Applied Statistics with R

David Dalpiaz

2016-06-17

Contents

Chapter 1

Introduction

Welcome to Applied Statistics with `R`!

1.1 About This Book

This book was originally (and currently) designed for use with STAT 420, Methods of Applied Statistics, at the University of Illinois at Urbana-Champaign. It may certainly be used elsewhere, but any references to "this course" in this book specifically refer to STAT 420.

This book is under active development. When possible, it would be best to always access the text online to be sure you are using the most up-to-date version. (Also, the html version provides additional features such as changing text size, font, and colors.) If you are in need of a local copy, a **pdf version** is continuously maintained.

Since this book is under active development you may encounter errors ranging from typos to broken code to poorly explained topics. If you do, please let us know! Simply send an email and we'll make the changes ASAP. (`dalpiaz2 AT illinois DOT edu`) Or, if you know RMarkdown and are familiar with GitHub, make a pull request and fix an issue yourself! (This process is partially automated by the edit button in the top-left corner of the html version.)

This text uses MathJax to render mathematical notation for the web. Occasionally, but rarely, a JavaScript error will prevent MathJax from rendering correctly. (In which case, will see the "code" instead of the expected mathematical equations.) From experience, this is almost always fixed by simply refreshing the page. You'll also notice that if you right-click any equation you can obtain the MathML Code (for copying into Microsoft Word) or the TeX command used to generate the equation.

$$a^2 + b^2 = c^2$$

1.2 Conventions

`R` code will be typeset using a `monospace` font which is syntax highlighted.

```
a = 3
b = 4
sqrt(a ^ 2 + b ^ 2)
```

`R` output lines, which would appear in the console will begin with `##`. They will generally not be syntax highlighted.

```
## [1] 5
```

1.3 Acknowledgements

- Alex Stepanov
- David Unger
- James Balamuta

1.4 License

Chapter 2

Introduction to R

> "Measuring programming progress by lines of code is like measuring aircraft building progress by weight."
>
> — **Bill Gates**

After reading this chapter you will be able to:

- Interact with R using RStudio.
- Use R as a calculator.
- Work with data as vectors and data frames.
- Make basic data visualizations.
- Write your own R functions.
- Perform hypothesis tests using R.
- Perform basic simulations in R.

2.1 R Resources

R is both a programming language and software environment for statistical computing, which is *free* and *open-source*. To get started, you will need to install two pieces of software:

- R, the actual programming language.
  - Chose your operating system, and select the most recent version, 3.3.0.
- RStudio, an excellent IDE for working with R.
  - Note, you must have R installed to use RStudio. RStudio is simply an interface used to interact with R.

The popularity of R is on the rise, and everyday it becomes a better tool for statistical analysis. It even generated this book! (A skill you will learn in this course.) There are many good resources for learning R. They are not necessary for this course, but you may find them useful if you would like a deeper understanding of R:

- Try R from Code School.
  - An interactive introduction to the basics of R. Could be very useful for getting up to speed on R's syntax.

- Quick-R by Robert Kabacoff.
  - A good reference for R basics.
- R Tutorial by Chi Yau.
  - A combination reference and tutorial for R basics.
- R Markdown from RStudio.
  - Reference materials for RMarkdown.
- The Art of R Programming by Norman Matloff.
  - Gentle introduction to the programming side of R. (Whereas we will focus more on the data analysis side.) A free electronic version is available through the Illinois library.
- Advanced R by Hadley Wickham.
  - From the author of several extremely popular R packages. Good follow-up to The Art of R Programming. (And more up-to-date material.)
- R for Data Science by Hadley Wickham and Garrett Grolemund.
  - Similar to Advanced R, but focuses more on data analysis, while still introducing programming concepts. At the time of writing, currently under development.
- The R Inferno by Patrick Burns.
  - Likens learning the tricks of R to descending through the levels of hell. Very advanced material, but may be important if R becomes a part of your everyday toolkit.

RStudio has a large number of useful keyboard shortcuts. A list of these can be found using a keyboard shortcut – the keyboard shortcut to rule them all:

- On Windows: `Alt` + `Shift` + `K`
- On Mac: `Option` + `Shift` + `K`

The RStudio team has developed a number of "cheatsheets" for working with both R and RStudio. This particular cheatseet for Base R will summarize many of the concepts in this document.

When programming, it is often a good practice to follow a style guide. (Where do spaces go? Tabs or spaces? Underscores or CamelCase when naming variables?) No style guide is "correct" but it helps to be aware of what others do. The more import thing is to be consistent within your own code.

- Hadley Wickham Style Guide from Advanced R
- Google Style Guide

For this course, our main deviation from these two guides is the use of `=` in place of `<-`. (More on that later.)

2.2 R Basics

2.2.1 Basic Calculations

To get started, we'll use R like a simple calculator. Note, in R the `#` symbol is used for comments. In this book, lines which begin with two such symbols, `##`, indicate output.

Addition, Subtraction, Multiplication and Division

```r
3 + 2
```

```
## [1] 5
```

```r
3 - 2
```

```
## [1] 1
```

```r
3 * 2
```

```
## [1] 6
```

```r
3 / 2
```

```
## [1] 1.5
```

Exponents

```r
3 ^ 2
```

```
## [1] 9
```

```r
2 ^ (-3)
```

```
## [1] 0.125
```

```r
100 ^ (1 / 2)
```

```
## [1] 10
```

```r
sqrt(1 / 2)
```

```
## [1] 0.7071068
```

```r
exp(1)
```

```
## [1] 2.718282
```

Mathematical Constants

```r
pi
```

```
## [1] 3.141593
```

```
exp(1)
```

```
## [1] 2.718282
```

Logarithms

```
log(10)           # natural log
```

```
## [1] 2.302585
```

```
log10(1000)       # base 10 log
```

```
## [1] 3
```

```
log2(8)           # base 2 log
```

```
## [1] 3
```

```
log(16, base = 4) # base 4 log
```

```
## [1] 2
```

Trigonometry

```
sin(pi / 2)
```

```
## [1] 1
```

```
cos(0)
```

```
## [1] 1
```

2.2.2 Getting Help

In using R as a calculator, we have seen a number of functions: `sqrt()`, `exp()`, `log()` and `sin()`. To get documentation about a function in R, simply put a question mark in front of the function name and RStudio will display the documentation, for example:

```
?log
?sin
?paste
?lm
```

Frequently one of the most difficult things to do when learning R is asking for help. First, you need to decide to ask for help, then you need to know *how* to ask for help. Your very first line of defense should be to Google your error message or a short description of your issue. (The ability to solve problems using this method is quickly becoming an extremely valuable skill.) If that fails, and it eventually will, you should ask for help. There are a number of things you should include when emailing an instructor, or posting to a help website such as Stack Exchange.

- Describe what you expect the code to do.
- State the end goal you are trying to achieve. (Sometimes what you expect the code to do, is not what you want to actually do.)
- Provide the full text of any errors you have received.
- Provide enough code to recreate the error. Often for the purpose of this course, you could simply email your entire `.R` or `.Rmd` file.
- Sometimes it is also helpful to include a screenshot of your entire RStudio window when the error occurs.

If you follow these steps, you will get your issue resolved much quicker, and possibly learn more in the process. Do not be discouraged by running into errors and difficulties when learning R. (Or any technical skill.) It is simply part of the learning process.

2.2.3 Installing Packages

R comes with a number of built-in functions and datasets, but one of the main strengths of R as an open-source project is its package system. Packages add additional functions and data. Frequently if you want to do something in R, and it isn't available by default, there is a good chance that there is a package that will fulfill your needs.

To install a package, use the `install.packages()` function. Think of this as buying a recipe book from the store, bringing it home, and putting it on your shelf.

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

Once a package is installed, it must be loaded into your current R session before being used. Think of this as taking the book off of the shelf and opening it up to read.

```
library(ggplot2)
```

Once you close R, all the packages are closed and put back on the imaginary shelf. The next time you open R, you do not have to install the package again, but you do have to load any packages you intend to use by invoking `library()`.

2.2.4 Data Types

R has a number of basic data *types*.

- Numeric
  - Also known as Double. The default type when dealing with numbers.
  - Examples: `1`, `1.0`, `42.5`
- Integer
  - Examples: `1L`, `2L`, `42L`

- Complex
  - Example: `4 + 2i`
- Logical
  - Two possible values: `TRUE` and `FALSE`
  - You can also use `T` and `F`, but this is *not* recommended.
  - `NA` is also considered logical.
- Character
  - Examples: `"a"`, `"Statistics"`, `"1 plus 2."`

R also has a number of basic data *structures*. A data structure is either homogeneous (all elements are of the same data type) or heterogeneous (elements can be of more than one data type).

Dimension	**Homogeneous**	**Heterogeneous**
1	Vector	List
2	Matrix	Data Frame
3+	Array	

2.2.5 Vectors

Many operations in R make heavy use of **vectors**. Vectors in R are indexed starting at `1`. That is what the `[1]` in the output is indicating, that the first element of the row being displayed is the first element of the vector. Larger vectors will start additional rows with `[*]` where `*` is the index of the first element of the row.

Possibly the most common way to create a vector in R is using the `c()` function, which is short for "combine."" As the name suggests, it combines a list of numbers separated by commas.

```
c(1, 3, 5, 7, 8, 9)
```

```
## [1] 1 3 5 7 8 9
```

Here R simply outputs this vector. If we would like to store this vector in a **variable** we can do so with the **assignment** operator `=`. In this case the variable `x` now holds the vector we just created, and we can access the vector by typing `x`.

```
x = c(1, 3, 5, 7, 8, 9)
x
```

```
## [1] 1 3 5 7 8 9
```

As an aside, there is a long history of the assignment operator in R. For simplicity we will use `=`, but know that often you will see `<-` as the assignment operator. The pros and cons of these two are well beyond the scope of this book, but know that for our purposes you will have no issue if you simply use `=`.

Frequently you may wish to create a vector based on a sequence of numbers. The quickest and easiest way to do this is with the `:` operator, which creates a sequence of integers between two specified integers.

```
(y = 1:100)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

Here we see R labeling the rows after the first since this is a large vector. Also, we see that by putting parentheses around the assignment, R both stores the vector in a variable called y and automatically outputs y to the console.

To subset a vector, we use square brackets, `[]`.

```
x
```

```
## [1] 1 3 5 7 8 9
```

```
x[1]
```

```
## [1] 1
```

```
x[3]
```

```
## [1] 5
```

We see that `x[1]` returns the first element, and `x[3]` returns the third element.

```
x[-2]
```

```
## [1] 1 5 7 8 9
```

We can also exclude certain indexes, in this case the second element.

```
x[1:3]
```

```
## [1] 1 3 5
```

```
x[c(1,3,4)]
```

```
## [1] 1 5 7
```

Lastly we see that we can subset based on a vector of indices.

One of the biggest strengths of R is its use of vectorized operations. (Frequently the lack of understanding of this concept leads of a belief that R is *slow*. R is not the fastest language, but it has a reputation for being slower than it really is.)

```
x = 1:10
x + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
2 * x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2 ^ x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

```
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

We see that when a function like `log()` is called on a vector `x`, a vector is returned which has applied the function to each element of the vector `x`.

```
vec_1 = 1:10
vec_2 = 1:1000
vec_3 = 42
```

The length of a vector can be obtained with the `length()` function.

```
length(vec_1)
```

```
## [1] 10
```

```
length(vec_2)
```

```
## [1] 1000
```

```
length(vec_3)
```

```
## [1] 1
```

Note that scalars do not exists in `R`. They are simply vectors of length `1`.

If we want to create a sequence that isn't limited to integers and increasing by 1 at a time, we can use the `seq()` function.

```
seq(from = 1.5, to = 4.2, by = 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3
## [20] 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

We will discuss functions in detail later, but note here that the input labels `from`, `to`, and `by` are optional.

```
seq(1.5, 4.2, 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3
## [20] 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

Another common operation to create a vector is `rep()`, which can repeat a single value a number of times.

```
rep("A", times = 10)
```

```
##  [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"
```

Or, `rep()` can be used to repeat a vector a number of times.

```
rep(x, times = 3)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5
## [26]  6  7  8  9 10
```

We have now seen four different ways to create vectors:

- `c()`
- `:`
- `seq()`
- `rep()`

So far we have mostly used them in isolation, but they are often used together.

```
c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  1  3  5  7  9  1  3  5  7  9  1  3  5  7  9
## [26]  1  2  3 42  2  3  4
```

2.2.6 Summary Statistics

`R` has built in functions for a large number of summary statistics.

```
y
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

Central Tendency

```
mean(y)
```

```
## [1] 50.5
```

```
median(y)
```

```
## [1] 50.5
```

Spread

```
var(y)
```

```
## [1] 841.6667
```

```
sd(y)
```

```
## [1] 29.01149
```

```
IQR(y)
```

```
## [1] 49.5
```

```
min(y)
```

```
## [1] 1
```

```
max(y)
```

```
## [1] 100
```

```
range(y)
```

```
## [1]   1 100
```

2.2.7 Matrices

R can also be used for **matrix** calculations. Matrices have rows and columns containing a single data type. In a matrix, the order of rows and columns is important. (This is not true of *data frames*, which we will see later.)

Matrices can be created using the `matrix` function.

```
x = 1:9
x
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

```
X = matrix(x, nrow = 3, ncol = 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Note here that we are using two different variables: lower case `x`, which stores a vector and capital `X`, which stores a matrix. (Following the usual mathematical convention.) We can do this because `R` is case sensitive.

By default the `matrix` function reorders a vector into columns, but we can also tell `R` to use rows instead.

```
Y = matrix(x, nrow = 3, ncol = 3, byrow = TRUE)
Y
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

We can also create a matrix of a specified dimension where every element is the same, in this case `0`.

```
Z = matrix(0, 2, 4)
Z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
```

Like vectors, matrices can be subsetted using square brackets, `[]`. However, since matrices are two-dimensional, we need to specify both a row and a column when subsetting.

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
X[1, 2]
```

```
## [1] 4
```

Here we accessed the element in the first row and the second column. We could also subset an entire row or column.

```
X[1, ]
```

```
## [1] 1 4 7
```

```
X[, 2]
```

```
## [1] 4 5 6
```

We can also use vectors to subset more than one row or column at a time. Here we subset to the first and third column of the second row.

```
X[2, c(1, 3)]
```

```
## [1] 2 8
```

Matrices can also be created by combining vectors as columns, using `cbind`, or combining vectors as rows, using `rbind`.

```
x = 1:9
rev(x)
```

```
## [1] 9 8 7 6 5 4 3 2 1
```

```
rep(1, 9)
```

```
## [1] 1 1 1 1 1 1 1 1 1
```

```
cbind(x, rev(x), rep(1, 9))
```

```
##       x
##  [1,] 1 9 1
##  [2,] 2 8 1
##  [3,] 3 7 1
##  [4,] 4 6 1
##  [5,] 5 5 1
##  [6,] 6 4 1
##  [7,] 7 3 1
##  [8,] 8 2 1
##  [9,] 9 1 1
```

```
rbind(x, rev(x), rep(1, 9))
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## x    1    2    3    4    5    6    7    8    9
##      9    8    7    6    5    4    3    2    1
##      1    1    1    1    1    1    1    1    1
```

R can then be used to perform matrix calculations.

```
x = 1:9
y = 9:1
X = matrix(x, 3, 3)
Y = matrix(y, 3, 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
Y
```

```
##      [,1] [,2] [,3]
## [1,]    9    6    3
## [2,]    8    5    2
## [3,]    7    4    1
```

```
X + Y
```

```
##      [,1] [,2] [,3]
## [1,]   10   10   10
## [2,]   10   10   10
## [3,]   10   10   10
```

```
X - Y
```

```
##      [,1] [,2] [,3]
## [1,]   -8   -2    4
## [2,]   -6    0    6
## [3,]   -4    2    8
```

```
X * Y
```

```
##      [,1] [,2] [,3]
## [1,]    9   24   21
## [2,]   16   25   16
## [3,]   21   24    9
```

```
X / Y
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.1111111 0.6666667 2.333333
## [2,] 0.2500000 1.0000000 4.000000
## [3,] 0.4285714 1.5000000 9.000000
```

Note that `X * Y` is not matrix multiplication. It is element by element multiplication. (Same for `X / Y`). Instead, matrix multiplication uses `%*%`. Other matrix functions include `t()` which gives the transpose of a matrix and `solve()` which returns the inverse of a square matrix if it is invertible.

```
X %*% Y
```

```
##      [,1] [,2] [,3]
## [1,]   90   54   18
## [2,]  114   69   24
## [3,]  138   84   30
```

```
t(X)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
Z = matrix(c(9, 2, -3, 2, 4, -2, -3, -2, 16), 3, byrow = TRUE)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    9    2   -3
## [2,]    2    4   -2
## [3,]   -3   -2   16
```

```
solve(Z)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.12931034 -0.05603448 0.01724138
## [2,] -0.05603448  0.29094828 0.02586207
## [3,]  0.01724138  0.02586207 0.06896552
```

R has a number of matrix specific functions for obtaining dimension and summary information.

```
X = matrix(1:6, 2, 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(X)
```

```
## [1] 2 3
```

```
rowSums(X)
```

```
## [1]  9 12
```

```
colSums(X)
```

```
## [1]  3  7 11
```

```
rowMeans(X)
```

```
## [1] 3 4
```

```
colMeans(X)
```

```
## [1] 1.5 3.5 5.5
```

The `diag()` function can be used in a number of ways. We can extract the diagonal of a matrix.

```
diag(Z)
```

```
## [1]  9  4 16
```

Or create a matrix with specified elements on the diagonal. (And `0` on the off-diagonals.)

```
diag(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

Or, lastly, create a square matrix of a certain dimension with `1` for every element of the diagonal and `0` for the off-diagonals.

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

2.2.8 Data Frames

We have previously seen vectors and matrices for storing data as we introduced `R`. We will now introduce a **data frame** which will be the most common way that we store and interact with data in this course.

```
example_data = data.frame(x = c(1, 3, 5, 7, 9, 1, 3, 5, 7, 9),
                          y = rep("Hello", 10),
                          z = rep(c("TRUE", "FALSE"), 5))
```

Unlike a matrix, which can be thought of as a vector rearranged into rows and columns, a data frame is not required to have the same data type for each element. A data frame is a **list** of vectors. So, each vector must contain the same data type, but the different vectors can store different data types.

```
example_data
```

```
##    x     y     z
## 1  1 Hello  TRUE
## 2  3 Hello FALSE
## 3  5 Hello  TRUE
## 4  7 Hello FALSE
## 5  9 Hello  TRUE
## 6  1 Hello FALSE
## 7  3 Hello  TRUE
## 8  5 Hello FALSE
## 9  7 Hello  TRUE
## 10 9 Hello FALSE
```

The `data.frame()` function above is one way to create a data frame. We can also import data from various file types in into `R`, as well as use data stored in packages.

The example data above can also be found here as a .csv file. To read this data into `R`, we would use the `read.csv()` function.

```
example_data_from_csv = read.csv("data/example_data.csv")
```

This particular line of code assumes that the file `example_data.csv` exists in a folder called `data` in your current working directory.

Alternatively, we could use the "Import Dataset" feature in RStudio which can be found in the environment window. (By default, the top-right pane of RStudio.)

Once completed, this process will automatically generate the code to import a file. The resulting code will be shown in the console window.

Earlier we looked at installing packages, in particular the `ggplot2` package. (A package for visualization. While not necessary for this course, it is quickly growing in popularity.)

```
library(ggplot2)
```

Inside the `ggplot2` package is a dataset called `mpg`. By loading the package using the `library()` function, we can now access `mpg`.

When using data from inside a package, there are three things we would generally like to do:

- Look at the raw data.
- Understand the data. (Where did it come from? What are the variables? Etc.)
- Visualize the data.

To look at the data, we have two useful commands: `head()` and `str()`.

```
head(mpg, n = 10)
```

```
##    manufacturer      model displ year cyl      trans drv cty hwy fl   class
## 1          audi         a4   1.8 1999   4   auto(l5)   f  18  29  p compact
## 2          audi         a4   1.8 1999   4 manual(m5)   f  21  29  p compact
## 3          audi         a4   2.0 2008   4 manual(m6)   f  20  31  p compact
```

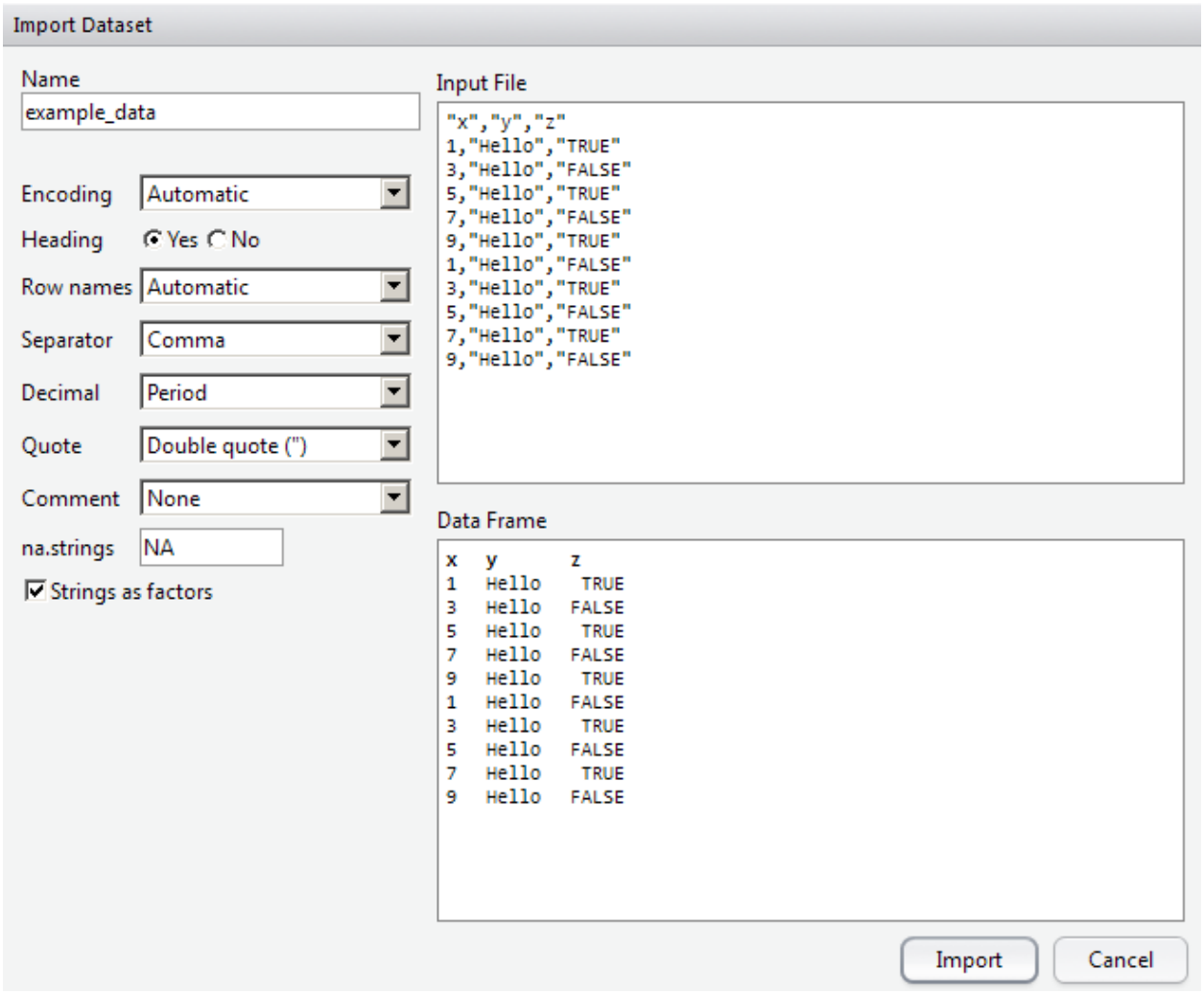


Figure 2.1: RStudio Import Screen

```
## 4          audi         a4   2.0 2008   4   auto(av)   f  21  30  p compact
## 5          audi         a4   2.8 1999   6   auto(l5)   f  16  26  p compact
## 6          audi         a4   2.8 1999   6 manual(m5)   f  18  26  p compact
## 7          audi         a4   3.1 2008   6   auto(av)   f  18  27  p compact
## 8          audi a4 quattro   1.8 1999   4 manual(m5)   4  18  26  p compact
## 9          audi a4 quattro   1.8 1999   4   auto(l5)   4  16  25  p compact
## 10         audi a4 quattro   2.0 2008   4 manual(m6)   4  20  28  p compact
```

The function `head()` will display the first `n` observations of the data frame.

```
str(mpg)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    234 obs. of  11 variables:
##  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
##  $ model       : chr  "a4" "a4" "a4" "a4" ...
##  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr  "p" "p" "p" "p" ...
##  $ class       : chr  "compact" "compact" "compact" "compact" ...
```

The function `str()` will display the "structure" of the data frame. It will display the number of **observations** and **variables**, list the variables, give the type of each variable, and show some elements of each variable.

It is important to note that while matrices have rows and columns, data frames instead have observations and variables. When displayed in the console or viewer, each row is an observation and each column is a variable. However generally speaking, their order does not matter, it is simply a side-effect of how the data was entered or stored.

In this dataset an observation is for a particular model-year of a car, and the variables describe attributes of the car, for example its highway fuel efficiency.

To understand more about the data set, we use the `?` operator to pull up the documentation for the data.

```
?mpg
```

`R` has a number of functions for quickly working with and extracting basic information from data frames. To quickly obtain a vector of the variable names, we use the `names()` function.

```
names(mpg)
```

```
##  [1] "manufacturer" "model"        "displ"        "year"         "cyl"
##  [6] "trans"        "drv"          "cty"          "hwy"          "fl"
## [11] "class"
```

To access one of the variables as a vector, we use the `$` operator.

```
mpg$year
```

```
##   [1] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008 2008
##  [16] 1999 2008 2008 2008 2008 2008 1999 2008 1999 1999 2008 2008 2008 2008 2008
##  [31] 1999 1999 1999 2008 1999 2008 2008 1999 1999 1999 1999 2008 2008 2008 1999
##  [46] 1999 2008 2008 2008 2008 1999 1999 2008 2008 2008 1999 1999 1999 2008 2008
##  [61] 2008 1999 2008 1999 2008 2008 2008 2008 2008 2008 1999 1999 2008 1999 1999
##  [76] 1999 2008 1999 1999 1999 2008 2008 1999 1999 1999 1999 1999 2008 1999 2008
##  [91] 1999 1999 2008 2008 1999 1999 2008 2008 2008 1999 1999 1999 1999 1999 2008
## [106] 2008 2008 2008 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 2008
## [121] 2008 2008 2008 2008 1999 1999 2008 2008 2008 2008 1999 2008 2008 1999 1999
## [136] 1999 2008 1999 2008 2008 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008
## [151] 1999 1999 2008 2008 1999 1999 1999 2008 2008 1999 1999 2008 2008 2008 2008
## [166] 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 1999 1999 2008 2008 1999
## [181] 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008 1999 1999
## [196] 1999 2008 2008 1999 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
## [211] 2008 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 1999 1999 1999 1999
## [226] 2008 2008 1999 1999 2008 2008 1999 1999 2008
```

```
mpg$hwy
```

```
##   [1] 29 29 31 30 26 26 27 26 25 28 27 25 25 25 25 24 25 23 20 15 20 17 17 26 23
##  [26] 26 25 24 19 14 15 17 27 30 26 29 26 24 24 22 22 24 24 17 22 21 23 23 19 18
##  [51] 17 17 19 19 12 17 15 17 17 12 17 16 18 15 16 12 17 17 16 12 15 16 17 15 17
##  [76] 17 18 17 19 17 19 19 17 17 17 16 16 17 15 17 26 25 26 24 21 22 23 22 20 33
## [101] 32 32 29 32 34 36 36 29 26 27 30 31 26 26 28 26 29 28 27 24 24 24 22 19 20
## [126] 17 12 19 18 14 15 18 18 15 17 16 18 17 19 19 17 29 27 31 32 27 26 26 25 25
## [151] 17 17 20 18 26 26 27 28 25 25 24 27 25 26 23 26 26 26 26 25 27 25 27 20 20
## [176] 19 17 20 17 29 27 31 31 26 26 28 27 29 31 31 26 26 27 30 33 35 37 35 15 18
## [201] 20 20 22 17 19 18 20 29 26 29 29 24 44 29 26 29 29 29 29 23 24 44 41 29 26
## [226] 28 29 29 29 28 29 26 26 26
```

We can use the `dim()`, `nrow()` and `ncol()` functions to obtain information about the dimension of the data frame.

```
dim(mpg)
```

```
## [1] 234  11
```

```
nrow(mpg)
```

```
## [1] 234
```

```
ncol(mpg)
```

```
## [1] 11
```

Here `nrow()` is also the number of observations, which in most cases is the *sample size*.

Subsetting data frames can work much like subsetting matrices using square brackets, `[,]`. Here, we find fuel efficient vehicles earning over 35 miles per gallon and only display `manufacturer`, `model` and `year`.

```
mpg[mpg$hwy > 35, c("manufacturer", "model", "year")]
```

```
##     manufacturer      model year
## 106        honda      civic 2008
## 107        honda      civic 2008
## 197       toyota    corolla 2008
## 213   volkswagen      jetta 1999
## 222   volkswagen new beetle 1999
## 223   volkswagen new beetle 1999
```

An alternative would be to use the `subset()` function, which has a much more readable syntax.

```
subset(mpg, subset = hwy > 35, select = c("manufacturer", "model", "year"))
```

Lastly, we could use the `filter` and `select` functions from the `dplyr` package which introduces the `%>%` operator from the `magrittr` package. This is not necessary for this course, however the `dplyr` package is something you should be aware of as it is becoming a popular tool in the `R` world.

```
library(dplyr)
mpg %>% filter(hwy > 35) %>% select(manufacturer, model, year)
```

All three approaches produce the same results. Which you use will be largely based on a given situation as well as user preference.

2.2.9 Plotting

Now that we have some data to work with, and we have learned about the data at the most basic level, our next tasks is to visualize the data. Often, a proper visualization can illuminate features of the data that can inform further analysis.

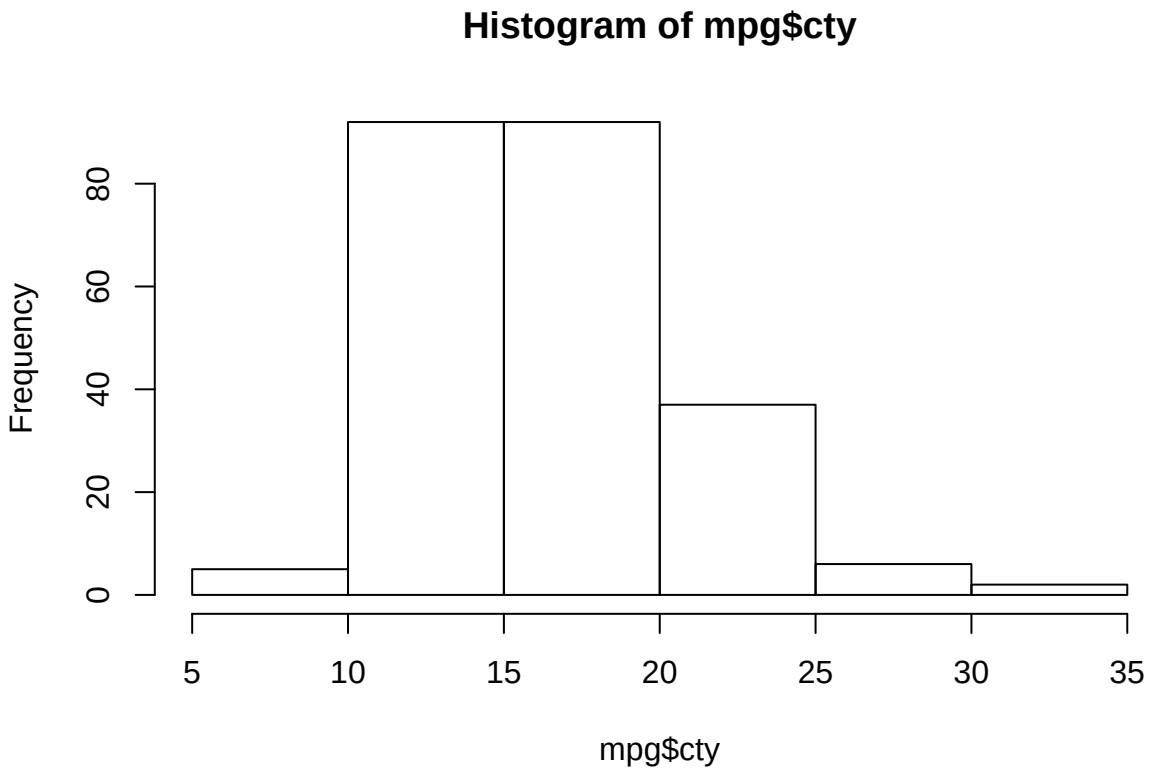
We will look at three methods of visualizing data that we will use throughout the course:

- Histograms
- Boxplots
- Scatterplots

2.2.9.1 Histograms

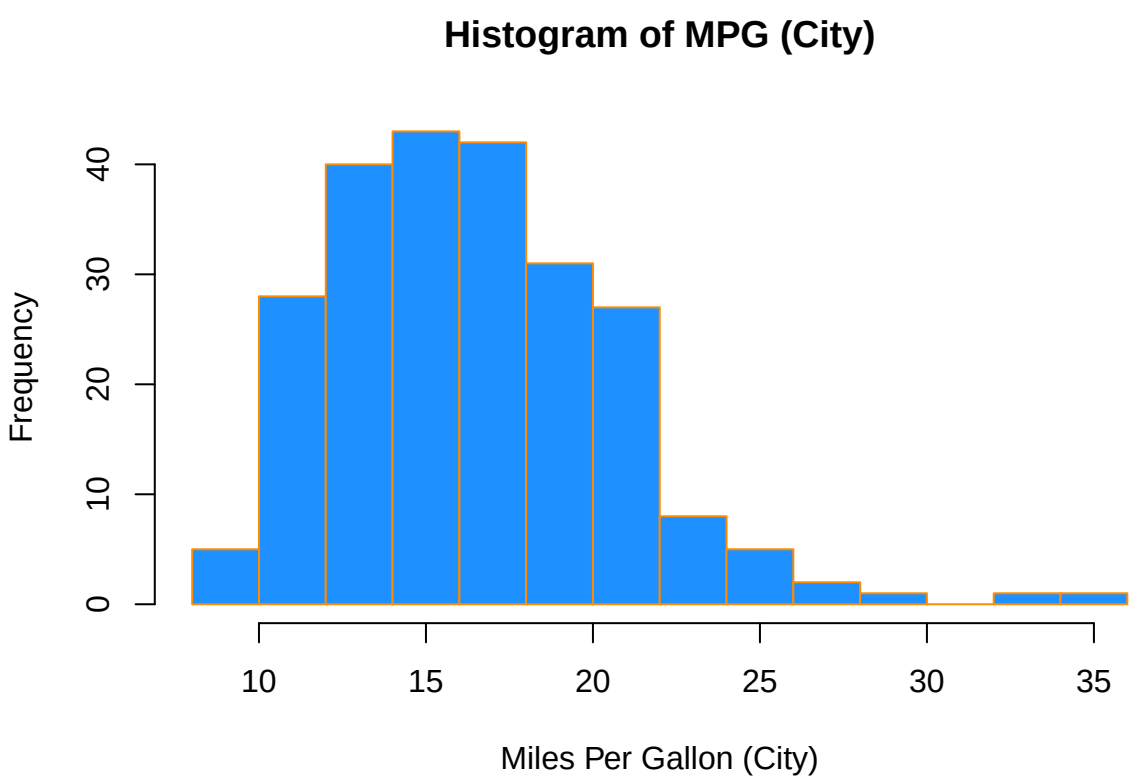
When visualizing a single numerical variable, a **histogram** will be our go-to tool, which can be created in `R` using the `hist()` function.

```
hist(mpg$cty)
```



The histogram function has a number of parameters which can be changed to make our plot look much nicer. Use the `?` operator to read the documentation for the `hist()` to see a full list of these parameters.

```
hist(mpg$cty,
     xlab   = "Miles Per Gallon (City)",
     main   = "Histogram of MPG (City)",
     breaks = 12,
     col    = "dodgerblue",
     border = "darkorange")
```



Importantly, you should always be sure to label your axes and give the plot a title. The argument `breaks` is specific to `hist()`. Entering an integer will give a suggestion to `R` for how many bars to use for the histogram. By default `R` will attempt to intelligently guess a good number of `breaks`, but as we can see here, it is sometimes useful to modify this yourself.

2.2.9.2 Boxplots

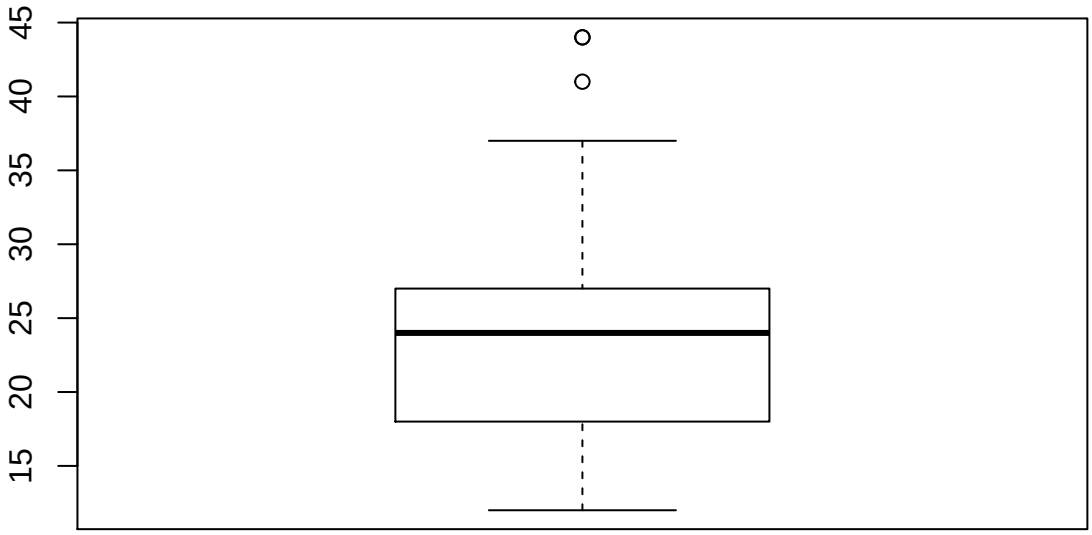
To visualize the relationship between a numerical and categorical variable, we will use a **boxplot**. In the `mpg` dataset, the `drv` variable takes a small, finite number of values. A car can only be front wheel drive, 4 wheel drive, or rear wheel drive.

```
unique(mpg$drv)
```

```
## [1] "f" "4" "r"
```

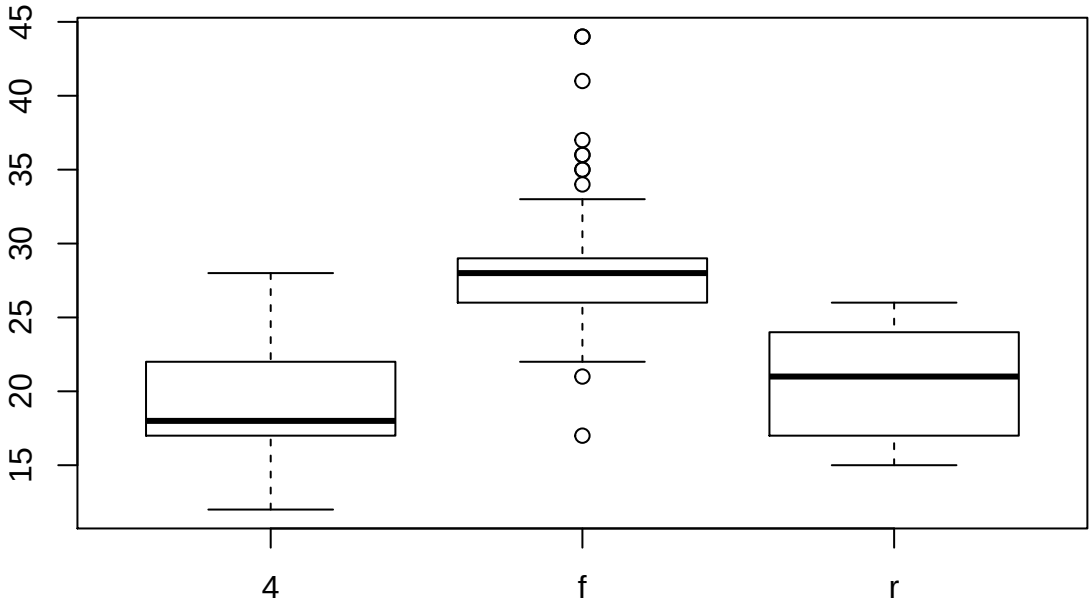
First note that we can use a single boxplot as an alternative to a histogram for visualizing a single numerical variable. To do so in `R`, we use the `boxplot()` function.

```
boxplot(mpg$hwy)
```



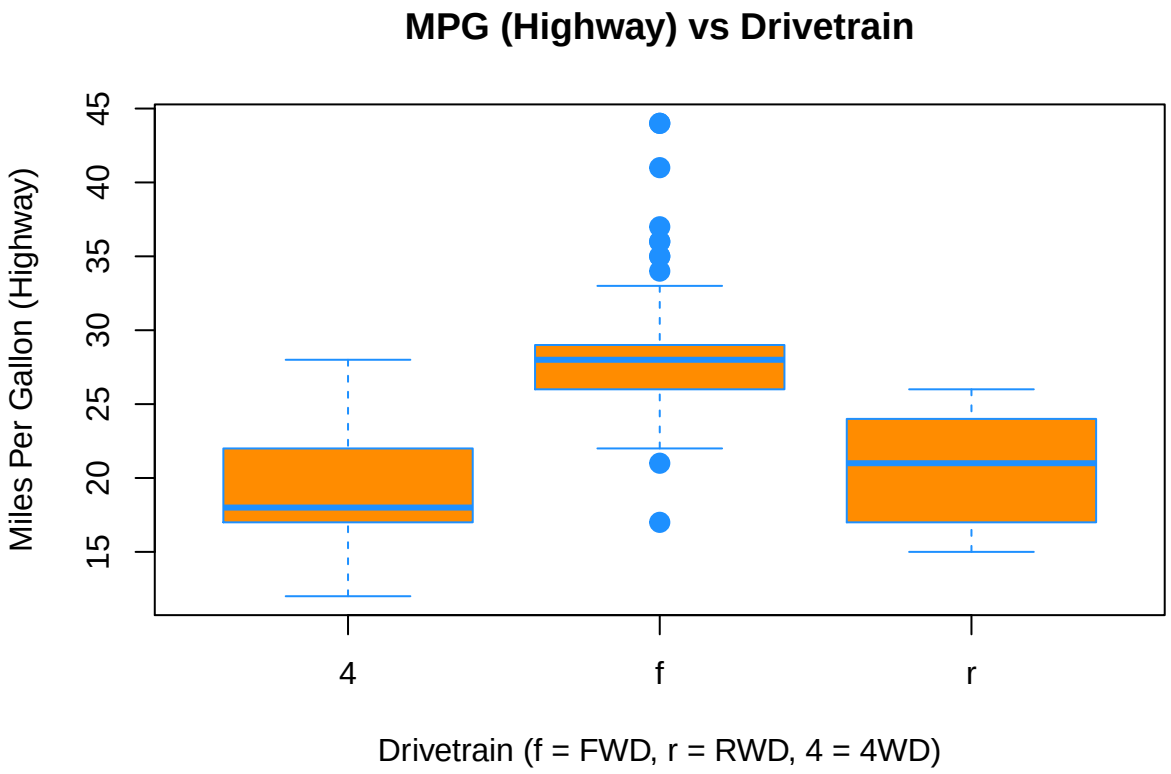
However, more often we will use boxplots to compare a numerical variable for different values of a categorical variable.

```
boxplot(hwy ~ drv, data = mpg)
```



Here used the `boxplot()` command to create side-by-side boxplots. However, since we are now dealing with two variables, the syntax has changed. The `R` syntax `hwy ~ drv, data = mpg` reads "Plot the `hwy` variable against the `drv` variable using the dataset `mpg`." We see the use of a `~` (which specifies a formula) and also a `data =` argument. This will be a syntax that is common to many functions we will use in this course.

```
boxplot(hwy ~ drv, data = mpg,
     xlab   = "Drivetrain (f = FWD, r = RWD, 4 = 4WD)",
     ylab   = "Miles Per Gallon (Highway)",
     main   = "MPG (Highway) vs Drivetrain",
     pch    = 20,
     cex    = 2,
     col    = "darkorange",
     border = "dodgerblue")
```

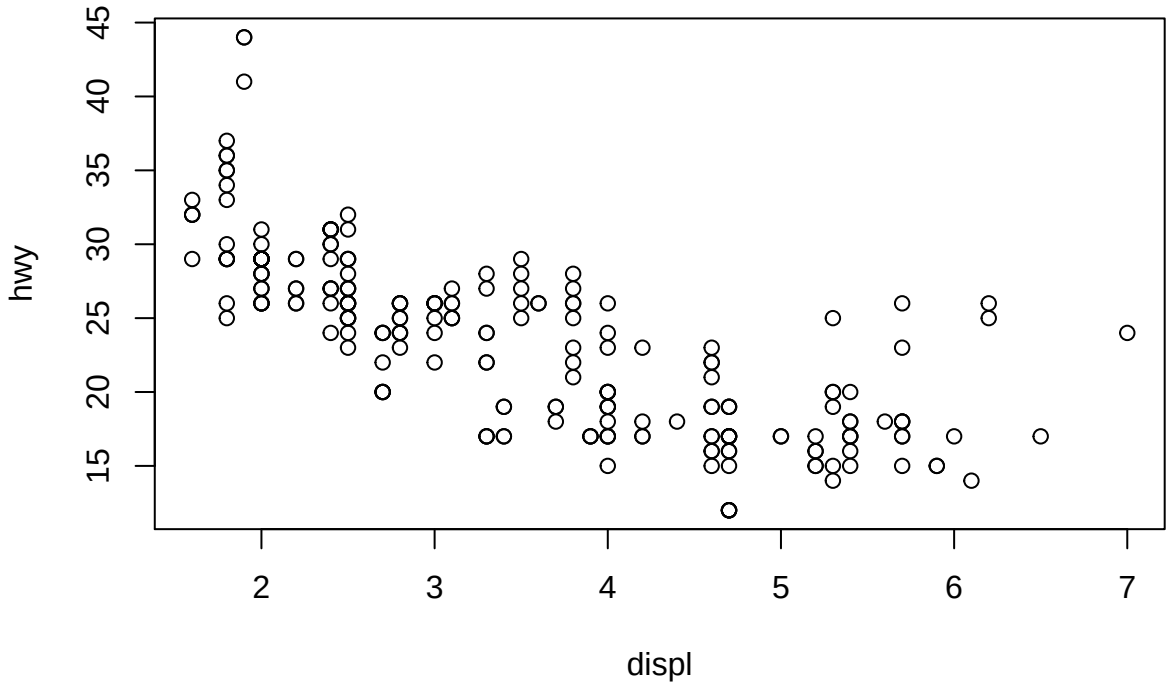


Again, `boxplot()` has a number of additional arguments which have the ability to make our plot more visually appealing.

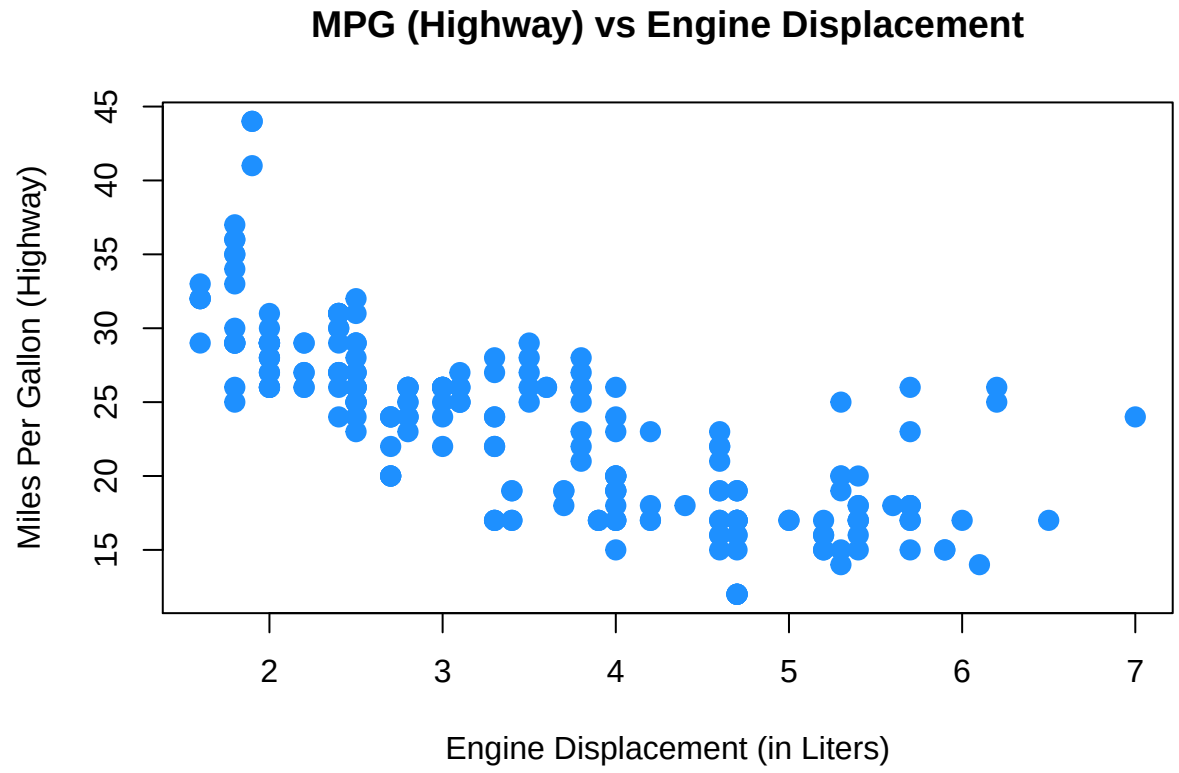
2.2.9.3 Scatterplots

Lastly, to visualize the relationship between two numeric variables we will use a **scatterplot**. This can be done with the `plot()` function and the `~` syntax we just used with a boxplot. (The function `plot()` can also be used more generally; see the documentation for details.)

```
plot(hwy ~ displ, data = mpg)
```




```
plot(hwy ~ displ, data = mpg,
     xlab = "Engine Displacement (in Liters)",
     ylab = "Miles Per Gallon (Highway)",
     main = "MPG (Highway) vs Engine Displacement",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
```



2.2.10 Distributions

When working with different statistical distributions, we often want to make probabilistic statements based on the distribution.

We typically want to know one of four things:

- The density (pdf) at a particular value.
- The distribution (cdf) at a particular value.
- The quantile value corresponding to a particular probability.
- A random draw of values from a particular distribution.

This used to be done with statistical tables printed in the back of textbooks. Now, R has functions for obtaining density, distribution, quantile and random values.

The general naming structure of the relevant R functions is:

- `dname` calculates density (pdf) at input `x`.
- `pname` calculates distribution (cdf) at input `x`.
- `qname` calculates the quantile at an input probability.
- `rname` generates a random draw from a particular distribution.

Note that `name` represents the name of the given distribution.

For example, consider a random variable X which is $N(\mu = 2, \sigma^2 = 25)$. (Note, we are parameterizing using the variance σ^2. R however uses the standard deviation.)

To calculate the value of the pdf at `x = 3`, that is, the height of the curve at `x = 3`, use:

```
dnorm(x = 3, mean = 2, sd = 5)
```

```
## [1] 0.07820854
```

To calculate the value of the cdf at `x = 3`, that is, $P(X \leq 3)$, the probability that X is less than or equal to 3, use:

```
pnorm(q = 3, mean = 2, sd = 5)
```

```
## [1] 0.5792597
```

Or, to calculate the quantile for probability 0.975, use:

```
qnorm(p = 0.975, mean = 2, sd = 5)
```

```
## [1] 11.79982
```

Lastly, to generate a random sample of size `n = 10`, use:

```
rnorm(n = 10, mean = 2, sd = 5)
```

```
##  [1]  -1.0371302  -0.1900134   2.5636793   2.4889558   5.6961443   0.6141446
##  [7]   6.9849835   6.9371509   0.7040066 -10.0461619
```

These functions exist for many other distributions, including but not limited to:

Command	Distribution
`*binom`	Binomial
`*t`	t
`*pois`	Poisson
`*f`	F
`*chisq`	Chi-Squared

Where `*` can be `d`, `p`, `q`, and `r`. Each distribution will have its own set of parameters which need to be passed to the functions as arguments. For example, `dbinom()` would not have arguments for `mean` and `sd`, since those are not parameters of the distribution. Instead a binomial distribution is usually parameterized by n and p, however `R` chooses to call them something else. To find the names that `R` uses we would use `?dbinom` and see that `R` instead calls the arguments `size` and `prob`. For example:

```
dbinom(x = 6, size = 10, prob = 0.75)
```

```
## [1] 0.145998
```

Also note that, when using the `dname` functions with discrete distributions, they are the pmf of the distribution. For example, the above command is $P(Y = 6)$ if $Y \sim b(n = 10, p = 0.75)$. (The probability of flipping an unfair coin `10` times and seeing `6` heads, if the probability of heads is `0.75`.)

2.3 Programming Basics

2.3.1 Logical Operators

Operator	Summary	Example	Result
`x < y`	x less than y	`3 < 42`	TRUE
`x > y`	x greater than y	`3 > 42`	FALSE
`x <= y`	x less than or equal to y	`3 <= 42`	TRUE
`x >= y`	x greater than or equal to y	`3 >= 42`	FALSE
`x == y`	xequal to y	`3 == 42`	FALSE
`x != y`	x not equal to y	`3 != 42`	TRUE
`!x`	not x	`!(3 > 42)`	TRUE
`x | y`	x or y	`(3 > 42) | TRUE`	TRUE
`x & y`	x and y	`(3 < 4) & ( 42 > 13)`	TRUE

In `R`, logical operators are vectorized. To demonstrate this, we will use the following height and weight data.

```
heights = c(110, 120, 115, 136, 205, 156, 175)
weights = c(64, 67, 62, 60, 77, 70, 66)
```

First, using the `<` operator, when can find which `heights` are less than `121`. Further, we could also find which `heights` are less than `121` or exactly equal to `156`.

```
heights < 121
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

```
heights < 121 | heights == 156
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

Often, a vector of logical values is useful for subsetting a vector. For example, we can find the `heights` that are larger than `150`. We can then use the resulting vector to subset the `heights` vector, thus actually returning the `heights` that are above `150`, instead of a vector of which values are above `150`. Here we also obtain the `weights` corresponding to `heights` above `150`.

```
heights > 150
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
heights[heights > 150]
```

```
## [1] 205 156 175
```

```
weights[heights > 150]
```

```
## [1] 77 70 66
```

When comparing vectors, be sure you are comparing vectors of the same length.

```
a = 1:10
b = 2:4
a < b
```

```
## Warning in a < b: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

What happened here? `R` still performed the operation, but it also gives us a warning. (To perform the operation, `R` automatically made `b` longer by repeating `b` as needed.)

The one exception to this behavior is comparing to a vector of length `1`. `R` does not warn us in this case, as comparing each value of a vector to a single value is a common operation that is usually reasonable to perform.

```
a > 5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Often we will want to convert `TRUE` and `FALSE` values to `1` and `0`. When performing mathematical operations on `TRUE` and `FALSE`, this is done automatically through type coercion.

```
5 + (a > 5)
```

```
##  [1] 5 5 5 5 5 6 6 6 6 6
```

By calling `sum()` on a vector of logical values, we can essentially count the number of `TRUE` values.

```
sum(a > 5)
```

```
## [1] 5
```

Here we count the elements of `a` that are larger than `5.` This is an extremely useful feature.

2.3.2 Control Flow

In `R`, the if/else syntax is:

```
if (...) {
  some R code
} else {
  more R code
}
```

For example,

```
x = 1
y = 3
if (x > y) {
  z = x * y
  print("x is larger than y")
} else {
  z = x + 5 * y
  print("x is less than or equal to y")
}
```

```
## [1] "x is less than or equal to y"
```

```
z
```

```
## [1] 16
```

`R` also has a special function `ifelse()` which is very useful. It returns one of two specified values based on a conditional statement.

```
ifelse(4 > 3, 1, 0)
```

```
## [1] 1
```

The real power of `ifelse()` comes from its ability to be applied to vectors.

```
fib = c(1, 1, 2, 3, 5, 8, 13, 21)
ifelse(fib > 6, "Foo", "Bar")
```

```
## [1] "Bar" "Bar" "Bar" "Bar" "Bar" "Foo" "Foo" "Foo"
```

Now a `for` loop example,

```
x = 11:15
for (i in 1:5) {
  x[i] = x[i] * 2
}

x
```

```
## [1] 22 24 26 28 30
```

Note that this `for` loop is very normal in many programming languages, but not in R. In R we would not use a loop, instead we would simply use a vectorized operation.

```
x = 11:15
x = x * 2
x
```

```
## [1] 22 24 26 28 30
```

2.3.3 Functions

So far we have been using functions, but haven't actually discussed some of their details.

```
function_name(arg1 = 10, arg2 = 20)
```

To use a function, you simply type its name, followed by an open parenthesis, then specify values of its arguments, then finish with a closing parenthesis.

An **argument** is a variable which is used in the body of the function. Specifying the values of the arguments is essentially providing the inputs to the function.

We can also write our own functions in R. For example, we often like to "standardize" variables, that is, subtracting the sample mean, and dividing by the sample standard deviation.

$$\frac{x - \bar{x}}{s}$$

In R we would write a function to do this. When writing a function, there are three thing you must do.

- Give the function a name. Preferably something that is short, but descriptive.
- Specify the arguments using `function()`
- Write the body of the function within curly braces, `{}`.

```
standardize = function(x) {
  m = mean(x)
  std = sd(x)
  result = (x - m) / std
  result
}
```

Here the name of the function is `standardize`, and the function has a single argument `x` which is used in the body of function. Note that the output of the final line of the body is what is returned by the function. In this case the function returns the vector stored in the variable `results`.

To test our function, we will take a random sample of size `n = 10` from a normal distribution with a mean of 2 and a standard deviation of 5.

```
(test_sample = rnorm(n = 10, mean = 2, sd = 5))
```

```
##  [1]   6.9256288   0.3512114  -0.6218648  -0.2082259  -2.2960603   3.1285190
##  [7]  13.6674746 -13.3707457  -4.4935080   7.9366455
```

```
standardize(x = test_sample)
```

```
##  [1]  0.7811827 -0.1006969 -0.2312235 -0.1757388 -0.4557968  0.2718458
##  [7]  1.6855209 -1.9413337 -0.7505582  0.9167985
```

This function could be written much more succinctly, simply performing all the operations on one line and immediately returning the result, without storing any of the intermediate results.

```
standardize = function(x) {
  (x - mean(x)) / sd(x)
}
```

When specifying arguments, you can provide default arguments.

```
power_of_num = function(num, power = 2) {
  num ^ power
}
```

Let's look at a number of ways that we could run this function to perform the operation `10^2` resulting in `100`.

```
power_of_num(10)
```

```
## [1] 100
```

```
power_of_num(10, 2)
```

```
## [1] 100
```

```
power_of_num(num = 10, power = 2)
```

```
## [1] 100
```

```
power_of_num(power = 2, num = 10)
```

```
## [1] 100
```

Note that without using the argument names, the order matters. The following code will not evaluate to the same output as the previous example.

```
power_of_num(2, 10)
```

```
## [1] 1024
```

Also, the following line of code would produce an error since arguments without a default value must be specified.

```
power_of_num(power = 5)
```

To further illustrate a function with a default argument, we will write a function that calculates sample standard deviation two ways.

By default, is will calculate the unbiased estimate of σ, which we will call s.

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x-\bar{x})^2}$$

It will also have the ability to return the biased estimate (based on maximum likelihood) which we will call $\hat{\sigma}$.

$$\hat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x-\bar{x})^2}$$

```
get_sd = function(x, biased = FALSE) {
  n = length(x) - 1 * !biased
  sqrt((1 / n) * sum((x - mean(x)) ^ 2))
}
```

```
get_sd(test_sample)
```

```
## [1] 7.455005
```

```
get_sd(test_sample, biased = FALSE)
```

```
## [1] 7.455005
```

```
sd(test_sample)
```

```
## [1] 7.455005
```

We see the function is working as expected, and when returning the unbiased estimate it matches R's built in function `sd()`. Finally, let's examine the biased estimate of σ.

```
get_sd(test_sample, biased = TRUE)
```

```
## [1] 7.072439
```

2.4 Hypothesis Tests in R

2.4.1 One Sample t-Test: Review

Suppose $x_i \sim \text{N}(\mu, \sigma^2)$ and we want to test $H_0 : \mu = \mu_0$ versus $H_1 : \mu \neq \mu_0$.

Assuming σ is unknown, we use the one-sample Student's t test statistic:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} \sim t_{n-1},$$

where $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$ and $s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$.

A $100(1-\alpha)\%$ confidence interval for μ is given by,

$$\bar{x} \pm t_{n-1}(\alpha/2)\frac{s}{\sqrt{n}}$$

where $t_{n-1}(\alpha/2)$ is the critical value such that $P\left(t > t_{n-1}(\alpha/2)\right) = \alpha/2$ for $n-1$ degrees of freedom.

2.4.2 One Sample t-Test: Example

Suppose a grocery store sells "16 ounce" boxes of *Captain Crisp* cereal. A random sample of 9 boxes was taken and weighed. The weight in ounces are stored in the data frame `capt_crisp`.

```
capt_crisp = data.frame(weight = c(15.5, 16.2, 16.1, 15.8, 15.6, 16.0, 15.8, 15.9, 16.2))
```

The company that makes *Captain Crisp* cereal claims that the average weight of a box is at least 16 ounces. We will assume the weight of cereal in a box is normally distributed and use a 0.05 level of significance to test the company's claim.

To test $H_0 : \mu \geq 16$ versus $H_1 : \mu < 16$, the test statistic is

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

The sample mean $\bar{x}$ and the sample standard deviation s can be easily computed using `R`. We also create variables which store the hypothesized mean and the sample size.

```
x_bar = mean(capt_crisp$weight)
s     = sd(capt_crisp$weight)
mu_0  = 16
n     = 9
```

We can then easily compute the test statistic.

```
t = (x_bar - mu_0) / (s / sqrt(n))
t
```

```
## [1] -1.2
```

Under the null hypothesis, the test statistic has a t distribution with $n-1$ degrees of freedom, in this case 8.

To complete the test, we need to obtain the p-value of the test. Since this is a one-sided test with a less-than alternative, we need to area to the left of -1.2 for a t distribution with 8 degrees of freedom. That is,

$$P(t_8 < -1.2)$$

```
pt(t, df = n - 1)
```

```
## [1] 0.1322336
```

We now have the p-value of our test, which is greater than our significance level (0.05), so we fail to reject the null hypothesis.

Alternatively, this entire process could have been completed using one line of R code.

```
t.test(x = capt_crisp$weight, mu = 16, alternative = c("less"), conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  capt_crisp$weight
## t = -1.2, df = 8, p-value = 0.1322
## alternative hypothesis: true mean is less than 16
## 95 percent confidence interval:
##      -Inf 16.05496
## sample estimates:
## mean of x
##      15.9
```

We supply R with the data, the hypothesized value of μ, the alternative, and the confidence level. R then returns a wealth of information including:

- The value of the test statistic.
- The degrees of freedom of the distribution under the null hypothesis.
- The p-value of the test.
- The confidence interval which corresponds to the test.
- An estimate of μ.

Since the test was one-sided, R returned a one-sided confidence interval. If instead we wanted a two-sided interval for the mean weight of boxes of *Captain Crisp* cereal we could modify our code.

```
capt_test_results = t.test(capt_crisp$weight, mu = 16,
                           alternative = c("two.sided"), conf.level = 0.95)
```

This time we have stored the results. By doing so, we can directly access portions of the output from `t.test()`. To see what information is available we use the `names()` function.

```
names(capt_test_results)
```

```
## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

We are interested in the confidence interval which is stored in `conf.int`.

```
capt_test_results$conf.int
```

```
## [1] 15.70783 16.09217
## attr(,"conf.level")
## [1] 0.95
```

Let's check this interval "by hand." The one piece of information we are missing is the critical value, $t_{n-1}(\alpha/2) = t_8(0.025)$, which can be calculated in R using the `qt()` function.

```
qt(0.975, df = 8)
```

```
## [1] 2.306004
```

So, the 95% CI for the mean weight of a cereal box is calculated by plugging into the formula,

$$\bar{x} \pm t_{n-1}(\alpha/2)\frac{s}{\sqrt{n}}$$

```
c(mean(capt_crisp$weight) - qt(0.975, df = 8) * sd(capt_crisp$weight) / sqrt(9),
  mean(capt_crisp$weight) + qt(0.975, df = 8) * sd(capt_crisp$weight) / sqrt(9))
```

```
## [1] 15.70783 16.09217
```

2.4.3 Two Sample t-Test: Review

Suppose $x_i \sim \mathrm{N}(\mu_x, \sigma^2)$ and $y_i \sim \mathrm{N}(\mu_y, \sigma^2)$.

Want to test $H_0 : \mu_x - \mu_y = \mu_0$ versus $H_1 : \mu_x - \mu_y \neq \mu_0$.

Assuming σ is unknown, use the two-sample Student's t test statistic:

$$t = \frac{(\bar{x} - \bar{y}) - \mu_0}{s_p\sqrt{\frac{1}{n} + \frac{1}{m}}} \sim t_{n+m-2},$$

where $\bar{x} = \dfrac{\sum_{i=1}^{n} x_i}{n}$, $\bar{y} = \dfrac{\sum_{i=1}^{m} y_i}{m}$, and $s_p^2 = \dfrac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}$.

A $100(1-\alpha)\%$ CI for $\mu_x - \mu_y$ is given by

$$(\bar{x} - \bar{y}) \pm t_{n+m-2}(\alpha/2)\left(s_p\sqrt{\frac{1}{n} + \frac{1}{m}}\right),$$

where $t_{n+m-2}(\alpha/2)$ is the critical value such that $P\left(t > t_{n+m-2}(\alpha/2)\right) = \alpha/2$.

2.4.4 Two Sample t-Test: Example

Assume that the distributions of X and Y are $\mathrm{N}(\mu_1, \sigma^2)$ and $\mathrm{N}(\mu_2, \sigma^2)$, respectively. Given the $n = 6$ observations of X,

```
x = c(70, 82, 78, 74, 94, 82)
n = length(x)
```

and the $m = 8$ observations of Y,

```
y = c(64, 72, 60, 76, 72, 80, 84, 68)
m = length(y)
```

we will test $H_0 : \mu_1 = \mu_2$ versus $H_1 : \mu_1 > \mu_2$.

First, note that we can calculate the sample means and standard deviations.

```
x_bar = mean(x)
s_x   = sd(x)
y_bar = mean(y)
s_y   = sd(y)
```

We can then calculate the pooled standard deviation.

$$s_p = \sqrt{\frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}}$$

```
s_p = sqrt(((n - 1) * s_x ^ 2 + (m - 1) * s_y ^ 2) / (n + m - 2))
```

Thus, the relevant t test statistic is given by

$$t = \frac{(\bar{x} - \bar{y}) - \mu_0}{s_p \sqrt{\frac{1}{n} + \frac{1}{m}}}.$$

```
t = ((x_bar - y_bar) - 0) / (s_p * sqrt(1 / n + 1 / m))
t
```

```
## [1] 1.823369
```

Note that $t \sim t_{n+m-2} = t_{12}$, so we can calculate the p-value, which is

$$P(t_{12} > 1.8233692).$$

```
1 - pt(t, df = n + m - 2)
```

```
## [1] 0.04661961
```

But, then again, we could have simply performed this test in one line of R.

```
t.test(x, y, alternative = c("greater"), var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  x and y
## t = 1.8234, df = 12, p-value = 0.04662
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  0.1802451       Inf
## sample estimates:
## mean of x mean of y
##        80        72
```

Recall that a two-sample t-test can be done with or without an equal variance assumption. Here `var.equal = TRUE` tells `R` we would like to perform the test under the equal variance assumption.

Above we carried out the analysis using two vectors `x` and `y`. In general, we will have a preference for using data frames.

```
t_test_data = data.frame(values = c(x, y),
                         group  = c(rep("A", length(x)), rep("B", length(y))))
```

We now have the data stored in a single variables (`values`) and have created a second variable (`group`) which indicates which "sample" the value belongs to.

```
t_test_data
```

```
##    values group
## 1      70     A
## 2      82     A
## 3      78     A
## 4      74     A
## 5      94     A
## 6      82     A
## 7      64     B
## 8      72     B
## 9      60     B
## 10     76     B
## 11     72     B
## 12     80     B
## 13     84     B
## 14     68     B
```

Now to perform the test, we still use the `t.test()` function but with the `~` syntax and a `data` argument.

```
t.test(values ~ group, data = t_test_data,
       alternative = c("greater"), var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  values by group
## t = 1.8234, df = 12, p-value = 0.04662
```

```
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  0.1802451       Inf
## sample estimates:
## mean in group A mean in group B
##              80              72
```

2.5 Simulation

One of the biggest strengths of R is its ability to carry out simulations. We'll look at two examples here, however simulation will be a topic we revisit several times throughout the course.

2.5.1 Paired Differences

Consider the model:

$$X_{11}, X_{12}, \ldots, X_{1n} \sim N(\mu_1, \sigma^2)$$
$$X_{21}, X_{22}, \ldots, X_{2n} \sim N(\mu_2, \sigma^2)$$

Assume that $\mu_1 = 6$, $\mu_2 = 5$, $\sigma^2 = 4$ and $n = 25$.

Let

$$\bar{X}_1 = \frac{1}{n}\sum_{i=1}^{n} X_{1i}$$
$$\bar{X}_2 = \frac{1}{n}\sum_{i=1}^{n} X_{2i}$$
$$D = \bar{X}_1 - \bar{X}_2.$$

Suppose we would like to calculate $P(0 < D < 2)$. First we will need to obtain the distribution of D.

Recall,

$$\bar{X}_1 \sim N\left(\mu_1, \frac{\sigma^2}{n}\right)$$

and

$$\bar{X}_2 \sim N\left(\mu_2, \frac{\sigma^2}{n}\right).$$

Then,

$$D = \bar{X}_1 - \bar{X}_2 \sim N\left(\mu_1 - \mu_2, \frac{\sigma^2}{n} + \frac{\sigma^2}{n}\right) = N\left(6 - 5, \frac{4}{25} + \frac{4}{25}\right).$$

So,

$$D \sim N(\mu = 1, \sigma^2 = 0.32).$$

Thus,

$$P(0 < D < 2) = P(D < 2) - P(D < 0).$$

This can then be calculated using `R` without a need to first standardize, or use a table.

```
pnorm(2, mean = 1, sd = sqrt(0.32)) - pnorm(0, mean = 1, sd = sqrt(0.32))
```

```
## [1] 0.9229001
```

An alternative approach, would be to **simulate** a large number of observations of D then use the **empirical distribution** to calculate the probability.

Our strategy will be to repeatedly:

- Generate a sample of 25 random observations from $N(\mu_1 = 6, \sigma^2 = 4)$. Call the mean of this sample $\bar{x}_{1s}$.
- Generate a sample of 25 random observations from $N(\mu_1 = 5, \sigma^2 = 4)$. Call the mean of this sample $\bar{x}_{2s}$.
- Calculate the differences of the means, $d_s = \bar{x}_{1s} - \bar{x}_{2s}$.

We will repeat the process a large number of times. Then we will use the distribution of the simulated observations of d_s as an estimate for the true distribution of D.

```
set.seed(42)
num_samples = 10000
differences = rep(0, num_samples)
```

Before starting our `for` loop to perform the operation, we set a seed for reproducibility, create and set a variable `num_samples` which will define the number of repetitions, and lastly create a variables `differences` which will store the simulate values, d_s.

By using `set.seed()` we can reproduce the random results of `rnorm()` each time starting from that line.

```
for (s in 1:num_samples) {
  x1 = rnorm(n = 25, mean = 6, sd = 2)
  x2 = rnorm(n = 25, mean = 5, sd = 2)
  differences[s] = mean(x1) - mean(x2)
}
```

To estimate $P(0 < D < 2)$ we will find the proportion of values of d_s (among the 10000 values of d_s generated) that are between 0 and 2.

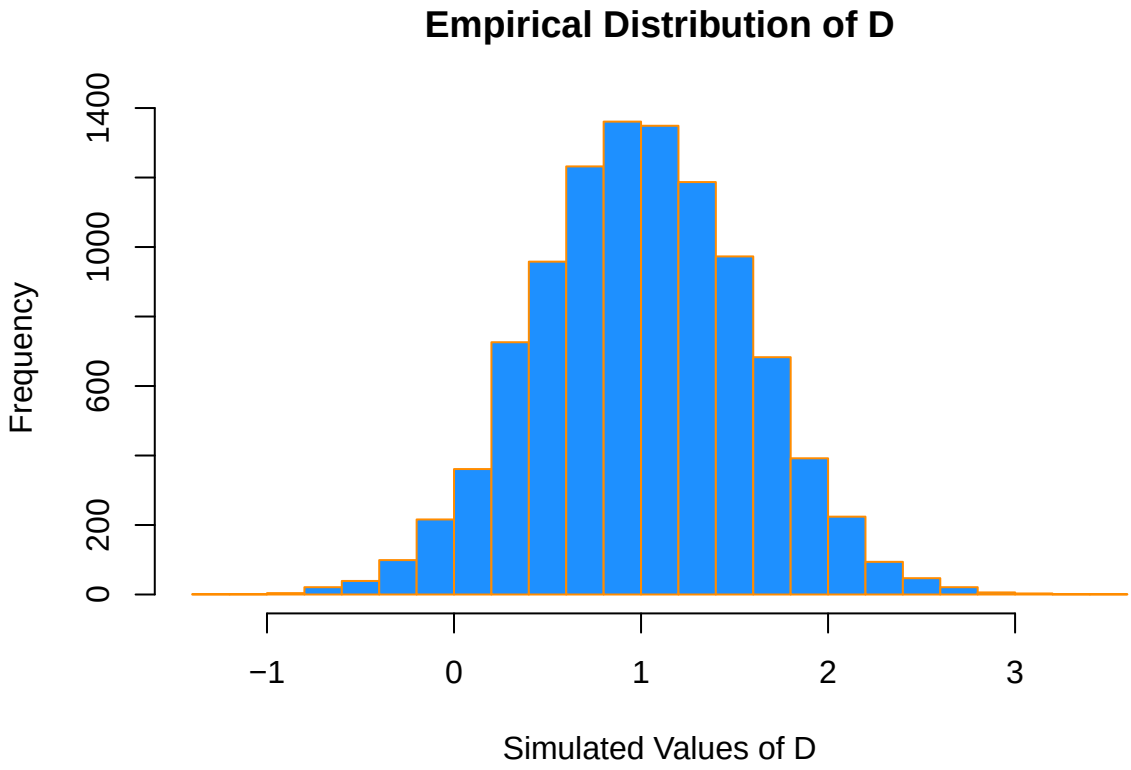
```
mean(0 < differences & differences < 2)
```

```
## [1] 0.9222
```

Recall that above we derived the distribution of D to be $N(\mu = 1, \sigma^2 = 0.32)$

If we look at a histogram of the differences, we find that it looks very much like a normal distribution.

```
hist(differences, breaks = 20,
     main   = "Empirical Distribution of D",
     xlab   = "Simulated Values of D",
     col    = "dodgerblue",
     border = "darkorange")
```



Also the sample mean and variance are very close to to what we would expect.

```
mean(differences)
```

```
## [1] 1.001423
```

```
var(differences)
```

```
## [1] 0.3230183
```

We could have also accomplished this task with a single line of more "idiomatic" R.

```
set.seed(42)
diffs = replicate(10000, mean(rnorm(25, 6, 2)) - mean(rnorm(25, 5, 2)))
```

Use `?replicate` to take a look at the documentation for the `replicate` function and see if you can understand how this line performs the same operations that our `for` loop above executed.

```
mean(differences == diffs)
```

```
## [1] 1
```

We see that by setting the same seed for the randomization, we actually obtain identical results!

2.5.2 Distribution of a Sample Mean

For another example of simulation, we will simulate observations from a Poisson distribution, and examine the empirical distribution of the sample mean of these observations.

Recall, if

$$X \sim Pois(\mu)$$

then

$$E[X] = \mu$$

and

$$Var[X] = \mu.$$

Also, recall that for a random variable X with finite mean μ and finite variance σ^2, the central limit theorem tells us that the mean, $\bar{X}$ of a random sample of size n is approximately normal for *large* values of n. Specifically, as $n \to \infty$,

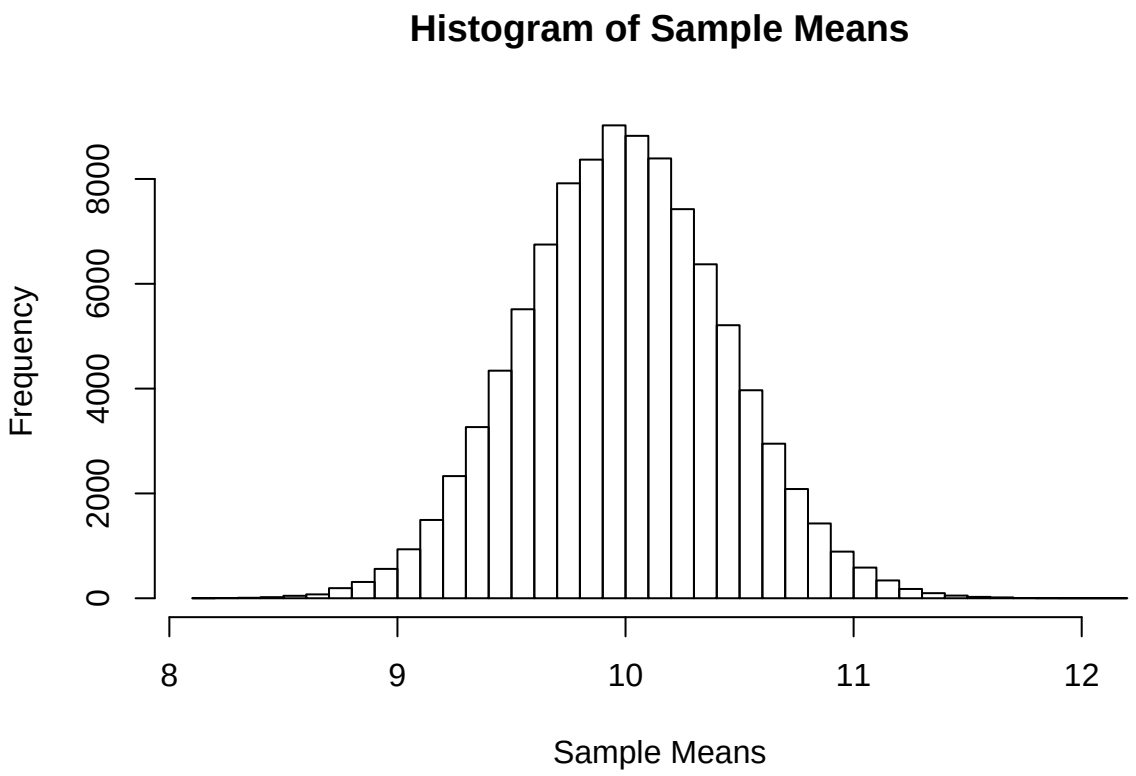
$$\bar{X} \xrightarrow{d} N\left(\mu, \frac{\sigma^2}{n}\right).$$

The following verifies this result for a Poisson distribution with $\mu = 10$ and a sample size of $n = 50$.

```
set.seed(1337)
mu          = 10
sample_size = 50
samples     = 100000
x_bars      = rep(0, samples)
```

```
for(i in 1:samples){
  x_bars[i] = mean(rpois(sample_size, lambda = mu))
}
```

```
x_bar_hist = hist(x_bars, breaks = 50,
                  main = "Histogram of Sample Means",
                  xlab = "Sample Means")
```



Now we will compare sample statistics from the empirical distribution with their known values based on the parent distribution.

```
c(mean(x_bars), mu)
```

```
## [1] 10.00008 10.00000
```

```
c(var(x_bars), mu / sample_size)
```

```
## [1] 0.1989732 0.2000000
```

```
c(sd(x_bars), sqrt(mu) / sqrt(sample_size))
```

```
## [1] 0.4460641 0.4472136
```

And here, we will calculate the proportion of sample means that are within 2 standard deviations of the population mean.

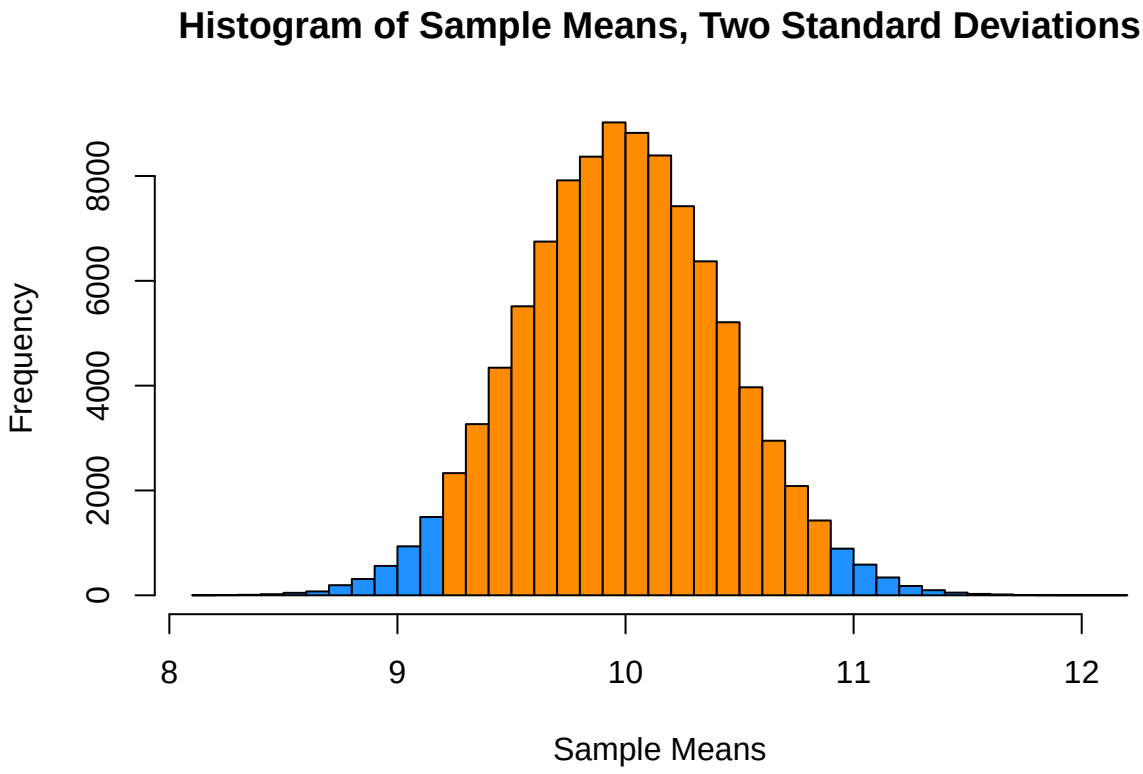
```
mean(x_bars > mu - 2 * sqrt(mu) / sqrt(sample_size) &
     x_bars < mu + 2 * sqrt(mu) / sqrt(sample_size))
```

```
## [1] 0.95429
```

This last histogram uses a bit of a trick to approximately shade the bars that are within two standard deviations of the mean.)

```
shading = ifelse(x_bar_hist$breaks > mu - 2 * sqrt(mu) / sqrt(sample_size) &
                   x_bar_hist$breaks < mu + 2 * sqrt(mu) / sqrt(sample_size),
                 "darkorange", "dodgerblue")

x_bar_hist = hist(x_bars, breaks = 50, col = shading,
                  main = "Histogram of Sample Means, Two Standard Deviations",
                  xlab = "Sample Means")
```



Chapter 3

Simple Linear Regression

> "All models are wrong, but some are useful."
>
> — **George E. P. Box**

After reading this chapter you will be able to:

- Understand the concept of a model.
- Describe two ways in which regression coefficients are derived.
- Estimate and visualize a regression model using `R`.
- Interpret regression coefficients and statistics in the context of real-world problems.
- Use a regression model to make predictions.

3.1 Modeling

Let's consider a simple example of how the speed of a car affects its stopping distance, that is, how far it travels before it comes to a stop. To examine this relationship, we will use the `cars` dataset which, is a default `R` dataset. Thus, we don't need to load a package first; it is immediately available.

To get a first look at the data you can use the `View()` function inside RStudio.

```
View(cars)
```

We could also take a look at the variable names, the dimension of the data frame, and some sample observations with `str()`.

```
str(cars)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

As we have seen before with data frames, there are a number of additional functions to access some of this information directly.

```
dim(cars)
```

```
## [1] 50  2
```

```
nrow(cars)
```

```
## [1] 50
```

```
ncol(cars)
```

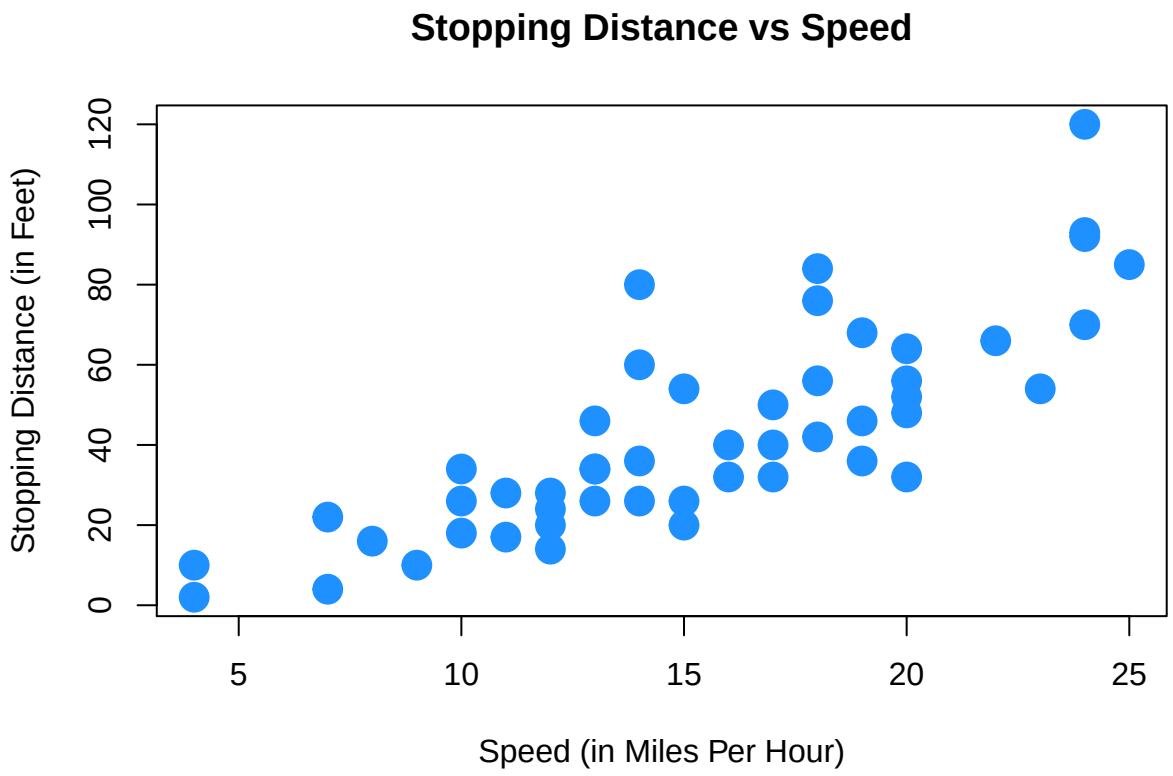
```
## [1] 2
```

Other than the two variable names and the number of observations, this data is still just a bunch of numbers, so we should probably obtain some context.

```
?cars
```

Reading the documentation we learn that this is data gathered during the 1920s about the speed of cars and the resulting distance it takes for the car to come to a stop. The interesting task here is to determine how far a car travels before stopping, when traveling at a certain speed. So, we will first plot the stopping distance against the speed.

```
plot(dist ~ speed, data = cars,
     xlab = "Speed (in Miles Per Hour)",
     ylab = "Stopping Distance (in Feet)",
     main = "Stopping Distance vs Speed",
     pch  = 20,
     cex  = 3,
     col  = "dodgerblue")
```



Let's now define some terminology. We have pairs of data, (x_i, y_i), for $i = 1, 2, \ldots n$, where n is the sample size of the dataset.

We use i as an index, simply for notation. We use x_i as the **predictor** (explanatory) variable. The predictor variable is used to help *predict* or explain the **response** (target, outcome) variable, y_i.

Other texts may use the term independent variable instead of predictor and dependent variable in place of response. However, those monikers imply mathematical characteristics that might not be true. While these others terms are not incorrect, independence is already a strictly defined concept in probability. For example, when trying to predict a person's weight given their height, would it be accurate to say that height is independent of weight? Certainly not, but that is an unintended implication of saying "independent variable." We prefer to stay away from this nomenclature.

In the `cars` example, we are interested in using the predictor variable `speed` to predict and explain the response variable `dist`.

Broadly speaking, we would like to model the relationship between X and Y using the form

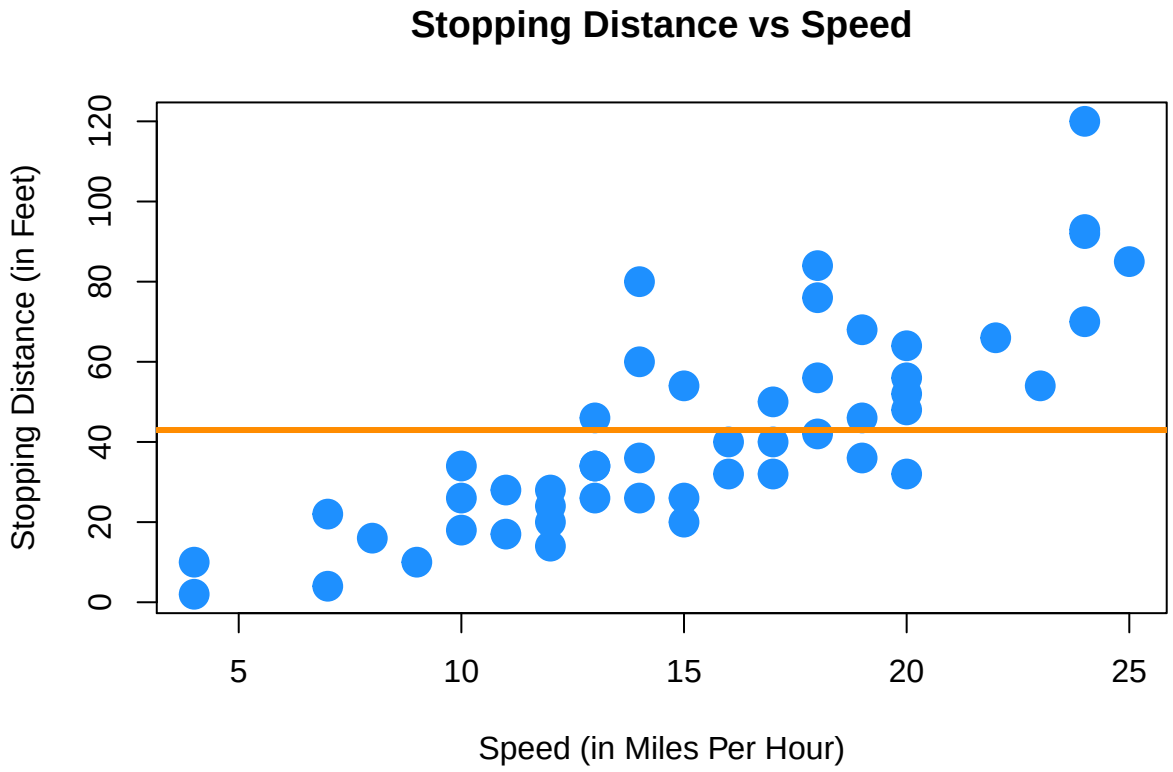
$$Y = f(X) + \epsilon.$$

The function f describes the functional relationship between the two variables, and the ϵ term is used to account for error. This indicates that if we plug in a given value of X as input, our output is a value of Y, within a certain range of error. You could think of this a number of ways:

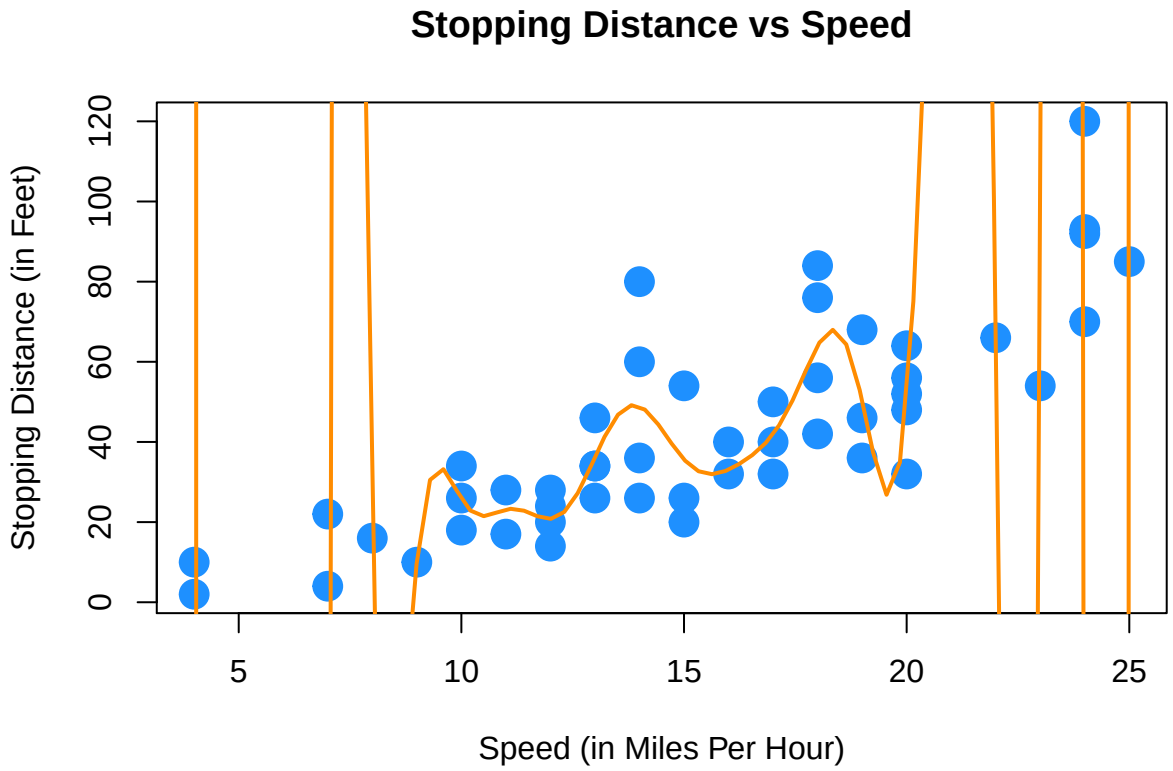
- Response = Prediction + Error
- Response = Signal + Noise
- Response = Model + Unexplained
- Response = Explainable + Unexplainable

What sort of function should we use for $f(X)$ for the `cars` data?

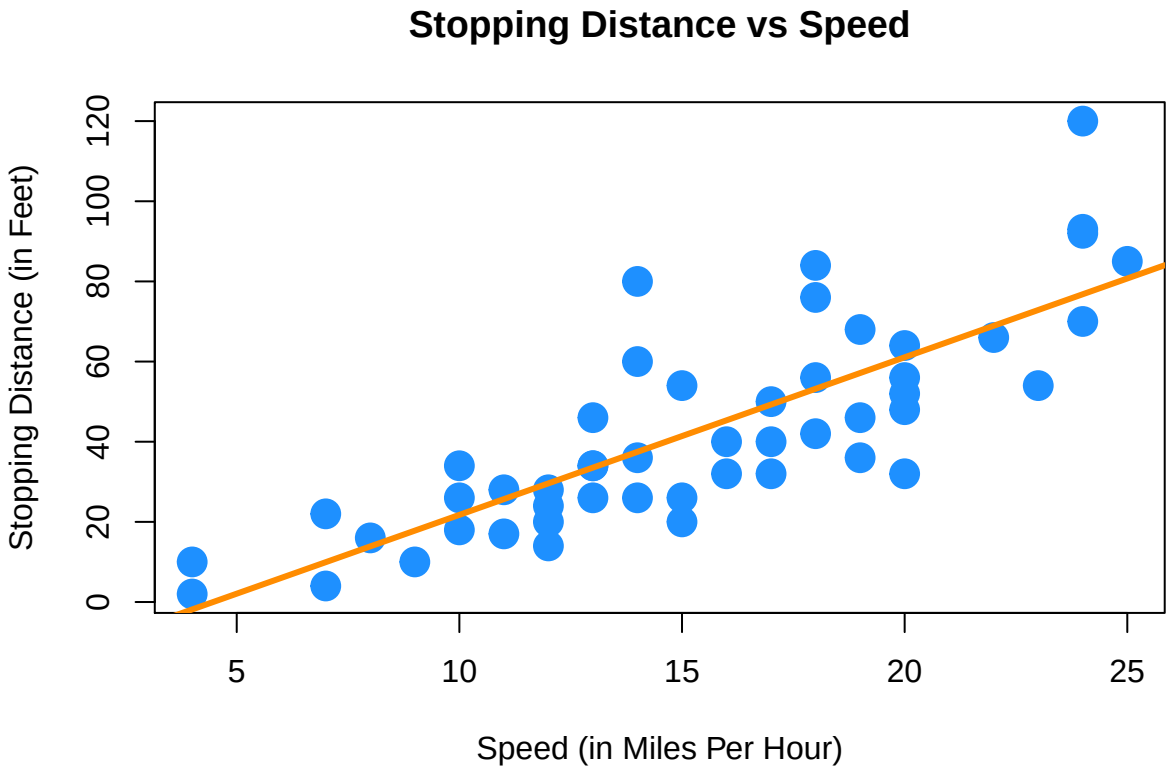
We could try to model the data with a horizontal line. That is, the model for y does not depend on the value of x. (Some function $f(X) = c$.) In the plot below, we see this doesn't seem to do a very good job. Many of the data points are very far from the orange line representing c. This is an example of **underfitting**. The obvious fix is to use make the function $f(X)$ actually depend on x.



We could also try to model the data with a very "wiggly" function that tries to go through as many of the data points as possible. This also doesn't seem to work very well. The stopping distance for a speed of 5 mph shouldn't be off the chart! (Even in 1920.) This is an example of **overfitting**. (Note that in this example no function will go through every point, since there are some x values that have several possible y values in the data.)



Lastly, we could try to model the data with a well chosen line rather than one of the two extremes previously attempted. The line on the plot below seems to summarize the relationship between stopping distance and speed quite well. As speed increases, the distance required to come to a stop increases. There is still some variation about this line, but it seems to capture the overall trend.



With this in mind, we would like to restrict our choice of $f(X)$ to *linear* functions of X. We will write our model using β_1 for the slope, and β_0 for the intercept,

$$Y = \beta_0 + \beta_1 X + \epsilon.$$

3.1.1 Simple Linear Regression Model

We now define what we will call the simple linear regression model,

$$Y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2)$. That is, the ϵ_i are *independent and identically distributed* (iid) normal random variables with mean 0 and variance σ^2. This model has three parameters to be estimated: β_0, β_1, and σ^2, which are fixed, but unknown constants.

We have slightly modified our notation here. We are now using Y_i and x_i, since we will be fitting this model to a set of n data points, for $i = 1, 2, \ldots n$.

Recall that we use capital Y to indicate a random variable, and lower case y to denote a potential value of the random variable. Since we will have n observations, we have n random variables Y_i and their possible values y_i.

In the simple linear regression model, the x_i are assumed to be fixed, known constants, and are thus notated with a lower case variable. The response Y_i remains a random variable because of the random behavior of the error variable, ϵ_i. That is, each response Y_i is tied to an observable x_i and a random, unobservable, ϵ_i.

The random Y_i are a function of x_i, thus we can write its mean as a function of x_i,

$$E[Y_i] = \beta_0 + \beta_1 x_i.$$

However, its variance remains constant for each x_i,

$$Var[Y_i] = \sigma^2.$$

This is visually displayed in the image below. We see that for any value x, the expected value of Y is $\beta_0 + \beta_1 x$. At each value of x, Y has the same variance σ^2.

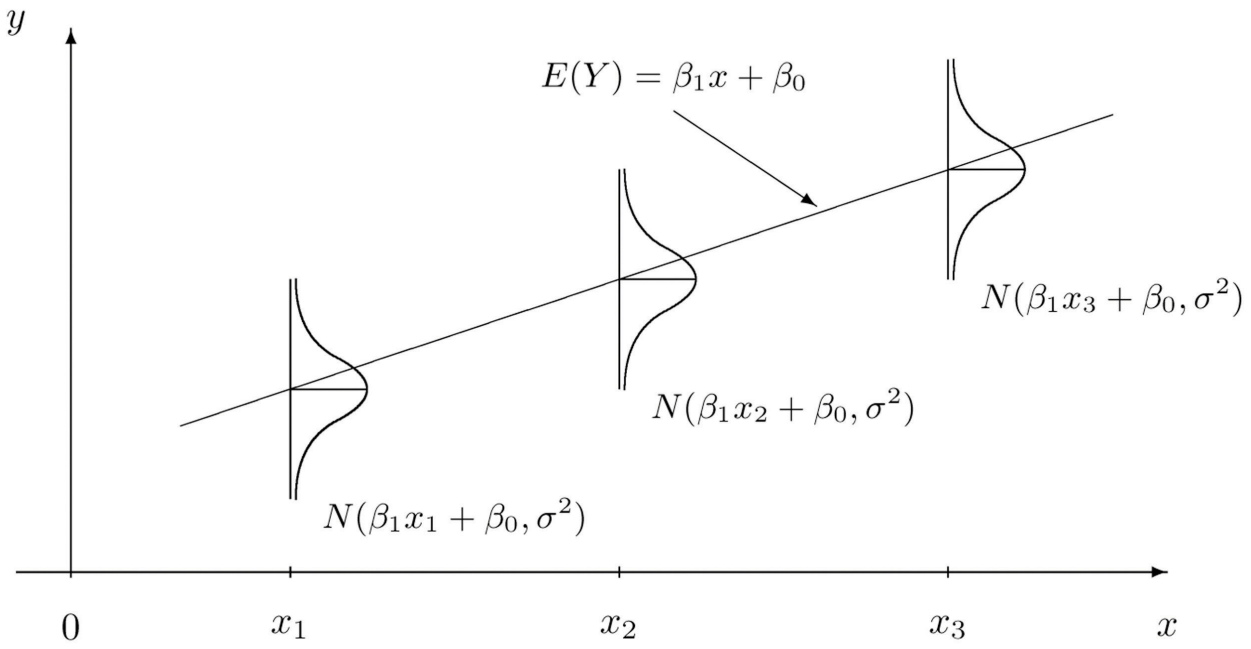


Figure 3.1: Simple Linear Regression Model UC David Stat Wiki

Often, we directly talk about the assumptions that this model makes. They can be cleverly shortened to **LINE**.

- **L**inear. The relationship between Y and x is linear, of the form $\beta_0 + \beta_1 x$.
- **I**ndepedent. The errors ϵ are independent.
- **N**ormal. The errors, ϵ are normally distributed. That is the "error" around the line follows a normal distribution.
- **E**qual Variance. At each value of x, the variance of Y is the same, σ^2.

As a side note, we will often refer to simple linear regression as **SLR**. Some explanation of the name SLR:

- **Simple** refers to the fact that we are using a single predictor variable. Later we will use multiple predictor variables.
- **Linear** tells us that our model for Y is a linear combination of the predictors X. (In this case just the one.) Right now, this always results in a model that is a line, but later we will see how this is not always the case.
- **Regression** simply means that we are attempting to measure the relationship between a response variable and (one or more) predictor variables.

So SLR models Y as a linear function of X, but how do we actually define a good line? There are an infinite number of lines we could use, so we will attempt to find one with "small errors." That is a line with as many points as close to it as possible. The questions now becomes, how do we find such a line? There are many approaches we could take.

We could find the line that has the smallest maximum distance from any of the points to the line. That is,

$$\underset{\beta_0,\beta_1}{\operatorname{argmin}} \max |y_i - (\beta_0 + \beta_1 x_i)|.$$

We could find the line that minimizes the sum of all the distances from the points to the line. That is,

$$\underset{\beta_0,\beta_1}{\operatorname{argmin}} \sum_{i=1}^{n} |y_i - (\beta_0 + \beta_1 x_i)|.$$

We could find the line that minimizes the sum of all the squared distances from the points to the line. That is,

$$\underset{\beta_0,\beta_1}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2.$$

This last option is called the method of **least squares**. It is essentially the de-facto method for fitting a line to data. (You may have even seen it before in a linear algebra course.) Its popularity is largely due to the fact that it is mathematically "easy." (Which was important historically, as computers are a modern contraption.) It is also very popular because many relationships are well approximated by a linear function.

3.2 Least Squares Approach

Given observations (x_i, y_i), for $i = 1, 2, \ldots n$, we want to find values of β_0 and β_1 which minimize

$$f(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2 = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2.$$

We will call these values $\hat{\beta}_0$ and $\hat{\beta}_1$.

First, we take a partial derivative with respect to both β_0 and β_1.

$$\frac{\partial f}{\partial \beta_0} = -2 \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)$$
$$\frac{\partial f}{\partial \beta_1} = -2 \sum_{i=1}^{n} (x_i)(y_i - \beta_0 - \beta_1 x_i)$$

We then set each of the partial derivatives equal to zero and solving the resulting system of equations.

$$\sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) = 0$$
$$\sum_{i=1}^{n} (x_i)(y_i - \beta_0 - \beta_1 x_i) = 0$$

While solving the system of equations, one common algebraic rearrangement results in the **normal equations**.

$$\sum_{i=1}^{n} y_i = n\beta_0 + \beta_1 \sum_{i=1}^{n} x_i$$
$$\sum_{i=1}^{n} x_i y_i = \beta_0 \sum_{i=1}^{n} x_i + \beta_1 \sum_{i=1}^{n} x_i^2$$

Finally, we finish solving the system of equations.

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} x_i y_i - \frac{(\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n}}{\sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n}} = \frac{S_{xy}}{S_{xx}}$$
$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

Here, we have defined some notation for the expression we've obtained. Note that they have alternative forms which are much easier to work with. (We won't do it here, but you can try to prove the equalities below on your own, for "fun.") We use the capital letter S to denote "summation" which replaces the capital letter Σ when we calculate these values based on observed data, (x_i, y_i). The subscripts such as xy denote over which variables the function $(z - \bar{z})$ is applied.

$$S_{xy} = \sum_{i=1}^{n} x_i y_i - \frac{(\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n} = \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$
$$S_{xx} = \sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n} = \sum_{i=1}^{n} (x_i - \bar{x})^2$$
$$S_{yy} = \sum_{i=1}^{n} y_i^2 - \frac{(\sum_{i=1}^{n} y_i)^2}{n} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Note that these summations S are not to be confused with sample standard deviation s.

By using the above alternative expressions for S_{xy} and S_{xx}, we arrive at a cleaner, more useful expression for $\hat{\beta}_1$.

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Traditionally we would now calculate $\hat{\beta}_0$ and $\hat{\beta}_1$ by hand for the `cars` dataset. However because we are living in the 21st century and are intelligent (or lazy or efficient, depending on your perspective), we will utilize `R` to do the number crunching for us.

To keep some notation consistent with above mathematics, we will store the response variable as `y` and the predictor variable as `x`.

```
x = cars$speed
y = cars$dist
```

We then calculate the three sums of squares defined above.

```
Sxy = sum((x - mean(x)) * (y - mean(y)))
Sxx = sum((x - mean(x)) ^ 2)
Syy = sum((y - mean(y)) ^ 2)
c(Sxy, Sxx, Syy)
```

```
## [1]  5387.40  1370.00 32538.98
```

Then finally calculate $\hat{\beta}_0$ and $\hat{\beta}_1$.

```
beta_1_hat = Sxy / Sxx
beta_0_hat = mean(y) - beta_1_hat * mean(x)
c(beta_0_hat, beta_1_hat)
```

```
## [1] -17.579095   3.932409
```

What do these values tell us about our dataset?

The slope *parameter* β_1 tells us that for an increase in speed of one mile per hour, the **mean** stopping distance increases by β_1. It is important to specify that we are talking about the mean. Recall that $\beta_0 + \beta_1 x$ is the estimated mean of Y, in this case stopping distance, for a particular value of x. (In this case speed.) So β_1 tells us how the mean of Y is affected by a change in x.

Similarly, the *estimate* $\hat{\beta}_1 = 3.932$ tells us that for an increase in speed of one mile per hour, the **estimated** *mean* stopping distance increases by 3.932 feet. Here we should be sure to specify we are discussing an estimated quantity. Recall that $\hat{y}$ is the estimated mean of Y, so $\hat{\beta}_1$ tells us how the estimated mean of Y is affected by changing x.

The intercept *parameter* β_0 tells us the **mean** stopping distance for a car traveling zero miles per hour. (Not moving.) The *estimate* $\hat{\beta}_0 = -17.579$ tells us that the **estimated** mean stopping distance for a car traveling zero miles per hour is -17.579 feet. So when you apply the brakes to a car that is not moving, it moves backwards? This doesn't seem right. (Extrapolation, which we will see later, is the issue here.)

3.2.1 Making Predictions

We can now write the **fitted** or estimated line,

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x.$$

In this case,

$$\hat{y} = -17.579 + 3.932x.$$

We can now use this line to make predictions. First, let's see the possible x values in the `cars` dataset. Since some x values may appear more than once, we use the `unique()` to return each unique value only once.

```
unique(cars$speed)
```

```
##  [1]  4  7  8  9 10 11 12 13 14 15 16 17 18 19 20 22 23 24 25
```

Let's make a prediction for the stopping distance of a car traveling at 8 miles per hour.

$$\hat{y} = -17.579 + 3.932 \times 8 = 13.88$$

```
beta_0_hat + beta_1_hat * 8
```

```
## [1] 13.88018
```

This tells us that the estimated mean stopping distance of a car traveling at 8 miles per hour is 13.88.

Now let's make a prediction for the stopping distance of a car traveling at 21 miles per hour. This is considered **interpolation** as 21 is not an observed value of x. (But is in the data range.) We can use the special `%in%` operator to quickly verify this in `R`.

```
8 %in% unique(cars$speed)
```

```
## [1] TRUE
```

```
21 %in% unique(cars$speed)
```

```
## [1] FALSE
```

```
min(cars$speed) < 21 & 21 < max(cars$speed)
```

```
## [1] TRUE
```

$$\hat{y} = -17.579 + 3.932 \times 21 = 65.001$$

```
beta_0_hat + beta_1_hat * 21
```

```
## [1] 65.00149
```

Lastly, we can make a prediction for the stopping distance of a car traveling at 50 miles per hour. This is considered **extrapolation** as 50 is not an observed value of x and is outside data range. We should be less confident in predictions of this type.

```
range(cars$speed)
```

```
## [1]  4 25
```

```
range(cars$speed)[1] < 50 & 50 < range(cars$speed)[2]
```

```
## [1] FALSE
```

$$\hat{y} = -17.579 + 3.932 \times 50 = 179.041$$

```
beta_0_hat + beta_1_hat * 50
```

```
## [1] 179.0413
```

Cars travel 50 miles per hour rather easily today, but not in the 1920s!

This is also an issue we saw when interpreting $\hat{\beta}_0 = -17.579$, which is equivalent to making a prediction at $x = 0$. We should not be confident in the estimated linear relationship outside of the range of data we have observed.

3.2.2 Residuals

If we think of our model as "Response = Prediction + Error," we can then write it as

$$y = \hat{y} + e.$$

We then define a **residual** to be the observed value minus the predicted value.

$$e_i = y_i - \hat{y}_i$$

Let's calculate the residual for the prediction we made for a car traveling 8 miles per hour. First, we need to obtain the observed value of y for this x value.

```
which(cars$speed == 8)
```

```
## [1] 5
```

```
cars[5, ]
```

```
##   speed dist
## 5     8   16
```

```
cars[which(cars$speed == 8), ]
```

```
##   speed dist
## 5     8   16
```

We can then calculate the residual.

$$e = 16 - 13.88 = 2.12$$

```
16 - (beta_0_hat + beta_1_hat * 8)
```

```
## [1] 2.119825
```

The positive residual value indicates that the observed stopping distance is actually 2.12 feet more than what was predicted.

3.2.3 Variance Estimation

We'll now use the residuals for each of the points to create an estimate for the variance, σ^2.

Recall that,

$$E[Y_i] = \beta_0 + \beta_1 x_i.$$

So,

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$$

is a natural estimate for the mean of Y_i for a given value of x_i.

Also, recall that when we specified the model, we had three unknown parameters; β_0, β_1, and σ^2. The method of least squares gave us estimates for β_0 and β_1, however, we have yet to see an estimate for σ^2. We will now define s_e^2 which will be an estimate for σ^2.

$$\begin{aligned} s_e^2 &= \frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \\ &= \frac{1}{n-2}\sum_{i=1}^{n} e_i^2 \end{aligned}$$

This probably seems like a natural estimate, aside from the use of $n-2$, which we will put off explaining until the next chapter. It should actually look rather similar to something we have seen before.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Here, s^2 is the estimate of σ^2 when we have a single random variable X. In this case $\bar{x}$ is an estimate of μ which is assumed to be the same for each x.

Now, in the regression case, with s_e^2 each y has a different mean because of the relationship with x. Thus, for each y_i, we use a different estimate of the mean, that is $\hat{y}_i$.

```
y_hat = beta_0_hat + beta_1_hat * x
e     = y - y_hat
n     = length(e)
s2_e  = sum(e^2) / (n - 2)
s2_e
```

```
## [1] 236.5317
```

Just as with the univariate measure of variance, this value of 236.532 doesn't have a practical interpretation in terms of stopping distance. Taking the square root, however, computes the standard deviation of the residuals, also known as *residual standard error*.

```
s_e = sqrt(s2_e)
s_e
```

```
## [1] 15.37959
```

This tells us that our estimates of mean stopping distance are "typically" off by 15.38 feet.

3.3 Decomposition of Variation

We can re-express $y_i - \bar{y}$, which measures the deviation of an observation from the sample mean, in the following way,

$$y_i - \bar{y} = (y_i - \hat{y}_i) + (\hat{y}_i - \bar{y}).$$

This is the common mathematical trick of "adding zero." In this case we both added and subtracted $\hat{y}_i$.

Here, $y_i - \hat{y}_i$ measures the deviation of an observation from the fitted regression line and $\hat{y}_i - \bar{y}$ measures the deviation of the fitted regression line from the sample mean.

If we square then sum both sides of the equation above, we can obtain the following,

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2.$$

This should be somewhat alarming or amazing. How is this true? For now we will leave this questions unanswered. (Think about this, and maybe try to prove it.) We will now define three of the quantities seen in this equation.

3.3.0.1 Sum of Squares Total

$$SST = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

The quantity "Sum of Squares Total," or SST, represents the **total variation** of the observed y values. This should be a familiar looking expression. Note that,

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2 = \frac{1}{n-1}SST.$$

3.3.0.2 Sum of Squares Regression

$$SSReg = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$

The quantity "Sum of Squares Regression," $SSReg$, represents the **explained variation** of the observed y values.

3.3.0.3 Sum of Squares Error

$$SSE = RSS = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

The quantity "Sum of Squares Error," SSE, represents the **unexplained variation** of the observed y values. You will often see SSE written as RSS, or "Residual Sum of Squares."

```
SST   = sum((y - mean(y)) ^ 2)
SSReg = sum((y_hat - mean(y)) ^ 2)
SSE   = sum((y - y_hat) ^ 2)
c(SST = SST, SSReg = SSReg, SSE = SSE)
```

```
##      SST    SSReg      SSE
## 32538.98 21185.46 11353.52
```

Note that,

$$s_e^2 = \frac{SSE}{n-2}.$$

```
SSE / (n - 2)
```

```
## [1] 236.5317
```

We can use `R` to verify that this matches our previous calculation of s_e^2.

```
s2_e == SSE / (n - 2)
```

```
## [1] TRUE
```

These three measures also do not have an important practical interpretation individually. But together, they're about to reveal a new statistic to help measure the strength of a SLR model.

3.3.1 Coefficient of Determination

The **coefficient of determination**, R^2, is defined as

$$R^2 = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = \frac{SSReg}{SST} = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{\sum_{i=1}^{n} e_i^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{SSE}{SST}$$

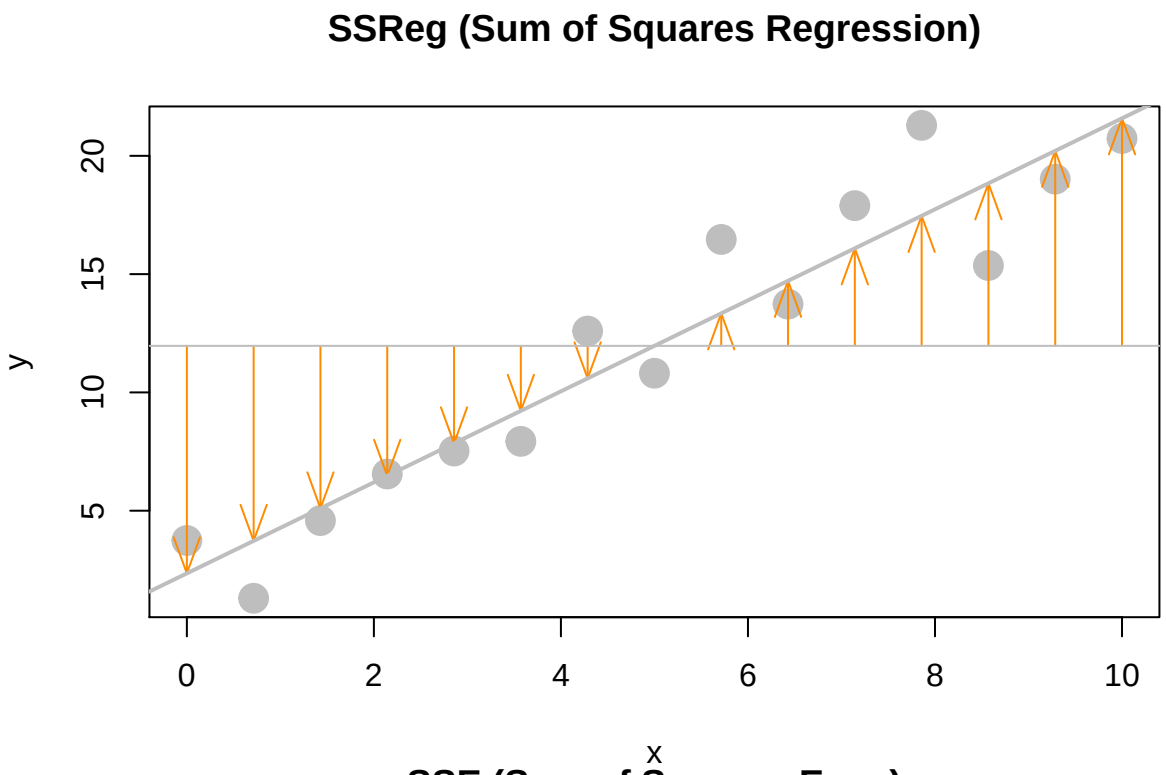
The coefficient of determination is interpreted as the proportion of observed variation in y that can be explained by the simple linear regression model.

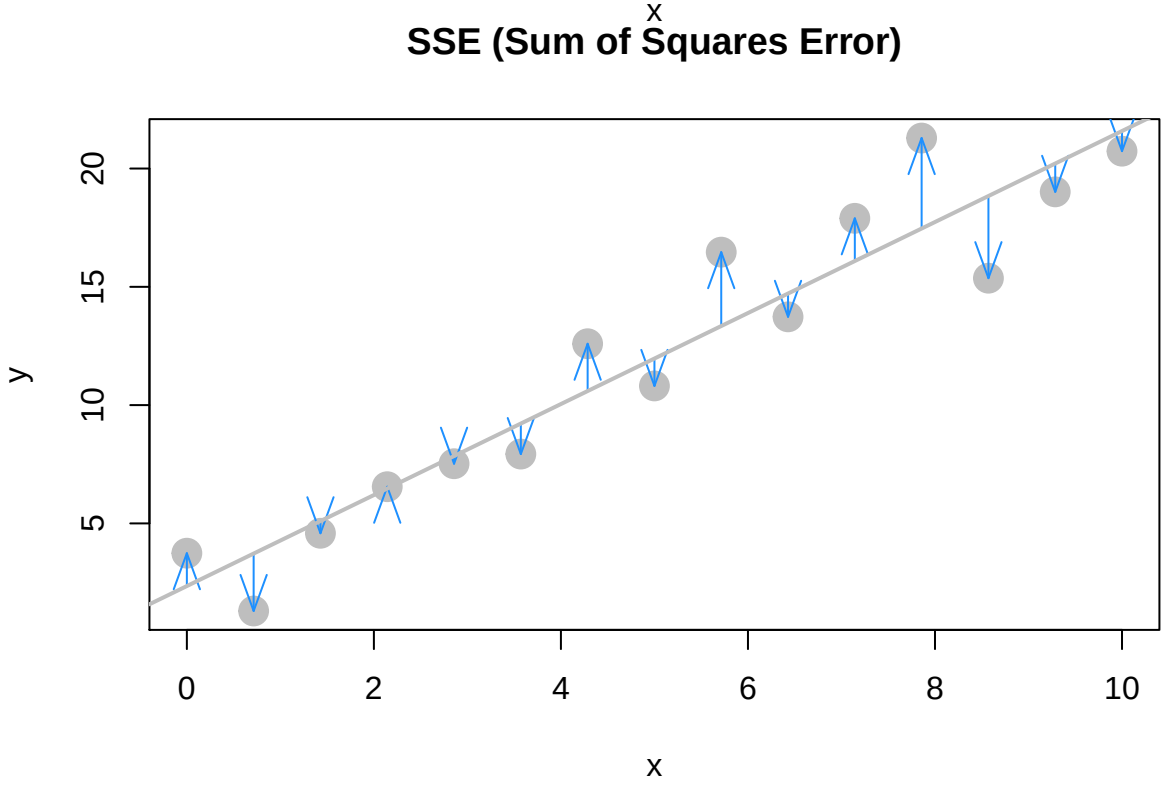
```
R2 = SSReg / SST
R2
```

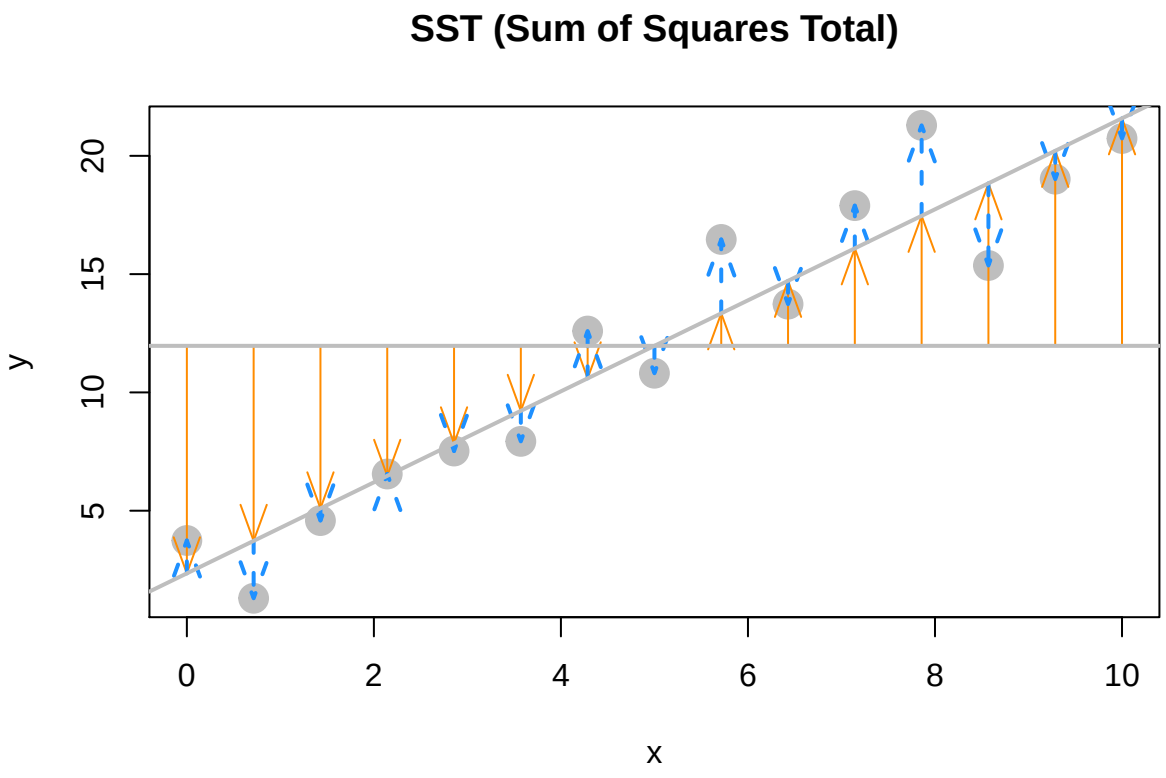
```
## [1] 0.6510794
```

For the `cars` example, we calculate $R^2 = 0.651$. We then say that 65.1% of the observed variability in stopping distance is explained by the linear relationship with speed.

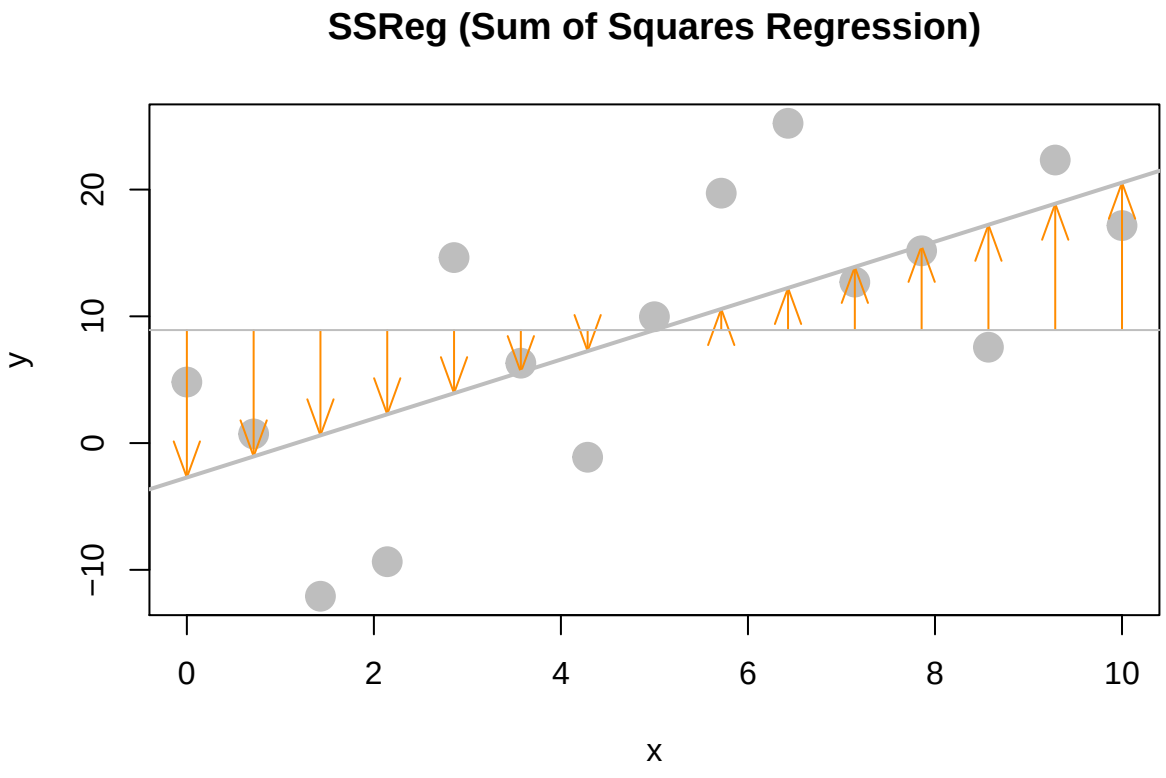
The following three plots visually demonstrate the three "sums of squares" for a simulated dataset which has $R^2 = 0.901$ which is a somewhat high value. Notice in the third plot, that the orange arrows account for a larger proportion of the total arrow.

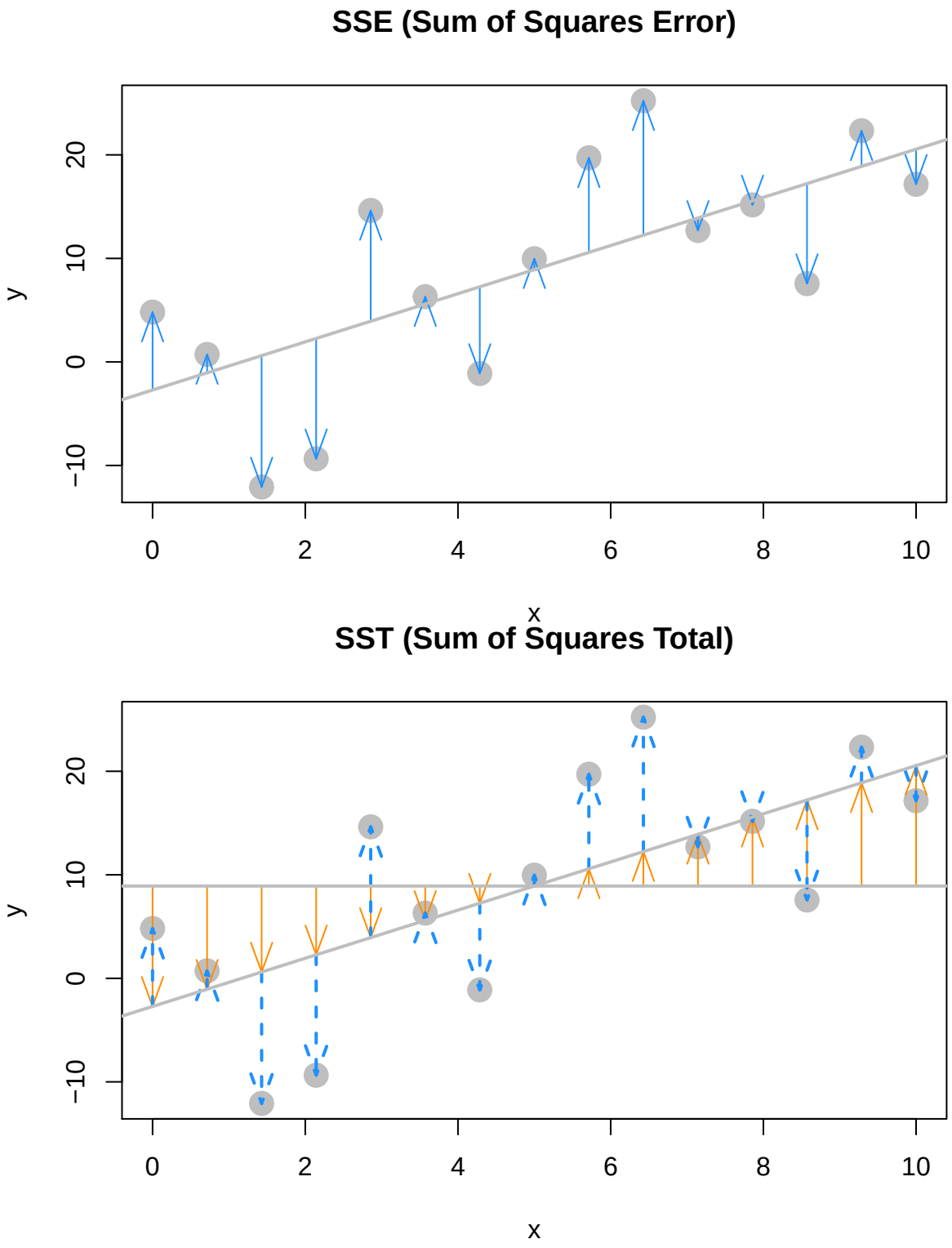

SSReg (Sum of Squares Regression)
y
x
0
2
4
6
8
10
5
10
15
20


SSE (Sum of Squares Error)
y
x
0
2
4
6
8
10
5
10
15
20



The next three plots again visually demonstrate the three "sums of squares," this time for a simulated dataset which has $R^2 = 0.459$. Notice in the third plot, that now the blue arrows account for a larger proportion of the total arrow.





3.4 The lm Function

So far we have done regression by deriving the least squares estimates, then writing simple R commands to perform the necessary calculations. Since this is such a common task, this is functionality that is built

directly into R via the lm() command.

The lm() command is used to fit **linear models** which actually account for a broader class of models than simple linear regression, but we will use SLR as our first demonstration of lm(). The lm() function will be one of our most commonly used tools, so you may want to take a look at the documentation by using ?lm. You'll notice there is a lot of information there, but we will start with just the very basics. This is documentation you will want to return to often.

We'll continue using the cars data, and essentially use the lm() function to check the work we had previously done.

```
stop_dist_model = lm(dist ~ speed, data = cars)
```

This line of code fits our very first linear model. The syntax should look somewhat familiar. We use the dist ~ speed syntax to tell R we would like to model the response variable dist as a linear function of the predictor variable speed. In general, you should think of the syntax as response ~ predictor. The data = cars argument then tells R that that dist and speed variables are from the dataset cars. We then store this result in a variable stop_dist_model.

The variable stop_dist_model now contains a wealth of information, and we will now see how to extract and use that information. The first thing we will do is simply output whatever is stored immediately in the variable stop_dist_model.

```
stop_dist_model
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

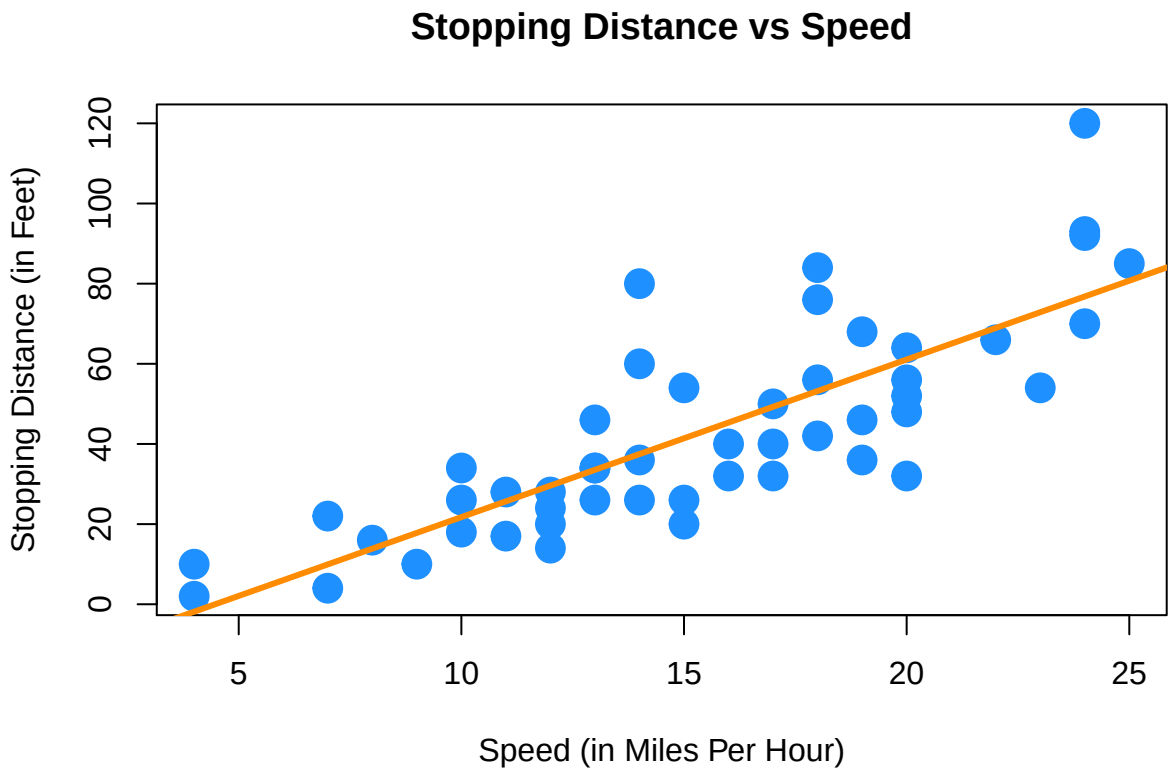
We see that it first tells us the formula we input into R, that is lm(formula = dist ~ speed, data = cars). We also see the coefficients of the model. We can check that these are what we had calculated previously. (Minus some rounding that R is doing to display the results.)

```
c(beta_0_hat, beta_1_hat)
```

```
## [1] -17.579095   3.932409
```

Next, it would be nice to add the fitted line to the scatterplot. To do so we will use the abline() function.

```
plot(dist ~ speed, data = cars,
     xlab = "Speed (in Miles Per Hour)",
     ylab = "Stopping Distance (in Feet)",
     main = "Stopping Distance vs Speed",
     pch  = 20,
     cex  = 3,
     col  = "dodgerblue")
abline(stop_dist_model, lwd = 3, col = "darkorange")
```



The `abline()` function is used to add lines of the form $a + bx$ to a plot. (Hence `abline`.) When we give it `stop_dist_model` as an argument, it automatically extracts the regression coefficient estimates ($\hat{\beta}_0$ and $\hat{\beta}_1$) and uses them as the slope and intercept of the line. Here we also use `lwd` to modify the width of the line, as well as `col` to modify the color of the line.

The "thing" that is returned by the `lm()` function is actually an object of class `lm` which is a list. The exact details of this are unimportant unless you are seriously interested in the inner-workings of `R`, but know that we can determine the names of the elements of the list using the `names()` command.

```
names(stop_dist_model)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

When can then use this information to, for example, access the residuals using the `$` operator.

```
stop_dist_model$residuals
```

```
##          1          2          3          4          5          6          7
##   3.849460  11.849460  -5.947766  12.052234   2.119825  -7.812584  -3.744993
##          8          9         10         11         12         13         14
##   4.255007  12.255007  -8.677401   2.322599 -15.609810  -9.609810  -5.609810
##         15         16         17         18         19         20         21
## -1.609810  -7.542219   0.457781   0.457781  12.457781 -11.474628  -1.474628
##         22         23         24         25         26         27         28
##  22.525372  42.525372 -21.407036 -15.407036  12.592964 -13.339445  -5.339445
##         29         30         31         32         33         34         35
```

```
## -17.271854  -9.271854   0.728146 -11.204263   2.795737  22.795737  30.795737
##         36         37         38         39         40         41         42
## -21.136672 -11.136672  10.863328 -29.069080 -13.069080  -9.069080  -5.069080
##         43         44         45         46         47         48         49
##   2.930920  -2.933898 -18.866307  -6.798715  15.201285  16.201285  43.201285
##         50
##   4.268876
```

Another way to access stored information in `stop_dist_model` are the `coef()`, `resid()`, and `fitted()` functions. These return the coefficients, residuals, and fitted values, respectively.

```
coef(stop_dist_model)
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

```
resid(stop_dist_model)
```

```
##          1          2          3          4          5          6          7
##   3.849460  11.849460  -5.947766  12.052234   2.119825  -7.812584  -3.744993
##          8          9         10         11         12         13         14
##   4.255007  12.255007  -8.677401   2.322599 -15.609810  -9.609810  -5.609810
##         15         16         17         18         19         20         21
##  -1.609810  -7.542219   0.457781   0.457781  12.457781 -11.474628  -1.474628
##         22         23         24         25         26         27         28
##  22.525372  42.525372 -21.407036 -15.407036  12.592964 -13.339445  -5.339445
##         29         30         31         32         33         34         35
## -17.271854  -9.271854   0.728146 -11.204263   2.795737  22.795737  30.795737
##         36         37         38         39         40         41         42
## -21.136672 -11.136672  10.863328 -29.069080 -13.069080  -9.069080  -5.069080
##         43         44         45         46         47         48         49
##   2.930920  -2.933898 -18.866307  -6.798715  15.201285  16.201285  43.201285
##         50
##   4.268876
```

```
fitted(stop_dist_model)
```

```
##         1         2         3         4         5         6         7         8
## -1.849460 -1.849460  9.947766  9.947766 13.880175 17.812584 21.744993 21.744993
##         9        10        11        12        13        14        15        16
## 21.744993 25.677401 25.677401 29.609810 29.609810 29.609810 29.609810 33.542219
##        17        18        19        20        21        22        23        24
## 33.542219 33.542219 33.542219 37.474628 37.474628 37.474628 37.474628 41.407036
##        25        26        27        28        29        30        31        32
## 41.407036 41.407036 45.339445 45.339445 49.271854 49.271854 49.271854 53.204263
##        33        34        35        36        37        38        39        40
## 53.204263 53.204263 53.204263 57.136672 57.136672 57.136672 61.069080 61.069080
##        41        42        43        44        45        46        47        48
## 61.069080 61.069080 61.069080 68.933898 72.866307 76.798715 76.798715 76.798715
##        49        50
## 76.798715 80.731124
```

An R function that is useful in many situations is `summary()`. We see that when it is called on our model, it returns a good deal of information. By the end of the course, you will know what every value here is used for. For now, you should immediately notice the coefficient estimates, and you may recognize the R^2 value we saw earlier.

```
summary(stop_dist_model)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value         Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601           0.0123 *
## speed         3.9324     0.4155   9.464 0.00000000000149 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 0.00000000000149
```

The `summary()` command also returns a list, and we can again use `names()` to learn what about the elements of this list.

```
names(summary(stop_dist_model))
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

So, for example, if we wanted to directly access the value of R^2, instead of copy and pasting it out of the printed statement from `summary()`, we could do so.

```
summary(stop_dist_model)$r.squared
```

```
## [1] 0.6510794
```

Another value we may want to access is s_e, which R calls `sigma`.

```
summary(stop_dist_model)$sigma
```

```
## [1] 15.37959
```

Note that this is the same result seen earlier as `s_e`. You may also notice that this value was display above as a result of the `summary()` command, which R labeled the "Residual Standard Error."

$$s_e = RSE = \sqrt{\frac{1}{n-2}\sum_{i=1}^{n} e_i^2}$$

Often it is useful to talk about s_e (or RSE) instead of s_e^2 because of their units. The units of s_e in the `cars` example is feet, while the units of s_e^2 is feet-squared.

Another useful function, which we will use almost as often as `lm()` is the `predict()` function.

```
predict(stop_dist_model, data.frame(speed = 8))
```

```
##        1
## 13.88018
```

The above code reads "predict the stopping distance of a car traveling 8 miles per hour using the `stop_dist_model`." Importantly, the second argument to `predict()` is a data frame that we make in place. We do this so that we can specify that `8` is a value of `speed`, so that predict knows how to use it with the model stored in `stop_dist_model`. We see that this result is what we had calculated "by hand" previously.

We could also predict multiple values at once.

```
predict(stop_dist_model, data.frame(speed = c(8, 21, 50)))
```

```
##         1         2         3
##  13.88018  65.00149 179.04134
```

$$\hat{y} = -17.579 + 3.932 \times 8 = 13.88$$
$$\hat{y} = -17.579 + 3.932 \times 21 = 65.001$$
$$\hat{y} = -17.579 + 3.932 \times 50 = 179.041$$

Or we could calculate the fitted value for each of the original data points.

```
predict(stop_dist_model, data.frame(speed = cars$speed))
```

```
##         1         2         3         4         5         6         7         8
## -1.849460 -1.849460  9.947766  9.947766 13.880175 17.812584 21.744993 21.744993
##         9        10        11        12        13        14        15        16
## 21.744993 25.677401 25.677401 29.609810 29.609810 29.609810 29.609810 33.542219
##        17        18        19        20        21        22        23        24
## 33.542219 33.542219 33.542219 37.474628 37.474628 37.474628 37.474628 41.407036
##        25        26        27        28        29        30        31        32
## 41.407036 41.407036 45.339445 45.339445 49.271854 49.271854 49.271854 53.204263
##        33        34        35        36        37        38        39        40
## 53.204263 53.204263 53.204263 57.136672 57.136672 57.136672 61.069080 61.069080
##        41        42        43        44        45        46        47        48
## 61.069080 61.069080 61.069080 68.933898 72.866307 76.798715 76.798715 76.798715
##        49        50
## 76.798715 80.731124
```

This is actually equivalent to simply calling `predict()` on `stop_dist_model` without a second argument.

```
predict(stop_dist_model)
```

```
##         1         2         3         4         5         6         7         8
## -1.849460 -1.849460  9.947766  9.947766 13.880175 17.812584 21.744993 21.744993
##         9        10        11        12        13        14        15        16
## 21.744993 25.677401 25.677401 29.609810 29.609810 29.609810 29.609810 33.542219
##        17        18        19        20        21        22        23        24
## 33.542219 33.542219 33.542219 37.474628 37.474628 37.474628 37.474628 41.407036
##        25        26        27        28        29        30        31        32
## 41.407036 41.407036 45.339445 45.339445 49.271854 49.271854 49.271854 53.204263
##        33        34        35        36        37        38        39        40
## 53.204263 53.204263 53.204263 57.136672 57.136672 57.136672 61.069080 61.069080
##        41        42        43        44        45        46        47        48
## 61.069080 61.069080 61.069080 68.933898 72.866307 76.798715 76.798715 76.798715
##        49        50
## 76.798715 80.731124
```

Note that then in this case, this is the same as using `fitted()`.

```
fitted(stop_dist_model)
```

```
##         1         2         3         4         5         6         7         8
## -1.849460 -1.849460  9.947766  9.947766 13.880175 17.812584 21.744993 21.744993
##         9        10        11        12        13        14        15        16
## 21.744993 25.677401 25.677401 29.609810 29.609810 29.609810 29.609810 33.542219
##        17        18        19        20        21        22        23        24
## 33.542219 33.542219 33.542219 37.474628 37.474628 37.474628 37.474628 41.407036
##        25        26        27        28        29        30        31        32
## 41.407036 41.407036 45.339445 45.339445 49.271854 49.271854 49.271854 53.204263
##        33        34        35        36        37        38        39        40
## 53.204263 53.204263 53.204263 57.136672 57.136672 57.136672 61.069080 61.069080
##        41        42        43        44        45        46        47        48
## 61.069080 61.069080 61.069080 68.933898 72.866307 76.798715 76.798715 76.798715
##        49        50
## 76.798715 80.731124
```

3.5 Maximum Likelihood Estimation (MLE) Approach

Recall the model,

$$Y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

where $e_i \sim N(0, \sigma^2)$.

Then we can find the mean and variance of each Y_i.

$$E[Y_i] = \beta_0 + \beta_1 x_i$$

and

$$Var[Y_i] = \sigma^2.$$

Recall that the pdf of a random variable $X \sim N(\mu, \sigma^2)$ is given by

$$f_X(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right].$$

Then we can write the pdf of each of the Y_i as

$$f_{Y_i}(y_i; x_i, \beta_0, \beta_1, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}\left(\frac{y_i - (\beta_0 + \beta_1 x_i)}{\sigma}\right)^2\right].$$

Given n data points (x_i, y_i) we can write the likelihood, which is a function of the three parameters β_0, β_1, and σ^2. Since the data have been observed, we use lower case y_i to denote that these values are no longer random.

$$L(\beta_0, \beta_1, \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}\left(\frac{y_i - \beta_0 - \beta_1 x_i}{\sigma}\right)^2\right]$$

Our goal is to find values of β_0, β_1, and σ^2 which maximize this function, which is a straightforward multivariate calculus problem.

We'll start by doing a bit of rearranging to make our task easier.

$$L(\beta_0, \beta_1, \sigma^2) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2\right]$$

Then, as is often the case when finding MLEs, for mathematical convenience we will take the natural logarithm of the likelihood function since log is a monotonically increasing function. Then we will proceed to maximize the log-likelihood, and the resulting estimates will be the same as if we had not taken the log.

$$\log L(\beta_0, \beta_1, \sigma^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2$$

Note that we use log to mean the natural logarithm. We now take a partial derivative with respect to each of the parameters.

$$\begin{aligned}
\frac{\partial \log L(\beta_0, \beta_1, \sigma^2)}{\partial \beta_0} &= -\frac{2}{\sigma^2}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i) \\
\frac{\partial \log L(\beta_0, \beta_1, \sigma^2)}{\partial \beta_1} &= -\frac{2}{\sigma^2}\sum_{i=1}^{n}(x_i)(y_i - \beta_0 - \beta_1 x_i) \\
\frac{\partial \log L(\beta_0, \beta_1, \sigma^2)}{\partial \sigma^2} &= -\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2
\end{aligned}$$

We then set each of the partial derivatives equal to zero and solve the resulting system of equations.

$$\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i) = 0$$

$$\sum_{i=1}^{n}(x_i)(y_i - \beta_0 - \beta_1 x_i) = 0$$

$$-\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2 = 0$$

You may notice that the first two equations also appear in the least squares approach. Then, skipping the issue of actually checking if we have found a maximum, we then arrive at our estimates. We call these estimates the maximum likelihood estimates.

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} x_i y_i - \frac{(\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n}}{\sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n}} = \frac{S_{xy}}{S_{xx}}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Note that $\hat{\beta}_0$ and $\hat{\beta}_1$ are the same as the least squares estimates. However we now have a new estimate of σ^2, that is $\hat{\sigma}^2$. So we now have two different estimates of σ^2.

$$s_e^2 = \frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n-2}\sum_{i=1}^{n} e_i^2 \quad \text{Least Squares}$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n} e_i^2 \quad \text{MLE}$$

In the next chapter, we will discuss in detail the difference between these two estimates, which involves biasedness.

3.6 Simulating SLR

We return again to more examples of simulation. This will be a common theme!

In practice you will almost never have a true model, and you will use data to attempt to recover information about the unknown true model. With simulation, we decide the true model and simulate data from the it. Then we apply a method to the data, in this case least squares. Now, since we know the true model, we can asses how well it did.

For this example, we will simulate `n = 20` observations from the model

$$y_i = 5 + 2x + \epsilon_i.$$

That is $\beta_0 = 5$, $\beta_1 = 2$, and let $\epsilon_i \sim N(\mu = 0, \sigma^2 = 1)$.

We first set the parameters of the simulation.

```
n       = 20
beta_0  = 5
beta_1  = 2
sigma   = 1
```

Next, we obtain simulated values of ϵ_i.

```
epsilon = rnorm(n, mean = 0, sd = sigma)
```

Now, since the x_i values in SLR are considered fixed and known, we simply generate them from a uniform distribution. Know, that this is an arbitrary, but common practice.

```
x = runif(n, 0, 10)
```

We then generate the y values according the specified functional relationship.

```
y = beta_0 + beta_1 * x + epsilon
```

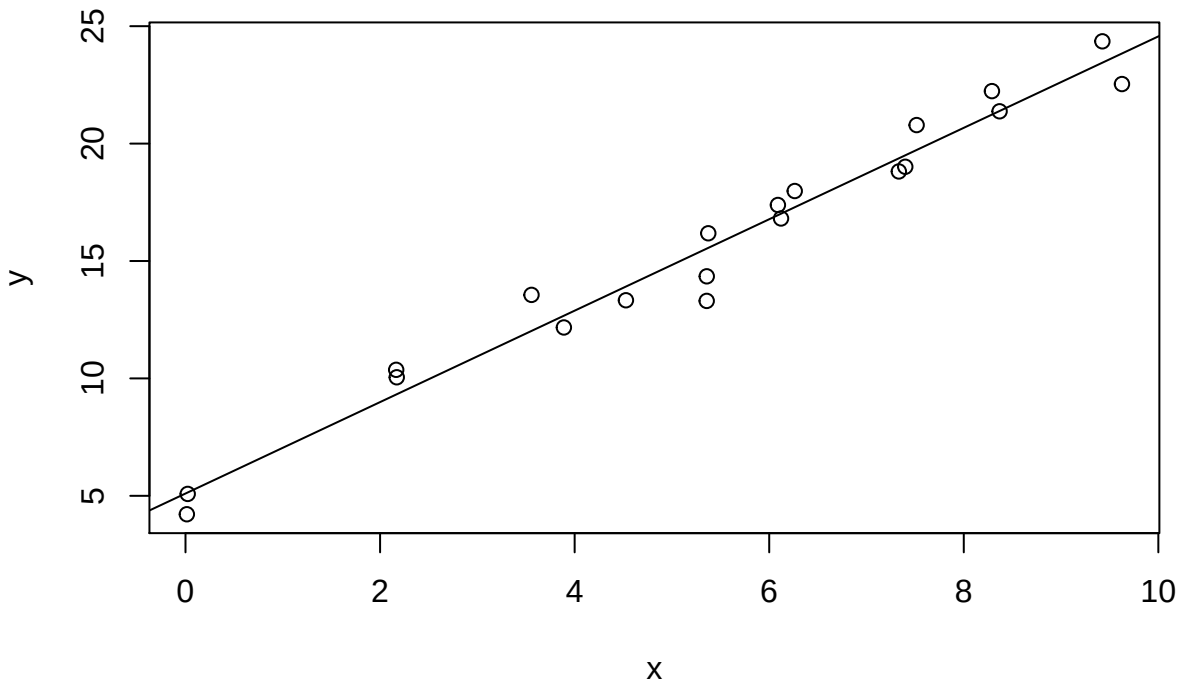
Now to check how well the method of least squares works, we use `lm()` to fit the model to our data, then take a look at the estimated coefficients.

```
sim_fit = lm(y ~ x)
coef(sim_fit)
```

```
## (Intercept)           x
##    5.098490    1.946622
```

And look at that, they aren't too far from the parameters we specified!

```
plot(y ~ x)
abline(sim_fit)
```



We should say here, that we're being sort of lazy, and not the good kinda of lazy that could be considered efficient. Any time you simulate data, you should consider doing two things: writing a function, and storing the data in a data frame.

The function below, `sim_slr()` can be used for the same task as above, but is much more flexible.

```
sim_slr = function(n, beta_0 = 10, beta_1 = 5, sigma = 1, xmin = 0, xmax = 10) {
  epsilon = rnorm(n, mean = 0, sd = sigma)
  x       = runif(n, xmin, xmax)
  y       = beta_0 + beta_1 * x + epsilon
  data.frame(predictor = x, response = y)
}
```

Here, we use the function to repeat the analysis above.

```
sim_data = sim_slr(n = 20, beta_0 = 5, beta_1 = 2, sigma = 1)
```

This time, the simulated observations are stored in a data frame.

```
head(sim_data)
```

```
##   predictor  response
## 1  9.348230 23.329224
## 2  5.504941 16.195112
## 3  6.017662 17.617148
## 4  1.969945 10.339627
## 5  5.352366 14.977440
## 6  1.795557  9.893657
```

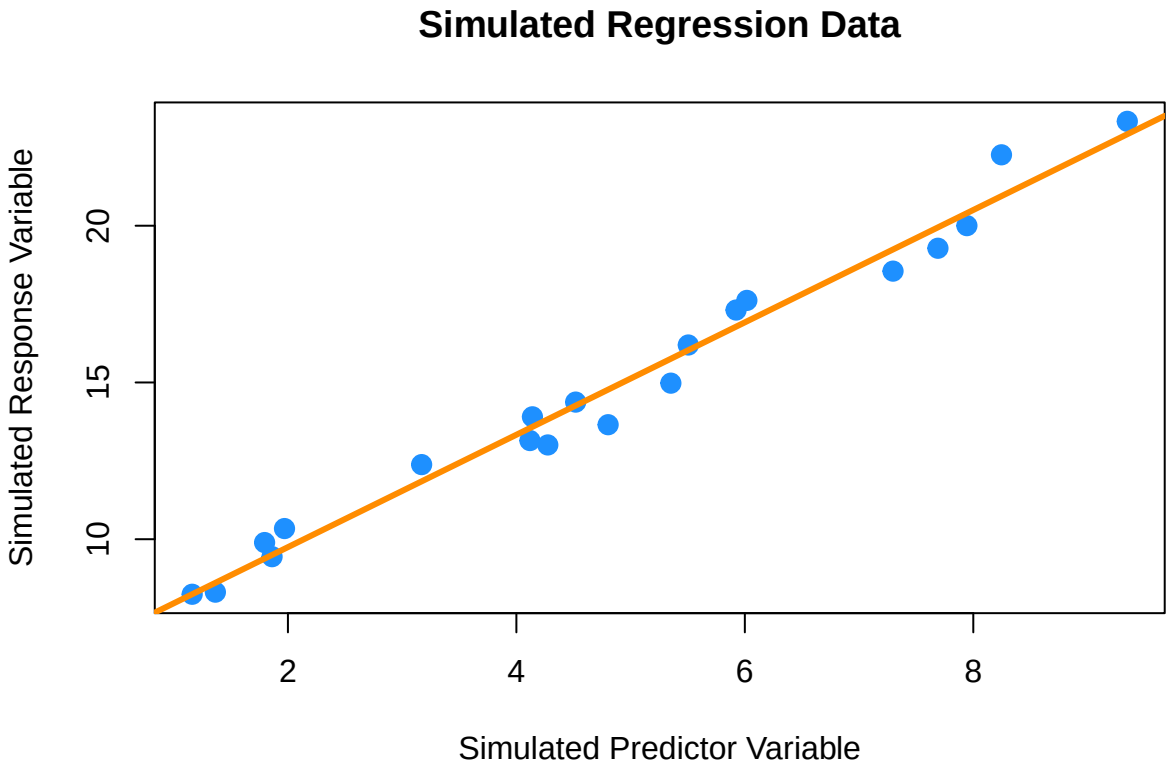
Now when we fit the model with `lm()` we can use a `data` argument, a very good practice.

```
sim_fit = lm(response ~ predictor, data = sim_data)
coef(sim_fit)
```

```
## (Intercept)   predictor
##    6.162023    1.792461
```

And this time, we'll make the plot look a lot nicer.

```
plot(response ~ predictor, data = sim_data,
     xlab = "Simulated Predictor Variable",
     ylab = "Simulated Response Variable",
     main = "Simulated Regression Data",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
abline(sim_fit, lwd = 3, col = "darkorange")
```



3.7 History

For some brief background on the history of linear regression, see "Galton, Pearson, and the Peas: A Brief History of Linear Regression for Statistics Instructors" from the Journal of Statistics Education as well as the Wikipedia page on the history of regression analysis and lastly the article for regression to the mean which details the origins of the term "regression."

Bibliography