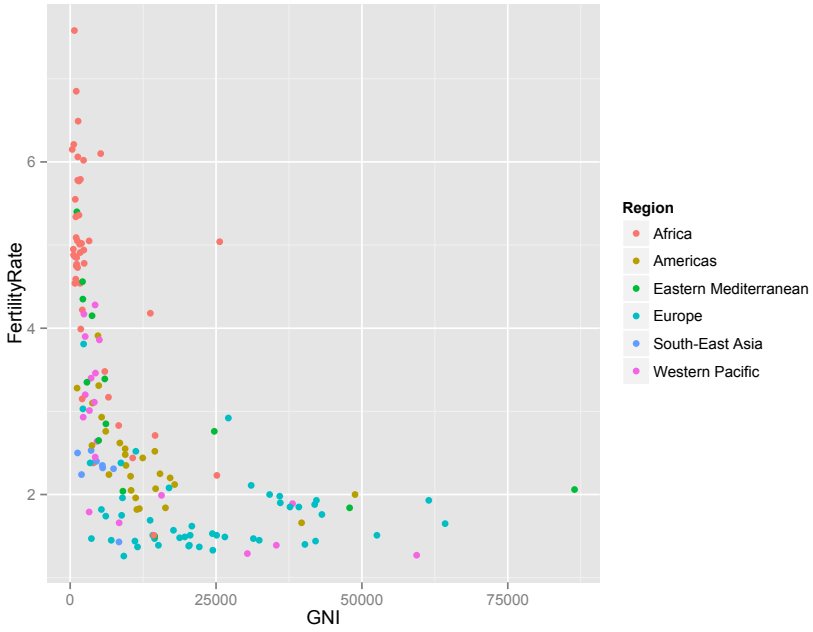
The resulting plot is shown in Figure 8. This plot easily allows us to see something we might not have observed before; the points from different regions are location in different areas of the plot. By changing the name of the variable after the `color` argument of the previous command, you can color the points by another attribute in the dataset.

Line Plots

In this section, we will use data from the City of Chicago about motor vehicle theft incidents, sometimes called *grand theft auto*. The dataset we will use is called "Crime.csv", and can be found in the Online Companion (do not open this file in any spreadsheet software before creating the plots because it might change the format of the date variable). The dataset just has three variables: the date and time the crime occurred (`Date`), the latitude coordinate of the location of the crime (`Latitude`), and the longitude coordinate of the location of the crime (`Longitude`).

We can use a line plot to communicate crime patterns over the course of an average week. Before we can do this, we need to count the number of crimes that occurred on each day of the week by using the following commands:

Figure 8: Scatterplot of gross national income versus fertility rate in the WHO dataset, with each point colored by the region it belongs to.




```
> Crime$Date = strptime(Crime$Date, format = "%m/%d/%y %H:%M")
> Crime$Weekdays = weekdays(Crime$Date)
> WeekdayCounts = as.data.frame(table(Crime$Weekdays))
```

The first line converts the `Date` variable to a format that R can understand, and the second line extracts the weekdays from `Date` and adds it as a new variable called `Weekdays` in the data frame `Crime`. The third line creates a new data frame called `WeekdayCounts` containing the counts of the number of crimes that occurred on each weekday. If you take a look at the structure of `WeekdayCounts`, you should see that there are two variables: `Var1`, which gives the name of the weekday, and `Freq` which gives the number of crimes in the dataset that occurred on the respective weekday.

Now, we are ready to create our plot using `ggplot2`:

```
> ggplot(WeekdayCounts, aes(x=Var1, y=Freq)) +
geom_line(aes(group=1))
```

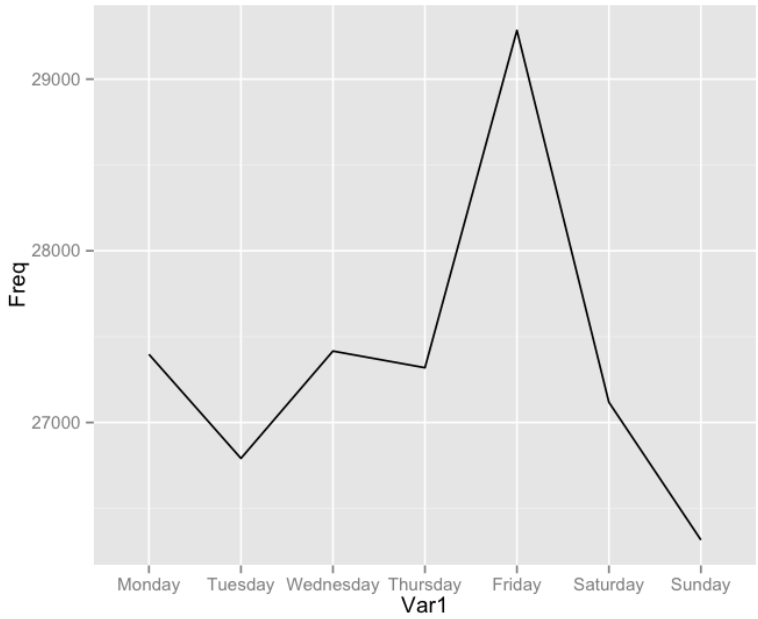
Here we use the `geom_line` geometry to create a line plot. The aesthetic option in `geom_line` just indicates that we want to plot one line. If you

create this plot, you should notice that the weekdays are in alphabetical order instead of chronological order. We can change this by making `Var1` an *ordered factor* variable, and generating our plot again:

```
> WeekdayCounts$Var1 = factor(WeekdayCounts$Var1, ordered = TRUE,
levels = c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"))
> ggplot(WeekdayCounts, aes(x=Var1, y=Freq)) +
geom_line(aes(group=1))
```

The first line changes `Var1` to an ordered factor, where the `levels` argument specifies the order for the values. The second line just generates the plot again. The resulting plot is shown in Figure 9. If you want to change the labels on the axes or add a title, you can do so using the same method we used for the scatterplot.

Figure 9: Line plot of the number of motor vehicle thefts in the city of Chicago by day of the week.



This line plot helps us observe that the number of motor vehicle thefts in Chicago (on average) is higher on Friday, and lower on Sunday. If we want to also observe how crime changes with the hour of the day, a heat map is a useful visualization.

Heat Maps

Let's create a heat map for our crime data to visualize the amount of crime in Chicago by hour and day of the week. We first need to add an `Hour` variable to our data frame, and then create a new table, this time with the crime counts for each day of the week and hour of the day:

```
> Crime$Hour = Crime$Date$hour
> WeekdayHourCounts = as.data.frame(table(Crime$Weekdays,
Crime$Hour))
```

The first line just creates a new variable in our `Crime` data frame called `Hour`, which is luckily easy to extract because date objects in R have an attribute called `hour`. The second line just creates a new data frame out of the two dimensional table with `Weekdays` and `Hour`. If you take a look at the structure of `WeekdayHourCounts`, you should see that the variables are called `Var1`, which corresponds to the weekday, `Var2`, which corresponds to the hour, and `Freq`, which corresponds to the crime counts.

The only problem we have left to fix before we can make our plot is to change how `Var1` and `Var2` are stored. We want to make sure the weekdays in `Var1` are in an intuitive order by making it an ordered factor, and we want to make sure the hours in `Var2` are numeric:

```
> WeekdayHourCounts$Var1 = factor(WeekdayHourCounts$Var1,
ordered=TRUE, levels=c("Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"))
> WeekdayHourCounts$Var2 =
as.numeric(as.character(WeekdayHourCounts$Var2))
```

The first line is exactly what we did when making our line plot. The second line converts `Var2` from a factor vector to a numeric vector. We have to use `as.character` because of how R stores factors.

Now we are ready to make our heat map. To do so, we will use the `geom_tile` geometry. We will also go ahead and change the axis labels:

```
> ggplot(WeekdayHourCounts, aes(x = Var2, y = Var1)) +
geom_tile(aes(fill = Freq)) + xlab("Hour of the Day") + ylab("")
```

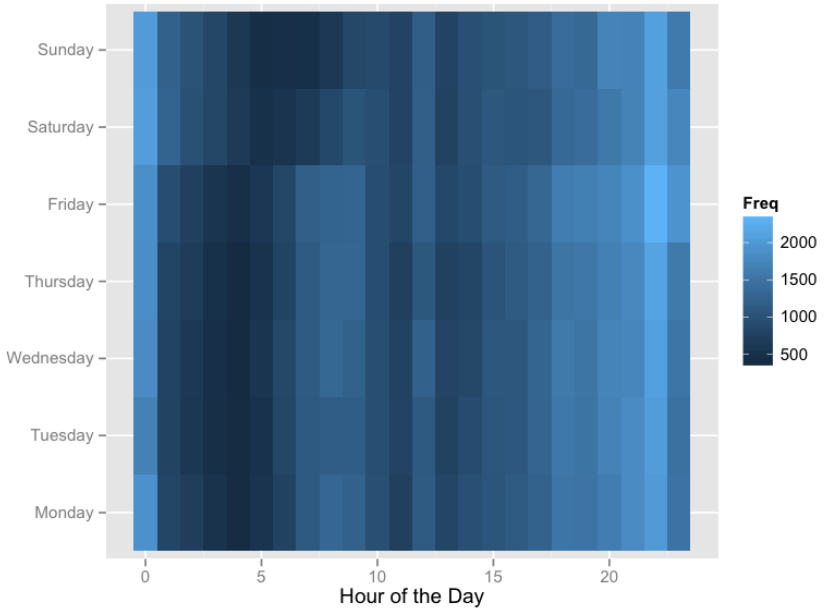
Note that here we just removed the y-axis label by using empty quotes. The resulting plot is shown in Figure 10. The legend on the right helps us understand the plot. The lighter the color is, the more motor vehicle thefts in that hour and day.

If you want to change the color scheme of the plot, you can do so by adding a `scale_fill_gradient` layer:

```
> ggplot(WeekdayHourCounts, aes(x = Var2, y = Var1)) +
geom_tile(aes(fill = Freq)) + xlab("Hour of the Day") + ylab(" ")
+ scale_fill_gradient(low="white", high="red")
```

This plot will have higher frequency values indicated by a more red color, and lower frequency values indicated by a whiter color. Changing the color

Figure 10: A heat map of the number of motor vehicle thefts in the city of Chicago by day of the week and hour of the day.



scheme is often useful depending on the application. A red color scheme is common in predictive policing, because it shows the "hotspots", or times and days with more crime, in red.

We can also create a heat map on a geographical map. Using the motor vehicle theft data, let's plot crime on a map of the city of Chicago. To do this, we need to install and load two new packages: `maps` and `ggmap`. Use the `install.packages` and `library` functions to install and load these packages.

Now, let's load a map of Chicago into R using the `get_map` function:

```
> chicago = get_map(location = "chicago", zoom = 11)
> ggmap(chicago)
```

In your graphics window, you should see a map of the city of Chicago. If we were to plot all of the motor vehicle thefts in our dataset as points on this map, we would obliterate the map of the city, since there are over 190,000 crime incidents. Instead, we will create a heat map of the amount of crime in each area of the city, where the areas are defined by rounding the latitude and longitude coordinates to two digits of accuracy.

We first need to create a new data frame that gives the crime counts by latitude and longitude coordinates. We will again use the `as.data.frame`

and `table` functions, together with the `round` function, which rounds a number to the digits of accuracy given as the second argument to the function:

```
> LatLongCounts = as.data.frame(table(round(Crime$Longitude, 2),
round(Crime$Latitude, 2)))
```

The data frame `LatLongCounts` gives us the total amount of crime in each geographical area defined by the rounded Latitude and Longitude coordinates. If you take a look at the structure of the data frame, you should see that there are 1638 observations, and three variables: `Var1` gives the longitude coordinate, `Var2` gives the latitude coordinate, and `Freq` gives the number of motor vehicle thefts that occurred in that area.

Just like we did with our previous tables, we first need to make sure that `Var1` and `Var2` are stored correctly. Currently, they are factor variables, but we want them to be numeric variables. We can fix this with the following commands:

```
> LatLongCounts$Var1 = as.numeric(as.character(LatLongCounts$Var1))
> LatLongCounts$Var2 = as.numeric(as.character(LatLongCounts$Var2))
```

Now, we are ready to create a heat map on the city of Chicago. We can do this with the `ggmap` and `geom_tile` functions:

```
> ggmap(chicago) + geom_tile(data=LatLongCounts, aes(x = Var1, y =
Var2, alpha = Freq), fill="red")
```

The resulting plot is shown in Figure 11. This is very similar to "hot spot" maps used in predictive policing. Note that the plot might take a few minutes to load on your computer.

United States Maps

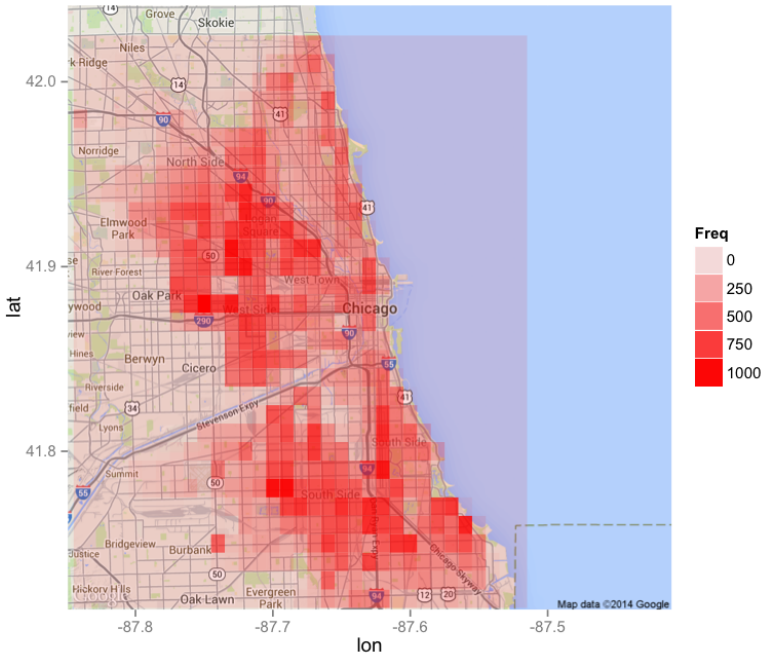
We will make an unemployment map of the United States, using unemployment rates from 2005, 2009, and 2013, to see how the Great Recession impacted various states. The data we will use is in the file "Unemployment.csv" (available in the Online Companion). If you read this dataset into R, you can see that it has four variables: the name of the state (`State`), the unemployment rate in 2005 (`Rate2005`), the unemployment rate in 2009 (`Rate2009`), and the unemployment rate in 2013 (`Rate2013`).

A map of the United States is included in R. We can load the map into a data frame called `StatesMap` using the `map_data` function:

```
> StatesMap = map_data("state")
```

If you take a look at the structure of `StatesMap`, you can see that it is a data frame with 15,537 observations. The observations tell R how to draw the map of the United States. To plot the map, we can use `ggplot` and the polygon geometry:

Figure 11: Heat map of the number of motor vehicle thefts in the city of Chicago by geographic location.




```
> ggplot(StatesMap, aes(x = long, y = lat, group = group)) +
geom_polygon(fill="white", color="black")
```

If you run this command in your R console, you should see a black and white map of the United States. To add our unemployment data to this map, we need to merge our `Unemployment` data frame with the `StatesMap` data frame, using the `merge` function. This is a very useful function in R, because it allows us to combine two data frames using a unique identifier. In our case, this identifier is the name of the state, called `State` in the `Unemployment` data frame, and `region` in the `StatesMap` data frame. The following command will create a new data frame called `UnemploymentMap`, which includes the unemployment data from the `Unemployment` data frame as new variables for each observation in the `StatesMap` data frame:

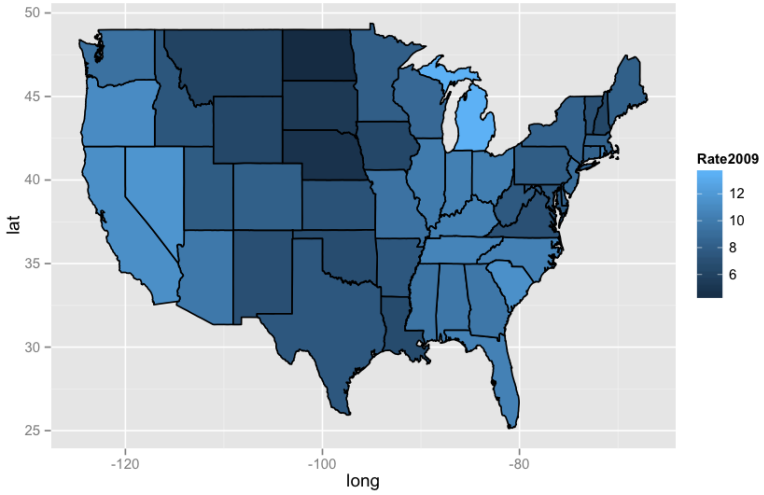
```
> UnemploymentMap = merge(StatesMap, Unemployment, by.x = "region",
by.y = "State")
```

Now, let's generate our map of the United States again, but this time shading each state by the unemployment rate in 2009:

```
> ggplot(UnemploymentMap, aes(x = long, y = lat, group = group,
fill=Rate2009)) + geom_polygon(color="black")
```

The resulting map is shown in Figure 12. Note that the difference between this command and the one to draw the blank map of the United States is the data frame we gave to the `ggplot` function, and the `fill` argument.

Figure 12: Heat map on a map of the United States, showing the unemployment rates in 2009.



To explore this dataset, it would be interesting to compare this map to the corresponding maps in 2005 and 2013. You can just change the variable being used to fill the states to generate these maps instead.

This section has just shown some basic plots you can generate using the `ggplot2` package. For more information, see `http://ggplot2.org`.