Statistics 111 ∝ Section 0

Zad Chin & Jarell Cheong Tze Wen
January 23, 2023

Logistics

- Section Times: Fridays 10:30 – 11:45 AM, Science Center 706.
- Office Hours: Tuesdays 7:00 – 9:00 PM, Quincy Dining Hall.
- Feel free to email us with questions at the following addresses:
  - → Zad: zadchin@college.harvard.edu
  - → Jarell: jarellcheong@college.harvard.edu

Contents

1 Introducing R and RStudio

The programming language R is a free, open-source software program for statistical analysis. You might have heard of languages like Python, C, HTML, and/or CSS. Just as HTML and CSS are customized for website development, the R language is customized for statistical analysis. We recommend doing your R coding in RStudio, an IDE (Integrated Development Environment) made for R.

Installing R and RStudio

To install R, follow these steps:

(a) Download the R installer.

- For Windows, install R here.
- For Mac, install R here.

(b) Install R by opening the installer and following the given steps.

To install RStudio, follow these steps:

(a) Download the RStudio Desktop installer here.

(b) Install RStudio by opening the installer and following the given steps.

Special R Quirks

- **Case Sensitive:** Technically, R is a *functional language* with very simple syntax. It's case sensitive, so A and a will mean different things.
- **Indexing in R:** As opposed to normal CS conventions where the indices to iterable objects (such as lists or vectors) start from 0, indexing in R begins with 1. We personally think that this makes life easier.
- **Commenting in R:** Use hashtags "#" to comment out lines in R.
- **Executing Commands in R:** Highlight the portion of the code you would love to run and click run in the top right corner of your script in RStudio, or copy and paste the subroutine you want to run into the R console!
- **Searching for Documentation?:** Just type help(...) in the R console for the functions you need help with! For example, try typing help(solve) for the documentation of the solve function in R.

Understanding the RStudio Interface

R studio may seem bunky and less cool than your normal editor, but every part of RStudio can help you when it comes to streamlining the process of statistical inference or simulation. The function of each panel in RStudio is as follows:

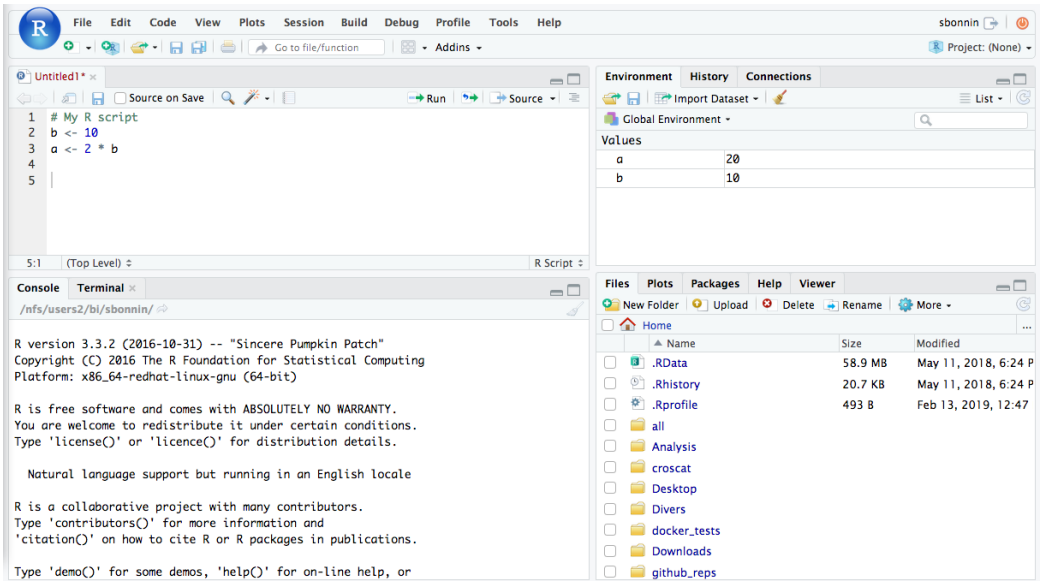

Figure 1: RStudio has 4 panels: in the top-left, you have scripts and files; in the bottom-left, you have the R console, which acts as a terminal; in the top-right, you have the objects, history, and environment tabs, which are helpful when finding and/or viewing the data you get from simulated results. Lastly, in the bottom-right, you have the tree of folders, a graph window, packages, a help window, and a viewer.

2 The Basics of R

Assignments and Variables in R

The assignment operator in R is <-. You can also use the equals sign = for assignment, but this symbol has different meanings in different contexts in R. So it is a good practice to use the former arrow for variable assignment. Also, recall that R is case-sensitive, so you can always define two different variables for each letter, one in uppercase and one in lowercase. For example, it is useful to define Y for predicted y values and y for observed y values. Lastly, commands in R can be separated by a new line or a semicolon ; as shown below:

```
A <- 10; a <- 1
print(A)
>> 10
print(a)
>> 1
```

R as a Calculator

Like any other programming language, R can act as a basic calculator and perform various calculations and operations. Example operations are as follows:

Command in R	Function
`10 + 20`	Addition
`20 - 10`	Subtraction
`10 * 20`	Multiplication
`10 / 20`	Division
`10 %% 20`	Modulus
`abs(x)`	$\lvert x \rvert$
`exp(x)`	e^x
`log(x)`	$\log x$
`log(x,b)`	$\log_b x$
`sqrt(x)`	$\sqrt{x}$
`floor(x)`	$\lfloor x \rfloor$
`ceiling(x)`	$\lceil x \rceil$
`factorial(n)`	$n!$

3 Loops, Conditionals, and Functions

Just like any other programming language, you have the usual ability in R to perform loops, set up conditionals, and create/use functions. We will cover each of these features in turn below.

Conditionals

Conditional statements are expressions that perform different computations or actions depending on whether a predefined boolean condition is TRUE or FALSE. Examples of conditionals with if, if else, and if else if else are shown below. Notice that we use curly brackets {} to start and end each conditional!

```
class <- 111
# An if statement
if (class == 111) {
    print("This is STAT111!")
}
>> "This is STAT111!"

class2 <- 110
# An if else statement
if (class2 == 111) {
  print("This is STAT111!")
} else {
  print("This is not STAT111!")
}
>> "This is not STAT111!"

# A one line if else: ifelse(condition, True result, False result)
ifelse(class2 %% 2 == 0, "STAT110", "STAT111")
>> "STAT110"

# An if else if else statement
if (class > 111) {
  print("This is a more advanced STAT course")
} else if (class < 111) {
  print("This is a less advanced STAT course")
} else {
  print("This is STAT111!")
}
>> "This is STAT111!"
```

Loops

Loops are one of the staples of all programming languages. It allows the certain operations to run repetitively until a certain conditions is met. There are three main types of loops in R: the for loop, the while loop and the rarely used repeat loop. Here, we will discuss examples of for loops and while loops. The while loop is used when you want to keep looping until a specific logical condition is satisfied, and the *for* loop which will always iterate through the given sequence.

```
# A for loop
for (i in 1:3) {
    print(i + 1)
}
>> 2
>> 3
>> 4

# A while loop
i <- 0
while (i <= 2) {
    i <- i + 1
    print(i)
}
>> 1
>> 2
>> 3
```

However, R would really prefer you not to use loops, and if you do, it will make you pay a price: running the same calculation with a loop can make the code run tens or even hundreds of times slower! Also, your code is much easier to read if you can avoid writing the loops. There are some instances where running for or while loops are extremely useful; for example, iterating through a dataset, running simulations, going through a recursive relationship (dynamic programming!), or implementing more complex algorithms that can't be simplified such as the EM algorithm or an MCMC algorithm (it's fine if you don't know what these are!).

So, if you are not encouraged to use loop, what should you use? Generally, it is advisable to vectorize your expression and apply some in-built function such as apply(), lapply(), tapply(), sapply(), vapply(), and mapply(), which are generally faster and have a lower risk of error!

Functions

Another programming languages stample is the humble function, your loyal servant waiting patiently to do your bidding to the best of their ability. Functions that are made with upmost care and attention can make code more powerful, concise, and clean. Examples on how to define and use functions are given below:

```r
# A function that returns the standard deviation of a given vector
stdev <- function(x) {
    return(sqrt(var(x)))
}

# A one-line function
stdev <- function(x) { return(sqrt(var(x)))}

# Try this in RStudio and see what it outputs!
lapply(0:4, function(a) {a + 1})
```

Although I defined a standard deviation function, generally statistical functions, such as summary statistics which are commonly used in Statistics 111, have a predefined function in R. Many examples are given in the table below:

Command in R	Function
`sum(v)`	Sum of v
`mean(v)`	Sample mean of v
`var(v)`	Sample variance of v
`sd(v)`	Sample standard deviation of v
`median(v)`	Sample median of v
`summary(v)`	Summary statistics for v
`quantile(v,p)`	Sample pth quantile of v
`cov(x)`	Sample covariance of v and w
`cor(v)`	Sample correlation of v and w

4 Vectors and Matrices in R

Vectors

Vectors are the most basic R data objects. Vectors in R aren't like the scary vectors in your traumatizing linear algebra class; they're more like arrays in C or lists in Python. A vector can be made up of numerics or strings and is capable of all of the mathematical operations we discussed above. Examples of how to create and use vectors are as follows:

```
# Create a vector of 5 zeros
zeros <- rep(0, 5)
print(zeros)
>> 0 0 0 0 0

# Creare a vector with elements from 4 to 10 incrementing by 2
evens <- seq(4, 10, by = 2)
print(evens)
>> 4  6  8 10

# Bind a list of numbers
v1 <- c(3, 8, 4, 5)

# Mathematical operations
print(v1 + evens)
>> 7 14 12 15
print(mean(v1))
>> 5
print(sort(v1))
>> 3 4 5 8

# Accessing elements in vectors
v1[2]
>> 8
v1[2:3]
>> 8 4
v1[v1 <= 4]
>> 3 4

# Combining two arrays
v2 <- c(v1, evens)
print(v2)
>> 3  8  4  5  4  6  8 10
```

Matrices

While vectors in R need not remind you of your traumatizing linear algebra class, matrices in R just might! Matrices can be very useful in R especially when you wish to iteratively keep track of multiple parameters while doing inference. Examples of how to compute with and use matrices are as follows. Do try to print these out in RStudio!

```
# Create a 3 by 3 matrix with a sequence
matrix1 <- matrix(c(3:11), nrow = 3, byrow = TRUE)
# Create a 3 by 3 matrix for a given sequence
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4, 1, 5, 1), nrow = 3)
v_test <- c(1, 2, 3)
print(matrix1)
>>      [1] [2] [3]
>> [1] 3    4   5
>> [2] 6    7   8
>> [3] 9   10  11
print(matrix2)
>>      [1] [2] [3]
>> [1] 5    9   1
>> [2] 2    3   5
>> [3] 0    4   1

# Matrix addition
print(matrix1 + matrix2)
>>      [1] [2] [3]
>> [1] 8   13   6
>> [2] 8   10  13
>> [3] 9   14  12

# Element by element matrix multiplication
print(matrix1 * matrix2)
>>      [1] [2] [3]
>> [1] 15  26   5
>> [2] 12  21  40
>> [3] 0   40  11

# Actual matrix multiplication
print(matrix1 %*% matrix2)
>>      [1] [2] [3]
>> [1] 23  59   28
>> [2] 44  107  49
>> [3] 65  155  70
```

```r
# Matrix vector multiplication
print(matrix1 %*% v_test)
>>      [1]
>> [1] 26
>> [2] 44
>> [3] 62

# Matrix inverse
solve(matrix2)

# Solving equations
solve(matrix2, v_test)

# Singular value decomposition (SVD)
svd(matrix2)

# QR decomposition
qr(matrix2)

# Eigenvalues and eigenvectors of a matrix
eigen(matrix1)

# Diagnoal of a matrix
diag(matrix1)

# Column bind a matrix to a vector
cbind(matrix1, v_test)
>>      [1] [2] [3] [4]
>> [1] 3    4   5   1
>> [2] 6    7   8   2
>> [3] 9   10  11   3

# Row bind a matrix to a vector
rbind(matrix1, v_test)
>>      [1] [2] [3]
>> [1] 3    4   5
>> [2] 6    7   8
>> [3] 9   10  11
>> [4] 1    2   3
```

5 Plotting in R

In statistics, plotting the data gives a lot of information about the data in a visually appealing way. For example, plotting histogram can inform statisticians about the estimated distribution of the given dataset. Below are some functions which are extremely useful when it comes to actually plotting things.

Command in R	Function
`plot(x, y)`	Scatter plot
`plot(y, type = "l")`	Line plot
`abline(a, b)`	Intercept line
`abline(h = y)`	Horizontal line
`abline(v = x)`	Vertical line
`hist(x, breaks = b)`	Histogram

Now, we provide an example on how to plot a line plot and histogram using the dataset, SP500.csv, which was used in a previous Statistics111 homework.

```
# Link the location of the file in your system below
Y <- read.csv("SP500time.csv")$Y

# Line plot for S&P500
plot(Y, type = "l", col = "#009999", main = "S&P500 Line Plot")

# Line plot for S&P500 with red horizontal line y = 1
plot(Y,
     type = "l",
     col = "#009999",
     main = "S&P500 Line Plot with y = 1"
     )
abline(h = 1, col = "red")

# Histogram for S&P500 with 100 breaks and orange bins
hist(Y,
     breaks = 100,
     main = "S&P500 Histogram with 100 Bins",
     col = "orange"
     )
```

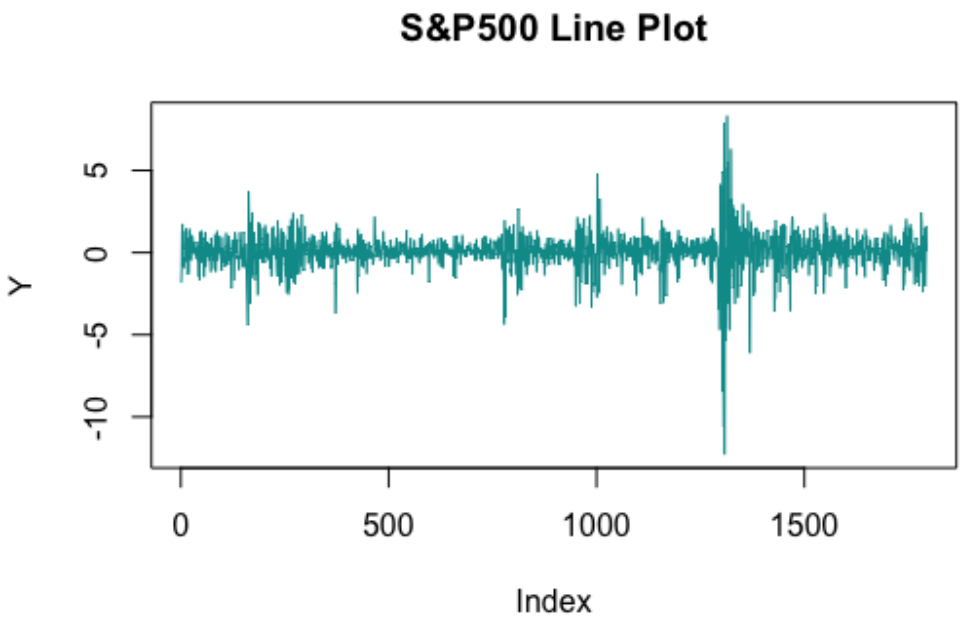

Figure 2: A line plot of S&P500 daily percentage changes.

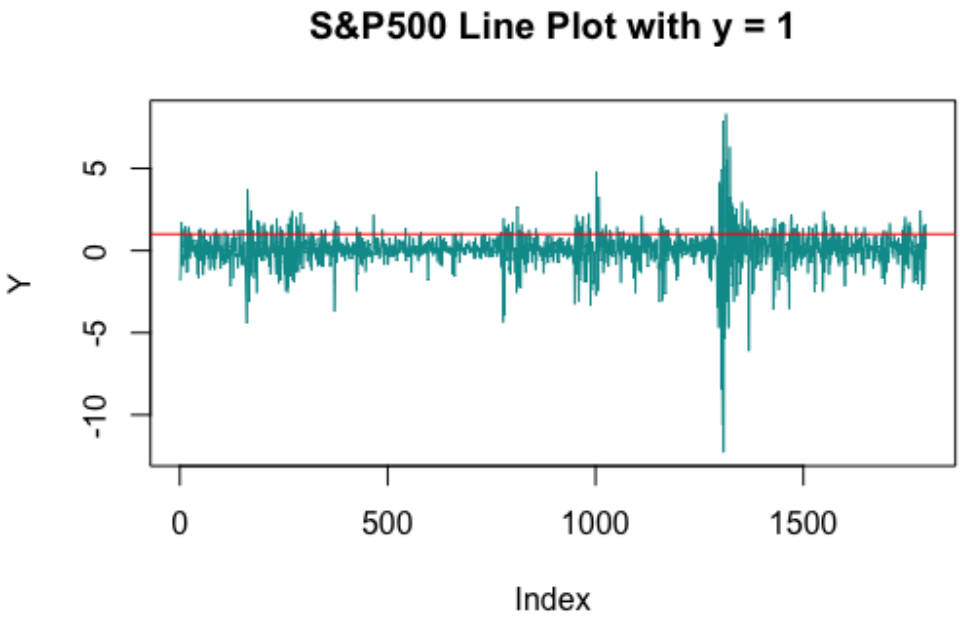

Figure 3: A line plot of S&P500 daily percentage changes with $y = 1$ drawn in red.

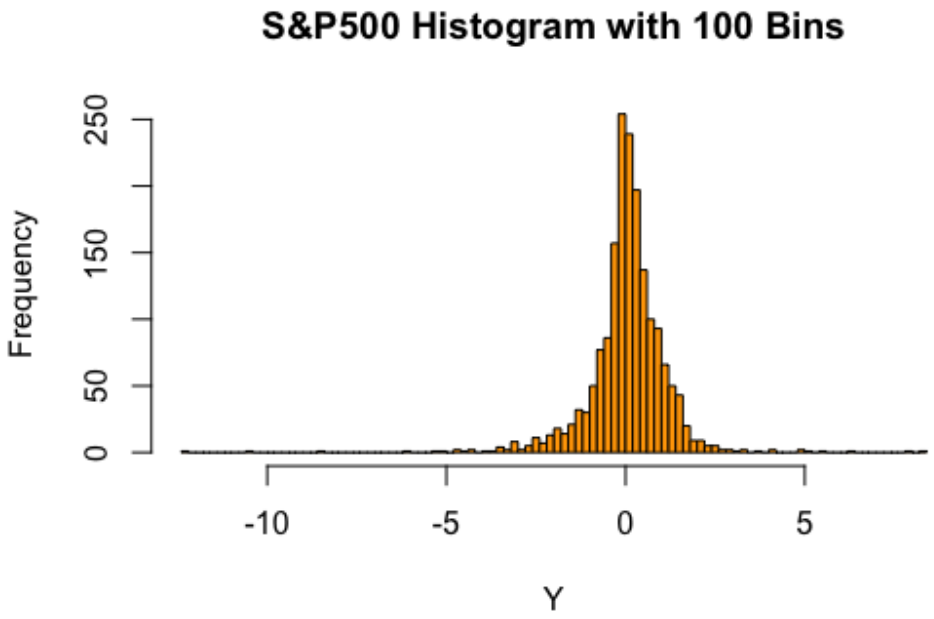

Figure 4: An orange histogram of S&P500 daily percentage changes.

6 Distributions in R

Finally, let's talk about obtaining various quantities related to distributions of interest in R. This is immensely useful in statistical inference since data is usually modeled after some known distribution! Consider the table shown below:

Command in R	Function
`help(distributions)`	General documentation
`dbinom(k, n, p)`	Binomial density
`pbinom(x, n, p)`	Binomial CDF
`qbinom(a, n, p)`	Binomial quantiles
`rbinom(r, n, p)`	Binomial samples

The commands shown above are completely analogous for different distributions. That is, d– yields the density of –, p– yields the CDF of –, q– yields the quantile function of –, and r– generates random samples from –, where – is a distribution we have to specify. The names of some common distributions with the d– command are displayed below:

Command in R	Function
`dgeom(k, p)`	Geometric density
`dhyper(k, w, b, n)`	Hypergeometric density
`dnbinom(k, r, p)`	Negative Binomial density
`dpois(k, r)`	Poisson density
`dbeta(x, a, b)`	Beta density
`dcauchy(x)`	Cauchy density
`dchisq(x, n)`	χ^2 density
`dexp(x, r)`	Exponential density
`dgamma(x, a, r)`	Gamma density
`dnorm(x, m, s)`	Normal density
`dt(x, n)`	t density
`dunif(x, a, b)`	Uniform density

7 An Example of the MLE

Problem. In Statistics 111, we are usually concerned with finding the value of a parameter that maximizes the probability of observing the data that we observed. This is called a *maximum likelihood estimate* (MLE). Let's use a specific example to visualize maximum likelihood.

(a) Create 20 random observations from a Pois(3) random variable. Suppose we know that the data are i.i.d. Poisson observations, but we do not know that $\lambda = 3$. For each $\lambda \in \{0.1, 0.2, \ldots, 4.9, 5.0\}$, find the likelihood of your data, i.e. find

$$\mathbb{P}(Y_1 = y_1, \ldots, Y_{20} = y_{20} \mid \lambda)$$

for each λ. Save these likelihoods in some vector.

(b) Now, make a *lineplot* of λ against the likelihoods. What seems to be the value of λ that maximizes the likelihood?

8 Generating Exponentials with the PIT

Problem. Let's use the PIT (or Universality of the Uniform for those more familiar with Statistics 110 terminology) to generate random observations from an Expo(1) distribution. Recall that $F(x) = 1 - e^{-x}$ for $X \sim \text{Expo}(1)$.

(a) The PIT says that if we plug a Uniform into the quantile function (inverse CDF), we will observe draws from the desired distribution. Find the needed quantile function analytically.

(b) Generate many draws from a Uniform distribution, plug them into the quantile function you have, and plot them in a histogram.

(c) Overlay the density of an Expo(1) distribution to the histogram that you get. Does the observed data appear to match the theoretical distribution?

---

9 Introducing Linear Regression

Problem. This problem explores *linear regression* using the lm command in the stats package, as well as via matrix operations. We use the iris dataset.

(a) Create a binary variable within iris for whether the species of each flower is setosa (i.e. should be 1 if the flower is setosa and 0 otherwise); call this setosa.

(b) Use the lm command to run a regression of setosa using all variables within iris *except* the Species variable. Call this model lm1.

(c) Find the regression coefficients for lm1.

(d) Let us try to get the same values with matrix operations. Create a dataframe called X which has our "predictor variables" (everything in iris except Species and setosa). Add a column to X that has all 1's (put this column to the left of the others, i.e. before the other columns).

(e) Make X into a matrix. Then, make a vector called y which has the response variable setosa.

(f) The matrix operations $(X'X)^{-1}X'y$ should also yield the coefficients we found in (c), where A' and A^{-1} are the transpose and inverse of A, respectively. Verify that this is the case.
