Analytics Using R

Margie Rosenberg, Wisconsin School of Business

January 16, 2021

Contents

Preface

The purpose of this book is to provide examples to help in learning the software R.

The approach presented in this book is for the R beginner. After you gain experience, you can learn how to do tasks in other ways that may be more efficient.

You can search for terms and R functions using the search icon at the top of this page.

These examples show some R code to illustrate different aspects of R. The intent is for you to type the code in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

Part I

R Basics

Chapter 1

Getting Started with R

You may have worked with Excel for some of your data analysis work. Microsoft's Excel is a good introductory package for learning how to analyze data, as the software provides a very visual interface with a menu bar to help you navigate to the appropriate place. With Excel, you are able to easily view your input, any formulas that you inserted, and your output. So you might wonder why other software is needed.

While there are many tasks for which Excel is appropriate, R is a programming language that provides more flexibility than Excel. R is not bound by a *spreadsheet*, where the data need to be entered in cells. For more complex analyses, Excel's spreadsheet format is too restrictive. And R is *freely* available on-line with new content being uploaded regularly.

1.1 What is RStudio?

RStudio is an interface that provides you with a greater ability to conduct your analyses in R. You can think of RStudio as a overlay on the software R to allow you to visually group together in one interface the input window, the output window, the objects in your workspace, and plots. The LinkedIn Learning (formerly lynda.com) videos on **Learning R** or **R Essential Statistics** are a good way to get going and see someone *implement* R with RStudio.

1.2 First Steps

In one sense, what makes R difficult to learn is that there are a million ways of doing the same thing. Everytime you pick up a new book or read something on-line, a new way will be shown and not reinforce what you have been already learned.

Of course there are other ways of learning R. *Googling* for support is certainly one option. *Youtube* has many videos in addition to the many now available on LinkedIn Learning (formerly lynda.com). Once you have loaded R and R Studio, you can access *Help* in a tab (lower right area of your screen). There are other resources for learning R. An on-line tutorial called DataCamp is one and there are on-line courses on Coursera.

To get started, follow the steps shown below.

1. Watch the first three videos of the course **Learning R** or **R Essential Statistics** on LinkedIn Learning (formerly lynda.com) in **Getting Started** with access through your MyUW interface.(Installing R on your computer, Using RStudio, Taking a First Look at the Interface)
2. Download [R] (https://www.r-project.org/) and [RStudio] (https://www.rstudio.com/).
3. Install R first, then install RStudio.

Chapter 2

Some Basic Hints

- One of the key points is to remember that R is *picky.* Really, really picky. R distinguishes upper case from lower case letters. Thus a variable named "Cost" differs from another variable named "cost".
- The way to learn new statistical software programs is through practice. There is a learning curve to R, but some text, the internet, or your classmates are good resources. You will feel really good when you figure things out, but you may struggle a bit in the process. The skills learned will be invaluable for you in the future.
- There are many options as to how to work in R and authors disagree on what is the best way. You can develop your own style.
- Check out Tools > Global Options > General tab in the menu bar of RStudio.
- Before you start programming, plan a strategy as to how to address the question at hand.
- You may want to familiarize yourself with a programming *style.* See Wickham's style guide in his book **Advanced R**.
- Comment your code thoroughly (using a # sign at the beginning of each line). Good documentation is a great reminder to you of what you are doing when you go back to the code another time and also helpful to your colleagues when you share your work.
- The character '>' at the far left in the R Console Window indicates that R is ready for you to enter a command.
- Sometimes when you are typing in code in the console, or highlighting code to include in the console, you do not include enough information and instead of a '>' to indicate that you have run the line, you end up with a

'+' instead. To delete the command and start anew, you can just click on the *Esc* key to return to the '>'.

2.1 At the Top of Your Code

- With RStudio, you can see the objects in the *Environment* window, usually in the upper right of your screen. The ls() code lists all of the objects in your workspace.
- The rm() code *removes* objects in your workspace. You can begin your code with the rm() function to clear all of the objects from your workspace to start with a clean environment. This way the workspace is empty and everything you create is clearly visible.

```
rm(list=ls())

ls()
```

The ls() command indicates that there is nothing in the workspace with the following response:

character(0)

showing that the workspace is empty. Also, in the top-right panel of RStudio, the Environment tab indicates that the workspace is empty.

- You can set the working directory so that you know where your data, code and output are stored. The setwd() and getwd() codes set the working directory and then verify the working directory. You can either hard code this with typing in commands:

```
setwd("C:/Margie/MEPS")
getwd()
```

or you can go to the *Session/Set Working Directory* menu item at the top of RStudio and set it with your mouse.

- A helpful command is **str(nameofdata)**, where *nameofdata* is the name of your data frame. This function provides a quick listing of the data, with the number of observations, the variables, and the type of the variables, whether they are an integer, real, or character.

2.2 Script Window vs. Console Window

- The *Script* Window is the place to enter and run your code so that it is easily edited and saved for future use. Usually the Script Window is shown at the top left in RStudio. If this window is *not* shown, it will be visible is you open a previously saved R script, or if you create a new R

Script. You create new R Script by clicking on File > New File > R Script in the RStudio menu bar.

- To execute your code in the R script, you can either highlight the code and click on *Run*, or you can highlight the code and press *CTRL + Enter* on your keyboard.
- If you prefer, you can enter code directly in the Console Window and click *Enter*. The commands that you run will be shown in the *History* Window on the top right of RStudio. You can save these commands for future use.

2.3 Environment, History, and Connections Window

RStudio provides nice tools to keep you informed of your work.

The *Environment* tab allows you to see what objects are in the workspace. If you create variables or data frames, you have a visual listing of everything in the current workspace. (You can similarly achieve this by listing everything with the command ls().)

The *History* tab allows you to see the commands that you have entered. You can save the output in the history window by clicking on the *save* button using a *.r extension on your file so that the file can be automatically opened in R. (You can similarly have a written record of your commands by using the script window and saving a file with the commands that you created.)

2.4 Getting Help

R is a complex language to learn and the syntax is very particular. You can get *quick* help in a number of ways.

- Note the text that appears in the *Console* Window regarding demos, help, or quitting. This approach exists prior to the introduction of RStudio.
- You can type ?something for help on something in R. If you type ?mean, documentation appears in the *Help* window on the Arithmetic Mean.
- You can go to the *Help* window and type what you need help by the search icon in the right-hand part of the window.
- Or you can google what you need.

2.5 options() Command

You can control the global options of your R workspace.

Two helpful changes are:

- options(scipen = 999)

  This command disables printing your results in scientific notation.

- options(max.print=999999)

  This command extends the number of lines of printing your results in the console.

- options(show.signif.stars=FALSE)

  This command removes the *stars* from regression output showing the statistical significance.

2.6 Some Reserved Names

Generally you have freedom to name your data frames and your variables any way that you prefer. However, there are some common *no-no*s as listed below:

- Words like FALSE/TRUE are reserved for testing logical variables.
- Words like Inf, NaN, NA are reserved for outcomes such as whether a variable is negative or positive infinity, not a number, or missing.
- If you build your own function (see Section 6.3), you would not want to name your function with something like *mean* or *var* that is built-into R. If you did, your function would override that in R.

2.7 Save Your Work

- Make sure you save your workspace and your code, especially your code. With your code, you can always regenerate your workspace but it could take a little time. Having your workspace saved allows you to start where you left off, with all of the variables you created and renamed saved.
- Your code (in your script window) is saved by clicking the SAVE button in the RStudio menu bar. The code will be saved in the working directory. See 2.1 for setting and getting the working directory.
- Do not overwrite the original data set and variables. Be sure, just in case you make a mistake, not to overwrite the data set that you know is good. Create new data sets just to be sure, especially when taking a subset from that data set.

Chapter 3

What are Packages?

Packages provide you ways of helping you manage your data or calculate some results. A package usually contains a group of functions, such as calculating an average of a group of numbers or completing a regression analysis. Sometimes packages build in data sets that you can use to help understand how to use the functions. One of the many advantages of R is that these packages are being constantly updated, as well as new ones created.

Some of these packages are included in the initial installation of R; others are available for download (and free as well). The latter are called *user-contributed* packages, designed by analtyics people who develop functions and share them with others.

Packages are to be used *at the risk of the user.* These packages make computations easier for the user so that they need not be recreated by you. Note that multiple packages may calculate the same quantity using different functions.

One package that I like is the *psych* package that has a great way to summarize the data. The *describe()* function provides different statistics with one command, like the average, the minimum, and the maximum.

Check out the LinkedIn Learning (formerly lynda.com) video on *Installing and Managing Packages* in the Getting Started video as part of the **R Essential Statistics** course.

3.1 Core R Packages

On the lower right panel of your RStudio screen there is a tab labeled packages. If you click on the *Packages* tab, you see all of the packages that are stored on your computer for use in R. Some are loaded with the initial installation, but the power of R is in your ability to download additional packages.

If the boxes next to a particular package are **not** checked, the packages are stored on your computer, but are **not** currently loaded in your R workspace. To load the package into R, you just need to check the box on the left of the package you want to load.

Suppose you click on the box next to MASS,. The result is shown in the Console in the lower left showing a command *library()*.

```
library(MASS)
```

Of course instead of clicking the box for a particular package, you could just type library(nameofpackage) in the console.

This command (or sometimes now authors are showing *require()*) loads the package in your session so that it available for use. Note that unclicking this box unloads (or as R says, "detaches") the package from your session. This action does not uninstall the package from your computer, but just removes the functions from the current workspace. If you want to delete a package, you click on the "x" that is to the right of the package (next to Version) in the Package tab in RStudio.

You can find help on the package by clicking on the package name.

Be aware that different packages have functions that are named the same. If you load both packages in the same session, the last one loaded will be the one used when you type in the function. Pay attention to the messages in the Console Window as the warning messages will indicate that a function will be masked.

3.2 Additional R Packages

If a package that you want to use is not in the list on your computer, this means it is not stored on your computer and must be downloaded. You use the Install tab under the Packages tab to do this.

Try installing the *psych* package. Click on the *Install tab* and be sure that the box for *Install dependencies* is checked. Checking this box helps ensure that if a function in the psych package depends on another function in another package, then that package is also installed. Note the installation of the psych package adds to your list of available packages.

Be aware that now the psych package is available in your computer storage, but the package is not currently loaded for use. If you tried to use a function in the package, but the package is not loaded, R will let you know that something is amiss. How do you correct this? By using the library(psych) command to load the library into your workspace.

Finally, packages are modified over time. It is good practice to check whether updates are needed. This is done by clicking on the *Update* tab (right next to the *Install* tab).

Chapter 4

Typing Data Directly into R

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

To do any analysis requires data. One can directly type data into R for use, or one could import data from an external data set or from the internet. This chapter focuses on the direct route.

4.1 Basic Calculator

R can do many different tasks from the very complicated to the very simple. At a very basic level, we can use R as a calculator.

The character '>' (or 'ready' prompt) at the far left in the R Console Window indicates that R is ready for you to enter a command.

In the Console window of RStudio, type 4 + 9 at the '>' prompt.

```
4 + 9
```

You can make the function more complicated by taking the square root of the sum:

```
sqrt(4 + 9)
```

When you enter this command, the result is displayed. With the Basic Calculator approach, the input and the results are not stored in the workspace. If they are needed again, you would need to re-enter the information.

4.2 Storing Information

We can extend the calculator function to create variables to store the information. We can do the following:

```
x <- 4
y <- 9
z <- sqrt(x + y)
```

With the above R code, you would *assign* (using the <- operator) x to have the value of 4; y to have the value of 9; and z to have the result of the square root of the sum of $x + y$.

Hint: you can use a keyboard shortcut to obtain the <- symbol. For PC users, click the *Alt key* along with - key while for those using a MAC, click the *option key* along with the - key.

Once you execute the code (i.e. run the code), nothing seems to have happened, other than R showing the *ready* prompt. (Note: you can use an equal sign (=). There are differences of opinions as to which to use.)

That is nice, but how do you know the result of the calculation? If you type x in the console at the *ready* prompt, you see it is assigned the value of 4. Similarly if you type z in the console at the *ready* prompt, you see the result of that calculation.

```
x
y
z
```

4.3 Lists

You are not limited to storing just one value in a variable. For example, you could put the numbers 1 to 10 into x with the following code:

```
x <- 1:10
x
```

We can be more precise to change the increment to a different number.

```
x <- seq(from = 1, to = 10, by = 0.5)
x
```

Note that the *from*, *to*, and *by* are not required syntax, but is included here for better understandability of the code, as shown in the example that follows.

```
x1 <- seq(1, 10, 0.5)
x1
```

We could have a vector where the numbers do not follow any logical sequence, such as:

```
x <- c(1, 6, 9)
x
```

Here the notation *c()* is a way to *combine* data into one list. This example combines numbers, but this *c()* function also combines text.

Chapter 5

Basics of Vector Math

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

A fundamental way of grouping numbers, letters, or words together in one object is by using a vector. In this chapter, the very basics of what is a vector are shown.

5.1 Simple Vector Math

One of the advantages of R is in its ability to work with vectors.

- You can add a constant to each element of a vector.

  ```
  x <- 1:5
  2 + x
  ```

- You can multiply each element of a vector by a constant.

  ```
  x <- 1:5
  2 * x
  ```

- You can add two vectors together that are the same length.

  ```
  x <- c(1, 3, 5)
  y <- c(2, 4, 6)
  x + y
  ```

  Note that the elements in the same position of each vector are added together.

- You can multiply two vectors together that are the same length.

  ```
  x <- c(1, 3, 5)
  y <- c(2, 4, 6)
  x * y
  ```

  Note that the elements in the same position of each vector are multiplied together.

- You can raise each element of a vector to a power.

  ```
  x <- 1:5
  x^2
  ```

5.2 More Subtle Vector Math

What happens when things are not simple?

Suppose you try to add c(1, 2, 3, 4) + c(5, 6). What is the result? In a matrix algebra class, you would say that this answer is undefined. If we try it in R, we actually get an answer.

```
c(1, 2, 3, 4) + c(5, 6)
```

Essentially, if the vectors are not of the same length, the shorter one is duplicated sufficiently to allow the operation to proceed. In this case, the answer reflects an adjustment of the question to:

```
c(1, 2, 3, 4) + c(5, 6, 5, 6)
```

And we can do a similar example with multiplication and obtain a similar answer.

```
c(1, 2, 3, 4)*c(5, 6)
```

But, this kind of math does not work in all cases. For example, if we slightly changed the problem to c(1, 2, 3) + c(5, 6), then an error would be produced.

```
c(1, 2, 3) + c(5, 6)
```

As the error states, the smaller vector needs to be a multiple of the larger vector. The first example satisfied the criterion, but in this case it did not.

Chapter 6

Calculations in R

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

We know that there are many times where we need to repeat calculations over and over again. To be more effective, we can create a function of that calculation and then use R to calculate these values for us.

6.1 Simple Calculations

Suppose the function that need multiple calculations of $1 + i * t$ is needed. Suppose $i = 0.05$ and $t = 2$.

Then you could create the following code, where *accum* holds the result of the calculation of $1 + i \cdot t$ for the values of i and t.

```
i <- 0.05
t <- 2
accum <- 1 + i*t
accum
```

Calculate the value of *accum* yourself to verify the result.

Now suppose you insert the code for *accum* before i and t are defined. What would happen? Here I create a new value of *accum1* that is a function of d and n (that have not been set ahead of time).

```
accum1 <- 1 + d*n
```

When you run this code, you will receive an error, as d and n have not yet been defined. (Here the error initially describes that d is missing, however once you defined d, then another error would result as you had not yet defined n.) Thus, the order of input of the code is important.

You can modify the function:

```
accum2 <- (1 + i)^t
accum2
```

Here, the function exponentiates the quantity $(1+i)$ to the tth power.

6.2 Calculations with Lists of Numbers

Instead of calculating one value of t, you could include a list of numbers like:

```
t <- seq(from = 0, to = 5, by = 0.5)
t
```

Then you might think that *accum* would automatically be recalculated. Try it.

```
accum
```

Hum. That did not work, did it? The function *accum* did not change! Why? How can you make this work?

The simplest way to make this *work* is it to re-run the original code for *accum* and recalculate:

```
accum <- 1 + i*t
accum
```

Now you have the 11 values of *accum* calculated for each value of t. While the original *accum* calculation looked like a function, it was replaced with a value once you inserted specific values for i and t.

6.3 User Defined Functions

To simplify your coding, you could develop a more general function than the ones described above. These are called *user defined functions* in R, as the user creates them from scratch.

As an example, the function is called *simple* and requires two input parameters i and t. The value of $(1 + i * t)$ is calculated.

```
simple <- function(i1, t1){
  1 + i1*t1
}
```

Note that the choice of the input parameters $i1$ and $t1$ are arbitrary. If $i = 0.05$ as above and the list of t remains defined from $[0, 5]$ by 0.5 increments, we can evaluate the function as:

```r
i <- 0.05
t <- seq(from = 0, to = 5, by = 0.5)
simple(i, t)
```

You can verify that your results agree with what you had calculated above.

Try also varying i from $[0, 0.05]$ by increments of 0.01, while keeping t fixed at 2.

```r
i <- seq(from = 0, to = 0.05, by = 0.01)
t <- 2
simple(i, t)
```

Thus, by creating this new function, you can calculate many values of accum with just one function.

However, complications occur if you want both i and t to be flexible in this function. For example:

```r
t <- seq(from = 0, to = 5, by = 0.5)
i <- seq(from = 0, to = .05, by = 0.01)
simple(i, t)
```

An error is produced here, as the lengths of i and t differ by other than a multiple. (The vector t is of length 11, while the vector i is of length 6.) This is a good thing that an error is produced. As had the vectors been a multiple of one another, the end-result of the calculation of the function would not have been correct. Alternatives to this approach will be shown in another tutorial.

Note: Be careful about re-using the same variable names with different meanings, like i and t in the same code. Your results may not be what you intend them.

Chapter 7

Creating For Loops in R

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

***For Loops* are a programming technique used to iterate a calculation.**

7.1 Prior Work

Suppose we have the function $1 + i1 * t1$ for values $i1$ and $t1$.

```
simple <- function(i1, t1){
  1 + i1*t1
 }
```

We want to evaluate the function where for every value of $i1$, we would calculate the function at every value of $t1$.

We need to be in careful using R because of the *subtle math* calculations that occur. For example, suppose the choice of i and t were the following:

```
i <- seq(from = 0, to = .05, by = 0.01)
t <- seq(from = 0, to = 5, by = 1)
```

The vector t is of length 6 and the vector i is of length 6. Evaluate *simple* to illustrate the result.

```
simple(i, t)
```

This is **not** the result that we want. The result is adding one to the product by element of i and t.

Had we instead defined the length of i **not** as a multiple of t, we would have received an error to alert us to an issue. Try this next example.

```
t <- seq(from = 0, to = 6, by = 1)
simple(i, t)
```

To obtain the correct answer, we could use a *for loop*. First delete i and t to clear the workspace so that we start with a clean slate.

```
i <- NULL
t <- NULL
```

7.2 Simple For Loop

Suppose that for a fixed $i = 0.05$, we create a for loop for t ranging from 0 to 6 by 1.

This loop looks like:

```
i <- 0.05
  for (t in 1:6){
  print(simple(i,t))
}
```

With the user-defined function *simple*, we add the *print* function to see the results. For this purpose, the for loop is unnecessary as we can get at the same result by defining a vector t as we did in subsection 6.3 called *User-Defined Functions*.

7.3 Double For Loop

However, to have both i and t vary, we need to use a for loop. Note that the i-loop (the outer loop) is held constant while the t-loop (the inner loop) iterates first completely. The first iteration sets $i = 0$ for values in t, then increase i by 0.01 for all values in t.

```
for (i in seq(0, 0.05, 0.01) ){
  for (t in seq(0, 6, 1)){
  print(simple(i,t))
}}
```

Run each of these loops so that you can see the results.

7.4 Recursive For Loop

There are usually multiple ways of figuring out how to calculate certain items.

For example, consider a different function defined as:

```
compound <- function(i1, t1){
  (1 + i1)^t1
 }
```

Editing the code from subsection 7.2, this loop looks like:

```
i <- 0.05
  for (t in 1:6){
  print(compound(i,t))
}
```

Instead of finding the results in this way, we could have calculated it as follows:

```
i <- 0.05
y <- 1
  for (count in 1:6){
  y <- (1 + i)*y
  print(y)
}
```

The variable *count* in the loop is included as a counter, i.e. the loop iterating 6 times. Mathematically, you may want to show why this approach is equivalent to that previously.

Chapter 8

An Introduction to Plots

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

Plots are a useful way of visually communicating the results of an analysis. The following illustrates how to create basic plots with R and to annotate the plot with a title and legend.

Check out the LinkedIn Learning (formerly lynda.com) videos on *R Essential Statistics* for more on creating plots.

8.1 Basic Plots

This example follows from a previous tutorial about user-defined functions in subsection 6.3.

Suppose you a function called *compound* that has two arguments i and t.

```
compound <- function(i, t){
  (1 + i)^t
}
```

Specifically define $i1 = 0.05$ and t as a sequence of numbers between 0 and 10 inclusive by an increment of 1/4. Let $A1$ be the value of the function multiplied by 1000 at each time increment.

```
i1 <- 0.05

t <- seq(0, 10, 1/4)
```

```
A1 <- 1000*compound(i1, t)
```

We can plot $A1$ vs. t with the *plot()* function:

```
plot(t, A1)
```

Your plot appears in the bottom right corner of RStudio in the *Plots* tab. Note that the order $(t, A1)$ indicates that t is on the $x-axis$ and $A1$ is on the $y-axis$. Check to see what happens if you reverse $(t, A1)$ and plot that result.

All of the 41 values of $A1$ are plotted on the chart, with R automatically deciding on the numbers plotted on the x-axis and y-axis.

8.2 Graphical Parameters

R is very flexible in creating figures. There are different *parameters* that can be set to modify default settings such as plotting symbols, line types, or line widths.

Check out the URL QuickR Graphical Parameters for a more detailed description of the graphical parameters.

One way of seeing everything that is pre-set in R is with:

```
par()
```

Understanding the meaning of all of these symbols is beyond the scope of this chapter. If you are interested, you can find more detailed description by typing **help(par)** in R.

And if you are going to play around with the different settings, then best to start with this statement:

```
opar <- par() # saves default settings (if do first)
```

This way you have saved the default parameters and can reset them by reversing the code.

```
par(opar)  # restores default settings
```

Otherwise, you can always restart R to reset the parameters.

8.3 Improving Plot Display

This basic plot can be improved in several ways to better communicate your results. Instead of showing the plot as separate points, you can show the points as a solid line. You could add a title and add descriptors for the values that are plotted on the axes.

```
plot(t, A1, type = "l", xlab = "Time (in years)",
     ylab = "Accumulated Value (in $)",
     main = "Accumulated Value of $1000 Investment")
```

8.4 Add Lines To Figure

Suppose you want to extend the above analysis to consider three different values of i 0.03, 0.05, and 0.07, and to visually compare the graphs on one plot.

First, you calculate the values:

```
i2 <- 0.03
A2 <- 1000*compound(i2, t)

i3 <- 0.07
A3 <- 1000*compound(i3, t)
```

The following block of code repeats the same plot command as to what you did previously and includes the *lines()* function to add the plots when $i = 0.03, 0.07$. In addition, a legend is included.

```
plot(t, A1,  ylim = c(1000,2000),
     type = "l", xlab = "Time (in years)",
     ylab = "Accumulated Value (in $)",
     main = "Accumulated Value of $1000 Investment", col = "black")
lines(t, A2, type = "l", col = "blue", lty = 2, lwd = 2)
lines(t, A3, type = "l", col = "red", lty = 3, lwd = 3)
legend("bottomright",
       legend = c("3% Interest", "5% Interest", "7% Interest"),
       col = c("blue", "black", "red"),
       lty = c(2, 1, 3), bty = "o", cex = 0.75)
```

The y-axis now ranges from 1000 to 2000. See what happens to the plot if you delete the parameter *ylim = c(1000,2000),* from the plot above.

The lines are distinguished not only by color, but by line type, and line thickness. You can explore which parameters result in the appearance of the plot lines. The *legend()* function associates each line by color and type with the value of *i*. The “bty” option framed the legend, while the “cex” option determined the font size of the legend.

8.5 Multiple Plot Display

One potential helpful parameter setting is the ability to insert multiple plots in one window. The mfrow parameter changes the print setting to allow multiple graphs in one window. Here the mfrow=c(2,2) coding displays a 2x2 table.

The parameter oma stands for outer margin area.

The order of numbers when adjusting the margins is: bottom, left, top, right (clockwise). By setting the third parameter to a "2", we allow for a wider top-margin so that you can put a grand title over all graphs. You can experiment with changes to this setting on your own.

```
par(mfrow=c(2,2), oma = c(0,0,2,0)) # puts 4 plots in one window (2x2)

plot(t, A1, type = "l", xlab = "Time (in years)",
     ylab = "AV  (i = 0.05)", ylim = c(1000, 2500),
     main = "Accumulated Value \nof $1000 Investment")

plot(t, A2, type = "l", xlab = "Time (in years)",
     ylab = "AV  (i = 0.03)", ylim = c(1000, 2500),
     main = "Accumulated Value \nof $1000 Investment")

plot(t, A3, type = "l", xlab = "Time (in years)",
     ylab = "AV (i = 0.07)", ylim = c(1000, 2500),
     main = "Accumulated Value \nof $1000 Investment")

mtext("A Grand Title", outer=TRUE, cex = 1.5, col="olivedrab")
```

Notice that the plot titles have been printed on two lines, as well as the y-axes shown with common values.

Part II

Data

Chapter 9

Importing External Data

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

For any data analysis, we need data. We could manually type the data into R, but that would be time-consuming and fraught with errors. And if the data already exist in an electronic format, then why would we spend time typing?

Data come in all different formats. The data could be readable, sometimes called *ascii* format. Or the data could be unreadable without the original program, like an Excel workbook (*.xlsx) or other statistical software formats like Stata (*.dta) or SAS (*.sas7bdat).

As a note, RStudio continually is updating its software. Thus there may be other ways to import data.

9.1 Importing Dataset Tab

First, as with many things in R, there are many ways of bringing data into your workspace.

A flexible way to import data is to click on the *Environment* tab in the upper right window of RStudio and then click the *Import Dataset* tab. Multiple file type options are shown, such as *text*, Excel, SPSS, SAS, and Stata.

There are two options for the *text* option that rely on different R functions to bring the data into the workspace. Both of these options work very similarly.

For either option, you can browse to where your file is located on your computer and open it.

The *key* items that you will need pay attention to are the *column separators*, i.e. how the software knows where to split columns and whether the data include column headings.

Data with an *.csv extension refers to *comma separated values.* You could open a file in a program like Notepad and be able to read the data. The columns are separated with commas. For these **.csv* files, you choose *comma* for the *deliminator*. This concept is the same as in Excel for the function *Text to Columns* under the *Data* tab.

The default program on your computer that is attached to files with *.csv extensions is Excel. If you double-clicked on the file to open it in Excel, Excel would separate the data into columns as defined by where the commas are located. The *.csv file is created in Excel using a *Save As* command in the File menu. Note that only one worksheet at a time can be saved as a *.csv file.

The second key item is to choose whether there are column headings in the top row. In the context of reading data, these column headings would mean the name of the variables.

- Once the data are read successfully, your *Global Environment* space displays the name of the data frame, the number of observations, and the number of variables. You can type *ls()* in your *Console* to see what is now present.
- You can see the commands needed to bring the file into the workspace in the *History* tab on the top right in RStudio.
- You can review some of the features of the *Global Environment* space in Section 2.

9.2 Directly Reading CSV Files

You can create comma-separated-value (CSV) data files from Excel. For a particular worksheet, you select File > Save As and after you select the folder as to where to save the data, you select the CSV option for the *Save File Type* that is under your choice of file name. Note that the file extension of these data will be *.csv, not* .xlsx.

A direct way of importing your data that are in a CSV format is with the following command:

```
dat <- read.csv("your.path/filename.csv", header=TRUE)
```

Here you would substitute the location of your file (i.e. the file directory where it is stored), as well as the particular file name. Note that to reference a file in the path, you use either one forward slash (/) as above, or two backward slashes (\\).

For those working on a Mac, you can start your file path with a tilde (~) that takes the place of your basic user information. For example, you can enter one of the two statements to load MYMEPS.csv in R (if the MYMEPS.csv file is in your Downloads folder):

```
MEPS <-read.csv("~/Downloads/MyMEPS.csv", header=TRUE)
MEPS <- read.csv("C://Macintosh/Users/YourName/Downloads/MyMEPS.csv",
        header=TRUE)
```

The <- is called an assignment operator. The information on the right-hand side is assigned to the object with the name on the left-hand side. Be sure that there is NO space between the < and the - .

In this example, MEPS, the left-hand-side of the <-, is what is called a *data frame*. More specifics about what is a data frame and how to manipulate it, is discussed in Section 10.

9.3 Read RData Files

After reading in the raw data, as in a csv file, you do work, like creating new variables or modifying the ones that you have. While you can recreate this work by re-running your code, it is much easier to save your workspace in a *.RData file, especially if you have made a lot of changes/additions to the raw data.

The *workspace* represents your work. The *.RData file has the original data plus any changes that you made.

The easiest way to load the data into R is to double-click on the particular file *yourfile.RData* after you download it to your computer. This will open in RStudio only if you have associated the .RData files with RStudio. Otherwise the file will open in R.

To change your file associations, please see Change File Associations in Windows or Change File Associations in Mac.

You may want to set your working directory to where you stored the RData file..

These *.RData files are read into R with a *load()* statement, rather than a *read* statement.

```
load("yourdirectory/someworkspace.RData")
```

If you type ls() again, you can see that your data frame is showing up as in the workspace. The information can also be seen in the *Environment* Window.

9.4 RStudio Helpful Setting

You might check out the RStudio interface for features that could help you. One such setting that may be beneficial is the ability to have a default starting working directory when RStudio starts. You find this setting: Tools > Global Options > General tab in the menu bar in RStudio.

You can also adjust this working directory under the *Session* tab in RStudio and clicking on the *Set Working Directory* and browse to the directory you want to set. You can verify the location with the getwd() to verify your selection.

Of course, you can always set your working directory with the setwd(name your directory) function if you want it to be set to something else as also discussed in Section 2.1.

```
setwd("where you want to set the working directory")
getwd()  # to check where the working directory is set
```

Chapter 10

Data Frames

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

This chapter assumes that you have data in your workspace in R.

Data Frames are objects that contain data. The data can be a mix of numerical, categorical and text. This chapter introduces you to the basic features of a data frame and some basic manipulations of the data.

See Section 12.1 for information about including different data types in R.

10.1 Simple Data Frame

To allow a common example for illustration purposes, suppose we have three sequences of numbers, *time*, *t*, and *accum* that are in separate lists. The first command removes all objects in your workspace so that you start with a clean slate.

```
rm(list = ls())

i <- 0.05
t <- seq(from = 0, to = 5, by = 0.5)
accum <- (1 + i)^t
time <- c("Zero", "One Half", "One", "One+Half", "Two",
          "Two+Half", "Three", "Three+Half", "Four", "Four+Half", "Five")
```

After you run the code, check in the *Global Environment Window* to see that 4 values are listed and how they are listed. Note that *t* and *accum* are shown as numeric vectors, while *time* is shown as a character vector. You can check this with the following code:

```
is.vector(t)
is.vector(accum)
is.vector(time)
```

To combine these vectors together, we use the function *data.frame()* and name the data frame *dat*. You can view the data frame either by typing *dat* in the script/console window as indicated in the following code.

```
dat <- data.frame(time, t, accum)
```

```
dat
```

The data frame *dat* is shown in the *Global Environment Window* with 11 observations and 3 variables. You can click on the blue circle next to *dat* to see a listing of the variables in the data frame and their data type. If you double-click on *dat* you can view the data frame.

This approach is great for small data sets. For large data frames, i.e. with a large number of rows and/or a large number of variables, we need other tools to help navigate the information in the data frame object.

10.2 Data Frame Features

- The str() command is useful to see a listing of the variables and their variable types in the console.
- The nrow() command shows the length of the data frame.
- The ncol() command shows the number of columns in the data frame.
- The rownames() or row.names() functions indicate the names of the rows. The quotes around the numbers show that the *row names* are characters, rather than an actual number.
- The colnames() or names() functions indicate the names of the columns. Again, the quotes indicate that the names are character variables.
- The *typeof()* function allows us to know what is the object type.
- The *class()* function describes what is the object.
- The *attributes()* function summarizes the column names, the row names, and the class of the object.

Using the data frame *dat* created in subsection 10.1, you can see the output produced from each of the statements below.

```
str(dat)
nrow(dat)
ncol(dat)
rownames(dat)
row.names(dat)
colnames(dat)
names(dat)
typeof(dat)
class(dat)
attributes(dat)
```

You can view the the top 6 rows by typing *head(dat)* or the bottom 6 rows by typing *tail(dat)*.

```
head(dat)
tail(dat)
```

If your data frame is really large, either a lot of variables or observations, remember the options command to expand the printing in the *Console Window* that was introduced in subsection 2.5.

```
options(maxprint = 1000000)
```

10.3 Refer/Create Specific Variable

We can reference a particular variable (column) in the data frame with the following notation: *dataframe_name$variable_name*. The $ sign is a separator to distinguish the name of the data frame and the variable name.

Typing *mean(dat$t)* indicates the average of *t*.

```
mean(dat$t)
```

You can use the same syntax to create a new variable and add it directly to the data frame. The key is that the length of the variable needs to be the same length as the data frame itself.

For example, the following code successfully adds *ObsNum* to the data frame:

```
dat$ObsNum <- 1:nrow(dat)
```

You can see in the *Environment Window* that the number of variables has increased by one and you can view the edited data frame by typing *dat*.

The following code produces an error when trying to add *oops* to the data frame:

```
dat$oops <- 1:2
```

10.4 Subset the Data

Many times you have a lot of data and for a particular analysis you do not need the entire data frame.

(1) Instead of working with the entire data set, we can take just a few rows or a few columns from the data frame that we created in Section 10.1. Suppose we create a new object *dat1* from our data frame *dat* with the following code.

```
dat1 <- dat[1:3, ]
dat1
```

By doing this, you create a new data frame object that is based on our original data.

What data does *dat1* represent? Data frames are represented by rows and columns as we saw above. The code above results in copying the first 3 rows of *dat* and pasting it in the new data frame *dat1*. See how all of the variables are copied to dat1. Note now that the *Global Environment* window shows the new data frame is created, now with only 3 observations and all of the original variables.

(2) We could also take just a few of the columns using the following code. Note that there is a comma inside of the left bracket.

```
dat2 <- dat1[ , 2:3]
dat2
```

For those familiar with matrices, this notation of [r,c] is in the format of rows and columns. Entries before the comma represent the rows of the data frame, while entries after the comma represent the columns.

For larger data frames this approach could be troublesome. Instead you can take just a few of the columns using their column names:

```
dat3 <- dat1[ ,  c("t","accum") ]
dat3
```

Here we use quotes around the variable names (remember that R is picky for upper and lower case, as well as spaces), and we use the c() notation that operates as a combine operator.

Be especially careful with the ””. If you copy code from the tutorial, you may end up with *smart quotes* (where they might show as curly). If your code does not work if you copy the code, try re-entering the quotes so that they are really straight quotes and not curly quotes. Check out smart quotes vs curly quotes to see the different looks of quotation marks.

(3) Another way of subsetting the data frame to keep only a specific group of variables is:

```
keep <- c("t","accum")
keep

dat4 <- dat[1:3, keep]
dat4
```

Here we create a new object *keep* that contains just the names of the two variables we want to retain. Then we use that *keep* object in the subsetting. When subsetting a large number of variables this approach is useful as it helps keep the coding clear and well-documented.

(4) Or we might want to keep the rows of the data frame that satisfy a particular criterion:

```
dat5 <- dat[which(dat$t > 2), ]
dat5
```

What you notice after you use *which* is that your data frame (dat5) is smaller and only contains row where $t > 2$. You can have more complicated logical expressions by using the AND (&) and the OR (|) logical operators. An example is presented in Section 12.4.

10.5 Draw a Random Sample

In certain analyses, you may want to take a random sample from your original data.

Again, let us remind ourselves of the data in *dat* from Section 10.1 with 11 observations and 3 variables.

```
         time   t    accum
1        Zero 0.0 1.000000
2    One Half 0.5 1.024695
3         One 1.0 1.050000
4    One+Half 1.5 1.075930
5         Two 2.0 1.102500
6    Two+Half 2.5 1.129726
7       Three 3.0 1.157625
8  Three+Half 3.5 1.186213
9        Four 4.0 1.215506
10  Four+Half 4.5 1.245523
11       Five 5.0 1.276282
```

10.5.1 Without Replacement

Suppose we want to create a new data frame that randomly selects a unique row each with each draw. Thus, once a row is selected, that data are not permitted to be sampled again. This type of sampling is called *without replacement*.

To sample five rows without replacement from *dat* we use the following command:

```
dat.wo <- dat[sample(nrow(dat), size = 5, replace = FALSE), ]
dat.wo
```

Take a look at your new data frame *dat.wo*. You have 5 rows but note that they are not ordered by time as in the original *dat*.

The syntax for the function *sample()* is examines the length of *dat* and randomly samples row numbers. The rows associated with the sampled row numbers are retained in the new data frame. These row numbers are in the *r* part of the [r, c] of the data frame. Note that the code is before the comma. The coding of this sampling mechanism is similar to that when we took a small subset of the data in Section 10.4. Also the *replace = FALSE* part of the syntax indicates to sample without replacement.

10.5.2 With Replacement

Suppose we want to create a new data frame that randomly selects a new row each with each draw, but this time we do not care if we repeat rows selected previously. This type of sampling is called *with replacement*.

To sample five rows with replacement from *dat* we use the following command:

```
dat.with <- dat[sample(nrow(dat), size = 5, replace = TRUE), ]
dat.with
```

Take a look at your new data frame *dat.with*. You have 5 rows but note that they are not ordered by time as in the original *dat*, as the rows are sampled at random.

Here we could use the same coding as in Section 10.5.1 and just change the *replace = FALSE* to *replace = TRUE*.

The statistical technique called *bootstrapping* uses random sampling with replacement.

Chapter 11

Writing to .CSV File

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

This chapter assumes that you have data in your workspace in R.

Once you create data frames, you may want to save the output to an external file. One easy way to do save the data is with a write.csv() command.

11.1 Recreate Data

To provide us something to save, let us recreate the simple example shown in Section 10.1.

```
rm(list = ls())

i <- 0.05
t <- seq(from = 0, to = 5, by = 0.5)
accum <- (1 + i)^t
time <- c("Zero", "One Half", "One", "One+Half", "Two",
          "Two+Half", "Three", "Three+Half", "Four", "Four+Half", "Five")
dat <- data.frame(time, t, accum)
```

After you run the code, check in the *Global Environment Window* to see that 4 variables and the data frame *dat*.

11.2 Save dat

You can easily save *dat* using the code shown below.

```
write.csv(dat, file = "where_do_you_want_to_save_it/filename")
```

The *where_do_you_want_to_save_it* indicates the directory to which you want to write the file and *filename* is the name of the new output. Note that if you omit including a directory, the file will be written to your current working directory. You can find it with the command getwd().

Chapter 12

Variable Types

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

This chapter uses data that is part of the R Core.

Data are integral to any analysis. The data can be a mix of numerical, categorical, and text. This chapter introduces you to the basic features of a data types and how you might consider including data in an analysis.

This chapter incorporates the use of a data frame. See Section 10 for information about data frames.

12.1 Types of Data

For any analytics study, you need data. You can think of data as:

- Numerical (quantative)
- Categorical (qualitative)
- Textual (quantitative & qualitative)

Numerical data can be integer-based (discrete) or real numbers (continuous), such as Age, Income, or Education (in years).

Categorical data are represented by *buckets*. These categorical data are either nominal, like Employment Status, Marital Status, or Occupation, or ordinal such as student course letter grades.

We could define Employment Status as:

$$\textit{Employment Status} = \begin{cases} 1 & \text{If Employed} \\ 2 & \text{If Job Hunting} \\ 3 & \text{If Not Looking For Work} \end{cases}$$

The placeholders for Employed (1), If Job Hunting (2), and If Not Looking For Work (3) are not meaningful of themselves, but define the bucket numerical placeholders to code the data. We could just have easily labelled the variable Employment Status as:

$$\textit{Employment Status} = \begin{cases} 7 & \text{If Employed} \\ 18 & \text{If Job Hunting} \\ 33 & \text{If Not Looking For Work} \end{cases}$$

In certain settings it is beneficial to convert numerical data such as Age, Income or Education (in years) to a categorical variable. For Education, we could define the number of years in school as:

$$\textit{Education} = \begin{cases} 1 & \text{If Less Than High School} \\ 2 & \text{If High School Degree} \\ 3 & \text{If More Than High School Degree} \end{cases}$$

Of course, there is not a limit on the number of categories. You could have grouped the Education variable in other ways as you see appropriate.

A special case of a categorical variable is an indicator variable, sometimes referred to as a binary or dummy variable. Here we could define Employment Status as simply *Currently Employed*:

$$\textit{Currently Employed} = \begin{cases} 1 & \text{If Employed} \\ 0 & \text{Otherwise} \end{cases}$$

This approach collapsed the two categories *If Job Hunting* and *If Not Looking For Work* into one category. Careful consideration when collapsing categories is needed, as you want the categories to be what we say *homogeneous* (or similar) so that your results are valid.

In this case, you could use *Currently Employed* as either a categorical or numeric variable.

Textual Data refer to data that are collected from writings or electronic databases. Methods concerning the mining of text data are beyond the scope of this book.

12.2 Example Data

The following data are included in the R Core will be used as illustrative of the concepts in this chapter. This chapter assumes that you know about data frames introduced in Section 10.

Some of the data are combined into a data frame, called *dat*, for easier viewing.

```
 rm(list=ls())
?state # documentation for data
```

```
data(state) # loads state data into the workspace

dat <- data.frame(state.abb,state.division,state.region,state.x77)
str(dat)
```

```
'data.frame':   50 obs. of  11 variables:
 $ state.abb     : chr  "AL" "AK" "AZ" "AR" ...
 $ state.division: Factor w/ 9 levels "New England",..: 4 9 8 5 9 8 1 3 3 3 ...
 $ state.region  : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
 $ Population    : num  3615 365 2212 2110 21198 ...
 $ Income        : num  3624 6315 4530 3378 5114 ...
 $ Illiteracy    : num  2.1 1.5 1.8 1.9 1.1 0.7 1.1 0.9 1.3 2 ...
 $ Life.Exp      : num  69 69.3 70.5 70.7 71.7 ...
 $ Murder        : num  15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 10.7 13.9 ...
 $ HS.Grad       : num  41.3 66.7 58.1 39.9 62.6 63.9 56 54.6 52.6 40.6 ...
 $ Frost         : num  20 152 15 65 20 166 139 103 11 60 ...
 $ Area          : num  50708 566432 113417 51945 156361 ...
```

The output of the *str()* function of *dat* for the state data shows a *num* for numerical quantities. Variables from Population to Area in *dat* are numerical. Population or Income, for example, are shown as integer variables, while Illiteracy and Life Expectancy are continuous variables.

The variables in *dat* for state.abb, state.division, and state.region are categorical variables and shown as *factors* in the *str()*.

12.3 Summarize Data

We can summarize numerical data by examining their:

- Means
- Modes
- Medians
- Percentiles
- Ranges
- Variances/Standard Deviations

We can summarize categorical data by examining their:

- Frequencies
- Proportions

Using our example data frame, *dat*, from Section 12.2 we can summarize the data with the *summary()* function.

```
    summary(dat)
```

Note that the function automatically creates statistics appropriate for the type of the data. For numerical data, the minimum, 25% percentile, median (50% percentile), mean (average), 75% percentile, and maximum are shown. For categorical data, the frequencies are shown, although limited to 7 lines with the last being *Other*.

Note that using the summary() function does not show the variance or the standard deviation.

A nice package to obtain descriptive statistics is the *psych* package. Refer to Sections 3.1 and 3.2 for details on installing and loading packages. To use this package, be sure that it is listed in the *Packages* tab in the lower right-hand window of RStudio. If not shown, then click *Install* and follow the instructions in 3.1.

```
library(psych)
describe(dat)
```

Now we see the standard deviation (sd), as well as a number of other statistics that are calculated. The psych package calculates statistics (meaningless) for the categorical variables, but does include an (*) to indicate that they are categorical.

12.4 Subset Data Revisited

As suggestion in Section 10.4, you might be interested in only having certain observations in a new data set.

Using the state data, we could see which states satisfy both of the following conditions:

- Illiteracy is less than 1.2; and
- Murder is greater than 7.5

```
dat.subset <- dat[which(dat$Illiteracy < 1.2 & dat$Murder > 7.5), ]
dat.subset
```

You can check your results to see that all of the observations meet the appropriate criteria.

If instead of an *AND* criteria, we had a *OR* criteria such as: - South region; or - Murder is less than 7

```r
dat.subset <- dat[which(dat$state.region == "South" | dat$Murder < 7), ]
dat.subset
```

Again, you can check your results to see that at least one of the criteria are satisfied. Where the murder rate is not less than 5, then the region must be south. You can verify that all southern states are included plus those whose murder rates are less than 5. Do you see any where both conditions are satisfied? (The answer is yes, e.g. Delaware.)

12.5 Convert Numerical Data to Categorical

Suppose that you wanted to use the *Income* variable as a categorical variable instead of a numerical variable. We would need to define how we want to parse the data into buckets. The first decision is to decide the number of buckets. The second decision is to decide how to allocate the data into the buckets.

For the *Education* variable example in Section 12.1, we chose three buckets, but also suggested that more (or less) could be completed.

For *Income*, one way would be to create equally sized buckets of some number of buckets. Another way is to examine the distribution and decide on reasonable *split* points (sometimes called *cut* points).

We could plot a histogram of the data, we could do a summary of the data (as in 12.3), or since the data set is small, we could order the data from smallest to largest.

```r
hist(dat$Income)
summary(dat$Income)
sort(dat$income)
```

To define the new categorical variable we use the following code:

```r
dat <- within(dat, {
  Income.cat <- NA # need to initialize variable
  Income.cat[Income < 4000] <- "Low"
  Income.cat[Income >= 4000 & Income < 5000] <- "Middle"
  Income.cat[Income >= 5000] <- "High"
   } )
```

This code defines the new categorical income variable *Income.cat* and automatically includes the new variable in the data frame (*dat*). The cut-points are set so that the median is in the middle of the *Middle* category. You could stop with this code and feel good.

However note in the code that follows. Income.cat is shown as a *chr*, or *character* variable. And the results of the summary() function are not meaningful.

```
str(dat)
summary(dat$Income.cat)
```

The next line defines *Income.cat* as a *factor* variable and sets the ordering of the buckets with the *levels()* parameter. The category noted first is called the **Reference category**. This ordering is important for some analytic methods that you will complete.

You observe now that the results reflect Income.cat as a factor variable.

```
dat$Income.cat <- factor(dat$Income.cat, levels = c("High", "Middle", "Low"))

str(dat)

summary(dat$Income.cat)
```

12.6 Indicator Variables

A special case of a variable with multiple categories is an *indicator* variable. These variables are sometimes referred to also as *binary* or *dummy* variables.

You can think of these variables with just two categories: 0 and 1. Usually the level of 1 is reserved for the characteristic of interest.

For example, you may want to indicate those with incomes higher than average. For the states data, you could do this multiple ways.

- One way is the following:

```
dat$HighIncome <- 1*(dat$Income > mean(dat$Income))
head(dat)
```

Listing the first few lines of the data show a 1 for Alaska, Arizone, California, and Colorado, and a 0 for Alabama and Arkansas. With a mean of 4435.8 we can compare the income for the state with the indicator.

- A second way is the following:

```
dat$HighIncome1 <- ifelse(dat$Income > mean(dat$Income), 1, 0)
head(dat)
```

Part III

Specific Topics

Chapter 13

Simulation Basics

Reminders: R is picky in its notation; you need to distinguish between upper/lower case. There are many ways of doing the same thing in R.

Type the code provided below in a R script to see the results. Typing is better than copying the code, as you might make a mistake in typing that allows you to learn how to debug your coding.

Simulation is a method used to examine the "what if" without having *real* data. We just make it up! We can use pre-programmed functions in R to simulate data from different probability distributions or we can design our own functions to simulate data from distributions not available in R.

This chapter illustrates a few built-in functions to generate values from particular probability distributions. Methods to generate these values are beyond the scope of this chapter, as well as simulating values using methods other than with probability distributions.

13.1 Introduction

When we **do** a simulation, we have to make many assumptions. One major assumption is the choice of the distribution to use for a particular variable. Each particular distribution has *parameters* that are integral to generating data from the distribution. We need to *set* a value for these parameters to simulate a value from a distribution.

How we decide that choice of distribution and the manner in which the values of the parameters are to be estimated are beyond the scope of this chapter.

Given a particular distribution and known parameters, we can generate *values* from that distribution. However, in reality, we never know the *true* distribution, and we never know the exact parameter values needed to generate values from that distribution. Practical statistical analysis helps us to identify a good choice of the distribution and estimate reasonable parameters.

We can think of the simulated values as a sample of a larger population. Samples with a large number of values will better reflect the properties of the distribution of the larger population.

In what follows, examples are shown to generate values from a normal, Bernoulli, Poisson, and Gamma. The last subsection shows how to combine simulating first from a Poisson and then from Gamma distribution to illustrate how more complex processes can be simulated simply.

13.2 Normal Distribution

The normal distribution is used if the variable is continuous. We usually refer to the density of a normal random variable as a *bell-shaped* curve. We require a value for the mean and another for the standard deviation to simulate a value from a normal distribution.(The mean and standard deviation (or variance) are the parameters of a normal distribution.)

We can easily simulate 1000 values from a normal distribution with a mean of 10 and a standard deviation of 4 as follows:

```
x <- rnorm(n = 1000, mean = 10, sd = 4)
```

Notice that notation of $n =$ (number of values to be simulated), $mean =$, and $sd =$. If we plot a histogram, we can see a somewhat *bell-shaped* curve.

```
hist(x)
```

We can check what are the mean and standard deviation of the sample values. We could start with the summary() function:

```
summary(x)
```

Note in the results that the mean is shown. While the mean not equal 10, it is *close* to 10. Samples of variables will never exactly be equal to the parameters used (in this case the mean and standard deviation). The larger the sample, the closer the simulated sample values will be to those set in the rnorm() function.

The standard deviation is not shown. We can use the sd() function or load the psych package.

```
sd(x)

library(psych)
describe(x)
```

Again we see that the standard deviation is *close* to the parameter of 4.

Simulation is a nice tool as you can re-do everything and get different samples. This way you can see how quantities (like the mean, standard deviation, histogram) vary from one sample to another, even though they were generated from the same underlying distribution.

You can change the sample size, or mean, or standard deviation. Plotting the values helps you *see* the data. See what happens if we just change the sample size to 10, instead of 1000.

```
x <- rnorm(n = 10, mean = 10, sd = 4)
describe(x)
hist(x)
```

For more depth on choosing the number of values to simulate, see your favorite statistics book.

13.3 Setting the Seed

Technically, when we simulate from a particular distribution, we are using a pre-programmed function to generate the numbers. Thus, we may call these pseudo-random numbers.

Because there is a recursive function behind the generation of numbers, we can control the sequence of the values generated. It might be that you want to be able to reproduce the simulation results at another time.

To do this, we initialize the *seed* of the data-generating process:

```
set.seed(1)
```

Then generate 10 values from a normal distribution with mean of 10 and standard deviation of 4.

```
x <- rnorm(n = 10, mean = 10, sd = 4)
describe(x)
```

And then, if we do this again:

```
set.seed(1)
y <- rnorm(n = 10, mean = 10, sd = 4)
describe(y)
```

Then we get the same values again. You can absolutely verify this by printing the values of x and y:

```
x
y
```

Note that you need to re-set the seed so that the recursive function starts at the same place to generate values. If you do not re-set the seed, the recursive function to generate values continues from where it left off.

13.4 Indicator (Bernoulli) Variables

A special case of a categorical variable is an indicator variable, sometimes referred to as a binary or dummy variable. The underlying distribution of an indicator variable is called a *Bernoulli* distribution.

Suppose we are interested in evaluating the whether a flip of a coin would be a head or a tail. Here we could define *Head* as the variable of interest.

$$Head = \begin{cases} 1 & \text{If Flip a Head} \\ 0 & \text{If Flip a Tail} \end{cases}$$

If you are more interested in evaluating *Tails*, you could define the random variable as:

$$Tail = \begin{cases} 1 & \text{If Flip a Tail} \\ 0 & \text{If Flip a Head} \end{cases}$$

We can simulate this random variable using a Binomial distribution. (Technically, the Bernoulli distribution is a special case of a Binomial.) We need to set values for $n =$, *size =* , and *prob =* , where n is the number of values you want to simulate, *size* in this case is 1 (as we want to simulate an indicator variable), and *prob*, is the probability that you will flip a head (or tail, depending on your random variable).

Simulating indicator variables is completed using the *rbinom* function. Here, we simulate 5 values of heads with a probability of 1/2 of getting a head on each flip (or a *fair* coin).

```r
rbinom(n = 6, size=1, prob=0.5)
```

If you are simulating 6 coin flips using a fair coin, how many heads do you expect? What did you get?

Re-run the above code several times to see how the sequence of 6 coin flips varies.

13.5 Poisson Variables (Optional)

Another discrete distribution that you may learn is called the *Poisson* distribution that is used to predict counts of some event that occur within a given time interval. For example, recording the number of car accidents that you have in a year.

As you would guess, there is a R function for simulating this random variable. Here in addition to the number of values to simulate, we just need the parameter for the mean (called *lambda*).

```r
rpois(n = 6, lambda = 20)
```

13.6 Gamma Variables (Optional)

Another continuous distribution that you may learn is called the *Gamma* distribution. This distribution is used for random variables that have some skewness and is not symmetrical, like the Normal Distribution.

The Gamma distribution requires a little more background to understand how to define the parameters.

There is a R function for simulating this random variable. Here in addition to the number of values to simulate, we just need two parameters, one for the *shape* and one for **either** the *rate* or the *scale*. The rate is the inverse of the scale. The general formula is: rgamma(n, shape, rate = 1, scale = 1/rate).

Given that α is the shape parameter and β is the rate or scale parameter, then if you are thinking of a Gamma random variable, where the mean $= \alpha * \beta$, then you use the *scale* for simulating. Otherwise, you use the *rate*.

We can verify these concepts with the following code.

```r
set.seed(1)
rgamma(n = 5, shape = 3, scale = 2)

set.seed(1)
rgamma(n = 5, shape = 3, rate = 0.5)
```

And verifying the mean equal to 6 (shape*scale):

```r
set.seed(1)
x <- rgamma(n = 1000, shape = 3, scale = 2)
mean(x)
```

Or verifying the mean equal to 6 (shape*1/rate):

```r
set.seed(1)
x <- rgamma(n = 1000, shape = 3, rate = 0.5)
mean(x)
```

13.7 Compound Distribution (Optional)

An example of where simulation is useful is shown below. Suppose you have one variable, X that is assumed to have a Poisson distribution with *lambda = 3* and another random variable Y that is assumed to have a Gamma distribution with *shape = 3* and *rate = 0.5*.

A practical story where this scenario might be useful is that we want to explore the annual sum of costs of individuals with car accidents. The number of annual car accidents is Poisson and the cost per car accident is Gamma.

The code below contains a for-loop as discussed in Section 7.4 and data frames as discussed in Section 10.

The example assumes that there are 6 people to consider. The data frame is initialized in the first block of code where one Poisson is simulated for the number of accidents, Gamma values are generated for the number of accidents, and then summed over the total. The number of accidents and the sum is written to a data frame. A second block of code contains a for loop that does the same thing, but adds the results to the already created data frame.

The interim values x, y, z are printed so that you can verify the process and the final results.

```
  x <- rpois(n = 1, lambda = 3)
  y <- rgamma(n = x, shape = 3, rate = 0.5)
  z <- sum(y)
  print(x)
  print(y)
  print(z)
  dat <- data.frame(x,z)
  dat

for (count in 1:5){
  x <- rpois(n = 1, lambda = 3)
  y <- rgamma(n = x, shape = 3, rate = 0.5)
  z <- sum(y)
  print(x)
  print(y)
  print(z)
  dat[nrow(dat) + 1,] <- list(x, z)
        }
```

}

If the simulation is completed with more iterations, we can plot the new simulated distribution, along with some summary statistics.

```
  x <- rpois(n = 1, lambda = 3)
  y <- rgamma(n = x, shape = 3, rate = 0.5)
  z <- sum(y)
  dat1 <- data.frame(x,z)


for (count in 1:999){
  x <- rpois(n = 1, lambda = 3)
  y <- rgamma(n = x, shape = 3, rate = 0.5)
  z <- sum(y)
  dat1[nrow(dat1) + 1,] <- list(x, z)
}

  hist(dat1$z)
  describe(dat1)
```