Applied Statistics with R

David Dalpiaz

2016-06-08

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 you may find references to "this course" which is 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 the most features such as changing text size, font, and colors.) If you are in need of a local copy, a pdf version is continuously maintained at http://daviddalpiaz.github.io/appliedstats/applied_statistics.pdf.

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. Or, if you know RMarkdown and are familiar with GitHub, make a pull request and fix an issue yourself!

This text uses MathJax to render mathematical notation for the web. Occasionally but rarely a JavaScript error will prevent MathJax from rendering correctly. (And you 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

Chapter 2

Introduction to R

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

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, which can be installed from http://cran.r-project.org/
  - Chose your operating system, and select the most recent version. (As of writing, 3.3.0.)
- RStudio, an excellent IDE for working with R, which can be obtained from http://www.rstudio.com/ (Note, you must have R installed to use RStudio. RStudio is simply a way to interact with R.)

R's popularity 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.
- 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.
- R Markdown from RStudio.
    - Reference materials for RMarkdown.

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 which can be found here or from the help menu inside of RStudio. This one for Base R in particular 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 some others do. The more import thing is to be consistent with yourself.

- 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.)

2.1 R Basics

2.1.1 Basic Calculations

To get started, we'll use R like a simple calculator. Note, in R the `#` symbol is used for comments. Lines which begin with two, `##` will indicate output.

- Addition, Subtraction, Multiplication and Division

```
3 + 2
```

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

```
3 - 2
```

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

```
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
```

```r
exp(1)
```

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

- Logarithms

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

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

```r
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
```

2.1.2 Getting Help

In using `R` as a calculator, we have seen a number of functions. `sqrt()`, `exp()`, `log()` and `sin()` are all `R` functions. 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.

2.1.3 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.

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

Once a package is installed, it must be loaded into your current `R` session before being used.

```
library(ggplot2)
```

2.1.4 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: `1`, `2`, `42`
- 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*. Data structures can be either homogeneous (contain only a single data type) or heterogeneous. (Contain more than one data type.)

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

2.1.5 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.

```
x = 1:10
```

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
```

```
##  [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`.

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

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

```r
length(vec_1)
```

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

```r
length(vec_2)
```

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

```r
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.

```r
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 later, but note here that `from`, `to` and `by` are optional.

```r
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.

```r
rep(0.5, times = 10)
```

```
##  [1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
```

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

```r
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
```

2.1.6 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
```

```r
IQR(y)
```

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

```r
min(y)
```

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

```r
max(y)
```

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

```r
range(y)
```

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

2.1.7 Matrices

R can also be used for **matrix** calculations. Matrices have rows and columns containing a single data type. (And unlike a data frame, which we will see later, the order is important.)

Matrices can be created using the `matrix` function.

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

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

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

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

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

```r
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
```

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

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

```r
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
```

```r
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.

```r
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
```

```r
Y
```

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

```r
X + Y
```

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

```r
X - Y
```

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

```r
X * Y
```

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

```r
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 `%*%`. `t()` gives the transpose of a matrix, and `solve()` returns the inverse of a matrix.

```r
X %*% Y
```

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

```r
t(X)
```

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

```r
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
```

```r
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
```

2.1.8 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")
```

TODO: Note about header and sep and wd and folder struct. rproj?

Alternatively, we could use the "Import Dataset" feature in RStudio which can be found in the environment window. This process will automatically generate the code to import a file. The resulting code will be shown in the console window.

```r
library(ggplot2)
```

Early 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.) Inside the `ggplot2` package is a dataset called `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()`.

```r
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
## 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
```

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

```r
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" ...
```

`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.

```r
dim(mpg)
```

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

```r
nrow(mpg)
```

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

```r
ncol(mpg)
```

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

TODO: subsetting. dpylr?

2.1.9 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.

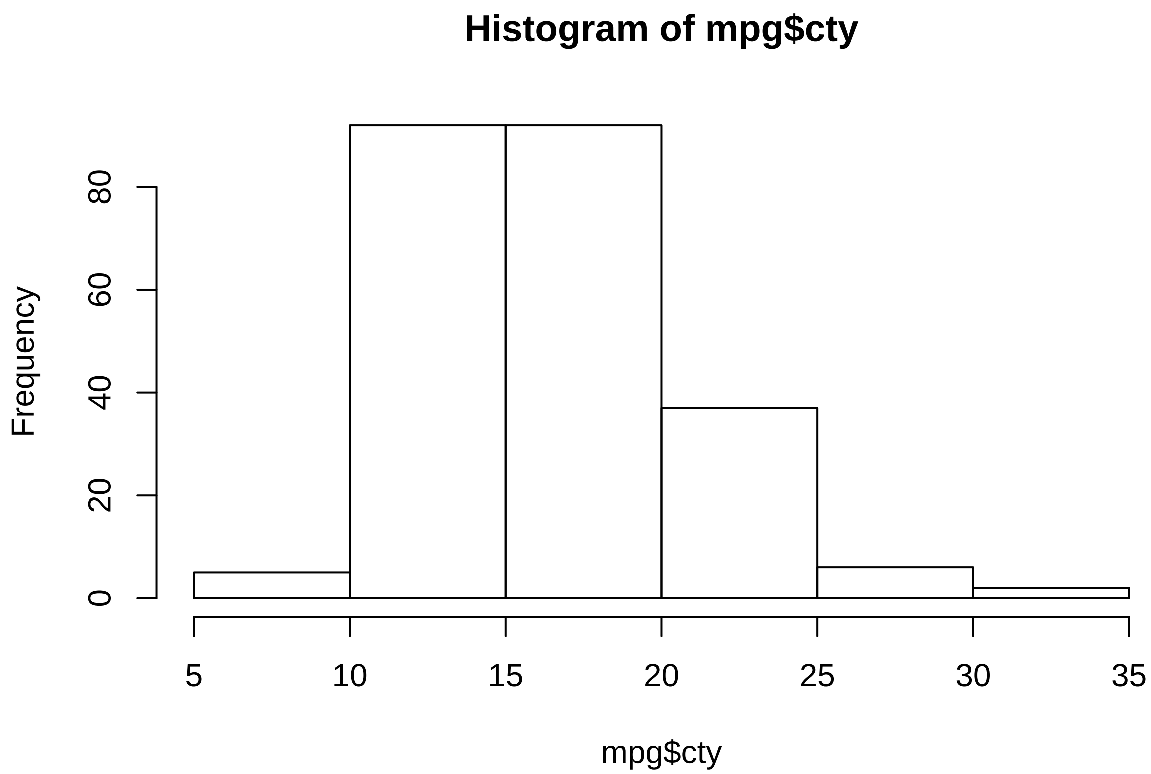
We will look at three methods of visualizing data that we will use throughout the course:

- Histograms
- Boxplots
- Scatterplots

2.1.9.1 Histograms

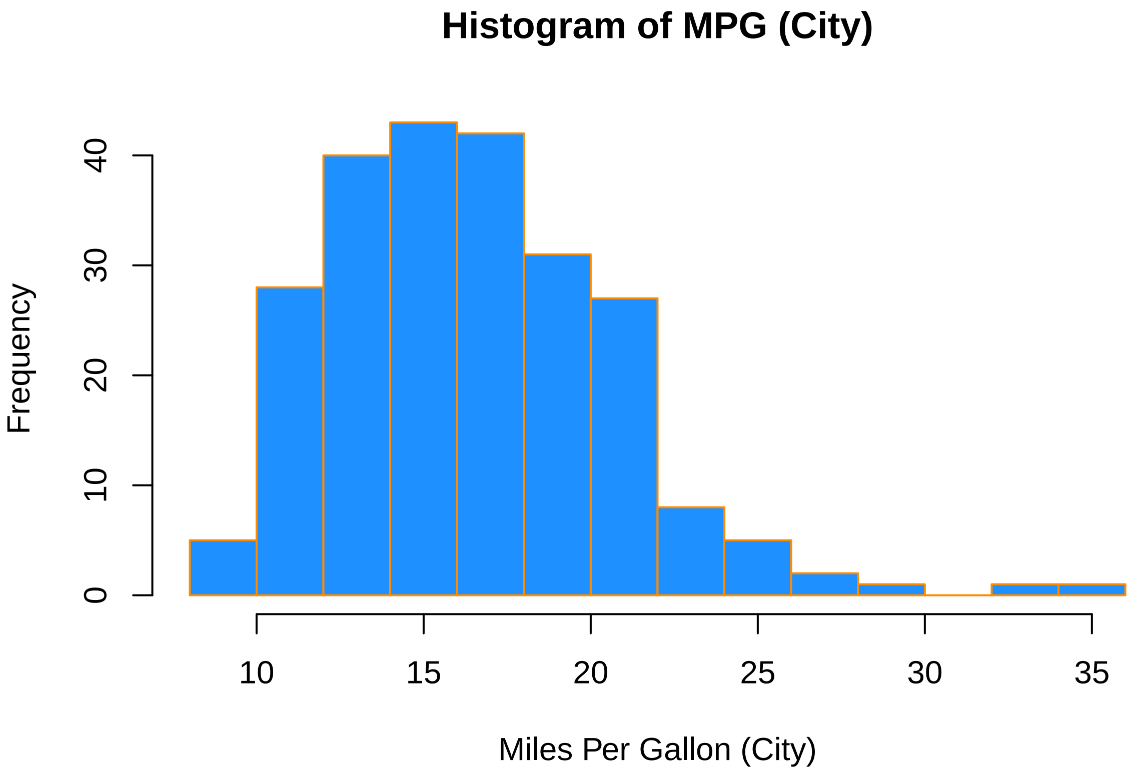
When visualizing a single numerical variable, a **histogram** will be our go-to tool, which can be created in `R` using the `hist()` function.

```r
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. `breaks` is an argument specific to `hist()` which controls how many bars `R` will 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.

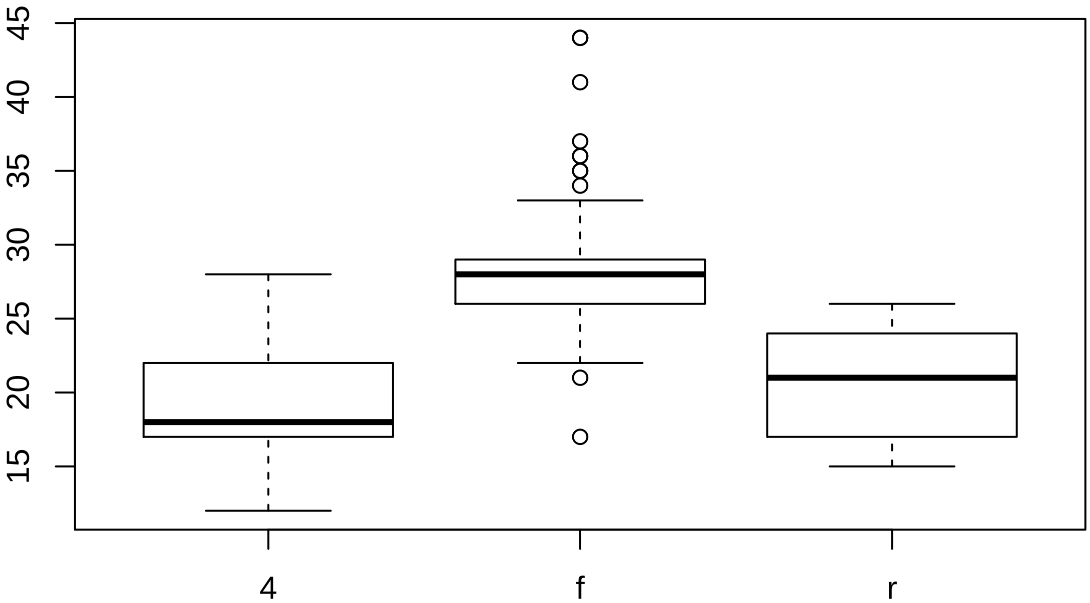
2.1.9.2 Boxplots

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, rear wheel drive, or 4 wheel drive. (FWD, RWD, or 4WD.)

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

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

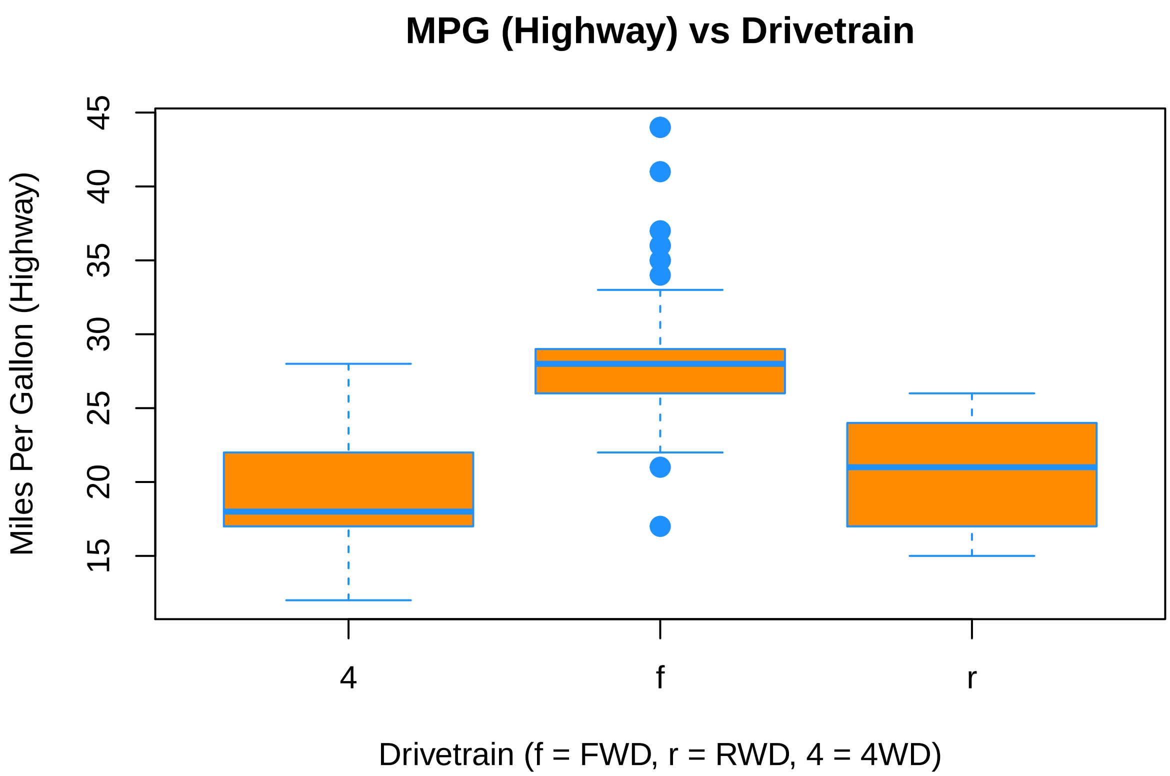
```
boxplot(hwy ~ drv, data = mpg)
```



Here used the `boxplot()` command to create the boxplot. However, since we are now dealing with two variables, the syntax has changed, and we see the use of a `~` and also a `data =` argument. This will be a syntax that is common to many functions we will use in this course.

The `R` syntax `hwy ~ drv, data = mpg` reads "Plot the `hwy` variable against the `drv` variable using the dataset `mpg`."

```
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")
```

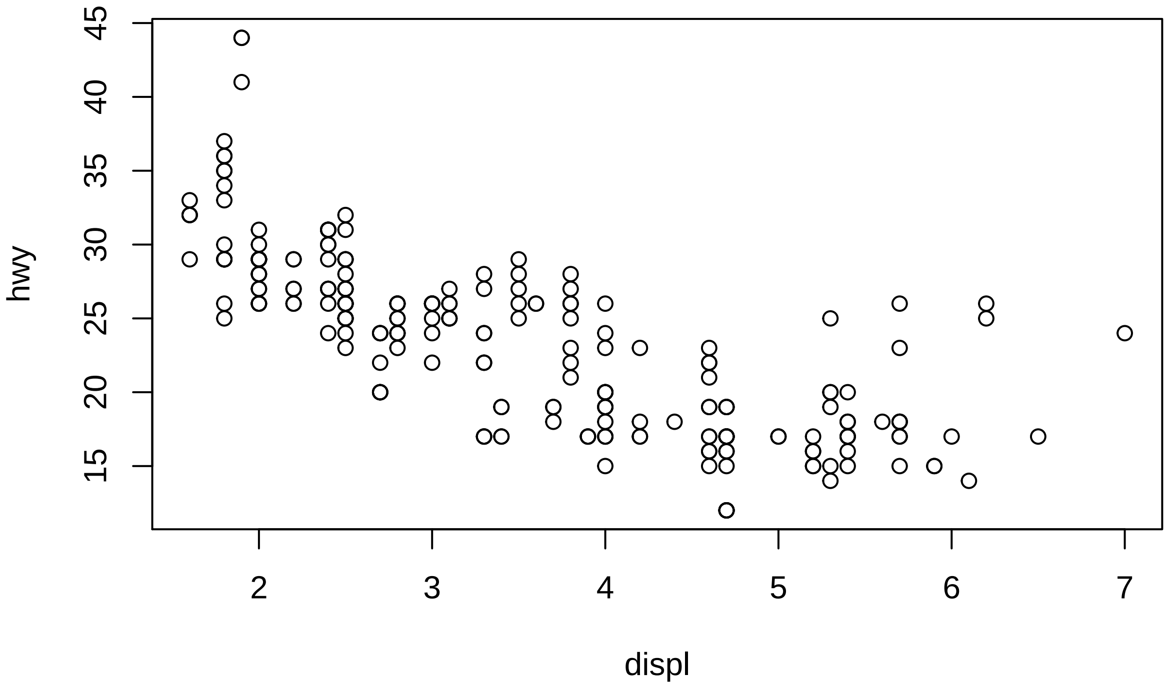


Again, `boxplot()` has a number of additional arguments which have the ability to make our plot more visually appealing.

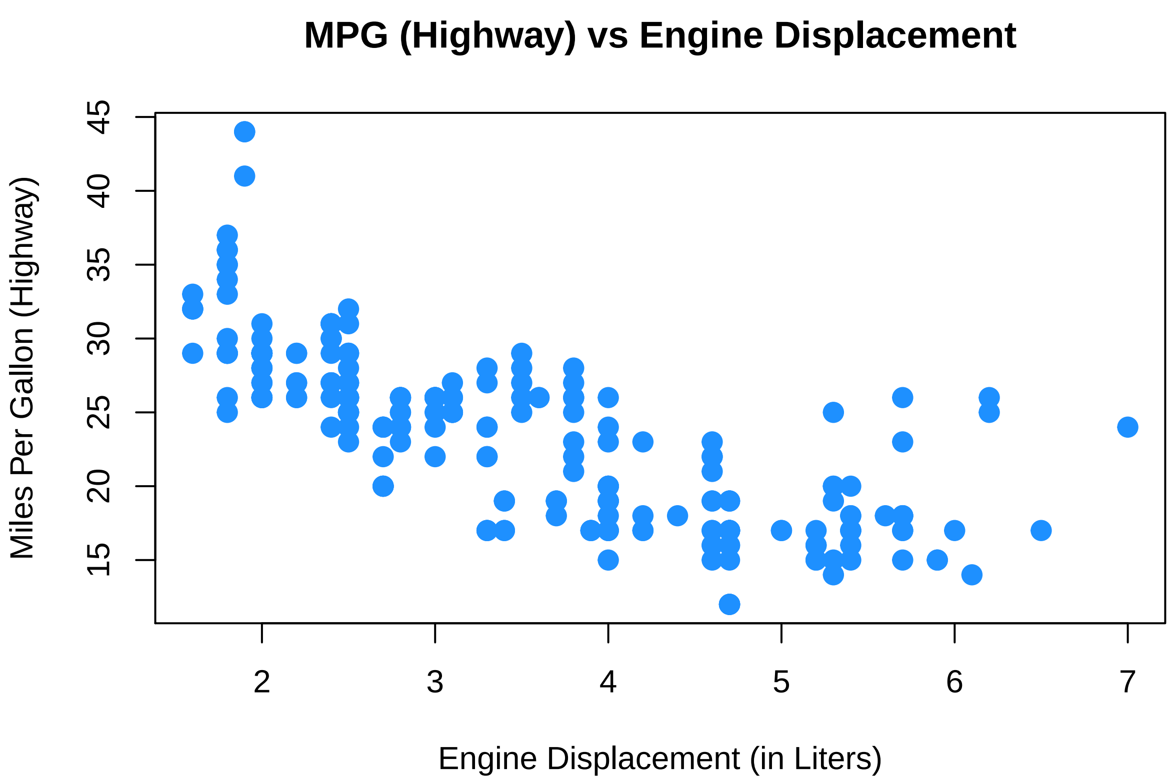
2.1.9.3 Scatterplots

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

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




```r
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")
```



2.1.10 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 corresponding to a particular probability.
- A random draw 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 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(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(3, mean = 2, sd = 5)
```

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

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

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

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

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

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

```
##  [1] -2.5916433  5.5193222 -3.3884937 -5.7928560 -2.2461998 -4.1607456
##  [7] -1.8257089 10.9596117 -3.0094302  0.2883016
```

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(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 $P(Y = 5)$ 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`.)

2.2 Programming Basics

2.2.1 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`	`x`equal 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
```

Be careful when comparing vectors that 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 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`. To do this, we simply "multiply" by `1`.

```
1 * (a > 5)
```

```
##  [1] 0 0 0 0 0 1 1 1 1 1
```

If we then sum the resulting vector, we have essentially counted the number of elements of `a` that are larger than `5`.

```
sum(1 * (a > 5))
```

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

2.2.2 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
```

2.2.3 Functions

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

TODO: use TODO: arguments, defaults (order?)

Lastly, we can 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.

TODO: syntax (argument, {}, return)

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

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`.

TODO: set.seed()

```
set.seed(42)
(test_sample = rnorm(10, mean = 2, sd = 5))
```

```
##  [1]  8.8547922 -0.8234909  3.8156421  5.1643130  4.0213416  1.4693774
##  [7]  9.5576100  1.5267048 12.0921186  1.6864295
```

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

```
##  [1]  0.9858912 -1.3310150 -0.2204424  0.1024190 -0.1711995 -0.7821201
##  [7]  1.1541404 -0.7683963  1.7608822 -0.7301595
```

```
result = standardize(x = test_sample)
mean(result)
```

```
## [1] 3.051487e-17
```

```
sd(result)
```

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

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)
}
```

To 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 (base on maximum likilhood) 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] 4.177244
```

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

```
## [1] 4.177244
```

```r
sd(test_sample)
```

```
## [1] 4.177244
```

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

```
## [1] 3.962882
```

2.3 Hypothesis Tests in R

2.3.1 One Sample t-Test: Review

Suppose $x_i \sim \mathrm{N}(\mu, \sigma^2)$ 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 $s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x - \bar{x})^2}$

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.

2.3.2 One Sample t-Test: Example

Suppose a 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`.

```r
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 for 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(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 hyopthesized 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 distribution under the null hyopthesis.
- 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 ouput from `t.test()`. To see what information is availible 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 insteaded in a confidence interval.

```
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 pluggin into the forumula,

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

```
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
```

2.3.3 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 $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$, $\bar{y} = \frac{\sum_{i=1}^{m} y_i}{m}$, and $s_p^2 = \frac{(n-1)s_1^2 + (m-1)s_2^2}{n+m-2}$

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

2.3.4 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.

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

```
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.

TODO: store in df and use formula?

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

```
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
```

```r
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
```

2.4 Simulation

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

2.4.1 Paired Differences

Consider the model:

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

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

Let

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

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

Recall,

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

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

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,

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

Which 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.

TODO: discuss rname functions. (or earlier?)

Generate $s = 10000$ samples for each of group 1 and group 2. For each of the 10000 samples, compute $d_s = \bar{x}_{1s} - \bar{x}_{2s}$.

What is the proportion of values of d (among the 10000 values of d generated) that are between 0 and 2?

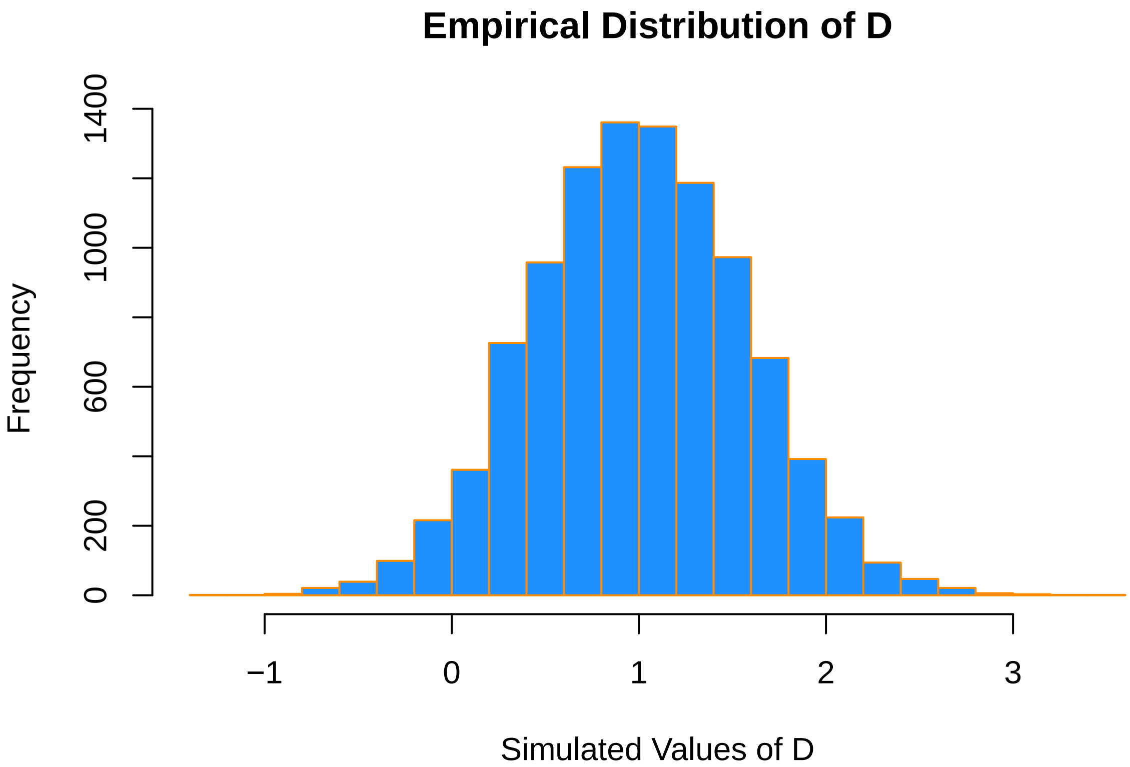
```
set.seed(42)
num_samples = 10000
differences = rep(0, num_samples)
```

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

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

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

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



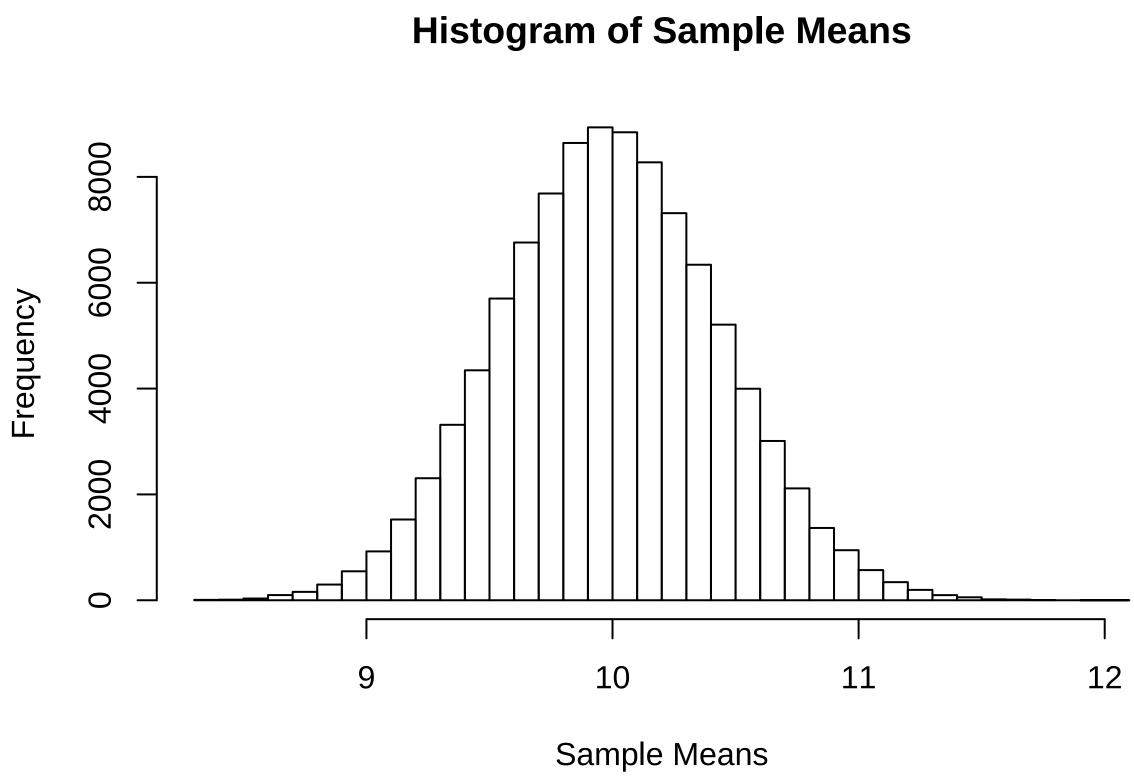
2.4.2 Distribution of a Sample Mean

For another example of simulation, we will simulate observations from a Poisson distribution, and examine the emperical distribution of these observatoins.

```
set.seed(42)
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")
```




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

```
## [1] 10.00009 10.00000
```

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

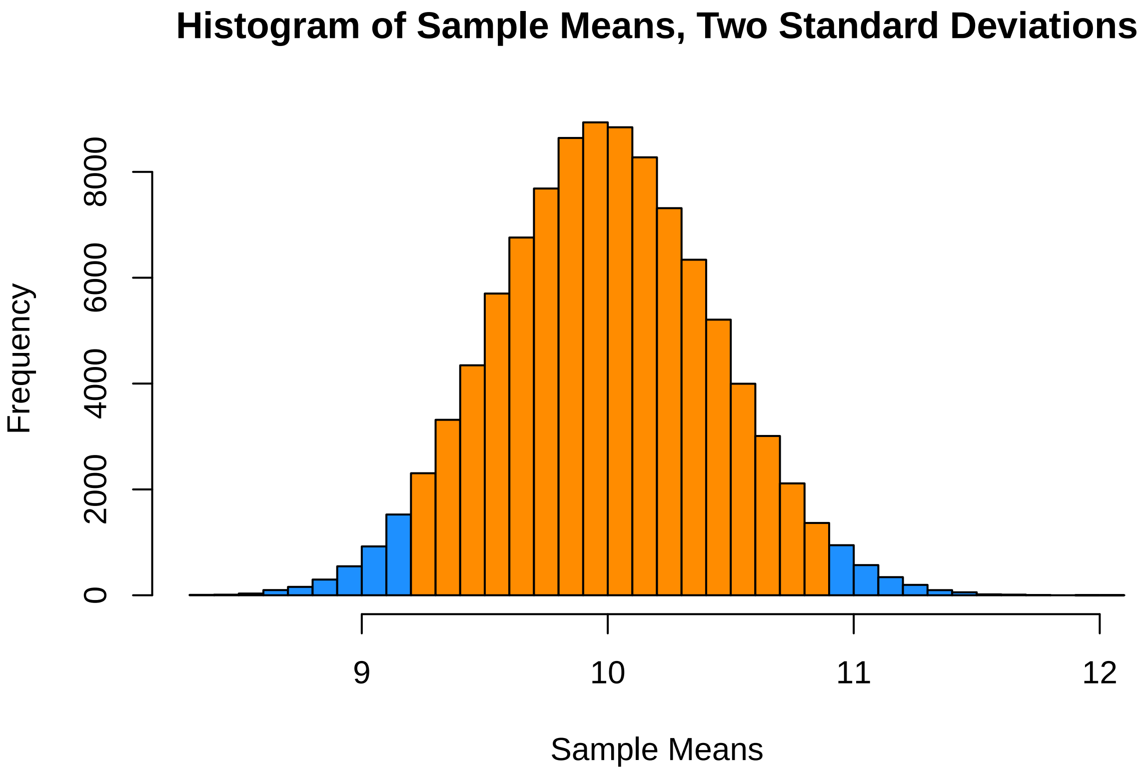
```
## [1] 0.4454965 0.4472136
```

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

```
## [1] 0.95459
```

```
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")
```



TODO: spell check

Chapter 3

Simple Linear Regression

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

Suppose you are the owner of the *Momma Leona's Pizza*, a restaurant chain located near several college campuses. You currently own 10 stores and have data on the size of the student population as well as quarterly sales for each. Your data is summarized in the table below.

Restaurant	Student Population	Quarterly Sales
1	2	58
2	6	105
3	8	88
4	8	118
5	12	117
6	16	137
7	20	157
8	20	169
9	22	149
10	26	202

Here,

- Restaurant, i
- Student Population, x_i, in 1000s
- Quarterly Sales, y_i, in \$1000s

```
momma_leona = data.frame(students = c(2, 6, 8, 8, 12, 16, 20, 20, 22, 26),
                         sales = c(58, 105, 88, 118, 117, 137, 157, 169, 149, 202))
```

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.

list of vectors observations and variables order "doesn't matter" (unlike a matrix)

```
names(momma_leona)
```

```
## [1] "students" "sales"
```

```r
momma_leona$sales
```

```
##  [1]  58 105  88 118 117 137 157 169 149 202
```

```r
momma_leona$students
```

```
##  [1]  2  6  8  8 12 16 20 20 22 26
```

```r
dim(momma_leona)
```

```
## [1] 10  2
```

```r
nrow(momma_leona)
```

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

```r
ncol(momma_leona)
```

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

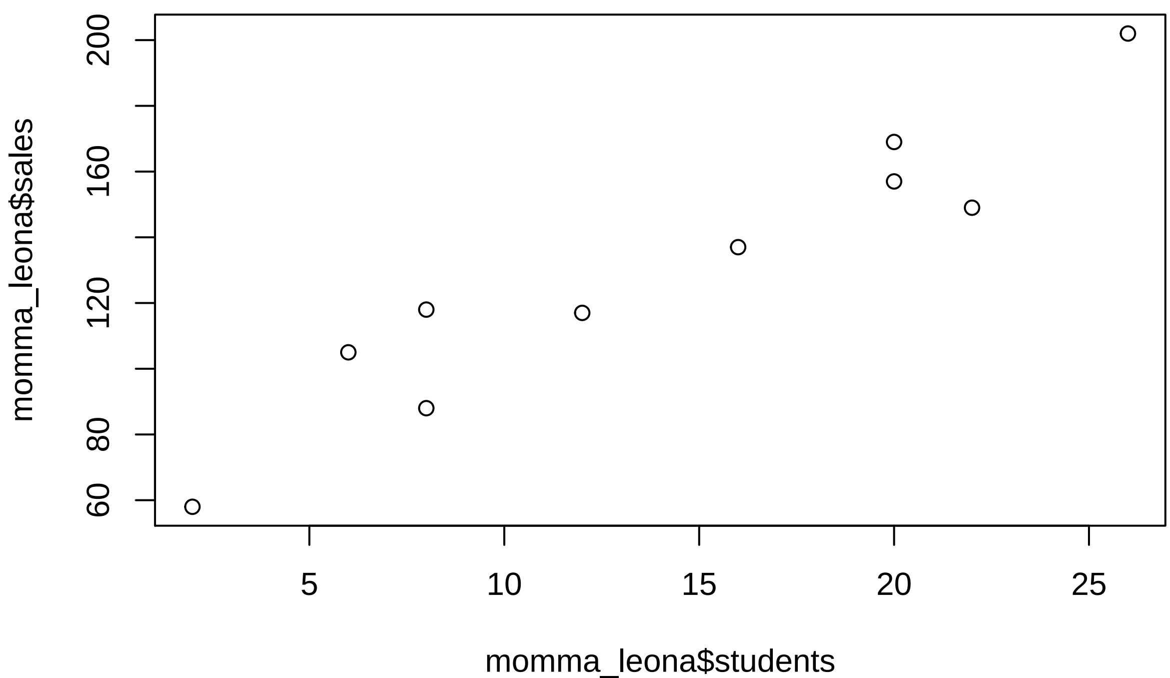
```r
str(momma_leona)
```

```
## 'data.frame':    10 obs. of  2 variables:
##  $ students: num  2 6 8 8 12 16 20 20 22 26
##  $ sales   : num  58 105 88 118 117 137 157 169 149 202
```

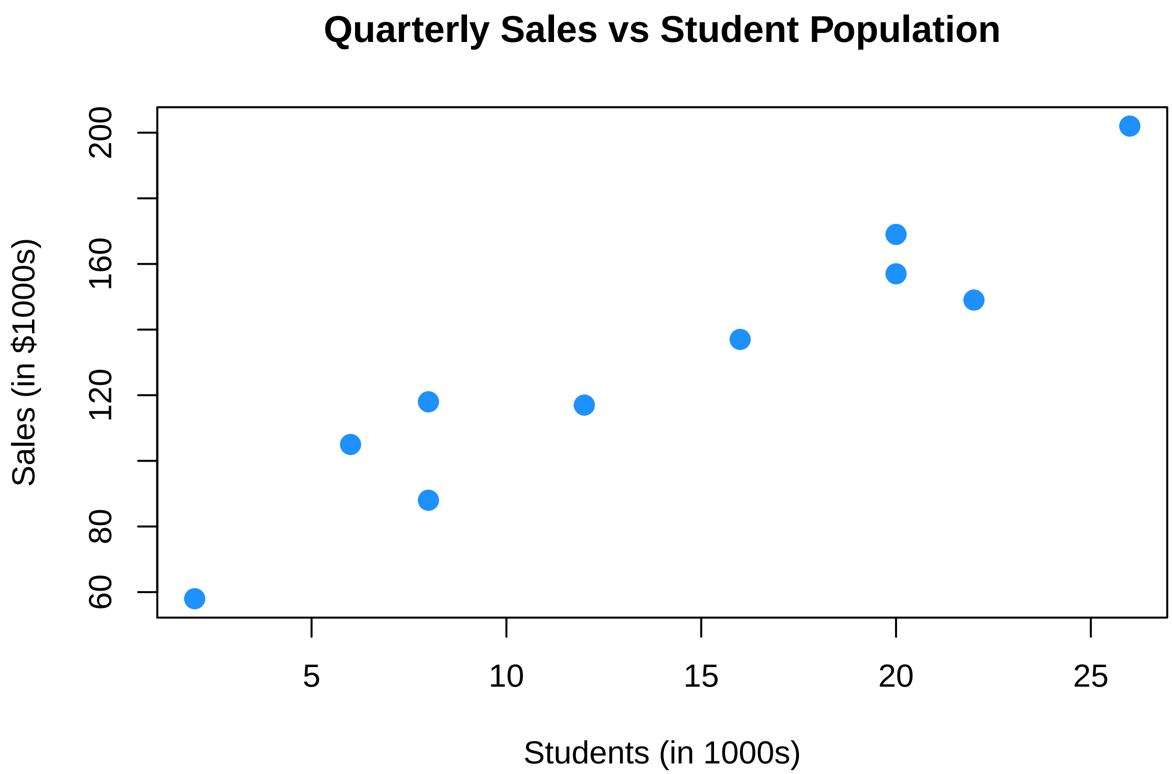
data is csv here

TODO: import into R/RStudio

```r
plot(momma_leona$students, momma_leona$sales)
```



```
plot(momma_leona$students, momma_leona$sales,
     xlab = "Students (in 1000s)",
     ylab = "Sales (in $1000s)",
     main = "Quarterly Sales vs Student Population",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
```



response, outcome predictor, explanatory

note about "independent"

How can you use this data to:

- Explain relationship
  - Significant?
  - Which *variables* (word?) are most important? (Say, a variable for public/private)
- Predict

One tool will do both, LINEAR REGRESSION

3.1 History

3.2 Model

$$y = f(x) + \epsilon$$

$$y = \beta_0 + \beta_1 x + \epsilon$$

PARAMETERS

DATA = PREDICTION + ERRORS = SIGNAL + NOISE = MODEL + ERROR

Unexplained:

- missing variables
- measurement error

3.3 What is the Best Line?

Goldilocks

- Underfitting (Just using y_bar)
- Just right (SLR)
- Overfitting (Connect the dots)

TODO: picture with lines. one for overall mean. one for LS?

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

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

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

historical, easy math

3.4 Least Squares Approach

Function to minimize.

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

Take derivatives.

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

Set equal to zero.

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

Rearrange, **normal equations**.

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

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

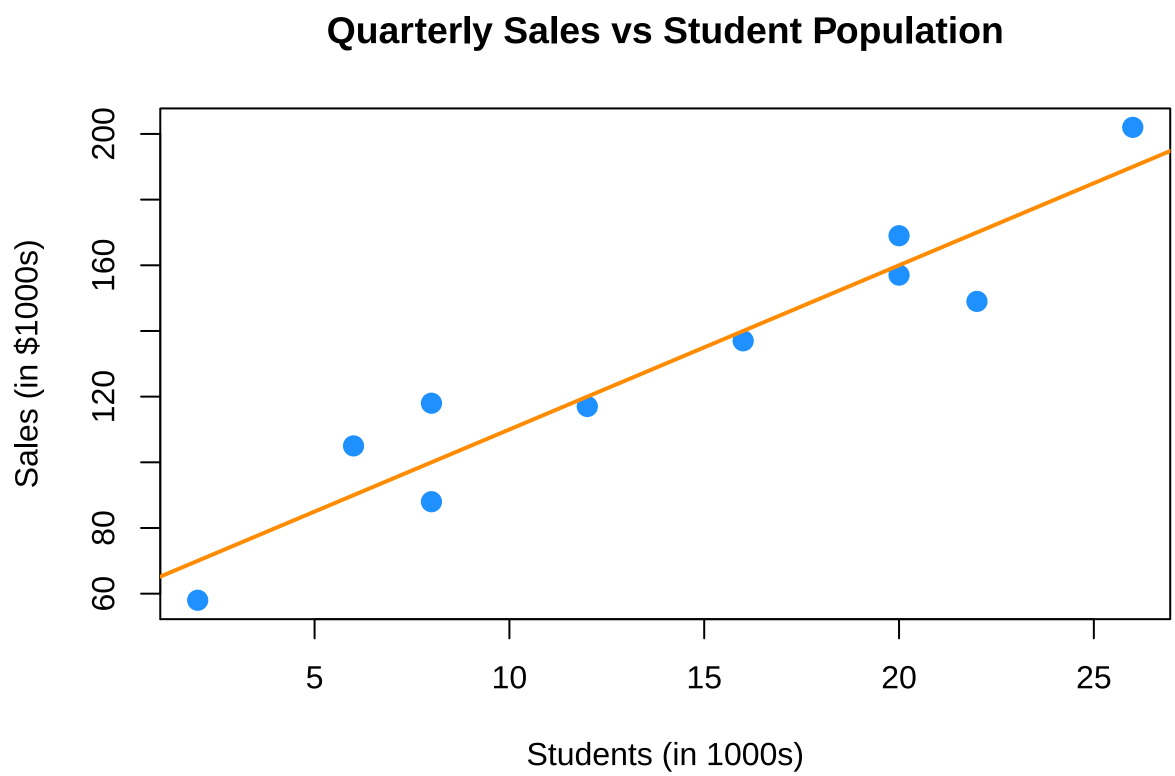
$$SXY = \sum_{i=1}^{n} x_i y_i - \frac{(\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n} = \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$
$$SXX = \sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n} = \sum_{i=1}^{n}(x_i - \bar{x})^2$$
$$SYY = \sum_{i=1}^{n} y_i^2 - \frac{(\sum_{i=1}^{n} y_i)^2}{n} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

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

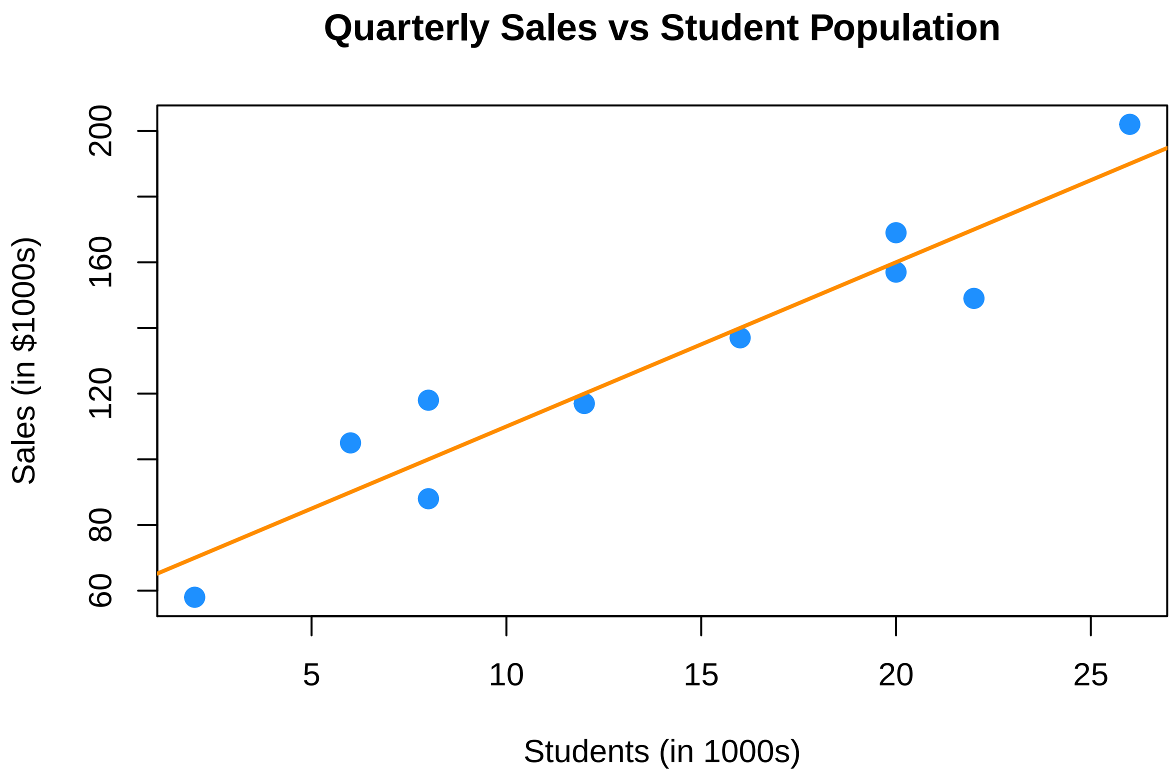
TODO: Fitted line. TODO: variance estimate

```r
momma_leona_model = lm(sales ~ students, data = momma_leona)
```

```r
plot(momma_leona$students, momma_leona$sales,
     xlab = "Students (in 1000s)",
     ylab = "Sales (in $1000s)",
     main = "Quarterly Sales vs Student Population",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
abline(momma_leona_model, lwd = 2, col = "darkorange")
```




```
plot(sales ~ students, data = momma_leona,
     xlab = "Students (in 1000s)",
     ylab = "Sales (in $1000s)",
     main = "Quarterly Sales vs Student Population",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
abline(momma_leona_model, lwd = 2, col = "darkorange")
```



TODO: fit in R TODO: extrapolation TODO: interpretation

3.5 Decomposition of Variation

TODO: Residuals

3.6 Coefficient of Determination

R^2

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

Decomposing variation.

3.7 MLE Approach

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

Chapter 4

Inference for Simple Linear Regression

> "There are three types of lies: lies, damn lies, and statistics."
> — **Benjamin Disraeli**

TODO: test push for will.

Bibliography