

Statistical Programming with Case Studies in R

Brian Muse¹ and Charles Carter²

August 8, 2016

¹muse.william@columbusstate.edu

²carter.charles@columbusstate.edu

DRAFT

Contents

| | |
|--|-----------|
| Preface | v |
| 1 An Introduction to the Research Process | 1 |
| 2 Computer-Assisted Analysis | 3 |
| 2.1 The R programming language | 4 |
| 2.1.1 Invoking and quitting R | 4 |
| 2.1.2 Directory and file operations | 6 |
| 2.1.3 Data structures and types | 8 |
| 2.2 R packages | 12 |
| 2.2.1 CRAN | 12 |
| 2.2.2 knitr | 12 |
| 2.2.3 ggplot2 | 12 |
| 2.2.4 dplyr | 12 |
| 2.3 Example analyses | 12 |
| 2.3.1 Running scripts | 12 |
| 2.3.2 Data entry | 15 |
| 2.3.3 Importing data | 15 |
| 3 Descriptive Statistics | 17 |
| A Installing and Running R() | 19 |
| B R Integrated Development Environments | 21 |
| Afterword | 23 |

DRAFT

Preface

this is the preface!

DRAFT

DRAFT

Chapter 1

An Introduction to the Research Process

this is chapter one!

DRAFT

DRAFT

Chapter 2

Computer-Assisted Analysis

Why do we need computer assisted analysis? In order to answer this question, we need to consider both what a computer does, and what a programmer does.

A computer computes, or more elegantly, a computer is just a computational machine, but an immensely powerful one. To illustrate its power, assume that we have in our employ a number of super mathematicians. Assume that one mathematician can perform one computation a second, 60 computations a minute, 60 minutes an hour, for 2000 hours a year, and that our mathematician is never tired, sick, in meetings, or needs professional time. How many computations can our mathematician do in a year?

```
#computations per second
number.of.computations <- 1
#computations per minute
number.of.computations <- number.of.computations * 60
#computations per hour
number.of.computations <- number.of.computations * 60
#computations per year
number.of.computations <- number.of.computations * 2000
# our mathematician is good for
number.of.computations

## [1] 7200000
```

Now, let's say that our computer is just an ordinary machine, nothing special, and that the processor runs at a clock speed of 2 GHz, or two billion computations a second. How many mathematicians do we need to equal the speed of one ordinary computer?

```
number.of.mathematicians <- 2000000000 / number.of.computations  
number.of.mathematicians  
  
## [1] 277.7778
```

As you can see, it takes 277.777778 mathematicians working for a year to perform the same number of computations that a computer can do in one second! This is amazing power, and leads us to the second question, what does a programmer do? What a programmer does is ... think. A computer is dumb, all it knows how to do is compute. A human, otherwise known as a programmer, tells the computer what to compute. Currently, we have at our fingertips incredible amounts of data. Ordinary computers have storage capacity of 100 gigabytes, and storage capacities of 1000 gigabytes are relatively common. Research computers can store terabytes (10^{12} bytes), petabytes (10^{15} bytes), and exabytes (10^{18} bytes) of data. For these amounts of data, computer assisted analysis is an absolute prerequisite.

This chapter introduces the R programming language, perhaps the most widely used computer assisted analytical software in the world.

Exercise: Name two or three commonly used technologies for data analytics.

2.1 The R programming language

2.1.1 Invoking and quitting R

This chapter assumes that you have R up and running. If you do not, please read appendix A on page 19. Details of invocation differ according to the operating system you use. This tutorial assumes that you use some current version of Microsoft Windows. If you have a shortcut on our desktop, you invoke R by opening the shortcut. You can also invoke R via the start menu. If you choose to use an integrated development (IDE), you invoke R by opening your IDE, see appendix B on page 21. If you open R program on Windows, you will see the R graphical user interface (GUI), a bare bones command interpreter, shown in figure 2.1. It's also possible to invoke R on the command line interface (the DOS or Powershell prompt), but we will not cover this.

You quit R by typing the command `q()` at the R prompt. You will see an alert box asking you whether you wish to save. You should always save your image unless you have a good reason not to. See figure 2.2. You may also give the command `q('yes')` to directly save the image, or `q('no')` to quit without saving the image. On Windows, if you save the image, R creates a file in your working directory named `.RData`. You can invoke R by clicking on this link in the normal fashion, and your previous workspace will be restored. See figure 2.3.

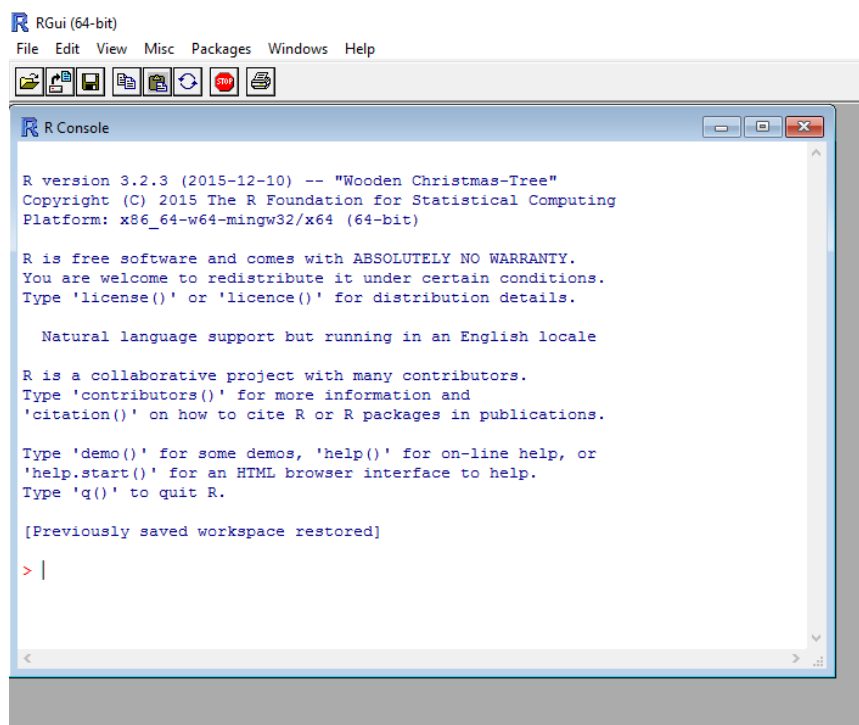


Figure 2.1: R GUI

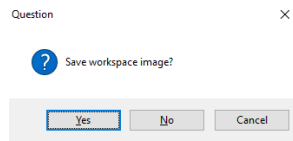


Figure 2.2: R quit alert

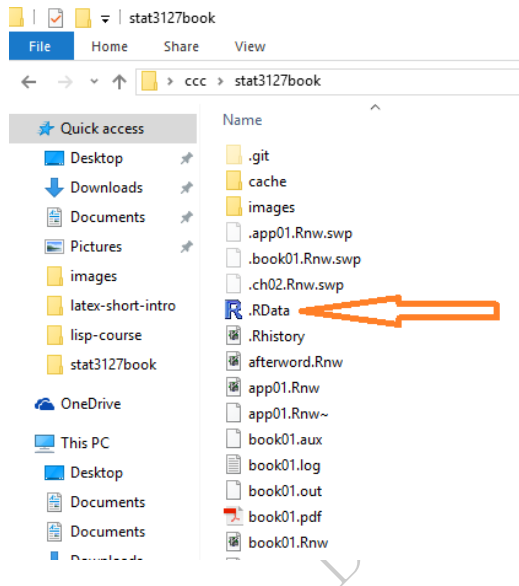


Figure 2.3: R.data file

Exercise: Invoke R in the usual manner, then quit R saving the workspace, and invoke R by clicking on the `.RData` file icon.

2.1.2 Directory and file operations

We will begin a tour of R with directory and file operations. These allow you to use R as a command shell, which may come in handy occasionally, however, you will mostly use your operating system's file and directory manipulation facilities. Here are the commands you should run.

```

1 #get help for getwd()
2 ?getwd
3 #list the current working directory
4 getwd()
5 #create a new child directory named 'testing'
6 dir.create("testing")
7 #change to the testing child directory
8 setwd("testing")
9 #create a numeric vector
10 num.vec <- c(1,2,3,4,5)

```

```

11 #print the num.vec
12 num.vec
13 #save the vector as a CSV file
14 write.csv(num.vec, "my-num-vec.csv")
15 #copy the file
16 file.copy("my-num-vec.csv", "another-num-vec.csv")
17 #list the directory contents
18 dir()
19 #delete the files you just created
20 file.remove("my-num-vec.csv")
21 file.remove("another-num-vec.csv")
22 #check to see if it's still there
23 dir()
24 #return to the parent directory
25 setwd("..")
26 #see what unlink does
27 ?unlink
28 #remove the testing directory
29 unlink("testing")
30 #is it still there?
31 dir("testing")
32

```

Let's see how this works.

```

#list the current working directory
getwd()

## [1] "C:/Users/ccc/stat3127book"

#create a new child directory named 'testing'
dir.create("testing")
#change to the testing child directory
setwd("testing")
#create a numeric vector
num.vec <- c(1,2,3,4,5)
#print the num.vec
num.vec

## [1] 1 2 3 4 5

#save the vector as a CSV file
write.csv(num.vec, "my-num-vec.csv")
#copy the file
file.copy("my-num-vec.csv", "another-num-vec.csv")

## [1] TRUE

#list the directory contents
dir()

## [1] "another-num-vec.csv" "my-num-vec.csv"

```

```

#delete the files you just created
file.remove("my-num-vec.csv")

## [1] TRUE

file.remove("another-num-vec.csv")

## [1] TRUE

#check to see if it's still there
dir()

## character(0)

#return to the parent directory
setwd("..")
#remove the testing directory
unlink("testing")
#is it still there?
dir("testing")

## character(0)

```

2.1.3 Data structures and types

R has a number of different data structures, including vectors, matrices, lists, and data frame. We will mostly be using data frames for statistical and analytical work, but it's worthwhile to have some sort of idea for the other kinds of data structures. Run the code in this listing.

```

1 #create a vector with four integers named 'a'
2 a <- c(1,2,3,4)
3 #create a vector with four integers named 'b'
4 b <- c(5,6,7,8)
5 #create a vector by listwise multiplication named 'c'
6 c <- a * b
7 #look at a, b, and c
8 a
9 b
10 c
11 #create a matrix named 'm' by row
12 m <- matrix(data = c(a, b, c), nrow = 3, byrow = TRUE)
13 m
14 #create a matrix named 'n' by column
15 n <- matrix(data = c(a, b, c), ncol = 3, byrow = FALSE)
16 n
17 #create three lists named 'd', 'e', 'f' and 'g'
18 d <- list("Washington", "Adams", "Jefferson", "Madison")
19 e <- list(1,2,3,4)
20 f <- list(1788, 1796, 1800, 1808)
21 #from Virginia?

```

```

22 g <- list(TRUE, FALSE, TRUE, TRUE)
23 #create a dataframe named 'presidents'
24 presidents <- data.frame(cbind(e, d, f, g), stringsAsFactors =
    FALSE)
25 names(presidents) <- c("order", "name", "year", "from.virginia?")
26 presidents
27

```

Let's see how this works.

```

#create a vector with four integers named 'a'
a <- c(1,2,3,4)
#create a vector with four integers named 'b'
b <- c(5,6,7,8)
#create a vector by listwise multiplication named 'c'
c <- a * b
#look at a, b, and c
a

## [1] 1 2 3 4

b

## [1] 5 6 7 8

c

## [1] 5 12 21 32

#create a matrix named 'm' by row
m <- matrix(data = c(a, b, c), nrow = 3, byrow = TRUE)
m

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    5   12   21   32

#create a matrix named 'n' by column
n <- matrix(data = c(a, b, c), ncol = 3, byrow = FALSE)
n

##      [,1] [,2] [,3]
## [1,]    1    5    5
## [2,]    2    6   12
## [3,]    3    7   21
## [4,]    4    8   32

```

```
#create three lists named 'd', 'e', 'f' and 'g'
d <- list("Washington", "Adams", "Jefferson", "Madison")
e <- list(1,2,3,4)
f <- list(1788, 1796, 1800, 1808)
#from Virginia?
g <- list(TRUE, FALSE, TRUE, TRUE)
#create a dataframe named 'presidents'
presidents <- data.frame(cbind(e, d, f, g), stringsAsFactors = FALSE)
names(presidents) <- c("order", "name", "year", "from.virginia?")
presidents
```

| ## | order | name | year | from.virginia? |
|------|-------|------------|------|----------------|
| ## 1 | 1 | Washington | 1788 | TRUE |
| ## 2 | 2 | Adams | 1796 | FALSE |
| ## 3 | 3 | Jefferson | 1800 | TRUE |
| ## 4 | 4 | Madison | 1808 | TRUE |

As a programming language, R has the usual types: integers, floats, doubles, strings, Booleans, etc. R contains no surprises here. However, there is one “data type” that may cause some confusion, *factors*. Factors can be any data type, but they are most frequently strings, such as “male” and “female.” Factors are categorical variables that correspond to levels in a study. We will look at two factors next, one consisting of strings, and the other consisting of “integers.”

```
1 #create a string vector
2 sex <- c("Male", "Female", "Male", "Female", "Female", "Female",
3         "Male", "Male", "Male", "Female", "Female", "Male", "Female",
4         "Male", "Female", "Male")
5 #look at it
6 sex
7 #convert it to a factor
8 sex.fac <- factor(sex, levels = c("Male", "Female"))
9 #look at the factor
10 sex.fac
11 #how many of each level?
12 table(sex.fac)
13 #create an integer vector
14 zips <- c(31901, 31906, 31901, 31909, 31907, 31907, 31909,
15         31901, 31906, 31906, 31907, 31909, 31901)
16 #look at zips
17 zips
18 #convert zips to a factor
19 zips.fac <- factor(zips, levels = c(31901, 31906, 31909,
20         31907), labels=c("Downtown", "Wynnton", "CPS",
21         "Lindsey_Creek"))
22 #look at the zips.fac
23 zips.fac
24 #see how many of each exist
25 table(zips.fac)
26
```

Let's see what happens when we run this.


```

#create a string vector
sex <- c("Male", "Female", "Male", "Female", "Female", "Female",
        "Male", "Male", "Male", "Female", "Female", "Male", "Female",
        "Male", "Female", "Male")
#look at it
sex

## [1] "Male" "Female" "Male" "Female" "Female" "Female" "Male"
## [8] "Male" "Male" "Male" "Female" "Female" "Male" "Female" "Male"
## [15] "Female" "Male"

#convert it to a factor
sex.fac <- factor(sex, levels = c("Male", "Female"))
#look at the factor
sex.fac

## [1] Male Female Male Female Female Female Male Male Male Female
## [11] Female Male Female Male Female Male
## Levels: Male Female

#how many of each level?
table(sex.fac)

## sex.fac
## Male Female
## 8 8

#create an ``integer'' vector
zips <- c(31901, 31906, 31901, 31909, 31907, 31907, 31909,
          31901, 31906, 31906, 31907, 31909, 31901)
#look at zips
zips

## [1] 31901 31906 31901 31909 31907 31907 31909 31901 31906 31906 31907
## [12] 31909 31901

#convert zips to a factor
zips.fac <- factor(zips, levels = c(31901, 31906, 31909,
                                   31907), labels=c("Downtown", "Wynnton", "CPS",
                                   "Lindsey Creek"))
#look at the zips.fac
zips.fac

## [1] Downtown Wynnton Downtown CPS Lindsey Creek
## [6] Lindsey Creek CPS Downtown Wynnton Wynnton
## [11] Lindsey Creek CPS Downtown
## Levels: Downtown Wynnton CPS Lindsey Creek

```

```
#see how many of each exist
table(zips.fac)

## zips.fac
##      Downtown      Wynnton      CPS Lindsey Creek
##           4           3           3           3
```

2.2 R packages

As an open technology, the R language is available for anyone to examine, copy, and change. An important benefit is that anyone can contribute to R, and many people have. These contributions are collected into *packages* and made available for everyone, with no strings attached. Popular repositories for R packages include Bioconductor,¹ GitHub,² and the Comprehensive R Archive Network. In this section we will take a look at CRAN and look at three popular packages.

2.2.1 CRAN

2.2.2 knitr

2.2.3 ggplot2

2.2.4 dplyr

2.3 Example analyses

2.3.1 Running scripts

You can run R scripts one of four ways. First, you can enter commands at the R prompt using the interactive shell. This is an excellent way to explore your data and experiment with various R functions, both built-in functions and user defined functions. Second, you can run R scripts as stand alone programs. This tutorial does not cover stand alone R programs, although this is not an advanced topic, and such programs are commonly used in research and industry. Third, you can run R scripts via the *knitr* package, which was covered in section 2.2.2 on 12. Finally, you can create a text file containing a series of R commands, and instruct your R console to execute the commands.

Open a text editor and copy the following listing. Save the file in the same directory in which your R is running as “script01.R”. In our R console, type this

¹Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, 1211 software packages, and an active user community.

²GitHub contains projects in a great number of languages, R included. In a way, it is a social networking site for people who write software.

command: `source("script01.R")`. The commands you listed in your script will run. Why should you do this? If you have a series of commands that you need to execute repeatedly, or if you are working on building a script and you need to test it many times, this will allow you to automate your work flow.

```

1 print("This is script01.R")
2 data <- LakeHuron
3 print("LakeHuron data")
4 print(data)
5 sumy <- summary(LakeHuron)
6 print("summary of data")
7 print(sumy)
8 avg <- mean(LakeHuron)
9 print("mean of data")
10 print(avg)
11 med <- median(LakeHuron)
12 print("median of data")
13 print(med)
14 sd <- sd(LakeHuron)
15 print("standard deviation of data")
16 print(sd)
17 print("histogram of data")
18 hist(LakeHuron)
19 print("script01.R done")
20

```

This is what you will see when you do this.

```

print("This is script01.R")

## [1] "This is script01.R"

data <- LakeHuron
print("LakeHuron data")

## [1] "LakeHuron data"

print(data)

## Time Series:
## Start = 1875
## End = 1972
## Frequency = 1
## [1] 580.38 581.86 580.97 580.80 579.79 580.39 580.42 580.82 581.40 581.32
## [11] 581.44 581.68 581.17 580.53 580.01 579.91 579.14 579.16 579.55 579.67
## [21] 578.44 578.24 579.10 579.09 579.35 578.82 579.32 579.01 579.00 579.80
## [31] 579.83 579.72 579.89 580.01 579.37 578.69 578.19 578.67 579.55 578.92
## [41] 578.09 579.37 580.13 580.14 579.51 579.24 578.66 578.86 578.05 577.79
## [51] 576.75 576.75 577.82 578.64 580.58 579.48 577.38 576.90 576.94 576.24
## [61] 576.84 576.85 576.90 577.79 578.18 577.51 577.23 578.42 579.61 579.05
## [71] 579.26 579.22 579.38 579.10 577.95 578.12 579.75 580.85 580.41 579.96
## [81] 579.61 578.76 578.18 577.21 577.13 579.10 578.25 577.91 576.89 575.96
## [91] 576.80 577.68 578.38 578.52 579.74 579.31 579.89 579.96

```

```
sumy <- summary(LakeHuron)
print("summary of data")

## [1] "summary of data"

print(sumy)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  576.0   578.1   579.1   579.0   579.9   581.9

avg <- mean(LakeHuron)
print("mean of data")

## [1] "mean of data"

print(avg)

## [1] 579.0041

med <- median(LakeHuron)
print("median of data")

## [1] "median of data"

print(med)

## [1] 579.12

sd <- sd(LakeHuron)
print("standard deviation of data")

## [1] "standard deviation of data"

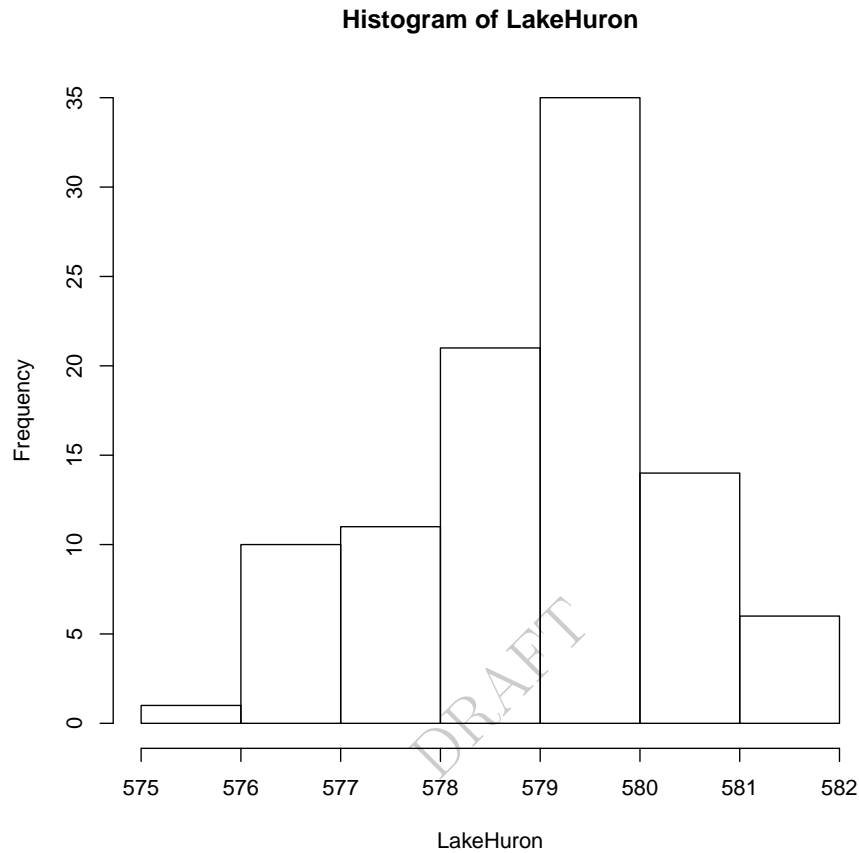
print(sd)

## [1] 1.318299

print("histogram of data")

## [1] "histogram of data"

hist(LakeHuron)
```



```
print("script01.R done")  
## [1] "script01.R done"
```

Exercise: R has a number of built-in data sets for exploration and learning. Enter the command `?data` and read the documentation. Then, enter the command `data()`,³ and review the available data sets. You will find the *LakeHuron* dataset here. Choose one, and create your own script to print the dataset and a summary of the dataset.

³Don't forget the parentheses after the word *data*. The parentheses tell R that you have called a function.

2.3.2 Data entry**2.3.3 Importing data**

DRAFT

Chapter 3

Descriptive Statistics

this is chapter tmplate!

DRAFT

DRAFT

Appendix A

Installing and Running R()

this is appendix one!

DRAFT

DRAFT

Appendix B

R Integrated Development Environments

this is appendix one part b!

DRAFT

DRAFT

Afterword

this is the afterword!

DRAFT