Rmd-ex-paper-02Oct2021

See GitHub

02/10/2021

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

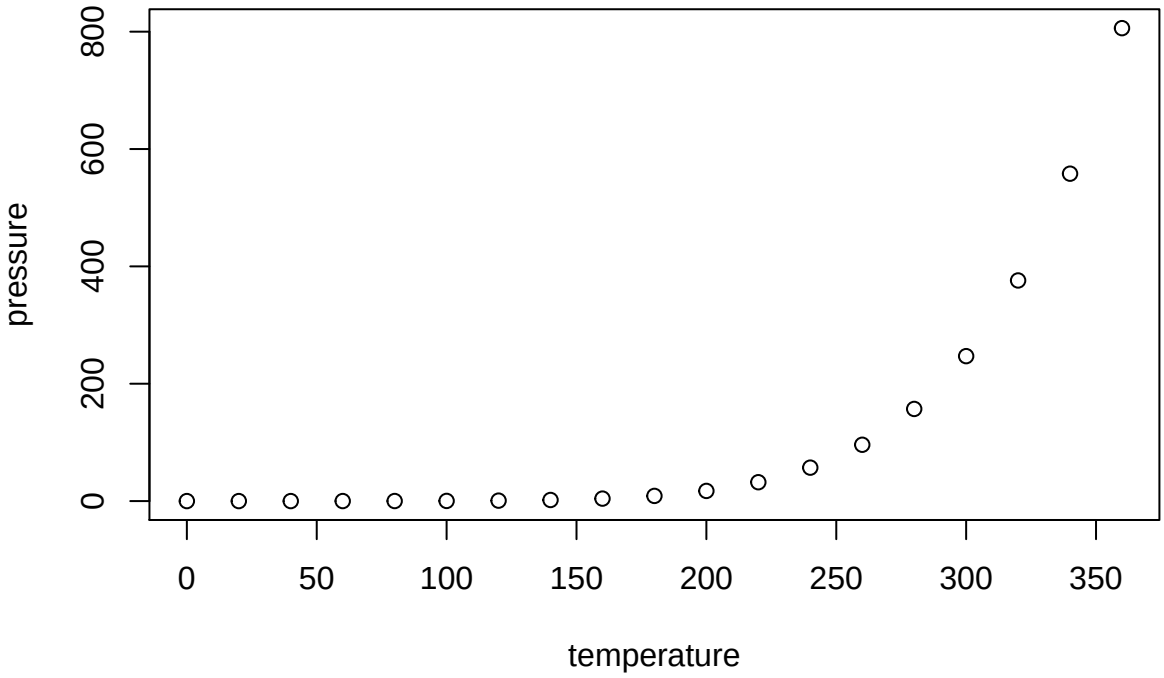
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00
```

Including Plots

You can also embed plots, for example:



Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Introduction to R

Getting Started

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, 4.0.5.
- 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.

The following few chapters will serve as a whirlwind introduction to R. They are by no means meant to be a complete reference for the R language, but simply an introduction to the basics that we will need along the way. Several of the more important topics will be re-stressed as they are actually needed for analyses.

These introductory R chapters may feel like an overwhelming amount of information. You are not expected to pick up everything the first time through. You should try all of the code from these chapters, then return to them a number of times as you return to the concepts when performing analyses.

R is used both for software development and data analysis. We will operate in a grey area, somewhere between these two tasks. Our main goal will be to analyze data, but we will also perform programming exercises that help illustrate certain concepts.

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 cheatsheet for "Base" R will summarize many of the concepts in this document. ("Base" R is a name used to differentiate the practice of using built-in R functions, as opposed to using functions from outside packages, in particular, those from the `tidyverse`. More on this later.)

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 important 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.)

Basic Calculations

To get started, we'll use R like a simple calculator.

Addition, Subtraction, Multiplication and Division

Math	R	Result
$3+2$	`3 + 2`	5
$3-2$	`3 - 2`	1
$3 \cdot 2$	`3 * 2`	6
$3/2$	`3 / 2`	1.5

Exponents

Math	R	Result
3^2	`3 ^ 2`	9
$2^{(-3)}$	`2 ^ (-3)`	0.125
$100^{1/2}$	`100 ^ (1 / 2)`	10
$\sqrt{100}$	`sqrt(100)`	10

Mathematical Constants

Math	R	Result
π	`pi`	3.1415927
e	`exp(1)`	2.7182818

Logarithms Note that we will use ln and log interchangeably to mean the natural logarithm. There is no `ln()` in R, instead it uses `log()` to mean the natural logarithm.

Math	R	Result
$\log(e)$	`log(exp(1))`	1
$\log_{10}(1000)$	`log10(1000)`	3
$\log_2(8)$	`log2(8)`	3
$\log_4(16)$	`log(16, base = 4)`	2

Trigonometry

Math	R	Result
$\sin(\pi/2)$	`sin(pi / 2)`	1
$\cos(0)$	`cos(0)`	1

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.

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 is not 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()`.

Data and Programming

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."`

Data Structures

`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	

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 elements 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`, partially due to the keys available on the keyboards of the creators of the `S` language. (Which preceded `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 `=`. If you are interested in the weird cases where the difference matters, check out The R Inferno.

If you wish to use `<-`, you will still need to use `=`, however only for argument passing. Some users like to keep assignment (`<-`) and argument passing (`=`) separate. No matter what you choose, the more important thing is that you **stay consistent**. Also, if working on a larger collaborative project, you should use whatever style is already in place.

Because vectors must contain elements that are all the same type, `R` will automatically coerce to a single type when attempting to create a vector that combines multiple types.

```
c(42, "Statistics", TRUE)
```

```
## [1] "42"         "Statistics" "TRUE"
```

```
c(42, TRUE)
```

```
## [1] 42  1
```

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.

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

```
2
```

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

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"
```

The `rep()` function can be used to repeat a vector some number of times.

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

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

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

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

```
length(x)
```

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

```
length(y)
```

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

Subsetting 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.

All of the above are subsetting a vector using a vector of indexes. (Remember a single number is still a vector.) We could instead use a vector of logical values.

```
z = c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE)
z
```

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

```
x[z]
```

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

Vectorization

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`.

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.

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

```
x > 3
```

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

```
x < 3
```

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

```
x == 3
```

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

```
x != 3
```

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

```
x == 3 & x != 3
```

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

```
x == 3 | x != 3
```

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

This is extremely useful for subsetting.

```
x[x > 3]
```

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

```
x[x != 3]
```

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

- TODO: coercion

```
sum(x > 3)
```

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

```
as.numeric(x > 3)
```

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

Here we see that using the `sum()` function on a vector of logical `TRUE` and `FALSE` values that is the result of `x > 3` results in a numeric result. `R` is first automatically coercing the logical to numeric where `TRUE` is `1` and `FALSE` is `0`. This coercion from logical to numeric happens for most mathematical operations.

```
which(x > 3)
```

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

```
x[which(x > 3)]
```

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

```
max(x)
```

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

```
which(x == max(x))
```

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

```
which.max(x)
```

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

More Vectorization

```
x = c(1, 3, 5, 7, 8, 9)
y = 1:100
```

```
x + 2
```

```
## [1]  3  5  7  9 10 11
```

```
x + rep(2, 6)
```

```
## [1]  3  5  7  9 10 11
```

```
x > 3
```

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

```
x > rep(3, 6)
```

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

```
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##   [1]   2   5   8  11  13  15   8  11  14  17  19  21  14  17  20  23  25  27
##  [19]  20  23  26  29  31  33  26  29  32  35  37  39  32  35  38  41  43  45
##  [37]  38  41  44  47  49  51  44  47  50  53  55  57  50  53  56  59  61  63
##  [55]  56  59  62  65  67  69  62  65  68  71  73  75  68  71  74  77  79  81
##  [73]  74  77  80  83  85  87  80  83  86  89  91  93  86  89  92  95  97  99
##  [91]  92  95  98 101 103 105  98 101 104 107
```

```
length(x)
```

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

```
length(y)
```

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

```
length(y) / length(x)
```

```
## [1] 16.66667
```

```
(x + y) - y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##   [1] 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1
##  [38] 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3
##  [75] 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7
```

```
y = 1:60
x + y
```

```
##  [1]  2  5  8 11 13 15  8 11 14 17 19 21 14 17 20 23 25 27 20 23 26 29 31 33 26
## [26] 29 32 35 37 39 32 35 38 41 43 45 38 41 44 47 49 51 44 47 50 53 55 57 50 53
## [51] 56 59 61 63 56 59 62 65 67 69
```

```
length(y) / length(x)
```

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

```
rep(x, 10) + y
```

```
##  [1]  2  5  8 11 13 15  8 11 14 17 19 21 14 17 20 23 25 27 20 23 26 29 31 33 26
## [26] 29 32 35 37 39 32 35 38 41 43 45 38 41 44 47 49 51 44 47 50 53 55 57 50 53
## [51] 56 59 61 63 56 59 62 65 67 69
```

```
all(x + y == rep(x, 10) + y)
```

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

```
identical(x + y, rep(x, 10) + y)
```

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

```
# ?any
# ?all.equal
```

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
```

```
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
```

```
cbind(col_1 = x, col_2 = rev(x), col_3 = rep(1, 9))
```

```
##       col_1 col_2 col_3
##  [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
```

When using `rbind` and `cbind` you can specify "argument" names that will be used as column names.

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
```

To verify that `solve(Z)` returns the inverse, we multiply it by `Z`. We would expect this to return the identity matrix, however we see that this is not the case due to some computational issues. However, `R` also has the `all.equal()` function which checks for equality, with some small tolerance which accounts for some computational issues. The `identical()` function is used to check for exact equality.

```
solve(Z) %*% Z
```

```
##              [,1]          [,2]         [,3]
## [1,] 1.000000e+00 -6.245005e-17 0.000000e+00
## [2,] 8.326673e-17  1.000000e+00 5.551115e-17
## [3,] 2.775558e-17  0.000000e+00 1.000000e+00
```

```
diag(3)
```

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

```
all.equal(solve(Z) %*% Z, diag(3))
```

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

`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
```

Calculations with Vectors and Matrices Certain operations in `R`, for example `%*%` have different behavior on vectors and matrices. To illustrate this, we will first create two vectors.

```
a_vec = c(1, 2, 3)
b_vec = c(2, 2, 2)
```

Note that these are indeed vectors. They are not matrices.

```
c(is.vector(a_vec), is.vector(b_vec))
```

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

```
c(is.matrix(a_vec), is.matrix(b_vec))
```

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

When this is the case, the `%*%` operator is used to calculate the **dot product**, also know as the **inner product** of the two vectors.

The dot product of vectors $\boldsymbol{a} = [a_1, a_2, \cdots a_n]$ and $\boldsymbol{b} = [b_1, b_2, \cdots b_n]$ is defined to be

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots a_n b_n.$$

```
a_vec %*% b_vec # inner product
```

```
##      [,1]
## [1,]   12
```

```
a_vec %o% b_vec # outer product
```

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

The `%o%` operator is used to calculate the **outer product** of the two vectors.

When vectors are coerced to become matrices, they are column vectors. So a vector of length n becomes an $n \times 1$ matrix after coercion.

```
as.matrix(a_vec)
```

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

If we use the `%*%` operator on matrices, `%*%` again performs the expected matrix multiplication. So you might expect the following to produce an error, because the dimensions are incorrect.

```
as.matrix(a_vec) %*% b_vec
```

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

At face value this is a 3×1 matrix, multiplied by a 3×1 matrix. However, when `b_vec` is automatically coerced to be a matrix, `R` decided to make it a "row vector", a 1×3 matrix, so that the multiplication has conformable dimensions.

If we had coerced both, then `R` would produce an error.

```
as.matrix(a_vec) %*% as.matrix(b_vec)
```

Another way to calculate a *dot product* is with the `crossprod()` function. Given two vectors, the `crossprod()` function calculates their dot product. The function has a rather misleading name.

```
crossprod(a_vec, b_vec)  # inner product
```

```
##      [,1]
## [1,]   12
```

```
tcrossprod(a_vec, b_vec)  # outer product
```

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

These functions could be very useful later. When used with matrices X and Y as arguments, it calculates

$$X^\top Y.$$

When dealing with linear models, the calculation

$$X^\top X$$

is used repeatedly.

```
C_mat = matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
D_mat = matrix(c(2, 2, 2, 2, 2, 2), 2, 3)
```

This is useful both as a shortcut for a frequent calculation and as a more efficient implementation than using `t()` and `%*%`.

```
crossprod(C_mat, D_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```
t(C_mat) %*% D_mat
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```
all.equal(crossprod(C_mat, D_mat), t(C_mat) %*% D_mat)
```

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

```
crossprod(C_mat, C_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```
t(C_mat) %*% C_mat
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```
all.equal(crossprod(C_mat, C_mat), t(C_mat) %*% C_mat)
```

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

Lists

A list is a one-dimensional heterogeneous data structure. So it is indexed like a vector with a single integer value, but each element can contain an element of any type.

```
# creation
list(42, "Hello", TRUE)
```

```
## [[1]]
## [1] 42
##
## [[2]]
## [1] "Hello"
##
## [[3]]
## [1] TRUE
```

```
ex_list = list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Hello!",
  d = function(arg = 42) {print("Hello World!")},
  e = diag(5)
)
```

Lists can be subset using two syntaxes, the `$` operator, and square brackets `[]`. The `$` operator returns a named **element** of a list. The `[]` syntax returns a **list**, while the `[[]]` returns an **element** of a list.

- `ex_list[1]` returns a list containing the first element.
- `ex_list[[1]]` returns the first element of the list, in this case, a vector.

```
# subsetting
ex_list$e
```

```
##      [,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
```

```
ex_list[1:2]
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] TRUE
```

```
ex_list[1]
```

```
## $a
## [1] 1 2 3 4
```

```
ex_list[[1]]
```

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

```
ex_list[c("e", "a")]
```

```
## $e
##      [,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
##
## $a
## [1] 1 2 3 4
```

```
ex_list["e"]
```

```
## $e
##      [,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
```

```
ex_list[["e"]]
```

```
##      [,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
```

```
ex_list$d
```

```
## function(arg = 42) {print("Hello World!")}
```

```
ex_list$d(arg = 1)
```

```
## [1] "Hello World!"
```

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 = c(rep("Hello", 9), "Goodbye"),
                          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 Goodbye FALSE
```

Unlike a list which has more flexibility, the elements of a data frame must all be vectors, and have the same length.

```
example_data$x
```

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

```
all.equal(length(example_data$x),
          length(example_data$y),
          length(example_data$z))
```

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

```
str(example_data)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ x: num  1 3 5 7 9 1 3 5 7 9
##  $ y: chr  "Hello" "Hello" "Hello" "Hello" ...
##  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

```
nrow(example_data)
```

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

```
ncol(example_data)
```

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

```
dim(example_data)
```

```
## [1] 10  3
```

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 from the `readr` package. Note that `R` has a built in function `read.csv()` that operates very similarly. The `readr` function `read_csv()` has a number of advantages. For example, it is much faster reading larger data. It also uses the `tibble` package to read the data as a tibble.

```
library(readr)
example_data_from_csv = read_csv("C:\\Users\\root\\Documents\\Rprojects\\Rmd-ex-paper-02Oct2021\\data\\
```

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

```
example_data_from_csv
```

```
## # A tibble: 10 x 3
##        x y       z
##    <dbl> <chr>   <lgl>
##  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 Goodbye FALSE
```

A tibble is simply a data frame that prints with sanity. Notice in the output above that we are given additional information such as dimension and variable type.

The `as_tibble()` function can be used to coerce a regular data frame to a tibble.

```
library(tibble)
example_data = as_tibble(example_data)
example_data
```

```
## # A tibble: 10 x 3
##        x y       z
##    <dbl> <chr>   <lgl>
##  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 Goodbye FALSE
```

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. In recent versions of RStudio, `read_csv()` is used by default, thus reading in a tibble.

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)
```

```
## # A tibble: 10 x 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4           1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4           2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4           2    2008     4 auto~ f        21    30 p     comp~
##  5 audi         a4           2.8  1999     6 auto~ f        16    26 p     comp~
##  6 audi         a4           2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4           3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 quattro   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 quattro   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 quattro   2    2008     4 manu~ 4        20    28 p     comp~
```

The function `head()` will display the first `n` observations of the data frame. The `head()` function was more useful before tibbles. Notice that `mpg` is a tibble already, so the output from `head()` indicates there are only `10` observations. Note that this applies to `head(mpg, n = 10)` and not `mpg` itself. Also note that tibbles print a limited number of rows and columns by default. The last line of the printed output indicates which rows and columns were omitted.

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4           1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4           2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4           2    2008     4 auto~ f        21    30 p     comp~
##  5 audi         a4           2.8  1999     6 auto~ f        16    26 p     comp~
```

```
##  6 audi         a4           2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4           3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 quattro   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 quattro   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 quattro   2    2008     4 manu~ 4        20    28 p     comp~
## # ... with 224 more rows
```

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. This information can also be found in the "Environment" window in RStudio.

```
str(mpg)
```

```
## tibble [234 x 11] (S3: tbl_df/tbl/data.frame)
##  $ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
##  $ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
##  $ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr [1:234] "f" "f" "f" "f" ...
##  $ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr [1:234] "p" "p" "p" "p" ...
##  $ class       : chr [1:234] "compact" "compact" "compact" "compact" ...
```

It is important to note that while matrices have rows and columns, data frames (tibbles) 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")]
```

```
## # A tibble: 6 x 3
##   manufacturer model       year
##   <chr>        <chr>      <int>
## 1 honda        civic       2008
## 2 honda        civic       2008
## 3 toyota       corolla     2008
## 4 volkswagen   jetta       1999
## 5 volkswagen   new beetle  1999
## 6 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.

When subsetting a data frame, be aware of what is being returned, as sometimes it may be a vector instead of a data frame. Also note that there are differences between subsetting a data frame and a tibble. A data frame operates more like a matrix where it is possible to reduce the subset to a vector. A tibble operates more like a list where it always subsets to another tibble.

Programming Basics

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
```

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 `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`.

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

```
##  [1]  7.0688430 -1.7987889 -6.5048051 -1.7021691  2.2691447  6.2076940
##  [7]  0.0403988  3.3345911  9.8179999  0.8423618
```

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

```
##  [1]  1.05313181 -0.77394860 -1.74357275 -0.75404112  0.06420549  0.87570131
##  [7] -0.39500370  0.28372928  1.61956611 -0.22976783
```

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 variance two ways.

By default, it will calculate the unbiased estimate of σ^2, which we will call s^2.

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

It will also have the ability to return the biased estimate (based on maximum likelihood) which we will call $\hat{\sigma}^2$.

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

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

```
get_var(test_sample)
```

```
## [1] 23.55592
```

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

```
## [1] 23.55592
```

```
var(test_sample)
```

```
## [1] 23.55592
```

We see the function is working as expected, and when returning the unbiased estimate it matches `R`'s built in function `var()`. Finally, let's examine the biased estimate of σ^2.

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

```
## [1] 21.20032
```

Summarizing Data

Summary Statistics

`R` has built in functions for a large number of summary statistics. For numeric variables, we can summarize data with the center and spread. We'll again look at the `mpg` dataset from the `ggplot2` package.

```
library(ggplot2)
```

Central Tendency

Measure	`R`	Result
Mean	`mean(mpg$cty)`	16.8589744
Median	`median(mpg$cty)`	17

Spread

Measure	`R`	Result
Variance	`var(mpg$cty)`	18.1130736
Standard Deviation	`sd(mpg$cty)`	4.2559457
IQR	`IQR(mpg$cty)`	5
Minimum	`min(mpg$cty)`	9
Maximum	`max(mpg$cty)`	35
Range	`range(mpg$cty)`	9, 35

Categorical

For categorical variables, counts and percentages can be used for summary.

```
table(mpg$drv)
```

```
##
##   4   f   r
## 103 106  25
```

```
table(mpg$drv) / nrow(mpg)
```

```
##
##         4         f         r
## 0.4401709 0.4529915 0.1068376
```

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.

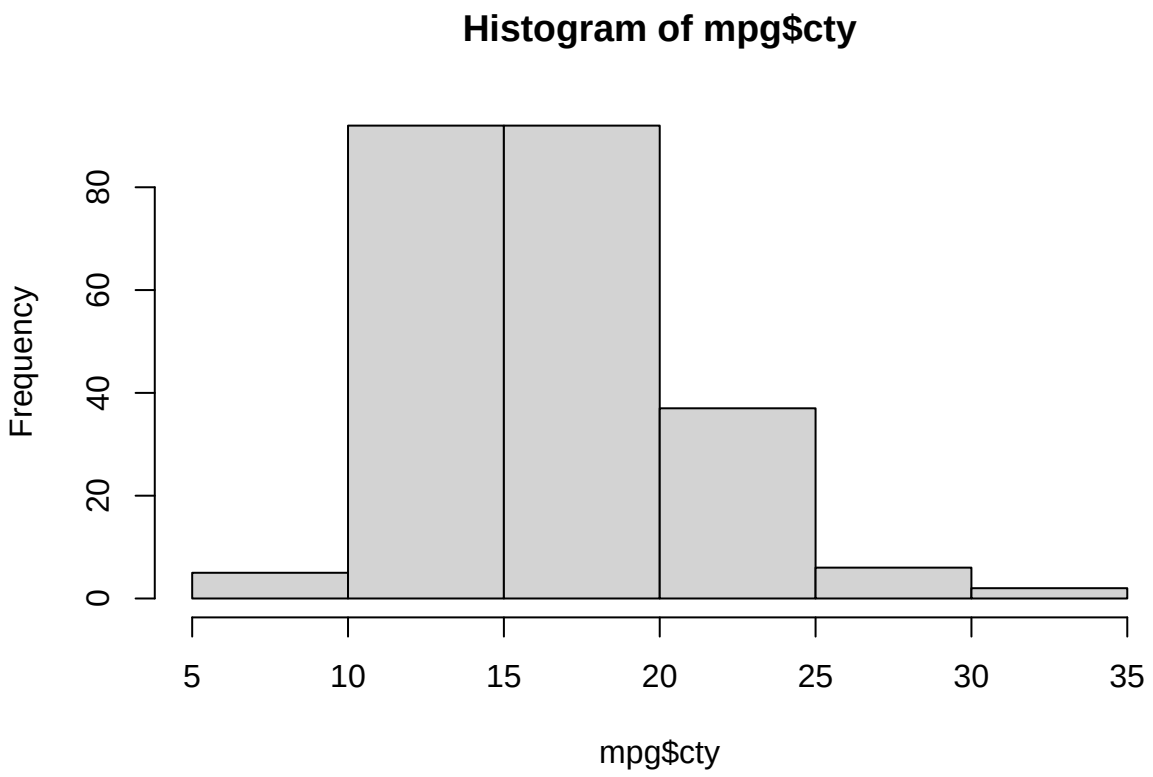
We will look at four methods of visualizing data that we will use throughout the course:

- Histograms
- Barplots
- Boxplots
- Scatterplots

Histograms

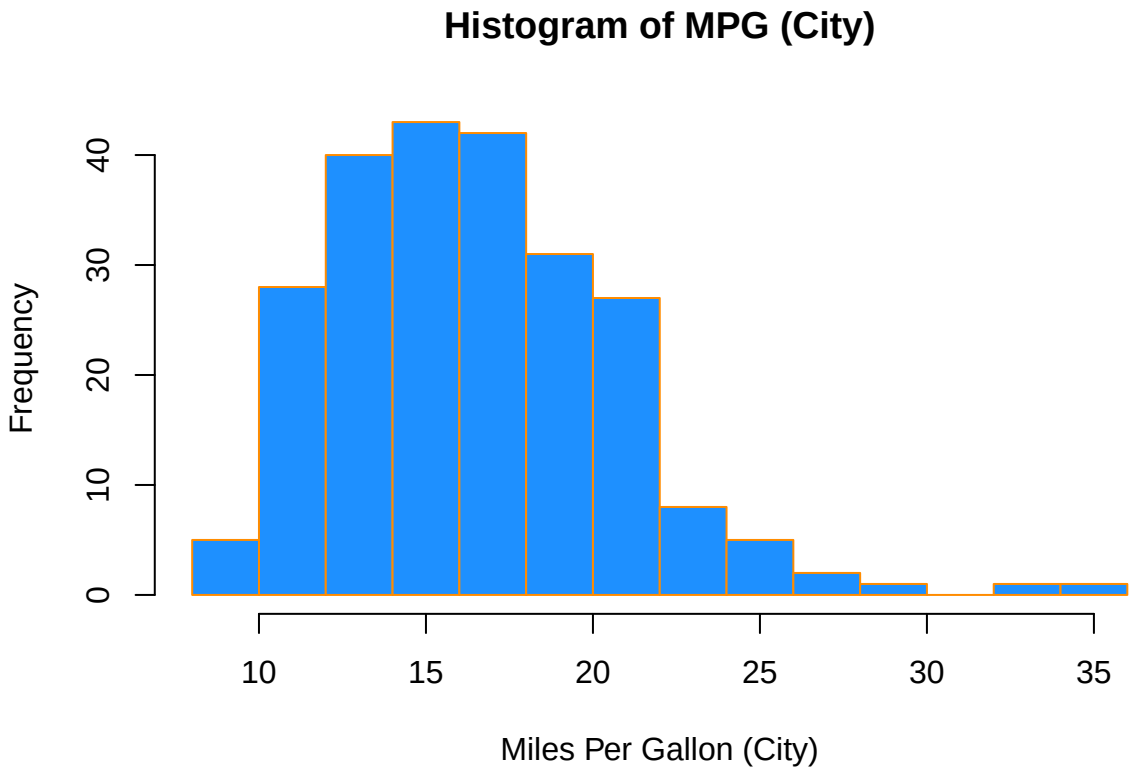
When visualizing a single numerical variable, a **histogram** will be our go-to tool, which can be created in `R` using the `hist()` function.

```
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")
```

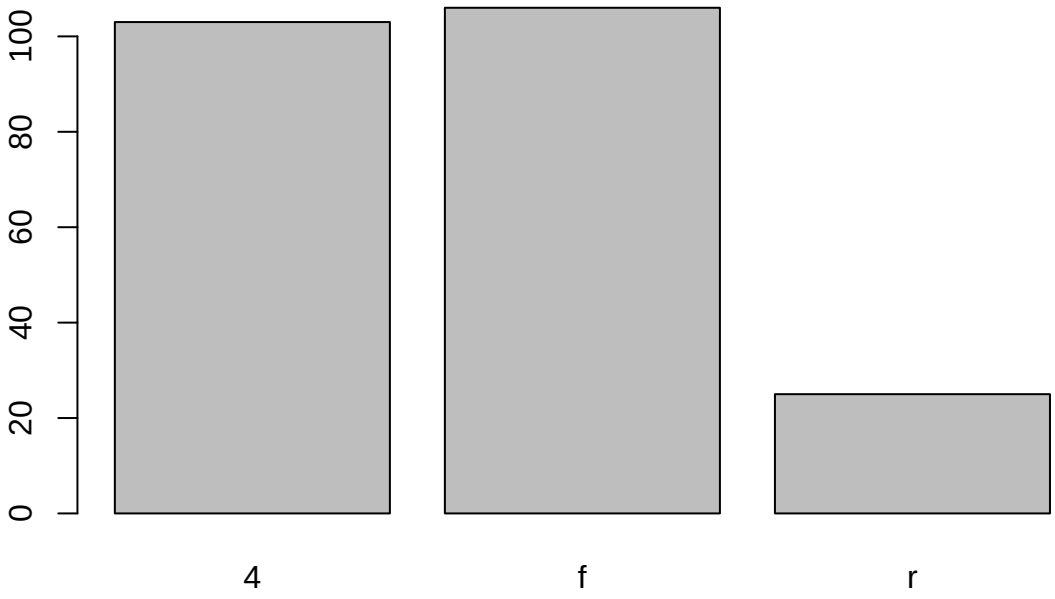


Importantly, you should always be sure to label your axes and give the plot a title. The argument `breaks` is specific to `hist()`. Entering an integer will give a suggestion to `R` for how many bars to use for the histogram. By default `R` will attempt to intelligently guess a good number of `breaks`,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

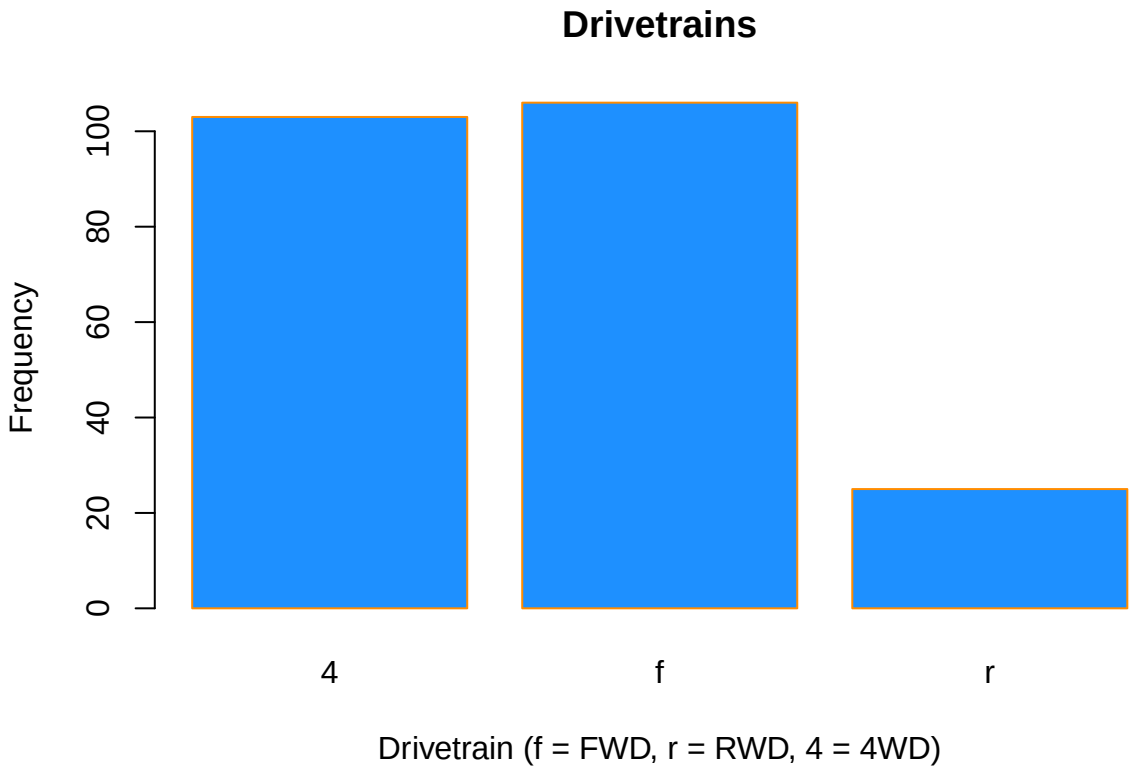
Barplots

Somewhat similar to a histogram, a barplot can provide a visual summary of a categorical variable, or a numeric variable with a finite number of values, like a ranking from 1 to 10.

```
barplot(table(mpg$drv))
```




```
barplot(table(mpg$drv),
        xlab   = "Drivetrain (f = FWD, r = RWD, 4 = 4WD)",
        ylab   = "Frequency",
        main   = "Drivetrains",
        col    = "dodgerblue",
        border = "darkorange")
```



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

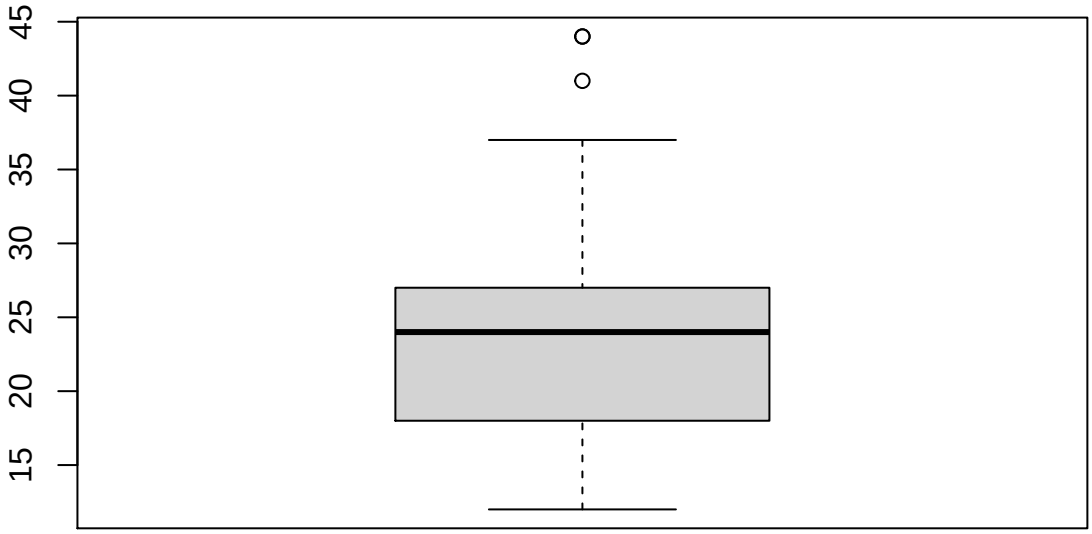
drive, or rear wheel drive.

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

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

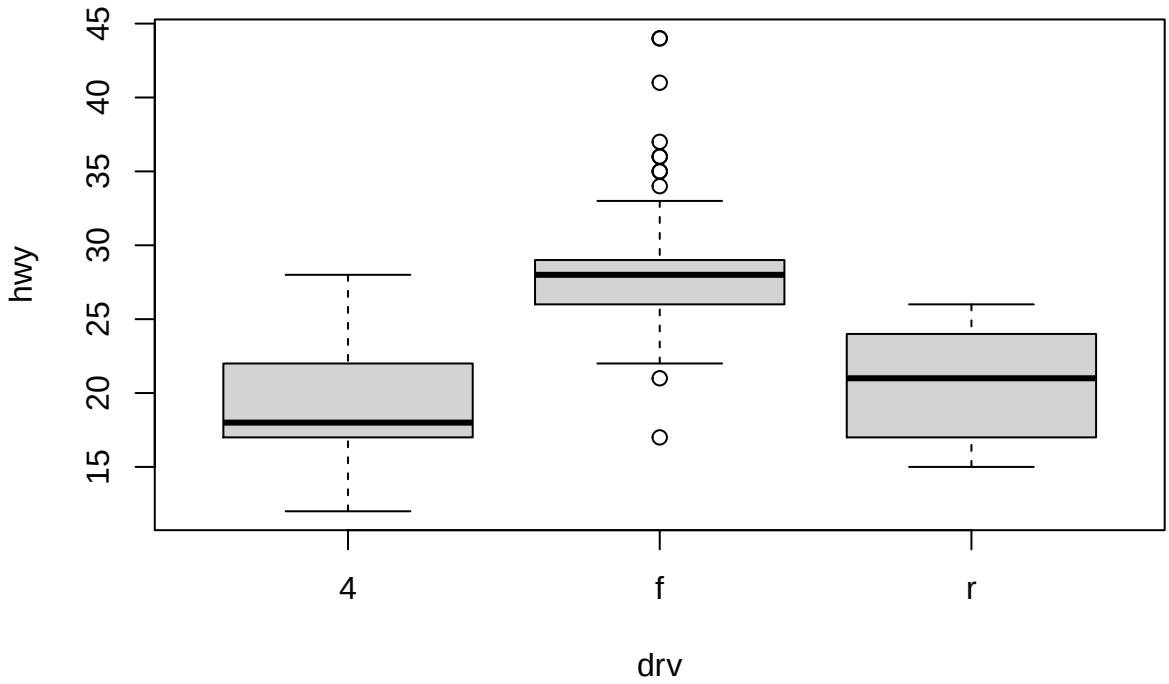
First note that we can use a single boxplot as an alternative to a histogram for visualizing a single numerical variable. To do so in R, we use the `boxplot()` function.

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



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

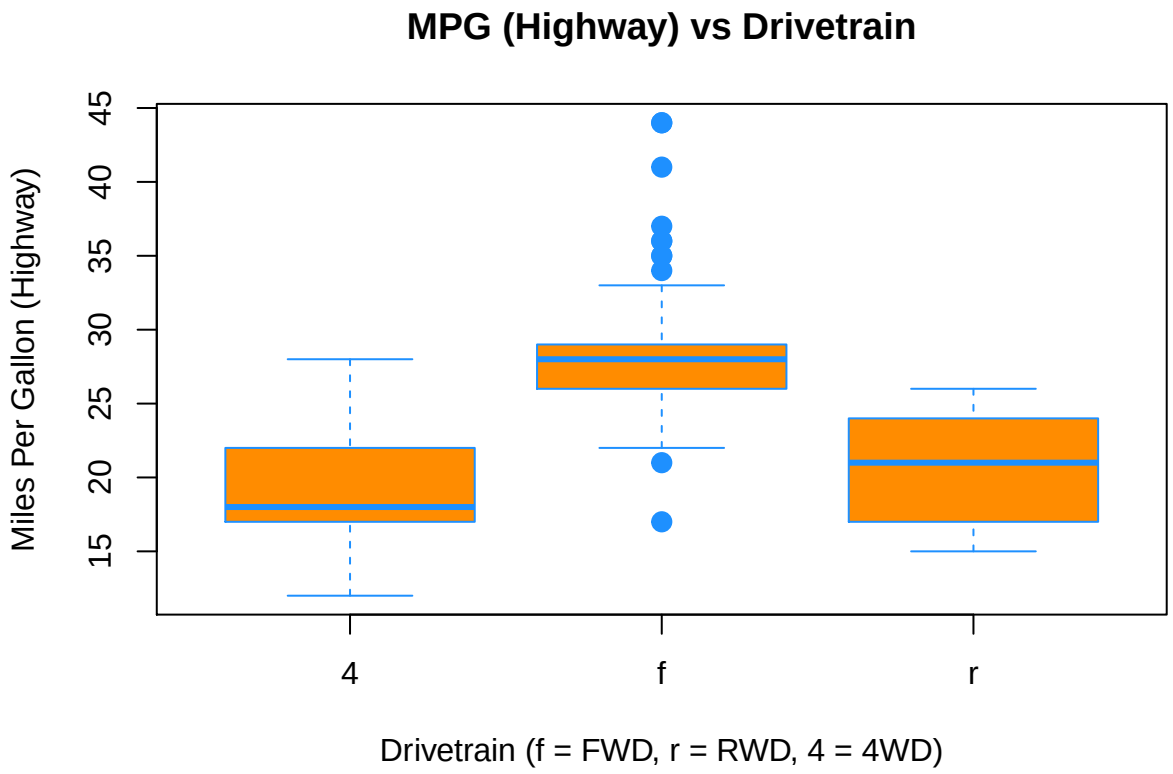
```
boxplot(hwy ~ drv, data = mpg)
```



Here we 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")
```

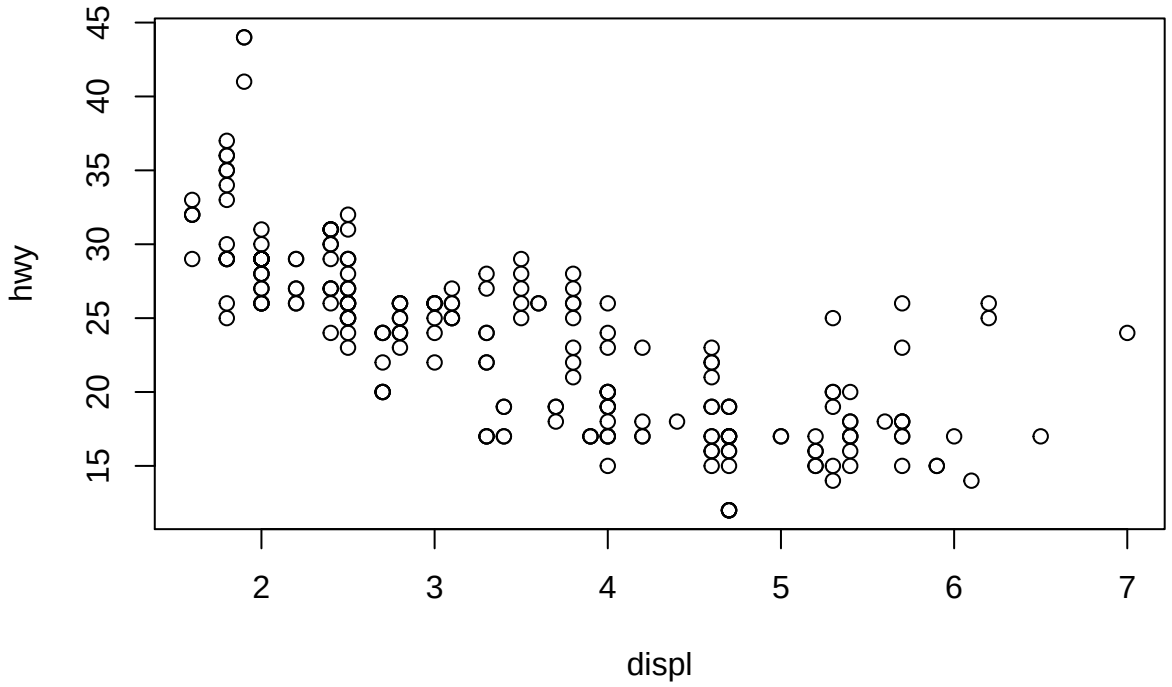


Again, `boxplot()` has a number of additional arguments which have the ability to make our plot more visually appealing.

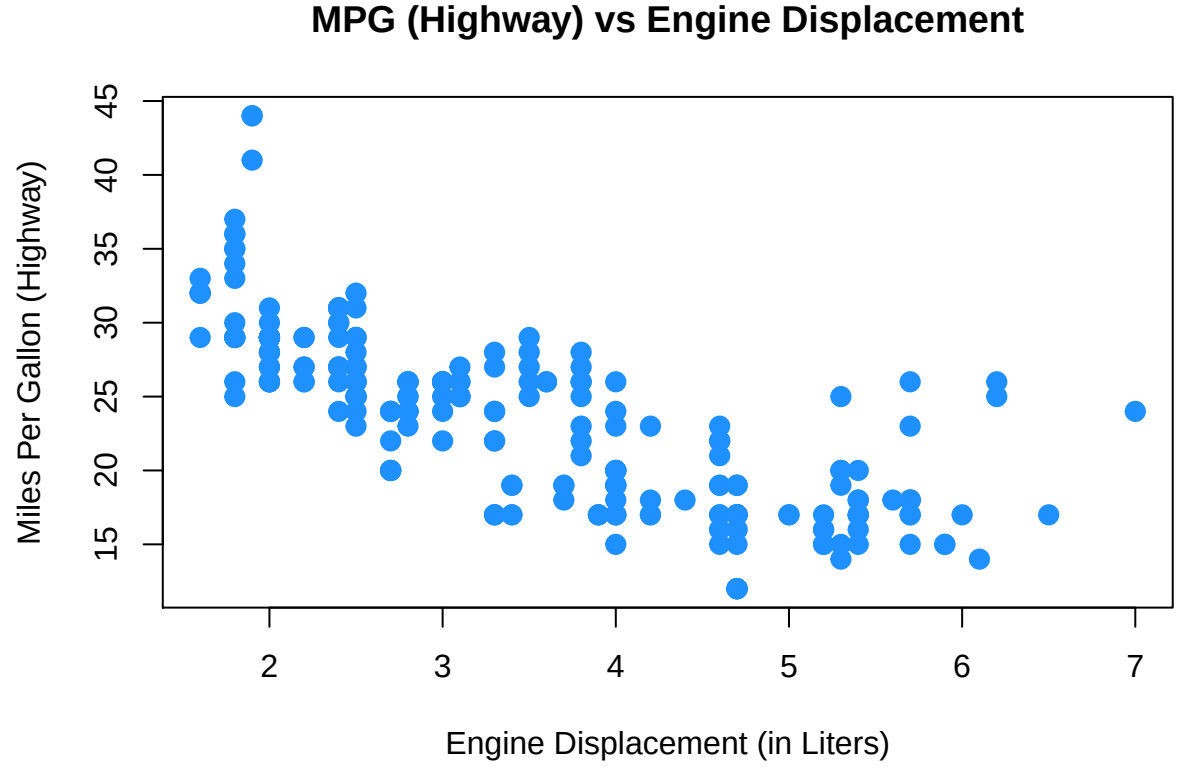
Scatterplots

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")
```



Probability and Statistics in R

Probability in R

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]  6.3122579 -4.8021015 10.8551126  8.1351438  9.6971849 12.1260337
##  [7] -0.9100311 -0.7309197 -0.1885474 -4.0982959
```

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

Command	Distribution
`*binom`	Binomial
`*t`	t
`*pois`	Poisson

Command	Distribution
`*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`.)

Hypothesis Tests in `R`

A prerequisite for STAT 420 is an understanding of the basics of hypothesis testing. Recall the basic structure of hypothesis tests:

- An overall model and related assumptions are made. (The most common being observations following a normal distribution.)
- The **null** (H_0) and **alternative** (H_1 or H_A) hypothesis are specified. Usually the null specifies a particular value of a parameter.
- With given data, the **value** of the *test statistic* is calculated.
- Under the general assumptions, as well as assuming the null hypothesis is true, the **distribution** of the *test statistic* is known.
- Given the distribution and value of the test statistic, as well as the form of the alternative hypothesis, we can calculate a **p-value** of the test.
- Based on the **p-value** and pre-specified level of significance, we make a decision. One of:
  - Fail to reject the null hypothesis.
  - Reject the null hypothesis.

We'll do some quick review of two of the most common tests to show how they are performed using `R`.

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.

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 the 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"  "stderr"      "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
```

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

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

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

Two Sample t-Test: Example

Assume that the distributions of X and Y are $\text{N}(\mu_1, \sigma^2)$ and $\text{N}(\mu_2, \sigma^2)$, 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
```

Simulation

Simulation and model fitting are related but opposite processes.

- In **simulation**, the *data generating process* is known. We will know the form of the model as well as the value of each of the parameters. In particular, we will often control the distribution and parameters which define the randomness, or noise in the data.
- In **model fitting**, the *data* is known. We will then assume a certain form of model and find the best possible values of the parameters given the observed data. Essentially we are seeking to uncover the truth. Often we will attempt to fit many models, and we will learn metrics to assess which model fits best.

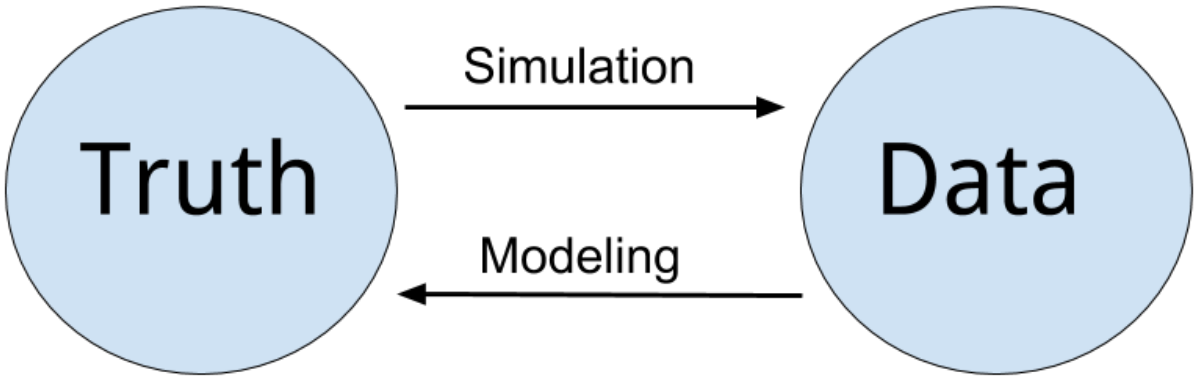


Figure 1: Simulation vs Modeling

Often we will simulate data according to a process we decide, then use a modeling method seen in class. We can then verify how well the method works, since we know the data generating process.

One of the biggest strengths of `R` is its ability to carry out simulations using built-in functions for generating random samples from certain distributions. We'll look at two very simple examples here, however simulation will be a topic we revisit several times throughout the course.

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 10^4 values of d_s generated) that are between 0 and 2.

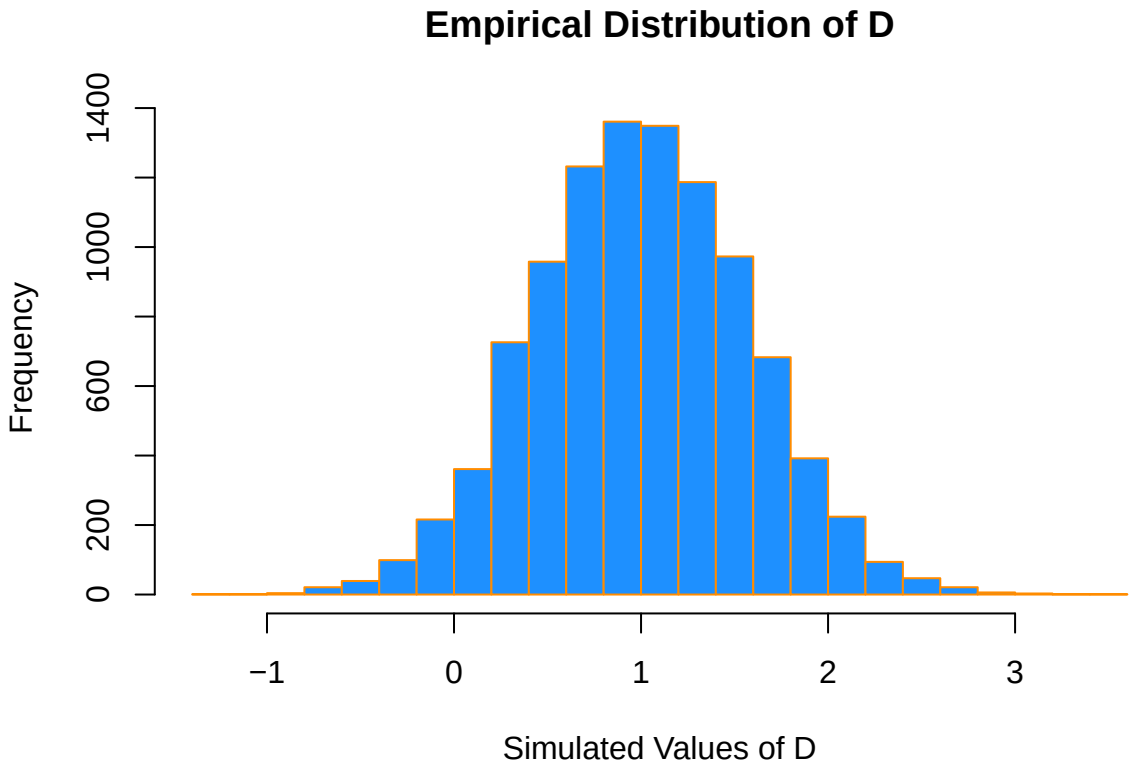
```
mean(0 < differences & differences < 2)
```

```
## [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!

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 $n \to \infty$,

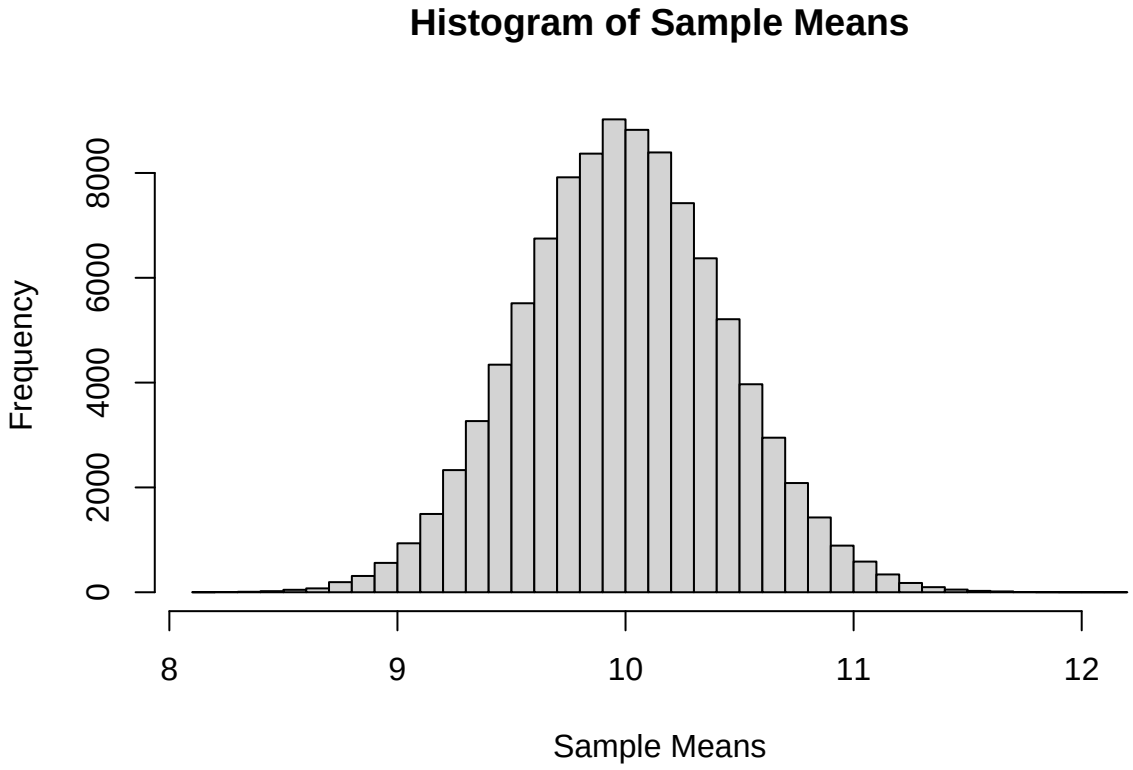
$$\bar{X} \xrightarrow{d} N\left(\mu, \frac{\sigma^2}{n}\right).$$

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.

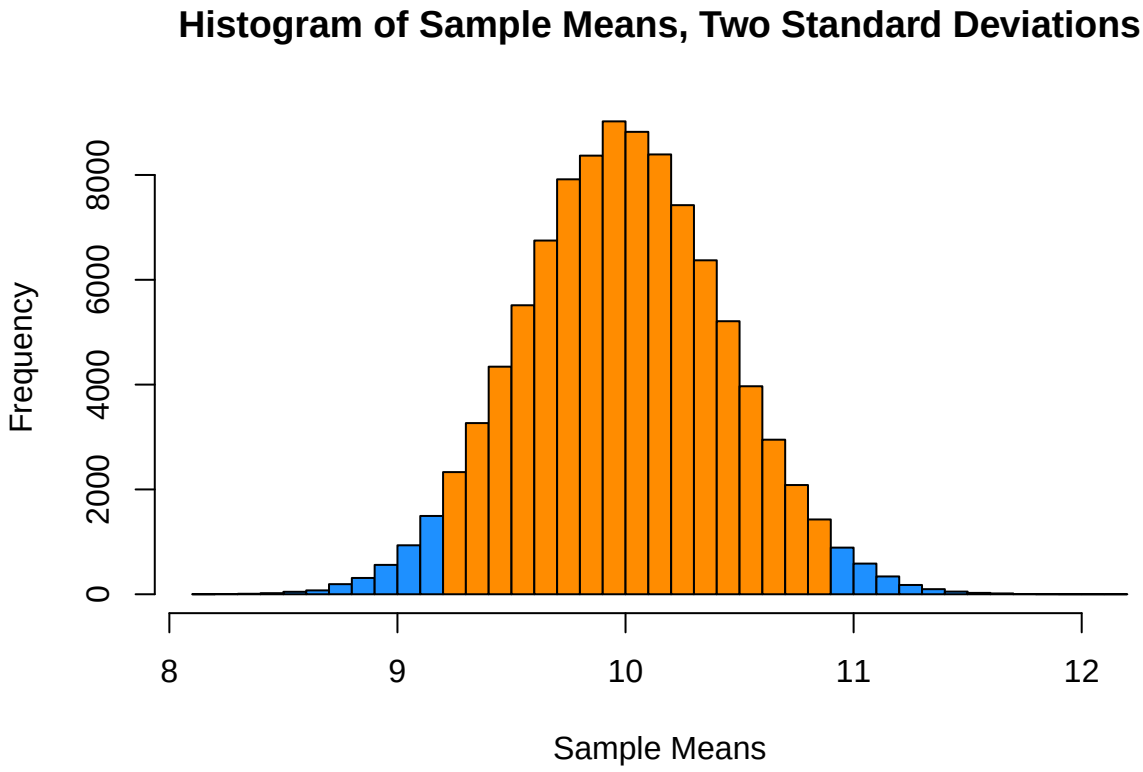
```
mean(x_bars > mu - 2 * sqrt(mu) / sqrt(sample_size) &
     x_bars < mu + 2 * sqrt(mu) / sqrt(sample_size))
```

```
## [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")
```



R Resources

So far, we have seen a lot of R, and a lot of R quickly. Again, the preceding chapters were in no way meant to be a complete reference for the R language, but rather an introduction to many of the concepts we will need in this text. The following resources are not necessary for the remainder of this text, but you may find them useful if you would like a deeper understanding of R:

Beginner Tutorials and References

- Try R from Code School.
  - An interactive introduction to the basics of R. 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 Programming for Data Science by Roger Peng
  - A great text for R programming beginners. Discusses R from the ground up, highlighting programming details we might not discuss.

Intermediate References

- 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. Especially useful for working in the tidyverse.
- 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 References

- 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.)
- 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.
- Efficient R Programming by Colin Gillespie and Robin Lovelace
  - Discusses both efficient R programs, as well as programming in R efficiently.

Quick Comparisons to Other Languages

Those who are familiar with other languages may find the following "cheatsheets" helpful for transitioning to R.

- MATLAB, NumPy, Julia
- Stata
- SAS - Look for a resource still! Suggestions welcome.

RStudio and RMarkdown Videos

The following video playlists were made as an introduction to R, RStudio, and RMarkdown for STAT 420 at UIUC. If you are currently using this text for a Coursera course, you can also find updated videos there.

- R and RStudio
- Data in R
- RMarkdown

Note that RStudio and RMarkdown are constantly receiving excellent support and updates, so these videos may already contain some outdated information.

RStudio provides their own tutorial for RMarkdown. They also have an excellent RStudio "cheatsheet" which visually identifies many of the features available in the IDE. You may also explore some additional "cheatsheets" on their website.

RMarkdown Template

This `.zip` file contains the files necessary to produce this rendered document. This document is a more complete version of a template than what is seen in the above videos.

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)
```

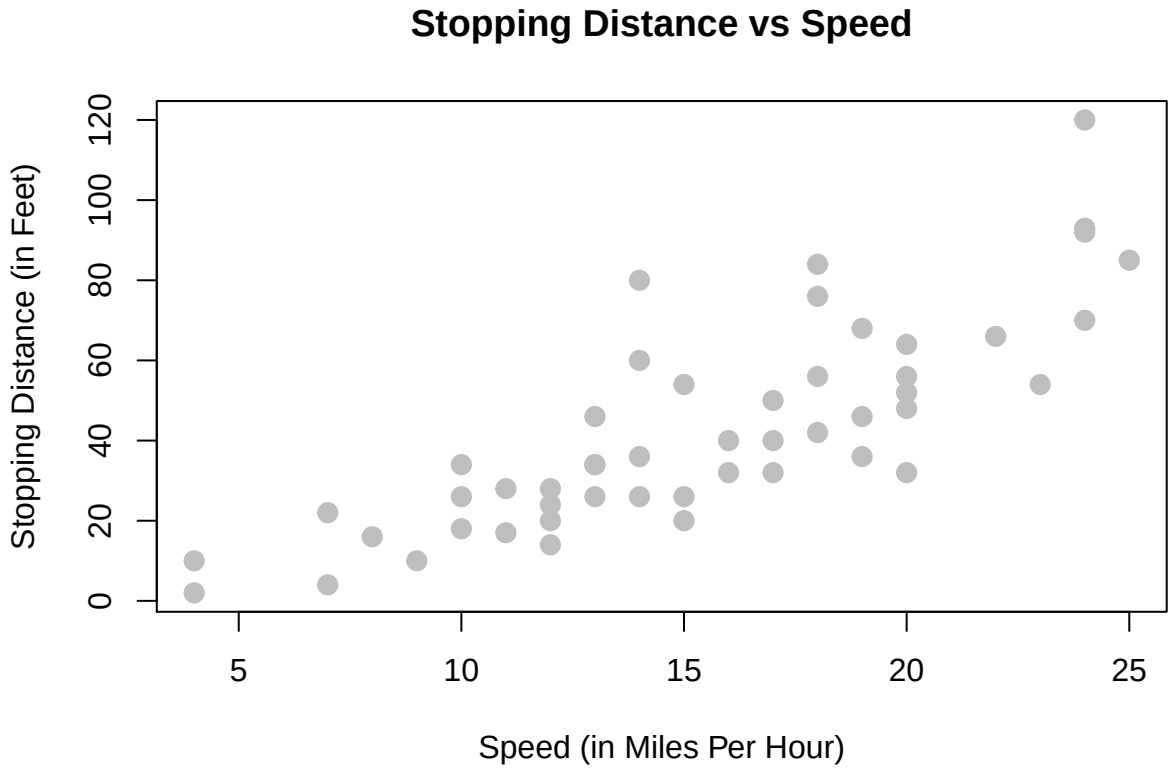
```
## [1] 2
```

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  = 2,
     col  = "grey")
```



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 other 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

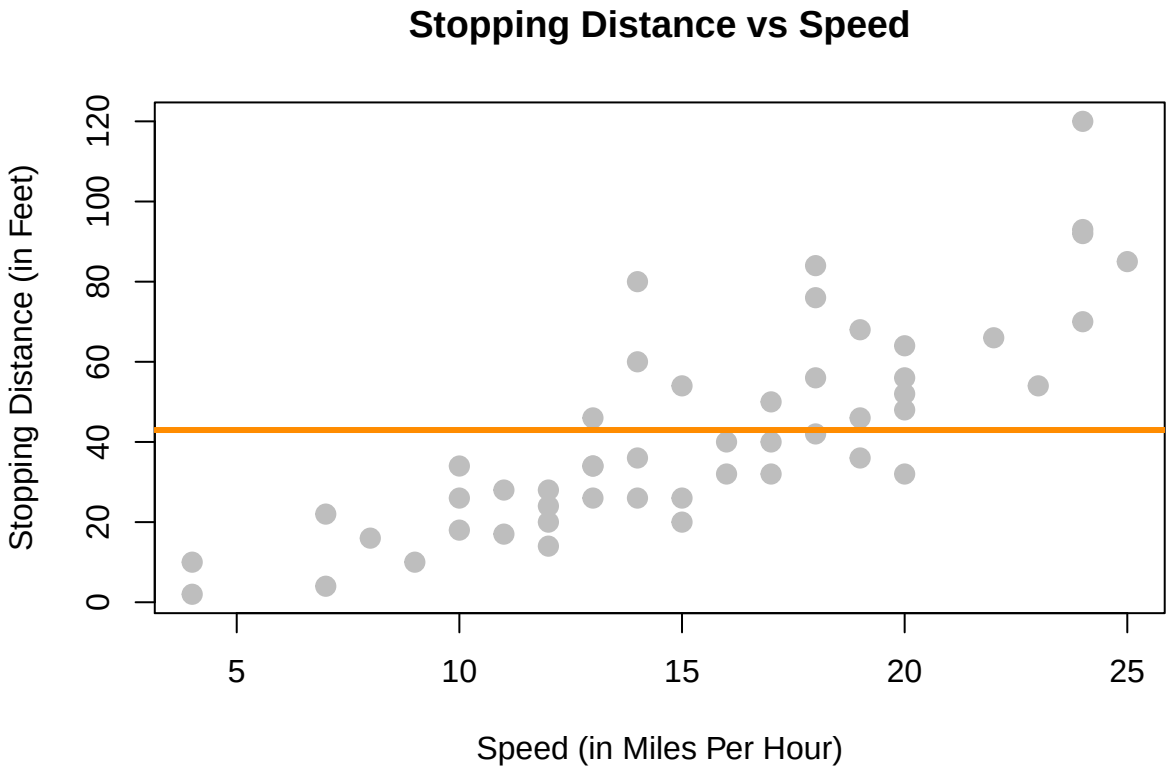
$$Y = f(X) + \epsilon.$$

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:

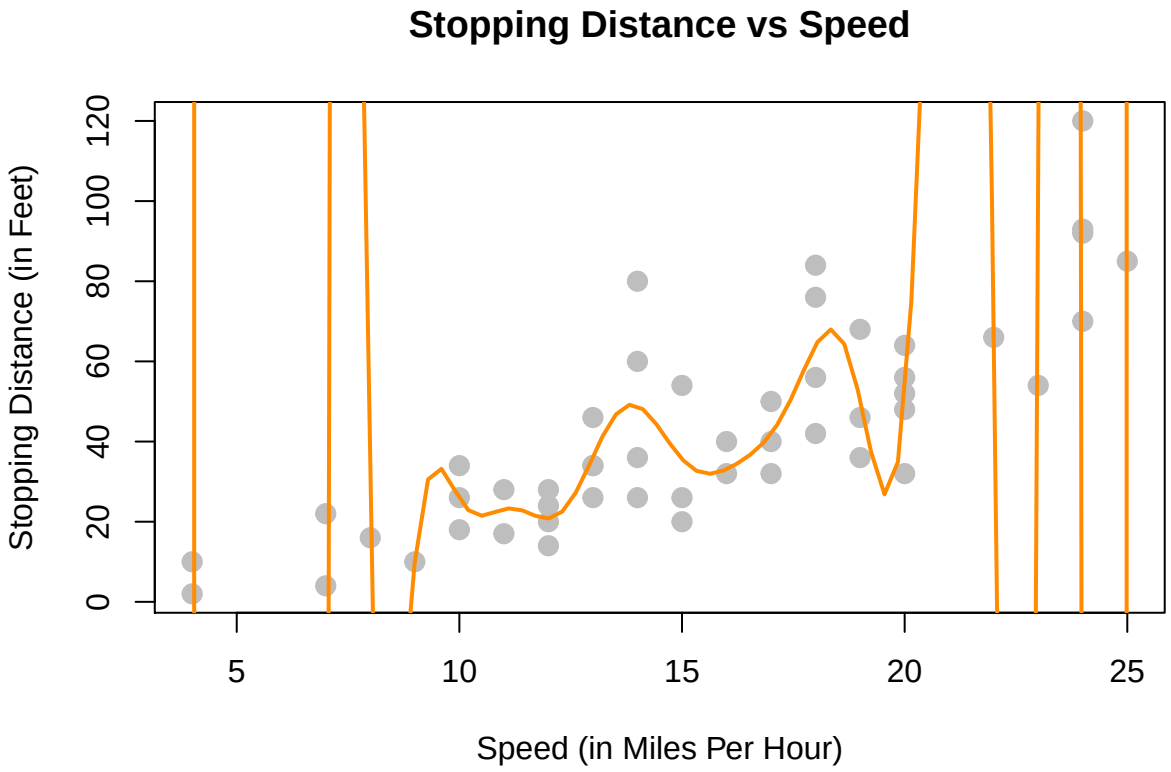
- Response = Prediction + Error
- Response = Signal + Noise
- Response = Model + Unexplained
- Response = Deterministic + Random
- Response = Explainable + Unexplainable

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

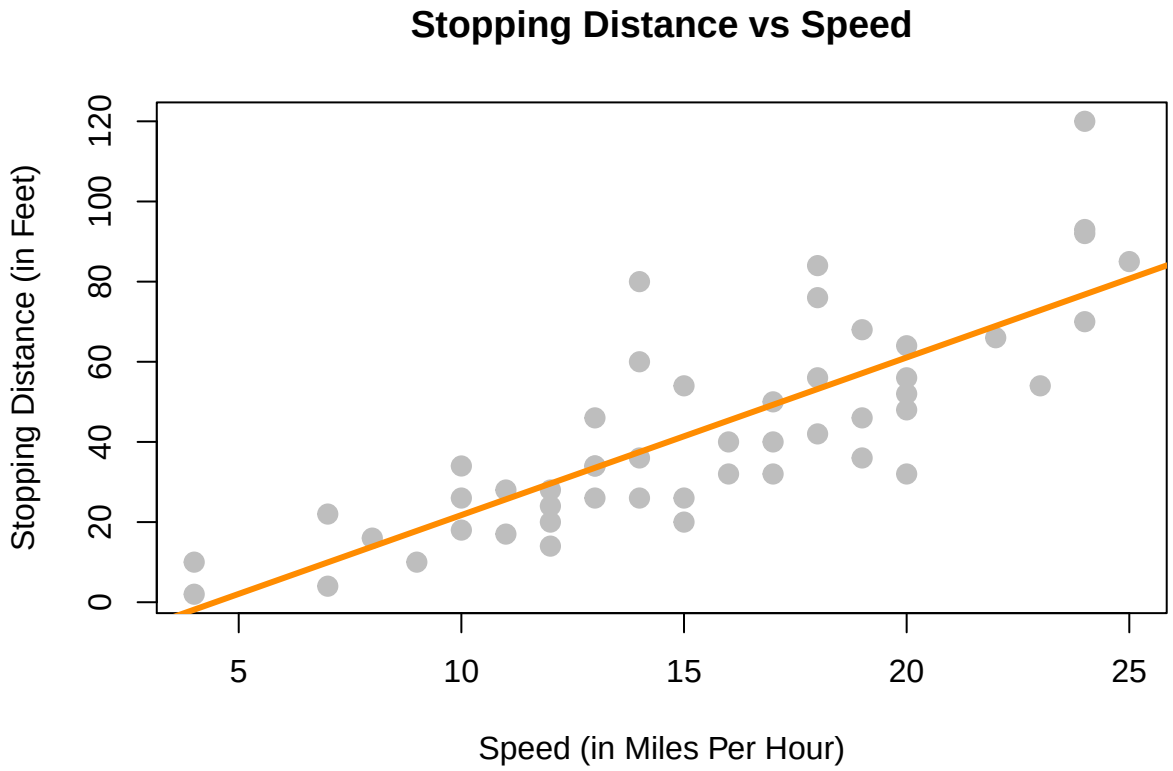
We could try to model the data with a horizontal line. That is, the model for y does not depend on the value of x. (Some function $f(X) = c$.) In the plot below, we see this doesn't seem to do a very good job. Many of the data points are very far from the orange line representing c. This is an example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

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

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.

Essentially, we could explicitly think of the Y_i as having a different distribution for each X_i. In other words, Y_i has a conditional distribution dependent on the value of X_i, written x_i. Doing so, we still make no distributional assumptions of the X_i, since we are only interested in the distribution of the Y_i for a particular value x_i.

$$Y_i \mid X_i \sim N(\beta_0 + \beta_1 x_i, \sigma^2)$$

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

$$\mathrm{E}[Y_i \mid X_i = x_i] = \beta_0 + \beta_1 x_i.$$

However, its variance remains constant for each x_i,

$$\mathrm{Var}[Y_i \mid X_i = x_i] = \sigma^2.$$

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.

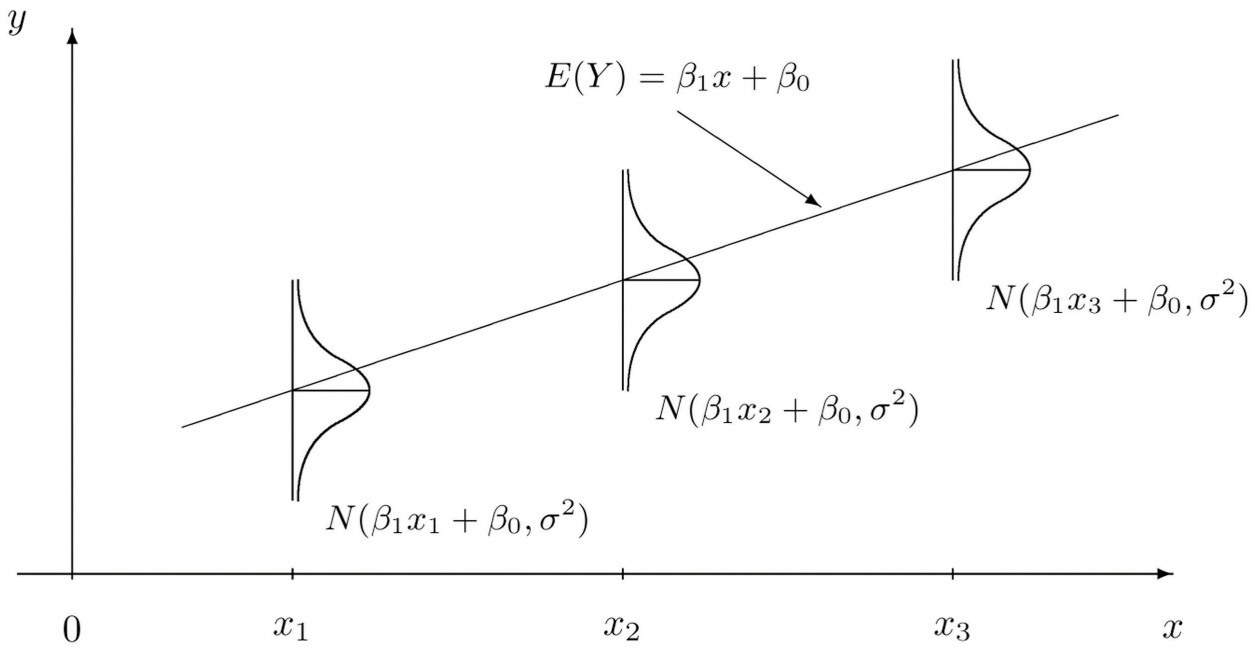


Figure 2: Simple Linear Regression Model Introductory Statistics (Shafer and Zhang), UC Davis Stat Wiki

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**ndependent. 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.

We are also assuming that the values of x are fixed, that is, not random. We do not make a distributional assumption about the predictor variable.

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. In the case of SLR, both the response and the predictor are *numeric* 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.

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**.

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

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 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* $\hat{\beta}_1 = 3.93$ tells us that for an increase in speed of one mile per hour, the **estimated** *mean* stopping distance increases by 3.93 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* $\hat{\beta}_0 = -17.58$ tells us that the **estimated** mean stopping distance for a car traveling zero miles per hour is -17.58 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.)

Making Predictions

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

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

In this case,

$$\hat{y} = -17.58 + 3.93x.$$

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.

$$\hat{y} = -17.58 + 3.93 \times 8$$

```
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
```

$$\hat{y} = -17.58 + 3.93 \times 21$$

```
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
```

$$\hat{y} = -17.58 + 3.93 \times 50$$

```
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 $\hat{\beta}_0 = -17.58$, 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.

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.

Variance Estimation

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

Recall that,

$$\mathrm{E}[Y_i \mid X_i = x_i] = \beta_0 + \beta_1 x_i.$$

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.

$$
\begin{aligned}
s_e^2 &= \frac{1}{n-2}\sum_{i=1}^{n}(y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i))^2 \\
&= \frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \\
&= \frac{1}{n-2}\sum_{i=1}^{n} e_i^2
\end{aligned}
$$

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 236.53 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.

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.

Sum of Squares Total

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

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,

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

Sum of Squares Regression

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

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

Sum of Squares Error

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

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,

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

```
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.

Coefficient of Determination

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

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

$$= \frac{\text{SST} - \text{SSE}}{\text{SST}} = 1 - \frac{\text{SSE}}{\text{SST}}$$

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

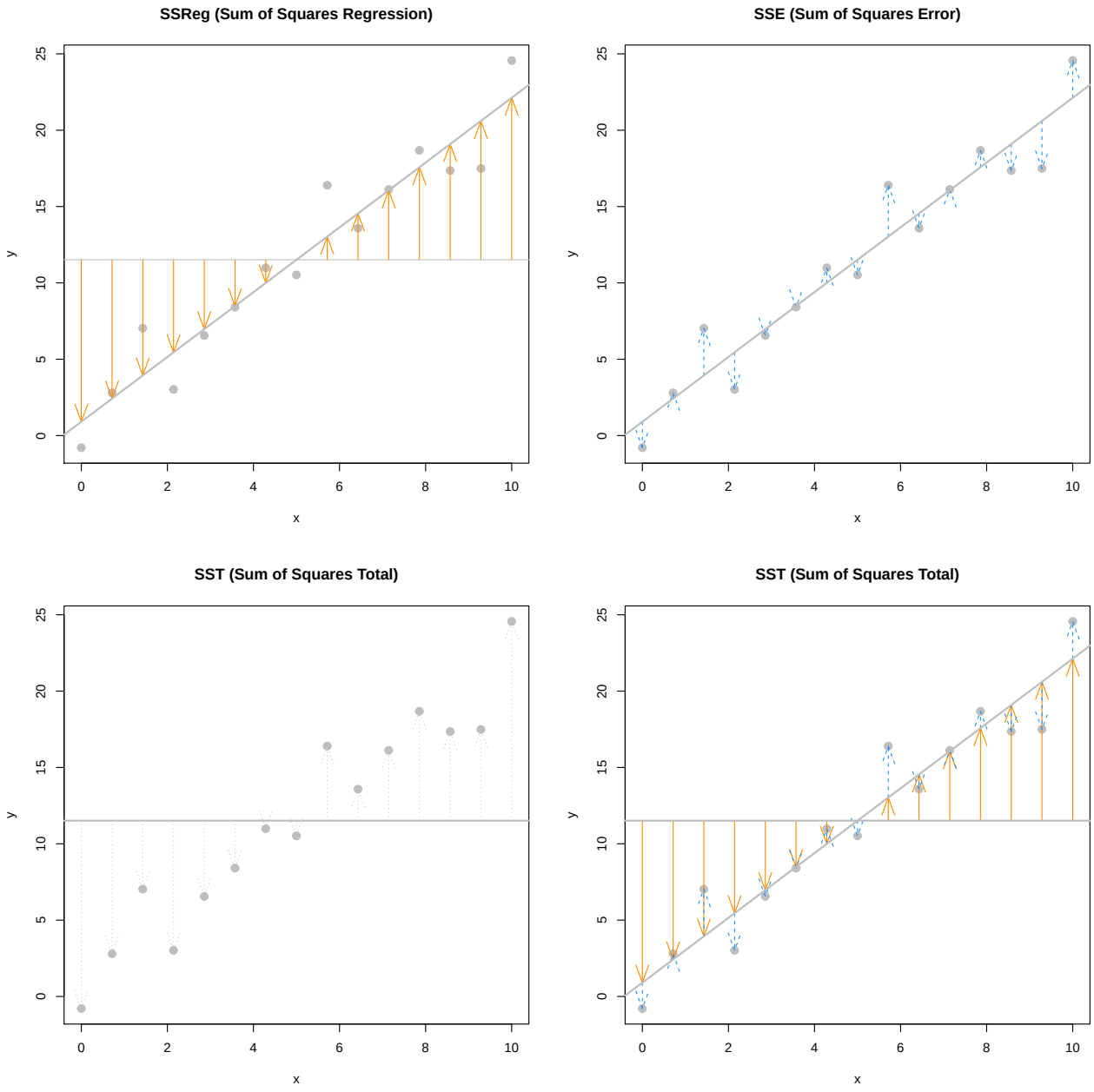
The coefficient of determination is interpreted as the proportion of observed variation in y that can be explained by the simple linear regression model.

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

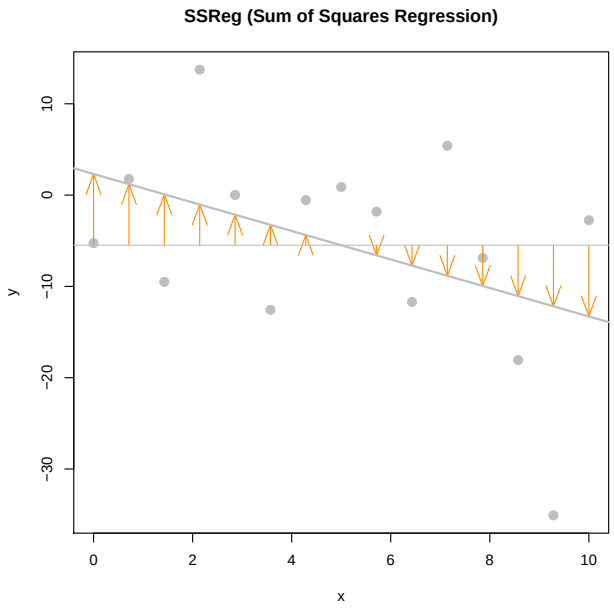
```
## [1] 0.6510794
```

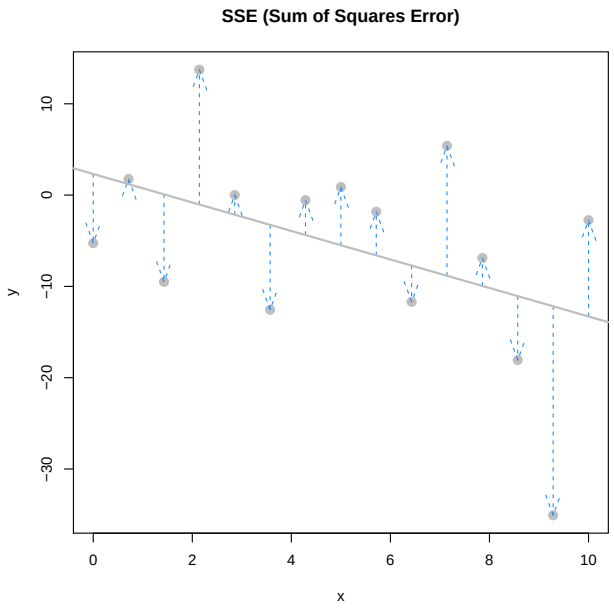
For the `cars` example, we calculate $R^2 = 0.65$. We then say that 65% of the observed variability in stopping distance is explained by the linear relationship with speed.

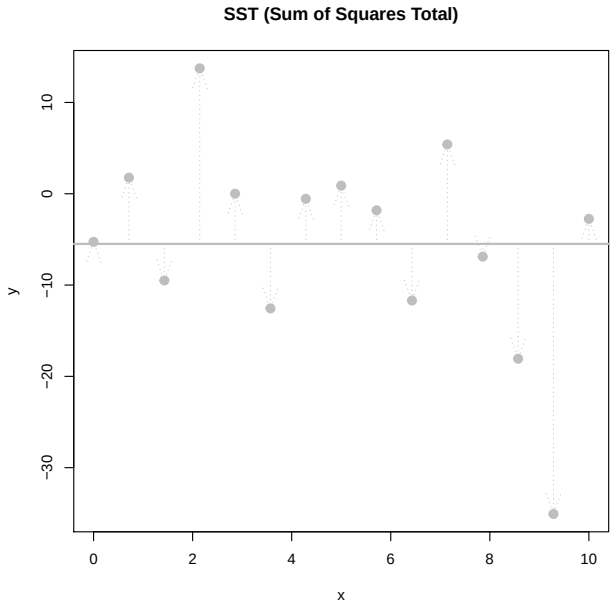
The following plots visually demonstrate the three "sums of squares" for a simulated dataset which has $R^2 = 0.92$ which is a somewhat high value. Notice in the final plot, that the orange arrows account for a larger proportion of the total arrow.

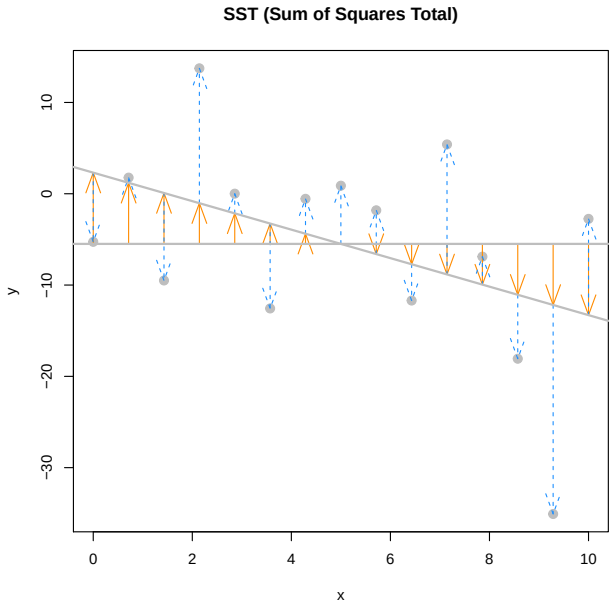


The next plots again visually demonstrate the three "sums of squares," this time for a simulated dataset which has $R^2 = 0.19$. Notice in the final plot, that now the blue arrows account for a larger proportion of the total arrow.









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
```

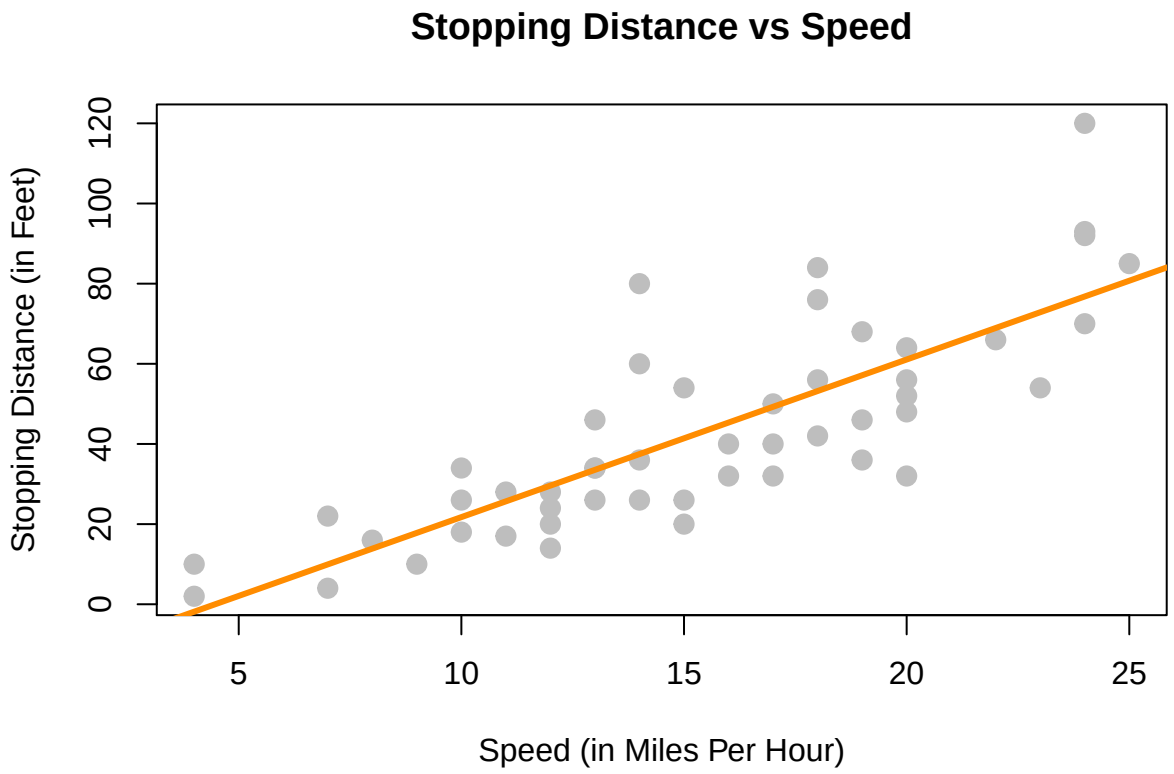
We see that it first tells us the formula we input into `R`, that is `lm(formula = dist ~ speed, data = cars)`. We also see the coefficients of the model. We can check that these are what we had calculated previously. (Minus some rounding that `R` is doing when displaying the results. They are stored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

```
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  = 2,
     col  = "grey")
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"
```

We 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 1.49e-12 ***
## ---
## 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: 1.49e-12
```

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 displayed above as a result of the `summary()` command, which `R` labeled the "Residual Standard Error."

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

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

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

$$\hat{y} = -17.58 + 3.93 \times 8 = 13.88$$
$$\hat{y} = -17.58 + 3.93 \times 21 = 65$$
$$\hat{y} = -17.58 + 3.93 \times 50 = 179.04$$

Or we could calculate the fitted value for each of the original data points. We can simply supply the original data frame, `cars`, since it contains a variables called `speed` which has the values we would like to predict at.

```
predict(stop_dist_model, newdata = cars)
```

```
##         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
```

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

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
```

Maximum Likelihood Estimation (MLE) Approach

Recall the model,

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

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

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

$$\mathrm{E}[Y_i \mid X_i = x_i] = \beta_0 + \beta_1 x_i$$

and

$$\mathrm{Var}[Y_i \mid X_i = x_i] = \sigma^2.$$

Additionally, the Y_i follow a normal distribution conditioned on the x_i.

$$Y_i \mid X_i \sim N(\beta_0 + \beta_1 x_i, \sigma^2)$$

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.

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

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

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

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.

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

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.

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 it. Then, we apply a method to the data, in this case least squares. Now, since we know the true model, we can assess how well it did.

For this example, we will simulate $n = 21$ observations from the model

$$Y = 5 - 2x + \epsilon.$$

That is $\beta_0 = 5$, $\beta_1 = -2$, and let $\epsilon \sim N(\mu = 0, \sigma^2 = 9)$. Or, even more succinctly we could write

$$Y \mid X \sim N(\mu = 5 - 2x, \sigma^2 = 9).$$

We first set the true parameters of the model to be simulated.

```
num_obs = 21
beta_0  = 5
beta_1  = -2
sigma   = 3
```

Next, we obtain simulated values of ϵ_i after setting a seed for reproducibility.

```
set.seed(1)
epsilon = rnorm(n = num_obs, mean = 0, sd = sigma)
```

Now, since the x_i values in SLR are considered fixed and known, we simply specify 21 values. Another common practice is to generate them from a uniform distribution, and then use them for the remainder of the analysis.

```
x_vals = seq(from = 0, to = 10, length.out = num_obs)
# set.seed(1)
# x_vals = runif(num_obs, 0, 10)
```

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

```
y_vals = beta_0 + beta_1 * x_vals + epsilon
```

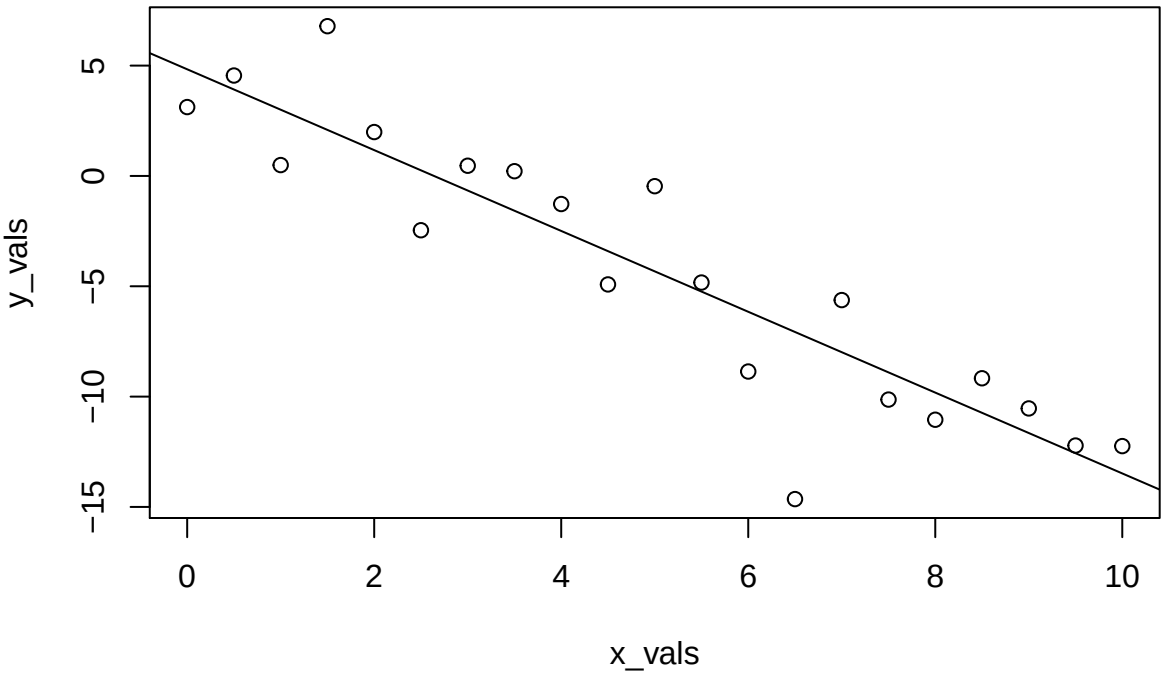
The data, (x_i, y_i), represent a possible sample from the true distribution. Now to check how well the method of least squares works, we use `lm()` to fit the model to our simulated data, then take a look at the estimated coefficients.

```
sim_fit = lm(y_vals ~ x_vals)
coef(sim_fit)
```

```
## (Intercept)      x_vals
##    4.832639   -1.831401
```

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

```
plot(y_vals ~ x_vals)
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. Notice that we provide `x` to the function, instead of generating `x` inside the function. In the SLR model, the x_i are considered known values. That is, they are not random, so we do not assume a distribution for the x_i. Because of this, we will repeatedly use the same `x` values across all simulations.

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

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

```
set.seed(1)
sim_data = sim_slr(x = x_vals, beta_0 = 5, beta_1 = -2, sigma = 3)
```

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

```
head(sim_data)
```

```
##   predictor   response
## 1       0.0  3.1206386
## 2       0.5  4.5509300
## 3       1.0  0.4931142
```

```
## 4       1.5  6.7858424
## 5       2.0  1.9885233
## 6       2.5 -2.4614052
```

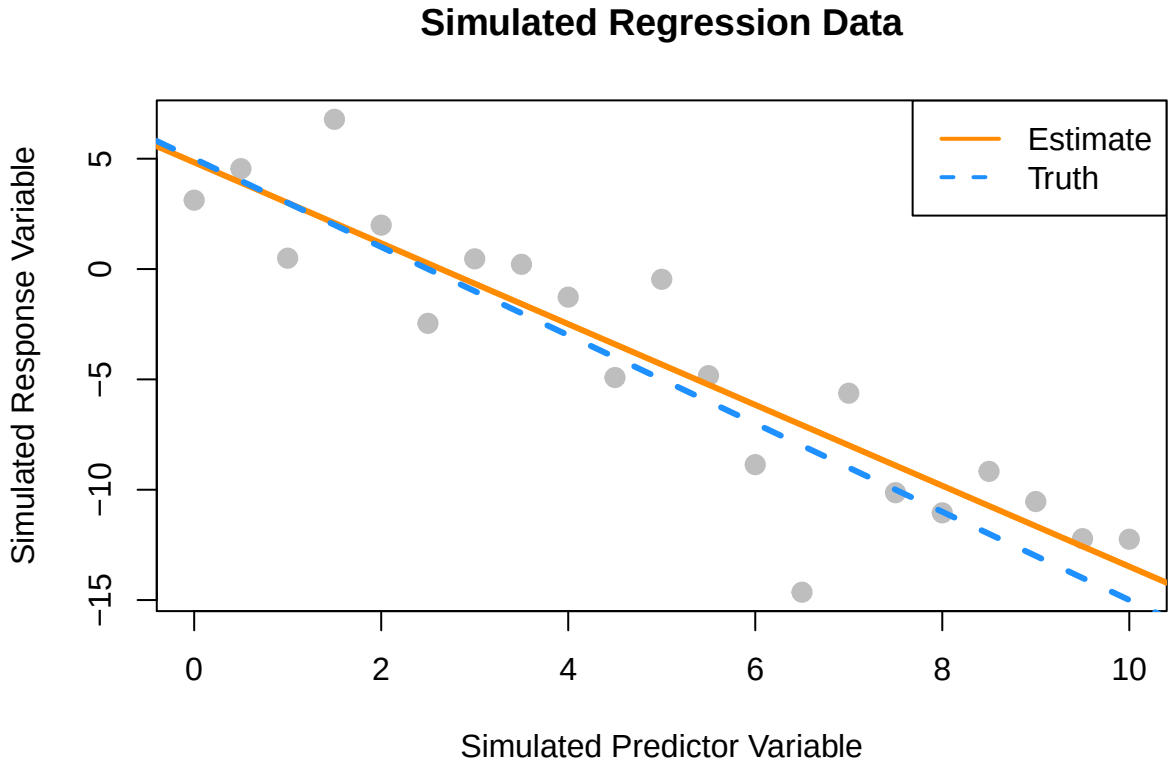
Now when we fit the model with `lm()` we can use a `data` argument, a very good practice.

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

```
## (Intercept)   predictor
##    4.832639   -1.831401
```

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  = "grey")
abline(sim_fit, lwd = 3, lty = 1, col = "darkorange")
abline(beta_0, beta_1, lwd = 3, lty = 2, col = "dodgerblue")
legend("topright", c("Estimate", "Truth"), lty = c(1, 2), lwd = 2,
       col = c("darkorange", "dodgerblue"))
```



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."

After reading this chapter you will be able to:

- Understand the distributions of regression estimates.
- Create interval estimates for regression parameters, mean response, and predictions.
- Test for significance of regression.

Last chapter we defined the simple linear regression model,

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

where $\epsilon_i \sim N(0, \sigma^2)$. We then used observations (x_i, y_i), for $i = 1, 2, \ldots n$, to find values of β_0 and β_1 which minimized

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

We called these values $\hat{\beta}_0$ and $\hat{\beta}_1$, which we found to be

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

We also estimated σ^2 using s_e^2. In other words, we found that s_e is an estimate of σ, where;

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

which we also called RSE, for "Residual Standard Error."

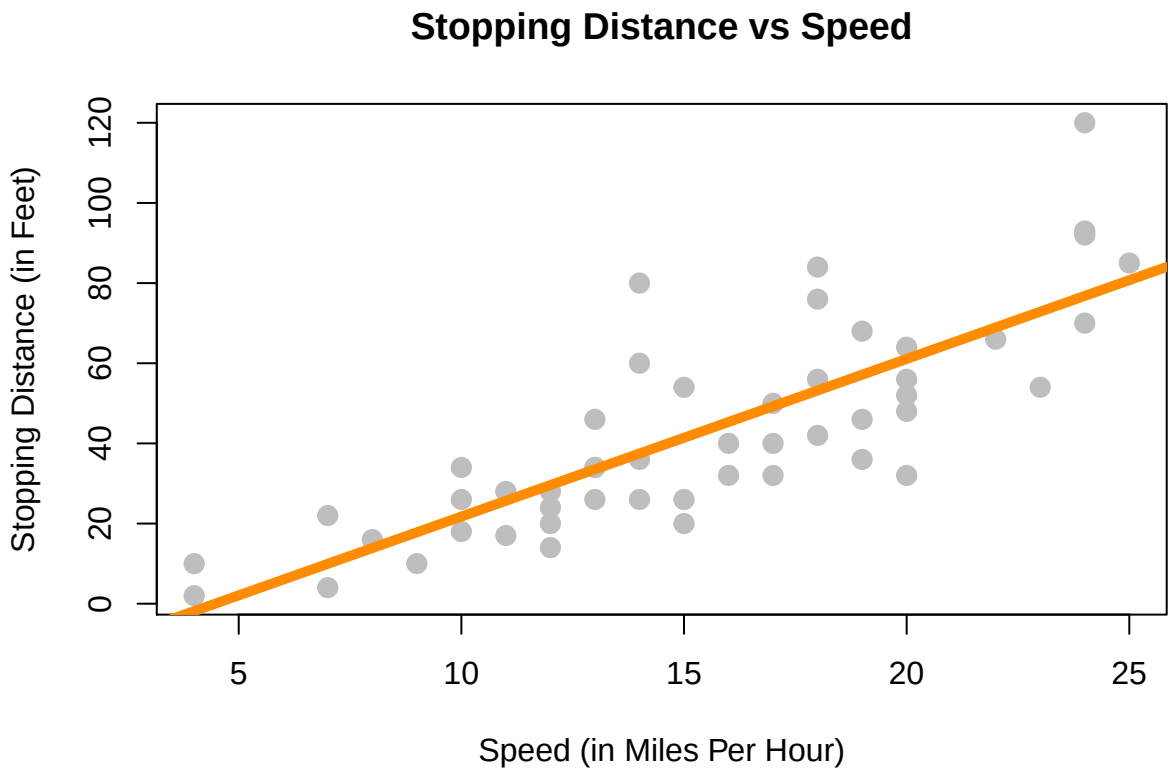
When applied to the `cars` data, we obtained the following results:

```
stop_dist_model = lm(dist ~ speed, data = cars)
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 1.49e-12 ***
## ---
## 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: 1.49e-12
```

Last chapter, we only discussed the `Estimate`, `Residual standard error`, and `Multiple R-squared` values. In this chapter, we will discuss all of the information under `Coefficients` as well as `F-statistic`.

```
plot(dist ~ speed, data = cars,
     xlab = "Speed (in Miles Per Hour)",
     ylab = "Stopping Distance (in Feet)",
     main = "Stopping Distance vs Speed",
     pch  = 20,
     cex  = 2,
     col  = "grey")
abline(stop_dist_model, lwd = 5, col = "darkorange")
```



To get started, we'll note that there is another equivalent expression for S_{xy} which we did not see last chapter,

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

This may be a surprising equivalence. (Maybe try to prove it.) However, it will be useful for illustrating concepts in this chapter.

Note that, $\hat{\beta}_1$ is a **sample statistic** when calculated with observed data as written above, as is $\hat{\beta}_0$.

However, in this chapter it will often be convenient to use both $\hat{\beta}_1$ and $\hat{\beta}_0$ as **random variables**, that is, we have not yet observed the values for each Y_i. When this is the case, we will use a slightly different notation, substituting in capital Y_i for lower case y_i.

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})Y_i}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$
$$\hat{\beta}_0 = \bar{Y} - \hat{\beta}_1\bar{x}$$

Last chapter we argued that these estimates of unknown model parameters β_0 and β_1 were good because we obtained them by minimizing errors. We will now discuss the Gauss–Markov theorem which takes this idea further, showing that these estimates are actually the "best" estimates, from a certain point of view.

Gauss–Markov Theorem

The **Gauss–Markov theorem** tells us that when estimating the parameters of the simple linear regression model β_0 and β_1, the $\hat{\beta}_0$ and $\hat{\beta}_1$ which we derived are the **best linear unbiased estimates**, or *BLUE* for short. (The actual conditions for the Gauss–Markov theorem are more relaxed than the SLR model.)

We will now discuss *linear*, *unbiased*, and *best* as is relates to these estimates.

Linear Recall, in the SLR setup that the x_i values are considered fixed and known quantities. Then a **linear** estimate is one which can be written as a linear combination of the Y_i. In the case of $\hat{\beta}_1$ we see

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})Y_i}{\sum_{i=1}^{n}(x_i - \bar{x})^2} = \sum_{i=1}^{n} k_i Y_i = k_1 Y_1 + k_2 Y_2 + \cdots k_n Y_n$$

where $k_i = \dfrac{(x_i - \bar{x})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$.

In a similar fashion, we could show that $\hat{\beta}_0$ can be written as a linear combination of the Y_i. Thus both $\hat{\beta}_0$ and $\hat{\beta}_1$ are linear estimators.

Unbiased Now that we know our estimates are *linear*, how good are these estimates? One measure of the "goodness" of an estimate is its **bias**. Specifically, we prefer estimates that are **unbiased**, meaning their expected value is the parameter being estimated.

In the case of the regression estimates, we have,

$$\mathrm{E}[\hat{\beta}_0] = \beta_0$$
$$\mathrm{E}[\hat{\beta}_1] = \beta_1.$$

This tells us that, when the conditions of the SLR model are met, on average our estimates will be correct. However, as we saw last chapter when simulating from the SLR model, that does not mean that each individual estimate will be correct. Only that, if we repeated the process an infinite number of times, on average the estimate would be correct.

Best Now, if we restrict ourselves to both *linear* and *unbiased* estimates, how do we define the *best* estimate? The estimate with the **minimum variance**.

First note that it is very easy to create an estimate for β_1 that has very low variance, but is not unbiased. For example, define:

$$\hat{\theta}_{BAD} = 5.$$

Then, since $\hat{\theta}_{BAD}$ is a constant value,

$$\mathrm{Var}[\hat{\theta}_{BAD}] = 0.$$

However since,

$$\mathrm{E}[\hat{\theta}_{BAD}] = 5$$

we say that $\hat{\theta}_{BAD}$ is a biased estimator unless $\beta_1 = 5$, which we would not know ahead of time. For this reason, it is a terrible estimate (unless by chance $\beta_1 = 5$) even though it has the smallest possible variance. This is part of the reason we restrict ourselves to *unbiased* estimates. What good is an estimate, if it estimates the wrong quantity?

So now, the natural question is, what are the variances of $\hat{\beta}_0$ and $\hat{\beta}_1$? They are,

$$\text{Var}[\hat{\beta}_0] = \sigma^2 \left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}} \right)$$
$$\text{Var}[\hat{\beta}_1] = \frac{\sigma^2}{S_{xx}}.$$

These quantify the variability of the estimates due to random chance during sampling. Are these "the best"? Are these variances as small as we can possibility get? You'll just have to take our word for it that they are because showing that this is true is beyond the scope of this course.

Sampling Distributions

Now that we have "redefined" the estimates for $\hat{\beta}_0$ and $\hat{\beta}_1$ as random variables, we can discuss their **sampling distribution**, which is the distribution when a statistic is considered a random variable.

Since both $\hat{\beta}_0$ and $\hat{\beta}_1$ are a linear combination of the Y_i and each Y_i is normally distributed, then both $\hat{\beta}_0$ and $\hat{\beta}_1$ also follow a normal distribution.

Then, putting all of the above together, we arrive at the distributions of $\hat{\beta}_0$ and $\hat{\beta}_1$.

For $\hat{\beta}_1$ we say,

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})Y_i}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \sim N\left(\beta_1, \ \frac{\sigma^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}\right).$$

Or more succinctly,

$$\hat{\beta}_1 \sim N\left(\beta_1, \frac{\sigma^2}{S_{xx}}\right).$$

And for $\hat{\beta}_0$,

$$\hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{x} \sim N\left(\beta_0, \ \frac{\sigma^2 \sum_{i=1}^{n} x_i^2}{n \sum_{i=1}^{n}(x_i - \bar{x})^2}\right).$$

Or more succinctly,

$$\hat{\beta}_0 \sim N\left(\beta_0, \sigma^2 \left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}} \right)\right)$$

At this point we have neglected to prove a number of these results. Instead of working through the tedious derivations of these sampling distributions, we will instead justify these results to ourselves using simulation.

A note to current readers: These derivations and proofs may be added to an appendix at a later time. You can also find these results in nearly any standard linear regression textbook. At UIUC, these results will likely be presented in both STAT 424 and STAT 425. However, since you will not be asked to perform derivations of this type in this course, they are for now omitted.

Simulating Sampling Distributions

To verify the above results, we will simulate samples of size $n = 100$ from the model

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

where $\epsilon_i \sim N(0, \sigma^2)$. In this case, the parameters are known to be:

- $\beta_0 = 3$
- $\beta_1 = 6$
- $\sigma^2 = 4$

Then, based on the above, we should find that

$$\hat{\beta}_1 \sim N\left(\beta_1, \frac{\sigma^2}{S_{xx}}\right)$$

and

$$\hat{\beta}_0 \sim N\left(\beta_0, \sigma^2\left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}\right)\right).$$

First we need to decide ahead of time what our x values will be for this simulation, since the x values in SLR are also considered known quantities. The choice of x values is arbitrary. Here we also set a seed for randomization, and calculate S_{xx} which we will need going forward.

```
set.seed(42)
sample_size = 100 # this is n
x = seq(-1, 1, length = sample_size)
Sxx = sum((x - mean(x)) ^ 2)
```

We also fix our parameter values.

```
beta_0 = 3
beta_1 = 6
sigma  = 2
```

With this information, we know the sampling distributions should be:

```
(var_beta_1_hat = sigma ^ 2 / Sxx)
```

```
## [1] 0.1176238
```

```
(var_beta_0_hat = sigma ^ 2 * (1 / sample_size + mean(x) ^ 2 / Sxx))
```

```
## [1] 0.04
```

$$\hat{\beta}_1 \sim N(6, 0.1176238)$$

and

$$\hat{\beta}_0 \sim N(3, 0.04).$$

That is,

$$\text{E}[\hat{\beta}_1] = 6$$
$$\text{Var}[\hat{\beta}_1] = 0.1176238$$

and

$$\text{E}[\hat{\beta}_0] = 3$$
$$\text{Var}[\hat{\beta}_0] = 0.04.$$

We now simulate data from this model 10,000 times. Note this may not be the most `R` way of doing the simulation. We perform the simulation in this manner in an attempt at clarity. For example, we could have used the `sim_slr()` function from the previous chapter. We also simply store variables in the global environment instead of creating a data frame for each new simulated dataset.

```
num_samples = 10000
beta_0_hats = rep(0, num_samples)
beta_1_hats = rep(0, num_samples)

for (i in 1:num_samples) {
  eps = rnorm(sample_size, mean = 0, sd = sigma)
  y   = beta_0 + beta_1 * x + eps

  sim_model = lm(y ~ x)

  beta_0_hats[i] = coef(sim_model)[1]
  beta_1_hats[i] = coef(sim_model)[2]
}
```

Each time we simulated the data, we obtained values of the estimated coefficiets. The variables `beta_0_hats` and `beta_1_hats` now store 10,000 simulated values of $\hat{\beta}_0$ and $\hat{\beta}_1$ respectively.

We first verify the distribution of $\hat{\beta}_1$.

```
mean(beta_1_hats) # empirical mean
```

```
## [1] 6.001998
```

```
beta_1            # true mean
```

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

```
var(beta_1_hats)  # empirical variance
```

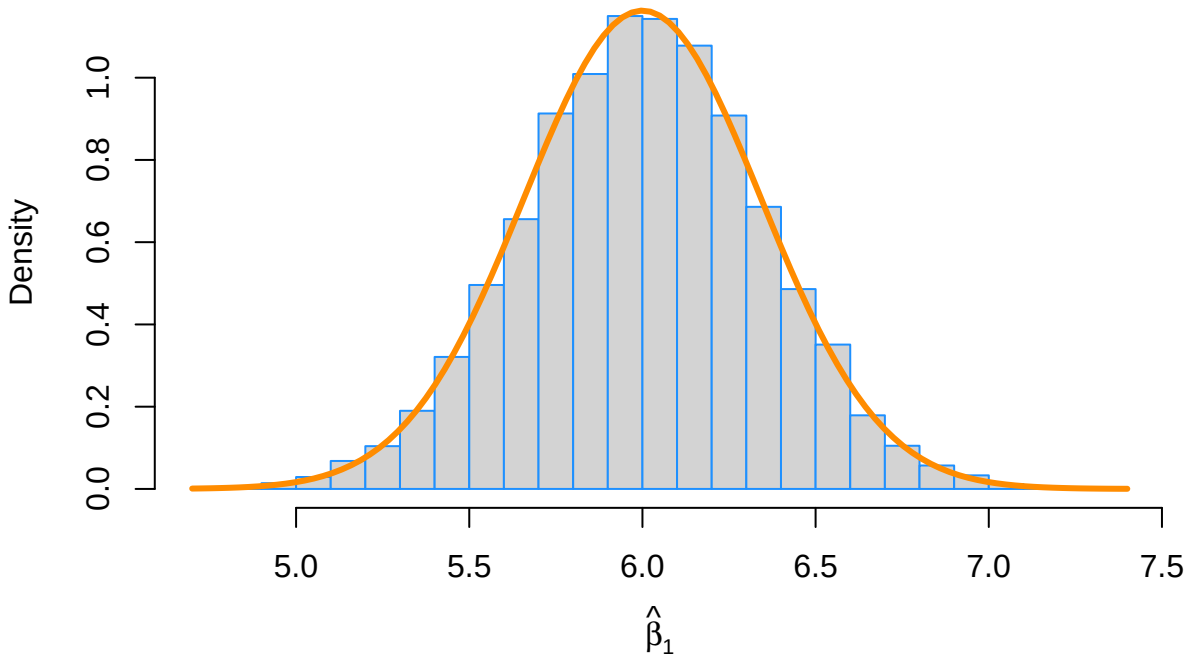
```
## [1] 0.11899
```

```
var_beta_1_hat    # true variance
```

```
## [1] 0.1176238
```

We see that the empirical and true means and variances are *very* similar. We also verify that the empirical distribution is normal. To do so, we plot a histogram of the `beta_1_hats`, and add the curve for the true distribution of $\hat{\beta}_1$. We use `prob = TRUE` to put the histogram on the same scale as the normal curve.

```
# note need to use prob = TRUE
hist(beta_1_hats, prob = TRUE, breaks = 20,
     xlab = expression(hat(beta)[1]), main = "", border = "dodgerblue")
curve(dnorm(x, mean = beta_1, sd = sqrt(var_beta_1_hat)),
      col = "darkorange", add = TRUE, lwd = 3)
```



We then repeat the process for $\hat{\beta}_0$.

```
mean(beta_0_hats) # empirical mean
```

```
## [1] 3.001147
```

```
beta_0            # true mean
```

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

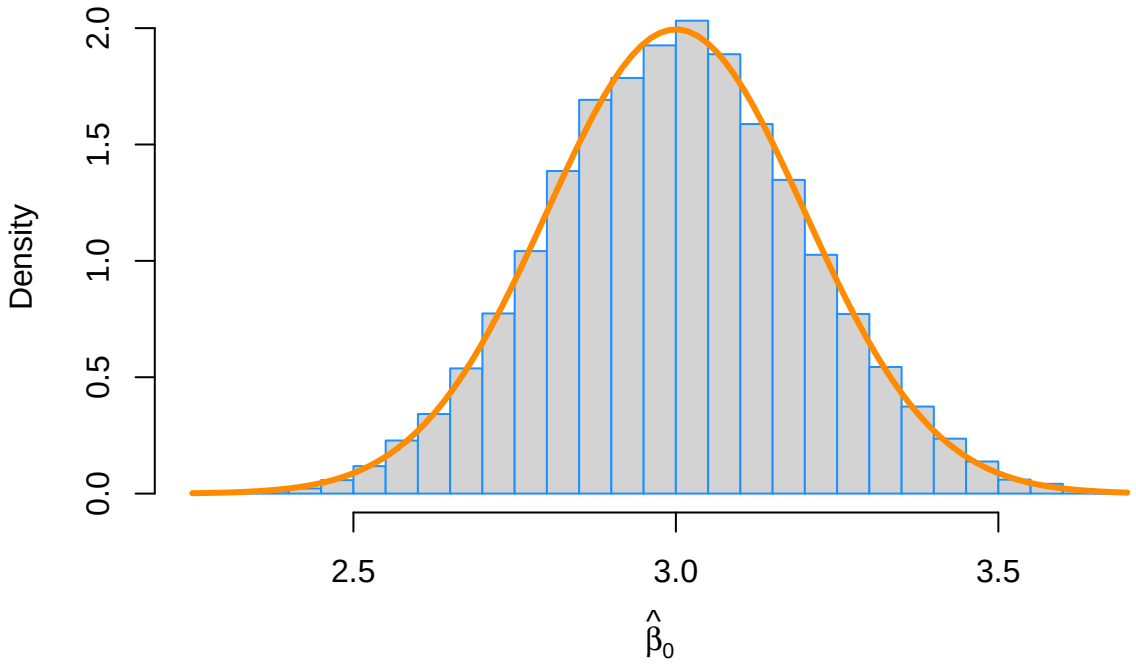
```
var(beta_0_hats)  # empirical variance
```

```
## [1] 0.04017924
```

```
var_beta_0_hat    # true variance
```

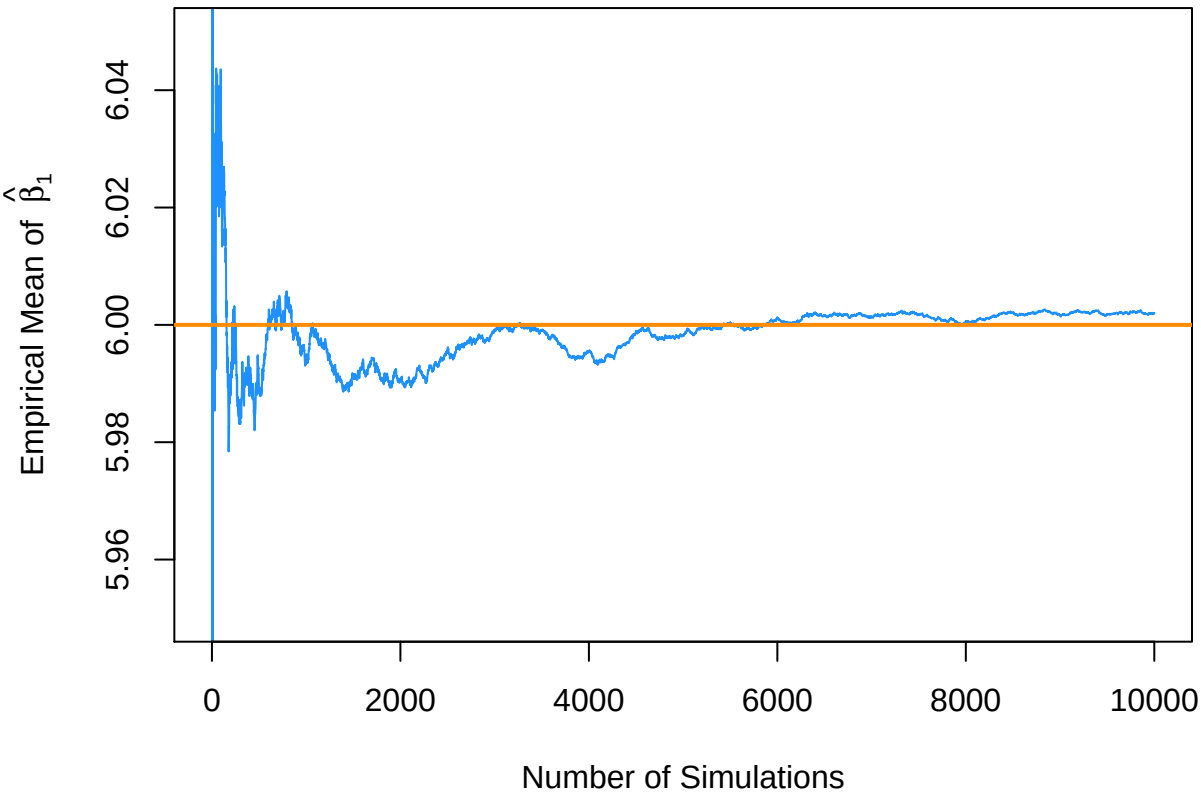
```
## [1] 0.04
```

```
hist(beta_0_hats, prob = TRUE, breaks = 25,
     xlab = expression(hat(beta)[0]), main = "", border = "dodgerblue")
curve(dnorm(x, mean = beta_0, sd = sqrt(var_beta_0_hat)),
      col = "darkorange", add = TRUE, lwd = 3)
```

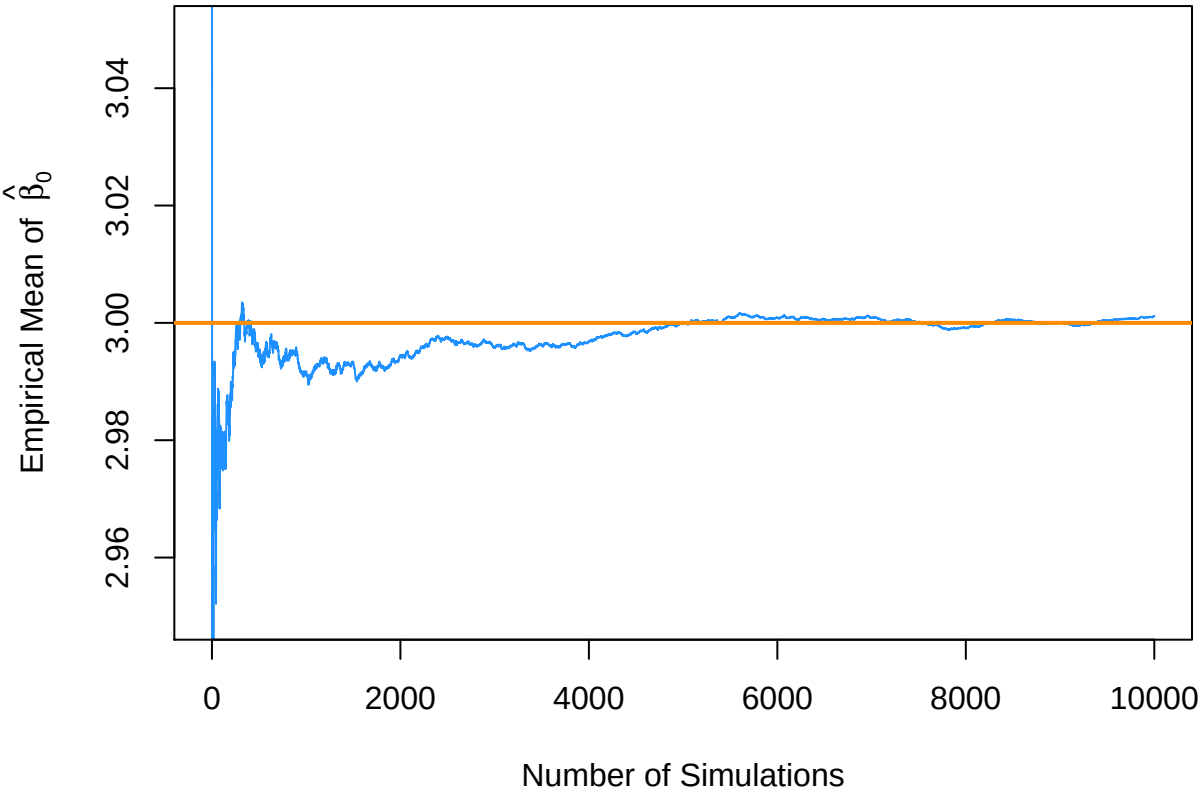


In this simulation study, we have only simulated a finite number of samples. To truly verify the distributional results, we would need to observe an infinite number of samples. However, the following plot should make it clear that if we continued simulating, the empirical results would get closer and closer to what we should expect.

```
par(mar = c(5, 5, 1, 1)) # adjusted plot margins, otherwise the "hat" does not display
plot(cumsum(beta_1_hats) / (1:length(beta_1_hats)), type = "l", ylim = c(5.95, 6.05),
     xlab = "Number of Simulations",
     ylab = expression("Empirical Mean of " ~ hat(beta)[1]),
     col  = "dodgerblue")
abline(h = 6, col = "darkorange", lwd = 2)
```




```
par(mar = c(5, 5, 1, 1)) # adjusted plot margins, otherwise the "hat" does not display
plot(cumsum(beta_0_hats) / (1:length(beta_0_hats)), type = "l", ylim = c(2.95, 3.05),
     xlab = "Number of Simulations",
     ylab = expression("Empirical Mean of " ~ hat(beta)[0]),
     col  = "dodgerblue")
abline(h = 3, col = "darkorange", lwd = 2)
```



Standard Errors

So now we believe the two distributional results,

$$\hat{\beta}_0 \sim N\left(\beta_0, \sigma^2\left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}\right)\right)$$
$$\hat{\beta}_1 \sim N\left(\beta_1, \frac{\sigma^2}{S_{xx}}\right).$$

Then by standardizing these results we find that

$$\frac{\hat{\beta}_0 - \beta_0}{\text{SD}[\hat{\beta}_0]} \sim N(0,1)$$

and

$$\frac{\hat{\beta}_1 - \beta_1}{\text{SD}[\hat{\beta}_1]} \sim N(0,1)$$

where

$$\text{SD}[\hat{\beta}_0] = \sigma\sqrt{\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}}$$

and

$$\text{SD}[\hat{\beta}_1] = \frac{\sigma}{\sqrt{S_{xx}}}.$$

Since we don't know σ in practice, we will have to estimate it using s_e, which we plug into our existing expression for the standard deviations of our estimates.

These two new expressions are called **standard errors** which are the *estimated* standard deviations of the sampling distributions.

$$\text{SE}[\hat{\beta}_0] = s_e\sqrt{\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}}$$

$$\text{SE}[\hat{\beta}_1] = \frac{s_e}{\sqrt{S_{xx}}}$$

Now if we divide by the standard error, instead of the standard deviation, we obtain the following results which will allow us to make confidence intervals and perform hypothesis testing.

$$\frac{\hat{\beta}_0 - \beta_0}{\text{SE}[\hat{\beta}_0]} \sim t_{n-2}$$

$$\frac{\hat{\beta}_1 - \beta_1}{\text{SE}[\hat{\beta}_1]} \sim t_{n-2}$$

To see this, first note that,

$$\frac{\text{RSS}}{\sigma^2} = \frac{(n-2)s_e^2}{\sigma^2} \sim \chi^2_{n-2}.$$

Also recall that a random variable T defined as,

$$T = \frac{Z}{\sqrt{\frac{\chi^2_d}{d}}}$$

follows a t distribution with d degrees of freedom, where χ^2_d is a χ^2 random variable with d degrees of freedom.

We write,

$$T \sim t_d$$

to say that the random variable T follows a t distribution with d degrees of freedom.

Then we use the classic trick of "multiply by 1" and some rearranging to arrive at

$$
\begin{aligned}
\frac{\hat{\beta}_1 - \beta_1}{\text{SE}[\hat{\beta}_1]} &= \frac{\hat{\beta}_1 - \beta_1}{s_e / \sqrt{S_{xx}}} \\
&= \frac{\hat{\beta}_1 - \beta_1}{s_e / \sqrt{S_{xx}}} \cdot \frac{\sigma / \sqrt{S_{xx}}}{\sigma / \sqrt{S_{xx}}} \\
&= \frac{\hat{\beta}_1 - \beta_1}{\sigma / \sqrt{S_{xx}}} \cdot \frac{\sigma / \sqrt{S_{xx}}}{s_e / \sqrt{S_{xx}}} \\
&= \frac{\hat{\beta}_1 - \beta_1}{\sigma / \sqrt{S_{xx}}} \Bigg/ \sqrt{\frac{s_e^2}{\sigma^2}} \\
&= \frac{\hat{\beta}_1 - \beta_1}{\text{SD}[\hat{\beta}_1]} \Bigg/ \sqrt{\frac{\frac{(n-2)s_e^2}{\sigma^2}}{n-2}} \sim \frac{Z}{\sqrt{\frac{\chi^2_{n-2}}{n-2}}} \sim t_{n-2}
\end{aligned}
$$

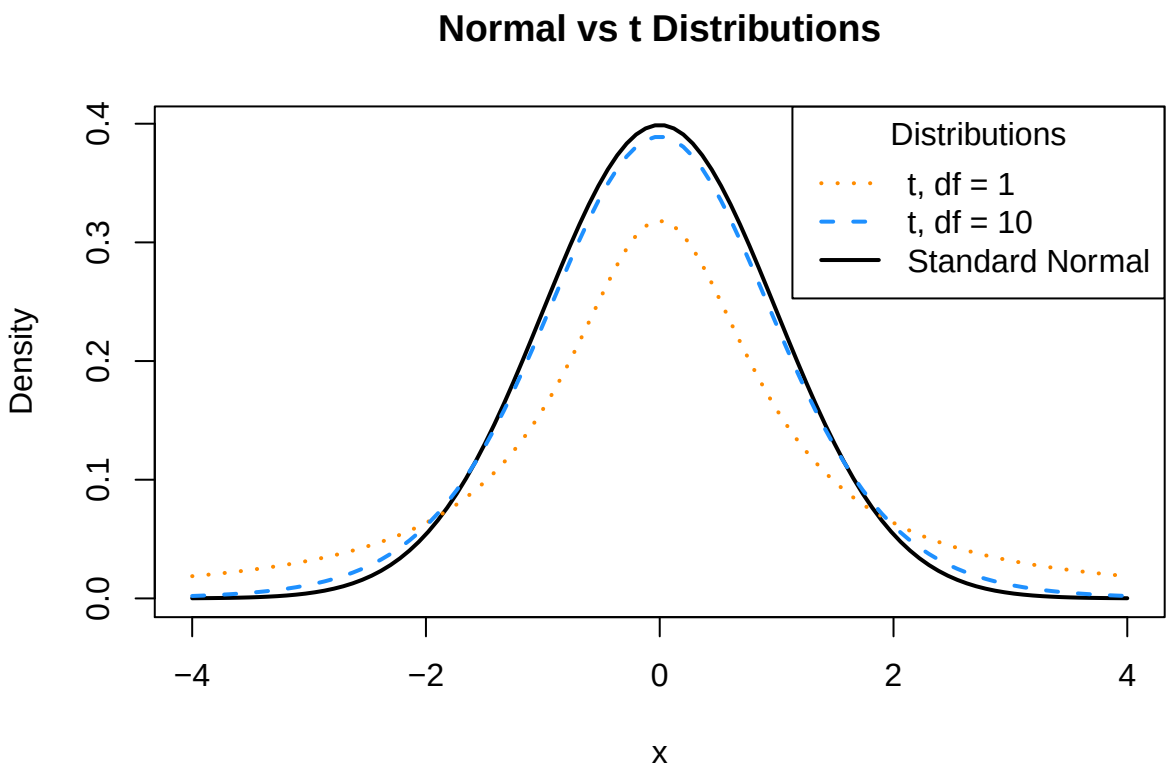
where $Z \sim N(0, 1)$.

Recall that a t distribution is similar to a standard normal, but with heavier tails. As the degrees of freedom increases, the t distribution becomes more and more like a standard normal. Below we plot a standard normal distribution as well as two examples of a t distribution with different degrees of freedom. Notice how the t distribution with the larger degrees of freedom is more similar to the standard normal curve.

```
# define grid of x values
x = seq(-4, 4, length = 100)

# plot curve for standard normal
plot(x, dnorm(x), type = "l", lty = 1, lwd = 2,
     xlab = "x", ylab = "Density", main = "Normal vs t Distributions")
# add curves for t distributions
lines(x, dt(x, df = 1), lty = 3, lwd = 2, col = "darkorange")
lines(x, dt(x, df = 10), lty = 2, lwd = 2, col = "dodgerblue")

# add legend
legend("topright", title = "Distributions",
       legend = c("t, df = 1", "t, df = 10", "Standard Normal"),
       lwd = 2, lty = c(3, 2, 1), col = c("darkorange", "dodgerblue", "black"))
```



Confidence Intervals for Slope and Intercept

Recall that confidence intervals for means often take the form:

$$\text{EST} \pm \text{CRIT} \cdot \text{SE}$$

or

$$\text{EST} \pm \text{MARGIN}$$

where EST is an estimate for the parameter of interest, SE is the standard error of the estimate, and $\text{MARGIN} = \text{CRIT} \cdot \text{SE}$.

Then, for β_0 and β_1 we can create confidence intervals using

$$\hat{\beta}_0 \pm t_{\alpha/2,n-2} \cdot \text{SE}[\hat{\beta}_0] \qquad \hat{\beta}_0 \pm t_{\alpha/2,n-2} \cdot s_e \sqrt{\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}}$$

and

$$\hat{\beta}_1 \pm t_{\alpha/2,n-2} \cdot \text{SE}[\hat{\beta}_1] \qquad \hat{\beta}_1 \pm t_{\alpha/2,n-2} \cdot \frac{s_e}{\sqrt{S_{xx}}}$$

where $t_{\alpha/2,n-2}$ is the critical value such that $P(t_{n-2} > t_{\alpha/2,n-2}) = \alpha/2$.

Hypothesis Tests

> "We may speak of this hypothesis as the 'null hypothesis', and it should be noted that the null hypothesis is never proved or established, but is possibly disproved, in the course of experimentation."
>
> — **Ronald Aylmer Fisher**

Recall that a test statistic (TS) for testing means often take the form:

$$\text{TS} = \frac{\text{EST} - \text{HYP}}{\text{SE}}$$

where EST is an estimate for the parameter of interest, HYP is a hypothesized value of the parameter, and SE is the standard error of the estimate.

So, to test

$$H_0 : \beta_0 = \beta_{00} \quad \text{vs} \quad H_1 : \beta_0 \neq \beta_{00}$$

we use the test statistic

$$t = \frac{\hat{\beta}_0 - \beta_{00}}{\text{SE}[\hat{\beta}_0]} = \frac{\hat{\beta}_0 - \beta_{00}}{s_e\sqrt{\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}}}$$

which, under the null hypothesis, follows a t distribution with $n - 2$ degrees of freedom. We use β_{00} to denote the hypothesized value of β_0.

Similarly, to test

$$H_0 : \beta_1 = \beta_{10} \quad \text{vs} \quad H_1 : \beta_1 \neq \beta_{10}$$

we use the test statistic

$$t = \frac{\hat{\beta}_1 - \beta_{10}}{\text{SE}[\hat{\beta}_1]} = \frac{\hat{\beta}_1 - \beta_{10}}{s_e/\sqrt{S_{xx}}}$$

which again, under the null hypothesis, follows a t distribution with $n - 2$ degrees of freedom. We now use β_{10} to denote the hypothesized value of β_1.

`cars` Example

We now return to the `cars` example from last chapter to illustrate these concepts. We first fit the model using `lm()` then use `summary()` to view the results in greater detail.

```
stop_dist_model = lm(dist ~ speed, data = cars)
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 1.49e-12 ***
## ---
## 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: 1.49e-12
```

Tests in `R`

We will now discuss the results displayed called `Coefficients`. First recall that we can extract this information directly.

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

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

```
summary(stop_dist_model)$coefficients
```

```
##               Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) -17.579095  6.7584402 -2.601058 1.231882e-02
## speed         3.932409  0.4155128  9.463990 1.489836e-12
```

The `names()` function tells us what information is available, and then we use the `$` operator and `coefficients` to extract the information we are interested in. Two values here should be immediately familiar.

$$\hat{\beta}_0 = -17.5790949$$

and

$$\hat{\beta}_1 = 3.9324088$$

which are our estimates for the model parameters β_0 and β_1.

Let's now focus on the second row of output, which is relevant to β_1.

```
summary(stop_dist_model)$coefficients[2,]
```

```
##     Estimate   Std. Error      t value     Pr(>|t|)
## 3.932409e+00 4.155128e-01 9.463990e+00 1.489836e-12
```

Again, the first value, `Estimate` is

$$\hat{\beta}_1 = 3.9324088.$$

The second value, `Std. Error`, is the standard error of $\hat{\beta}_1$,

$$\text{SE}[\hat{\beta}_1] = \frac{s_e}{\sqrt{S_{xx}}} = 0.4155128.$$

The third value, `t value`, is the value of the test statistic for testing $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$,

$$t = \frac{\hat{\beta}_1 - 0}{\text{SE}[\hat{\beta}_1]} = \frac{\hat{\beta}_1 - 0}{s_e/\sqrt{S_{xx}}} = 9.46399.$$

Lastly, `Pr(>|t|)`, gives us the p-value of that test.

$$\text{p-value} = 1.4898365 \times 10^{-12}$$

Note here, we are specifically testing whether or not $\beta_1 = 0$.

The first row of output reports the same values, but for β_0.

```
summary(stop_dist_model)$coefficients[1,]
```

```
##     Estimate   Std. Error      t value     Pr(>|t|)
## -17.57909489   6.75844017  -2.60105800   0.01231882
```

In summary, the following code stores the information of `summary(stop_dist_model)$coefficients` in a new variable `stop_dist_model_test_info`, then extracts each element into a new variable which describes the information it contains.

```
stop_dist_model_test_info = summary(stop_dist_model)$coefficients

beta_0_hat      = stop_dist_model_test_info[1, 1] # Estimate
beta_0_hat_se   = stop_dist_model_test_info[1, 2] # Std. Error
beta_0_hat_t    = stop_dist_model_test_info[1, 3] # t value
beta_0_hat_pval = stop_dist_model_test_info[1, 4] # Pr(>|t|)

beta_1_hat      = stop_dist_model_test_info[2, 1] # Estimate
beta_1_hat_se   = stop_dist_model_test_info[2, 2] # Std. Error
beta_1_hat_t    = stop_dist_model_test_info[2, 3] # t value
beta_1_hat_pval = stop_dist_model_test_info[2, 4] # Pr(>|t|)
```

We can then verify some equivalent expressions: the t test statistic for $\hat{\beta}_1$ and the two-sided p-value associated with that test statistic.

```
(beta_1_hat - 0) / beta_1_hat_se
```

```
## [1] 9.46399
```

```
beta_1_hat_t
```

```
## [1] 9.46399
```

```
2 * pt(abs(beta_1_hat_t), df = length(resid(stop_dist_model)) - 2, lower.tail = FALSE)
```

```
## [1] 1.489836e-12
```

```
beta_1_hat_pval
```

```
## [1] 1.489836e-12
```

Significance of Regression, t-Test

We pause to discuss the **significance of regression** test. First, note that based on the above distributional results, we could test β_0 and β_1 against any particular value, and perform both one and two-sided tests.

However, one very specific test,

$$H_0 : \beta_1 = 0 \quad \text{vs} \quad H_1 : \beta_1 \neq 0$$

is used most often. Let's think about this test in terms of the simple linear regression model,

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

If we assume the null hypothesis is true, then $\beta_1 = 0$ and we have the model,

$$Y_i = \beta_0 + \epsilon_i.$$

In this model, the response does **not** depend on the predictor. So then we could think of this test in the following way,

- Under H_0 there is not a significant linear relationship between x and y.
- Under H_1 there is a significant **linear** relationship between x and y.

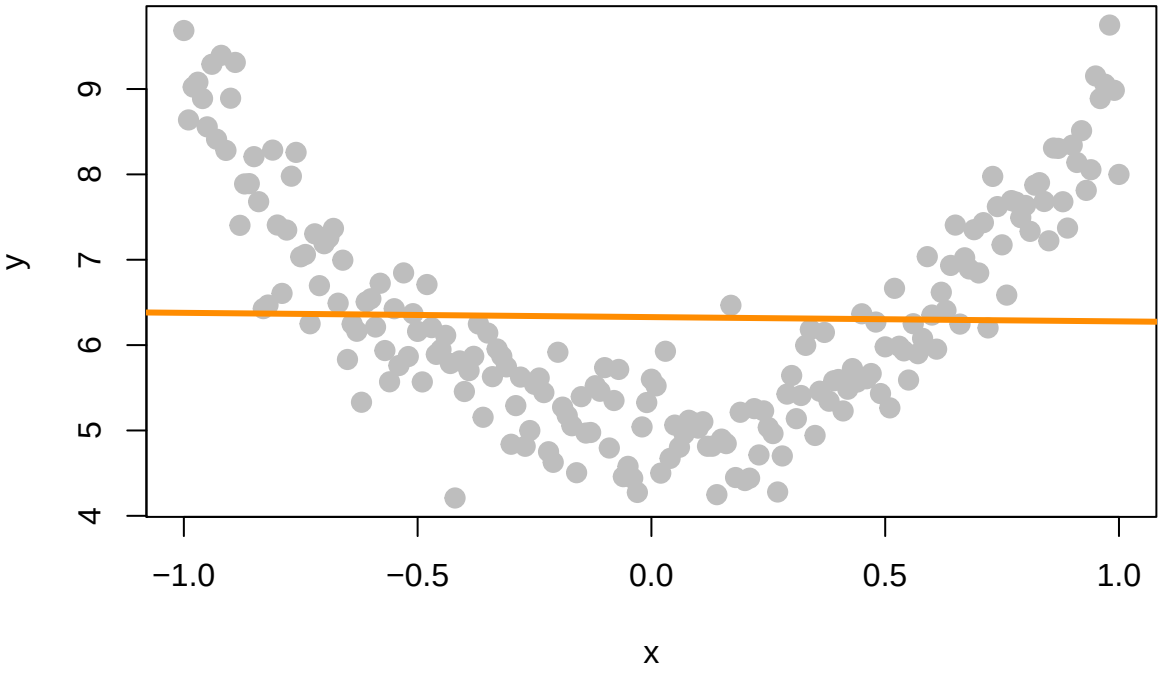
For the `cars` example,

- Under H_0 there is not a significant linear relationship between speed and stopping distance.
- Under H_1 there is a significant **linear** relationship between speed and stopping distance.

Again, that test is seen in the output from `summary()`,

$$\text{p-value} = 1.4898365 \times 10^{-12}.$$

With this extremely low p-value, we would reject the null hypothesis at any reasonable α level, say for example $\alpha = 0.01$. So we say there is a significant **linear** relationship between speed and stopping distance. Notice that we emphasize **linear**.



In this plot of simulated data, we see a clear relationship between x and y, however it is not a linear relationship. If we fit a line to this data, it is very flat. The resulting test for $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$ gives a large p-value, in this case 0.7564548, so we would fail to reject and say that there is no significant linear relationship between x and y. We will see later how to fit a curve to this data using a "linear" model, but for now, realize that testing $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$ can only detect straight line relationships.

Confidence Intervals in R

Using R we can very easily obtain the confidence intervals for β_0 and β_1.

```
confint(stop_dist_model, level = 0.99)
```

```
##                 0.5 %    99.5 %
## (Intercept) -35.706610 0.5484205
## speed         2.817919 5.0468988
```

This automatically calculates 99% confidence intervals for both β_0 and β_1, the first row for β_0, the second row for β_1.

For the `cars` example when interpreting these intervals, we say, we are 99% confident that for an increase in speed of 1 mile per hour, the average increase in stopping distance is between 2.8179187 and 5.0468988 feet, which is the interval for β_1.

Note that this 99% confidence interval does **not** contain the hypothesized value of 0. Since it does not contain 0, it is equivalent to rejecting the test of $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$ at $\alpha = 0.01$, which we had seen previously.

You should be somewhat suspicious of the confidence interval for β_0, as it covers negative values, which correspond to negative stopping distances. Technically the interpretation would be that we are 99% confident that the average stopping distance of a car traveling 0 miles per hour is between -35.7066103 and 0.5484205 feet, but we don't really believe that, since we are actually certain that it would be non-negative.

Note, we can extract specific values from this output a number of ways. This code is not run, and instead, you should check how it relates to the output of the code above.

```
confint(stop_dist_model, level = 0.99)[1,]
confint(stop_dist_model, level = 0.99)[1, 1]
confint(stop_dist_model, level = 0.99)[1, 2]
confint(stop_dist_model, parm = "(Intercept)", level = 0.99)
confint(stop_dist_model, level = 0.99)[2,]
confint(stop_dist_model, level = 0.99)[2, 1]
confint(stop_dist_model, level = 0.99)[2, 2]
confint(stop_dist_model, parm = "speed", level = 0.99)
```

We can also verify that calculations that R is performing for the β_1 interval.

```
# store estimate
beta_1_hat = coef(stop_dist_model)[2]

# store standard error
beta_1_hat_se = summary(stop_dist_model)$coefficients[2, 2]

# calculate critical value for two-sided 99% CI
crit = qt(0.995, df = length(resid(stop_dist_model)) - 2)

# est - margin, est + margin
c(beta_1_hat - crit * beta_1_hat_se, beta_1_hat + crit * beta_1_hat_se)
```

```
##    speed    speed
## 2.817919 5.046899
```

Confidence Interval for Mean Response

In addition to confidence intervals for β_0 and β_1, there are two other common interval estimates used with regression. The first is called a **confidence interval for the mean response**. Often, we would like an

interval estimate for the mean, $E[Y \mid X = x]$ for a particular value of x.

In this situation we use $\hat{y}(x)$ as our estimate of $E[Y \mid X = x]$. We modify our notation slightly to make it clear that the predicted value is a function of the x value.

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

Recall that,

$$\mathrm{E}[Y \mid X = x] = \beta_0 + \beta_1 x.$$

Thus, $\hat{y}(x)$ is a good estimate since it is unbiased:

$$\mathrm{E}[\hat{y}(x)] = \beta_0 + \beta_1 x.$$

We could then derive,

$$\mathrm{Var}[\hat{y}(x)] = \sigma^2 \left(\frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}} \right).$$

Like the other estimates we have seen, $\hat{y}(x)$ also follows a normal distribution. Since $\hat{\beta}_0$ and $\hat{\beta}_1$ are linear combinations of normal random variables, $\hat{y}(x)$ is as well.

$$\hat{y}(x) \sim N \left(\beta_0 + \beta_1 x, \sigma^2 \left(\frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}} \right) \right)$$

And lastly, since we need to estimate this variance, we arrive at the standard error of our estimate,

$$\mathrm{SE}[\hat{y}(x)] = s_e \sqrt{\frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}}.$$

We can then use this to find the confidence interval for the mean response,

$$\hat{y}(x) \pm t_{\alpha/2, n-2} \cdot s_e \sqrt{\frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}}$$

To find confidence intervals for the mean response using `R`, we use the `predict()` function. We give the function our fitted model as well as new data, stored as a data frame. (This is important, so that `R` knows the name of the predictor variable.) Here, we are finding the confidence interval for the mean stopping distance when a car is travelling 5 miles per hour and when a car is travelling 21 miles per hour.

```
new_speeds = data.frame(speed = c(5, 21))
predict(stop_dist_model, newdata = new_speeds,
        interval = c("confidence"), level = 0.99)
```

```
##         fit       lwr      upr
## 1  2.082949 -10.89309 15.05898
## 2 65.001489  56.45836 73.54462
```

Prediction Interval for New Observations

Sometimes we would like an interval estimate for a new observation, Y, for a particular value of x. This is very similar to an interval for the mean response, $\mathrm{E}[Y \mid X = x]$, but different in one very important way.

Our best guess for a new observation is still $\hat{y}(x)$. The estimated mean is still the best prediction we can make. The difference is in the amount of variability. We know that observations will vary about the true regression line according to a $N(0, \sigma^2)$ distribution. Because of this we add an extra factor of σ^2 to our estimate's variability in order to account for the variability of observations about the regression line.

$$\begin{aligned}
\mathrm{Var}[\hat{y}(x) + \epsilon] &= \mathrm{Var}[\hat{y}(x)] + \mathrm{Var}[\epsilon] \\
&= \sigma^2 \left(\frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}} \right) + \sigma^2 \\
&= \sigma^2 \left(1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}} \right)
\end{aligned}$$

$$\hat{y}(x) + \epsilon \sim N \left(\beta_0 + \beta_1 x, \ \sigma^2 \left(1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}} \right) \right)$$

$$\mathrm{SE}[\hat{y}(x) + \epsilon] = s_e \sqrt{1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}}$$

We can then find a **prediction interval** using,

$$\hat{y}(x) \pm t_{\alpha/2, n-2} \cdot s_e \sqrt{1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}}.$$

To calculate this for a set of points in `R` notice there is only a minor change in syntax from finding a confidence interval for the mean response.

```
predict(stop_dist_model, newdata = new_speeds,
        interval = c("prediction"), level = 0.99)
```

```
##         fit       lwr       upr
## 1  2.082949 -41.16099  45.32689
## 2 65.001489  22.87494 107.12803
```

Also notice that these two intervals are wider than the corresponding confidence intervals for the mean response.

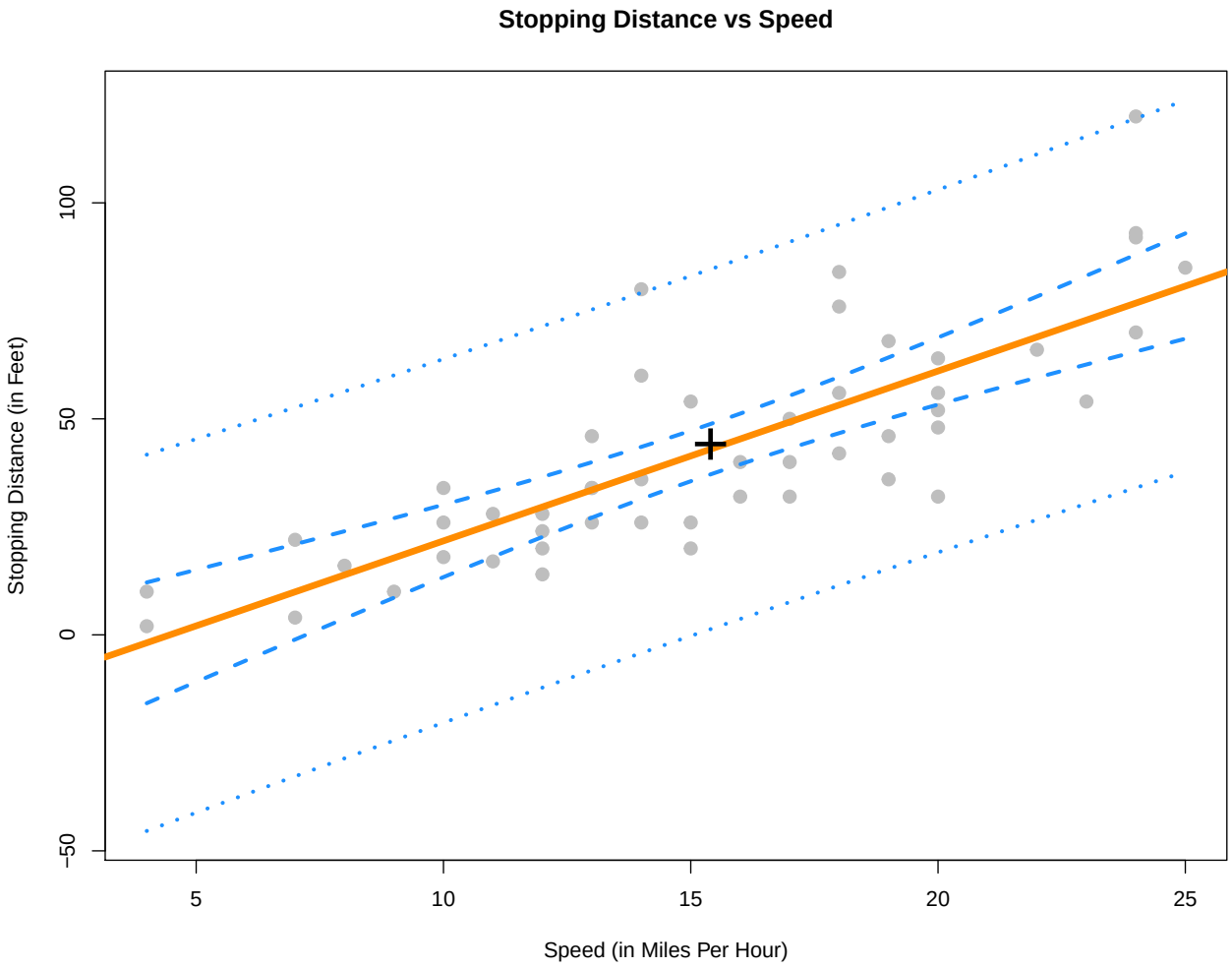
Confidence and Prediction Bands

Often we will like to plot both confidence intervals for the mean response and prediction intervals for all possible values of x. We calls these confidence and prediction bands.

```
speed_grid = seq(min(cars$speed), max(cars$speed), by = 0.01)
dist_ci_band = predict(stop_dist_model,
                       newdata = data.frame(speed = speed_grid),
                       interval = "confidence", level = 0.99)
dist_pi_band = predict(stop_dist_model,
                       newdata = data.frame(speed = speed_grid),
                       interval = "prediction", level = 0.99)
```

```r
plot(dist ~ speed, data = cars,
     xlab = "Speed (in Miles Per Hour)",
     ylab = "Stopping Distance (in Feet)",
     main = "Stopping Distance vs Speed",
     pch  = 20,
     cex  = 2,
     col  = "grey",
     ylim = c(min(dist_pi_band), max(dist_pi_band)))
abline(stop_dist_model, lwd = 5, col = "darkorange")

lines(speed_grid, dist_ci_band[,"lwr"], col = "dodgerblue", lwd = 3, lty = 2)
lines(speed_grid, dist_ci_band[,"upr"], col = "dodgerblue", lwd = 3, lty = 2)
lines(speed_grid, dist_pi_band[,"lwr"], col = "dodgerblue", lwd = 3, lty = 3)
lines(speed_grid, dist_pi_band[,"upr"], col = "dodgerblue", lwd = 3, lty = 3)
points(mean(cars$speed), mean(cars$dist), pch = "+", cex = 3)
```



Some things to notice:

- We use the `ylim` argument to stretch the y-axis of the plot, since the bands extend further than the points.
- We add a point at the point $(\bar{x}, \bar{y})$.
  - This is a point that the regression line will **always** pass through. (Think about why.)
  - This is the point where both the confidence and prediction bands are the narrowest. Look at the

standard errors of both to understand why.

- The prediction bands (dotted blue) are less curved than the confidence bands (dashed blue). This is a result of the extra factor of σ^2 added to the variance at any value of x.

Significance of Regression, F-Test

In the case of simple linear regression, the t test for the significance of the regression is equivalent to another test, the F test for the significance of the regression. This equivalence will only be true for simple linear regression, and in the next chapter we will only use the F test for the significance of the regression.

Recall from last chapter the decomposition of variance we saw before calculating R^2,

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

or, in short,

$$\text{SST} = \text{SSE} + \text{SSReg}.$$

To develop the F test, we will arrange this information in an **ANOVA** table,

Source	Sum of Squares	Degrees of Freedom	Mean Square	F
Regression	$\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$	1	SSReg/1	MSReg/MSE
Error	$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	$n-2$	SSE/$(n-2)$	
Total	$\sum_{i=1}^{n}(y_i - \bar{y})^2$	$n-1$		

ANOVA, or Analysis of Variance will be a concept we return to often in this course. For now, we will focus on the results of the table, which is the F statistic,

$$F = \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2/1}{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2/(n-2)} \sim F_{1,n-2}$$

which follows an F distribution with degrees of freedom 1 and $n-2$ under the null hypothesis. An F distribution is a continuous distribution which takes only positive values and has two parameters, which are the two degrees of freedom.

Recall, in the significance of the regression test, Y does **not** depend on x in the null hypothesis.

$$H_0 : \beta_1 = 0 \qquad Y_i = \beta_0 + \epsilon_i$$

While in the alternative hypothesis Y may depend on x.

$$H_1 : \beta_1 \neq 0 \qquad Y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

We can use the F statistic to perform this test.

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

In particular, we will reject the null when the F statistic is large, that is, when there is a low probability that the observations could have come from the null model by chance. We will let `R` calculate the p-value for us.

To perform the F test in `R` you can look at the last row of the output from `summary()` called `F-statistic` which gives the value of the test statistic, the relevant degrees of freedom, as well as the p-value of the test.

```
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 1.49e-12 ***
## ---
## 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: 1.49e-12
```

Additionally, you can use the `anova()` function to display the information in an ANOVA table.

```
anova(stop_dist_model)
```

```
## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value   Pr(>F)
## speed      1  21186 21185.5  89.567 1.49e-12 ***
## Residuals 48  11354   236.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This also gives a p-value for the test. You should notice that the p-value from the t test was the same. You might also notice that the value of the test statistic for the t test, 9.46399, can be squared to obtain the value of the F statistic, 89.5671065.

Note that there is another equivalent way to do this in `R`, which we will return to often to compare two models.

```
anova(lm(dist ~ 1, data = cars), lm(dist ~ speed, data = cars))
```

```
## Analysis of Variance Table
##
## Model 1: dist ~ 1
## Model 2: dist ~ speed
##   Res.Df   RSS Df Sum of Sq      F   Pr(>F)
## 1     49 32539
## 2     48 11354  1     21186 89.567 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The model statement `lm(dist ~ 1, data = cars)` applies the model $Y_i = \beta_0 + \epsilon_i$ to the cars data. Note that $\hat{y} = \bar{y}$ when $Y_i = \beta_0 + \epsilon_i$.

The model statement `lm(dist ~ speed, data = cars)` applies the model $Y_i = \beta_0 + \beta_1 x_i + \epsilon_i$.

We can then think of this usage of `anova()` as directly comparing the two models. (Notice we get the same p-value again.)

After reading this chapter you will be able to:

- Construct and interpret linear regression models with more than one predictor.
- Understand how regression models are derived using matrices.
- Create interval estimates and perform hypothesis tests for multiple regression parameters.
- Formulate and interpret interval estimates for the mean response under various conditions.
- Compare nested models using an ANOVA F-Test.

The last two chapters we saw how to fit a model that assumed a linear relationship between a response variable and a single predictor variable. Specifically, we defined the simple linear regression model,

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

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

However, it is rarely the case that a dataset will have a single predictor variable. It is also rarely the case that a response variable will only depend on a single variable. So in this chapter, we will extend our current linear model to allow a response to depend on *multiple* predictors.

```
# read the data from the web
autompg = read.table(
  "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data",
  quote = "\"",
  comment.char = "",
  stringsAsFactors = FALSE)
# give the dataframe headers
colnames(autompg) = c("mpg", "cyl", "disp", "hp", "wt", "acc", "year", "origin", "name")
# remove missing data, which is stored as "?"
autompg = subset(autompg, autompg$hp != "?")
# remove the plymouth reliant, as it causes some issues
autompg = subset(autompg, autompg$name != "plymouth reliant")
# give the dataset row names, based on the engine, year and name
rownames(autompg) = paste(autompg$cyl, "cylinder", autompg$year, autompg$name)
# remove the variable for name, as well as origin
autompg = subset(autompg, select = c("mpg", "cyl", "disp", "hp", "wt", "acc", "year"))
# change horsepower from character to numeric
autompg$hp = as.numeric(autompg$hp)
# check final structure of data
str(autompg)
```

```
## 'data.frame':    390 obs. of  7 variables:
##  $ mpg : num  18 15 18 16 17 15 14 14 14 15 ...
##  $ cyl : int  8 8 8 8 8 8 8 8 8 8 ...
##  $ disp: num  307 350 318 304 302 429 454 440 455 390 ...
##  $ hp  : num  130 165 150 150 140 198 220 215 225 190 ...
##  $ wt  : num  3504 3693 3436 3433 3449 ...
##  $ acc : num  12 11.5 11 12 10.5 10 9 8.5 10 8.5 ...
##  $ year: int  70 70 70 70 70 70 70 70 70 70 ...
```

We will once again discuss a dataset with information about cars. This dataset, which can be found at the UCI Machine Learning Repository contains a response variable `mpg` which stores the city fuel efficiency of cars, as well as several predictor variables for the attributes of the vehicles. We load the data, and perform some basic tidying before moving on to analysis.

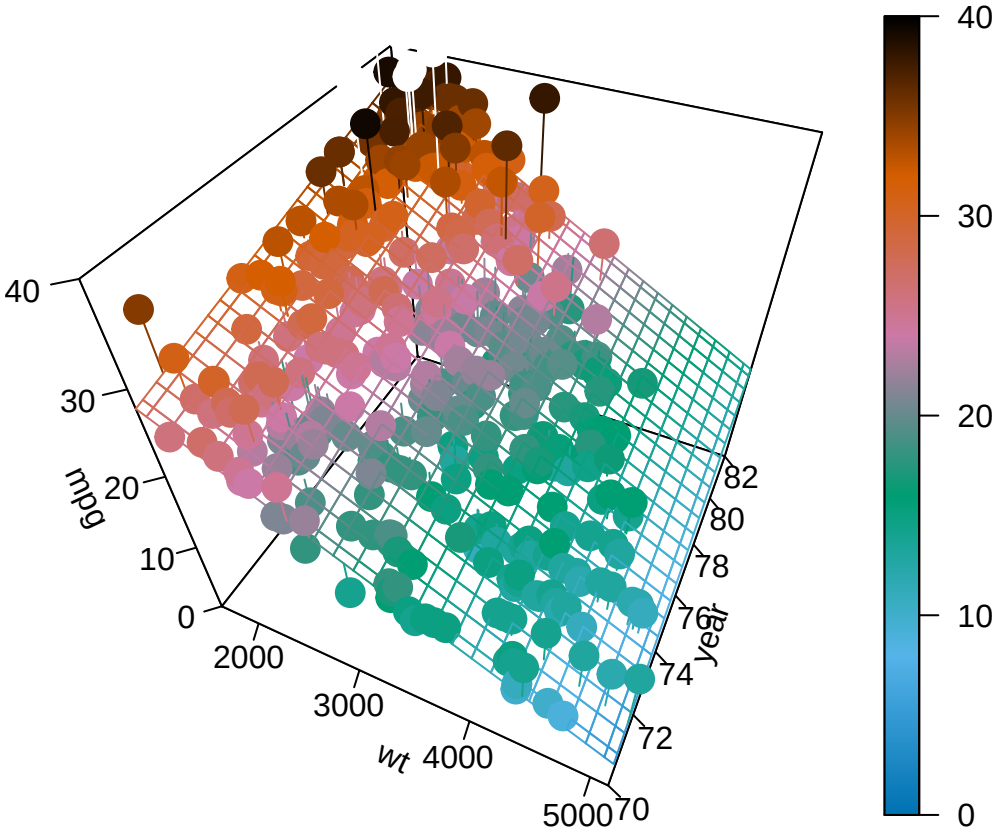
For now we will focus on using two variables, `wt` and `year`, as predictor variables. That is, we would like to model the fuel efficiency (`mpg`) of a car as a function of its weight (`wt`) and model year (`year`). To do so, we will define the following linear model,

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i, \qquad i = 1, 2, \ldots, n$$

where $\epsilon_i \sim N(0, \sigma^2)$. In this notation we will define:

- x_{i1} as the weight (`wt`) of the ith car.
- x_{i2} as the model year (`year`) of the ith car.

The picture below will visualize what we would like to accomplish. The data points (x_{i1}, x_{i2}, y_i) now exist in 3-dimensional space, so instead of fitting a line to the data, we will fit a plane. (We'll soon move to higher dimensions, so this will be the last example that is easy to visualize and think about this way.)



How do we find such a plane? Well, we would like a plane that is as close as possible to the data points. That is, we would like it to minimize the errors it is making. How will we define these errors? Squared distance of course! So, we would like to minimize

$$f(\beta_0, \beta_1, \beta_2) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2}))^2$$

with respect to β_0, β_1, and β_2. How do we do so? It is another straightforward multivariate calculus problem.

All we have done is add an extra variable since we did this last time. So again, we take a derivative with respect to each of β_0, β_1, and β_2 and set them equal to zero, then solve the resulting system of equations. That is,

$$\frac{\partial f}{\partial \beta_0} = 0$$
$$\frac{\partial f}{\partial \beta_1} = 0$$
$$\frac{\partial f}{\partial \beta_2} = 0$$

After doing so, we will once again obtain the **normal equations.**

$$n\beta_0 + \beta_1 \sum_{i=1}^{n} x_{i1} + \beta_2 \sum_{i=1}^{n} x_{i2} = \sum_{i=1}^{n} y_i$$
$$\beta_0 \sum_{i=1}^{n} x_{i1} + \beta_1 \sum_{i=1}^{n} x_{i1}^2 + \beta_2 \sum_{i=1}^{n} x_{i1}x_{i2} = \sum_{i=1}^{n} x_{i1}y_i$$
$$\beta_0 \sum_{i=1}^{n} x_{i2} + \beta_1 \sum_{i=1}^{n} x_{i1}x_{i2} + \beta_2 \sum_{i=1}^{n} x_{i2}^2 = \sum_{i=1}^{n} x_{i2}y_i$$

We now have three equations and three variables, which we could solve, or we could simply let `R` solve for us.

```
mpg_model = lm(mpg ~ wt + year, data = autompg)
coef(mpg_model)
```

```
##   (Intercept)            wt          year
## -14.637641945  -0.006634876   0.761401955
```

$$\hat{y} = -14.6376419 + -0.0066349x_1 + 0.761402x_2$$

Here we have once again fit our model using `lm()`, however we have introduced a new syntactical element. The formula `mpg ~ wt + year` now reads: "model the response variable `mpg` as a linear function of `wt` and `year`". That is, it will estimate an intercept, as well as slope coefficients for `wt` and `year`. We then extract these as we have done before using `coef()`.

In the multiple linear regression setting, some of the interpretations of the coefficients change slightly.

Here, $\hat{\beta}_0 = -14.6376419$ is our estimate for β_0, the mean miles per gallon for a car that weighs 0 pounds and was built in 1900. We see our estimate here is negative, which is a physical impossibility. However, this isn't unexpected, as we shouldn't expect our model to be accurate for cars from 1900 which weigh 0 pounds. (Because they never existed!) This isn't much of a change from SLR. That is, β_0 is still simply the mean when all of the predictors are 0.

The interpretation of the coefficients in front of our predictors are slightly different than before. For example $\hat{\beta}_1 = -0.0066349$ is our estimate for β_1, the average change in miles per gallon for an increase in weight (x_1) of one-pound **for a car of a certain model year**, that is, for a fixed value of x_2. Note that this coefficient is actually the same for any given value of x_2. Later, we will look at models that allow for a different change in mean response for different values of x_2. Also note that this estimate is negative, which we would expect since, in general, fuel efficiency decreases for larger vehicles. Recall that in the multiple linear regression setting, this interpretation is dependent on a fixed value for x_2, that is, "for a car of a certain model year." It is possible that the indirect relationship between fuel efficiency and weight does not hold when an additional factor, say year, is included, and thus we could have the sign of our coefficient flipped.

Lastly, $\hat{\beta}_2 = 0.761402$ is our estimate for β_2, the average change in miles per gallon for a one-year increase in model year (x_2) for a car of a certain weight, that is, for a fixed value of x_1. It is not surprising that the estimate is positive. We expect that as time passes and the years march on, technology would improve so that a car of a specific weight would get better mileage now as compared to their predecessors. And yet, the coefficient could have been negative because we are also including weight as variable, and not strictly as a fixed value.

Matrix Approach to Regression

In our above example we used two predictor variables, but it will only take a little more work to allow for an arbitrary number of predictor variables and derive their coefficient estimates. We can consider the model,

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_{p-1} x_{i(p-1)} + \epsilon_i, \qquad i = 1, 2, \ldots, n$$

where $\epsilon_i \sim N(0, \sigma^2)$. In this model, there are $p-1$ predictor variables, $x_1, x_2, \cdots, x_{p-1}$. There are a total of p β-parameters and a single parameter σ^2 for the variance of the errors. (It should be noted that almost as often, authors will use p as the number of predictors, making the total number of β parameters $p+1$. This is always something you should be aware of when reading about multiple regression. There is not a standard that is used most often.)

If we were to stack together the n linear equations that represent each Y_i into a column vector, we get the following.

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1(p-1)} \\ 1 & x_{21} & x_{22} & \cdots & x_{2(p-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{n(p-1)} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p-1} \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

$$Y = X\beta + \epsilon$$

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1(p-1)} \\ 1 & x_{21} & x_{22} & \cdots & x_{2(p-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{n(p-1)} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p-1} \end{bmatrix}, \quad \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

So now with data,

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Just as before, we can estimate β by minimizing,

$$f(\beta_0, \beta_1, \beta_2, \cdots, \beta_{p-1}) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_{p-1} x_{i(p-1)}))^2,$$

which would require taking p derivatives, which result in following **normal equations**.

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_{i1} & \sum_{i=1}^{n} x_{i2} & \cdots & \sum_{i=1}^{n} x_{i(p-1)} \\ \sum_{i=1}^{n} x_{i1} & \sum_{i=1}^{n} x_{i1}^2 & \sum_{i=1}^{n} x_{i1}x_{i2} & \cdots & \sum_{i=1}^{n} x_{i1}x_{i(p-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \sum_{i=1}^{n} x_{i(p-1)} & \sum_{i=1}^{n} x_{i(p-1)}x_{i1} & \sum_{i=1}^{n} x_{i(p-1)}x_{i2} & \cdots & \sum_{i=1}^{n} x_{i(p-1)}^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{p-1} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_{i1}y_i \\ \vdots \\ \sum_{i=1}^{n} x_{i(p-1)}y_i \end{bmatrix}$$

The normal equations can be written much more succinctly in matrix notation,

$$X^\top X\beta = X^\top y.$$

We can then solve this expression by multiplying both sides by the inverse of $X^\top X$, which exists, provided the columns of X are linearly independent. Then as always, we denote our solution with a hat.

$$\hat{\beta} = \left(X^\top X\right)^{-1} X^\top y$$

To verify that this is what `R` has done for us in the case of two predictors, we create an X matrix. Note that the first column is all 1s, and the remaining columns contain the data.

```
n = nrow(autompg)
p = length(coef(mpg_model))
X = cbind(rep(1, n), autompg$wt, autompg$year)
y = autompg$mpg

(beta_hat = solve(t(X) %*% X) %*% t(X) %*% y)
```

```
##                [,1]
## [1,] -14.637641945
## [2,]  -0.006634876
## [3,]   0.761401955
```

```
coef(mpg_model)
```

```
##   (Intercept)            wt          year
## -14.637641945  -0.006634876   0.761401955
```

$$\hat{\beta} = \begin{bmatrix} -14.6376419 \\ -0.0066349 \\ 0.761402 \end{bmatrix}$$

In our new notation, the fitted values can be written

$$\hat{y} = X\hat{\beta}.$$

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}$$

Then, we can create a vector for the residual values,

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}.$$

And lastly, we can update our estimate for σ^2.

$$s_e^2 = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n-p} = \frac{e^\top e}{n-p}$$

Recall, we like this estimate because it is unbiased, that is,

$$\mathrm{E}[s_e^2] = \sigma^2$$

Note that the change from the SLR estimate to now is in the denominator. Specifically we now divide by $n-p$ instead of $n-2$. Or actually, we should note that in the case of SLR, there are two β parameters and thus $p = 2$.

Also note that if we fit the model $Y_i = \beta + \epsilon_i$ that $\hat{y} = \bar{y}$ and $p = 1$ and s_e^2 would become

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

which is likely the very first sample variance you saw in a mathematical statistics class. The same reason for $n-1$ in this case, that we estimated one parameter, so we lose one degree of freedom. Now, in general, we are estimating p parameters, the β parameters, so we lose p degrees of freedom.

Also, recall that most often we will be interested in s_e, the residual standard error as `R` calls it,

$$s_e = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n-p}}.$$

In `R`, we could directly access s_e for a fitted model, as we have seen before.

```
summary(mpg_model)$sigma
```

```
## [1] 3.431367
```

And we can now verify that our math above is indeed calculating the same quantities.

```
y_hat = X %*% solve(t(X) %*% X) %*% t(X) %*% y
e     = y - y_hat
sqrt(t(e) %*% e / (n - p))
```

```
##          [,1]
## [1,] 3.431367
```

```
sqrt(sum((y - y_hat) ^ 2) / (n - p))
```

```
## [1] 3.431367
```

Sampling Distribution

As we can see in the output below, the results of calling `summary()` are similar to SLR, but there are some differences, most obviously a new row for the added predictor variable.

```
summary(mpg_model)
```

```
##
## Call:
## lm(formula = mpg ~ wt + year, data = autompg)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.852 -2.292 -0.100  2.039 14.325
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.464e+01  4.023e+00  -3.638 0.000312 ***
## wt          -6.635e-03  2.149e-04 -30.881  < 2e-16 ***
## year         7.614e-01  4.973e-02  15.312  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.431 on 387 degrees of freedom
## Multiple R-squared:  0.8082, Adjusted R-squared:  0.8072
## F-statistic: 815.6 on 2 and 387 DF,  p-value: < 2.2e-16
```

To understand these differences in detail, we will need to first obtain the sampling distribution of $\hat{\beta}$.

The derivation of the sampling distribution of $\hat{\beta}$ involves the multivariate normal distribution. Which will not be presented in full here.

Our goal now is to obtain the distribution of the $\hat{\beta}$ vector,

$$\hat{\beta} = \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_{p-1} \end{bmatrix}$$

Recall from last time that when discussing sampling distributions, we now consider $\hat{\beta}$ to be a random vector, thus we use Y instead of the data vector y.

$$\hat{\beta} = \left(X^\top X\right)^{-1} X^\top Y$$

Then it is a consequence of the multivariate normal distribution that,

$$\hat{\beta} \sim N\left(\beta, \sigma^2 \left(X^\top X\right)^{-1}\right).$$

We then have

$$\mathrm{E}[\hat{\beta}] = \beta$$

and for any $\hat{\beta}_j$ we have

$$\mathrm{E}[\hat{\beta}_j] = \beta_j.$$

We also have

$$\mathrm{Var}[\hat{\beta}] = \sigma^2 \left(X^\top X\right)^{-1}$$

and for any $\hat{\beta}_j$ we have

$$\mathrm{Var}[\hat{\beta}_j] = \sigma^2 C_{jj}$$

where

$$C = \left(X^\top X\right)^{-1}$$

and the elements of C are denoted

$$C = \begin{bmatrix} C_{00} & C_{01} & C_{02} & \cdots & C_{0(p-1)} \\ C_{10} & C_{11} & C_{12} & \cdots & C_{1(p-1)} \\ C_{20} & C_{21} & C_{22} & \cdots & C_{2(p-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{(p-1)0} & C_{(p-1)1} & C_{(p-1)2} & \cdots & C_{(p-1)(p-1)} \end{bmatrix}.$$

Essentially, the diagonal elements correspond to the β vector.

Then the standard error for the $\hat{\beta}$ vector is given by

$$\mathrm{SE}[\hat{\beta}] = s_e \sqrt{\left(X^\top X\right)^{-1}}$$

and for a particular $\hat{\beta}_j$

$$\mathrm{SE}[\hat{\beta}_j] = s_e \sqrt{C_{jj}}.$$

Lastly, each of the $\hat{\beta}_j$ follows a normal distribution,

$$\hat{\beta}_j \sim N\left(\beta_j, \sigma^2 C_{jj}\right).$$

thus

$$\frac{\hat{\beta}_j - \beta_j}{s_e \sqrt{C_{jj}}} \sim t_{n-p}.$$

Now that we have the necessary distributional results, we can move on to perform tests and make interval estimates.

Single Parameter Tests

The first test we will see is a test for a single β_j.

$$H_0 : \beta_j = 0 \quad \text{vs} \quad H_1 : \beta_j \neq 0$$

Again, the test statistic takes the form

$$\text{TS} = \frac{\text{EST} - \text{HYP}}{\text{SE}}.$$

In particular,

$$t = \frac{\hat{\beta}_j - \beta_j}{\text{SE}[\hat{\beta}_j]} = \frac{\hat{\beta}_j - 0}{s_e\sqrt{C_{jj}}},$$

which, under the null hypothesis, follows a t distribution with $n - p$ degrees of freedom.

Recall our model for `mpg`,

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i, \qquad i = 1, 2, \ldots, n$$

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

- x_{i1} as the weight (`wt`) of the ith car.
- x_{i2} as the model year (`year`) of the ith car.

Then the test

$$H_0 : \beta_1 = 0 \quad \text{vs} \quad H_1 : \beta_1 \neq 0$$

can be found in the `summary()` output, in particular:

```
summary(mpg_model)$coef
```

```
##                  Estimate   Std. Error    t value      Pr(>|t|)
## (Intercept) -14.637641945 4.0233913563  -3.638135  3.118311e-04
## wt           -0.006634876 0.0002148504 -30.881372 1.850466e-106
## year          0.761401955 0.0497265950  15.311765  1.036597e-41
```

The estimate (`Estimate`), standard error (`Std. Error`), test statistic (`t value`), and p-value (`Pr(>|t|)`) for this test are displayed in the second row, labeled `wt`. Remember that the p-value given here is specifically for a two-sided test, where the hypothesized value is 0.

Also note in this case, by hypothesizing that $\beta_1 = 0$ the null and alternative essentially specify two different models:

- H_0: $Y = \beta_0 + \beta_2 x_2 + \epsilon$
- H_1: $Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$

This is important. We are not simply testing whether or not there is a relationship between weight and fuel efficiency. We are testing if there is a relationship between weight and fuel efficiency, given that a term for year is in the model. (Note, we dropped some indexing here, for readability.)

Confidence Intervals

Since $\hat{\beta}_j$ is our estimate for β_j and we have

$$\mathrm{E}[\hat{\beta}_j] = \beta_j$$

as well as the standard error,

$$\mathrm{SE}[\hat{\beta}_j] = s_e\sqrt{C_{jj}}$$

and the sampling distribution of $\hat{\beta}_j$ is Normal, then we can easily construct confidence intervals for each of the $\hat{\beta}_j$.

$$\hat{\beta}_j \pm t_{\alpha/2, n-p} \cdot s_e\sqrt{C_{jj}}$$

We can find these in `R` using the same method as before. Now there will simply be additional rows for the additional β.

```
confint(mpg_model, level = 0.99)
```

```
##                      0.5 %       99.5 %
## (Intercept) -25.052563681 -4.222720208
## wt           -0.007191036 -0.006078716
## year          0.632680051  0.890123859
```

Confidence Intervals for Mean Response

As we saw in SLR, we can create confidence intervals for the mean response, that is, an interval estimate for $\mathrm{E}[Y \mid X = x]$. In SLR, the mean of Y was only dependent on a single value x. Now, in multiple regression, $\mathrm{E}[Y \mid X = x]$ is dependent on the value of each of the predictors, so we define the vector x_0 to be,

$$x_0 = \begin{bmatrix} 1 \\ x_{01} \\ x_{02} \\ \vdots \\ x_{0(p-1)} \end{bmatrix}.$$

Then our estimate of $\mathrm{E}[Y \mid X = x_0]$ for a set of values x_0 is given by

$$\begin{aligned} \hat{y}(x_0) &= x_0^\top \hat{\beta} \\ &= \hat{\beta}_0 + \hat{\beta}_1 x_{01} + \hat{\beta}_2 x_{02} + \cdots + \hat{\beta}_{p-1} x_{0(p-1)}. \end{aligned}$$

As with SLR, this is an unbiased estimate.

$$\begin{aligned} \mathrm{E}[\hat{y}(x_0)] &= x_0^\top \beta \\ &= \beta_0 + \beta_1 x_{01} + \beta_2 x_{02} + \cdots + \beta_{p-1} x_{0(p-1)} \end{aligned}$$

To make an interval estimate, we will also need its standard error.

$$\mathrm{SE}[\hat{y}(x_0)] = s_e\sqrt{x_0^\top \left(X^\top X\right)^{-1} x_0}$$

Putting it all together, we obtain a confidence interval for the mean response.

$$\hat{y}(x_0) \pm t_{\alpha/2, n-p} \cdot s_e \sqrt{x_0^\top \left(X^\top X\right)^{-1} x_0}$$

The math has changed a bit, but the process in `R` remains almost identical. Here, we create a data frame for two additional cars. One car that weighs 3500 pounds produced in 1976, as well as a second car that weighs 5000 pounds which was produced in 1981.

```
new_cars = data.frame(wt = c(3500, 5000), year = c(76, 81))
new_cars
```

```
##     wt year
## 1 3500   76
## 2 5000   81
```

We can then use the `predict()` function with `interval = "confidence"` to obtain intervals for the mean fuel efficiency for both new cars. Again, it is important to make the data passed to `newdata` a data frame, so that `R` knows which values are for which variables.

```
predict(mpg_model, newdata = new_cars, interval = "confidence", level = 0.99)
```

```
##        fit     lwr      upr
## 1 20.00684 19.4712 20.54248
## 2 13.86154 12.3341 15.38898
```

`R` then reports the estimate $\hat{y}(x_0)$ (`fit`) for each, as well as the lower (`lwr`) and upper (`upr`) bounds for the interval at a desired level (99%).

A word of caution here: one of these estimates is good while one is suspect.

```
new_cars$wt
```

```
## [1] 3500 5000
```

```
range(autompg$wt)
```

```
## [1] 1613 5140
```

Note that both of the weights of the new cars are within the range of observed values.

```
new_cars$year
```

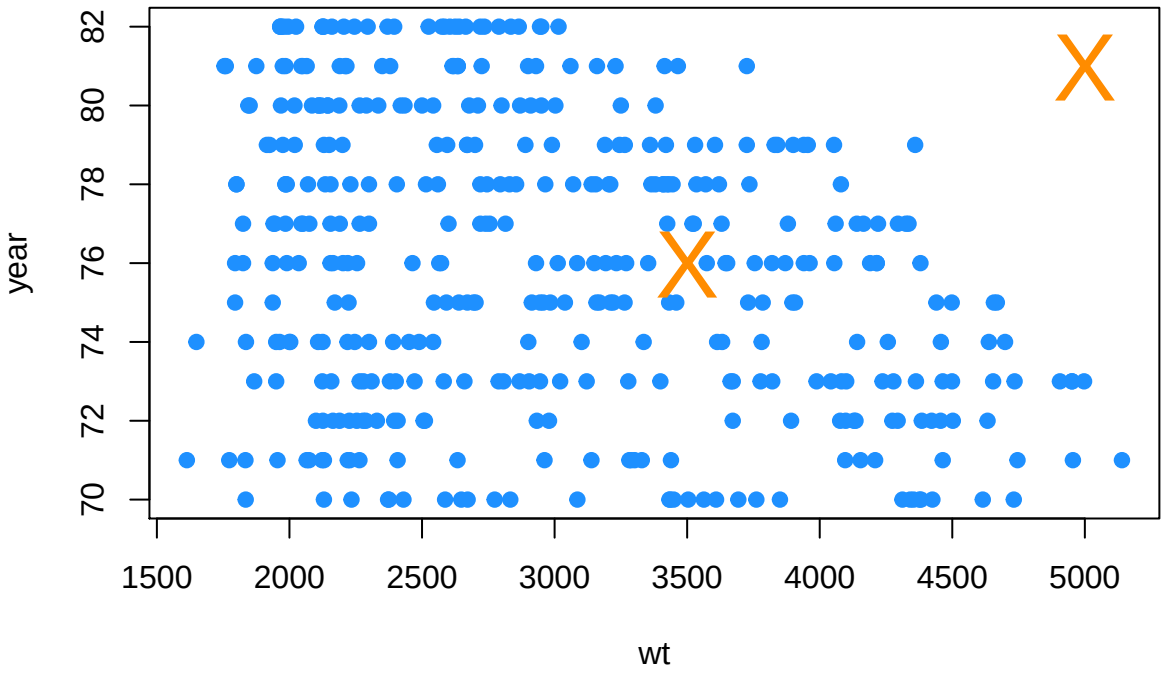
```
## [1] 76 81
```

```
range(autompg$year)
```

```
## [1] 70 82
```

As are the years of each of the new cars.

```
plot(year ~ wt, data = autompg, pch = 20, col = "dodgerblue", cex = 1.5)
points(new_cars, col = "darkorange", cex = 3, pch = "X")
```



However, we have to consider weight and year together now. And based on the above plot, one of the new cars is within the "blob" of observed values, while the other, the car from 1981 weighing 5000 pounds, is noticeably outside of the observed values. This is a hidden extrapolation which you should be aware of when using multiple regression.

Shifting gears back to the new data pair that can be reasonably estimated, we do a quick verification of some of the mathematics in R.

```
x0 = c(1, 3500, 76)
x0 %*% beta_hat
```

```
##           [,1]
## [1,] 20.00684
```

$$x_0 = \begin{bmatrix} 1 \\ 3500 \\ 76 \end{bmatrix}$$

$$\hat{\beta} = \begin{bmatrix} -14.6376419 \\ -0.0066349 \\ 0.761402 \end{bmatrix}$$

$$\hat{y}(x_0) = x_0^\top \hat{\beta} = \begin{bmatrix} 1 & 3500 & 76 \end{bmatrix} \begin{bmatrix} -14.6376419 \\ -0.0066349 \\ 0.761402 \end{bmatrix} = 20.0068411$$

Also note that, using a particular value for x_0, we can essentially extract certain $\hat{\beta}_j$ values.

```
beta_hat
```

```
##                [,1]
## [1,] -14.637641945
## [2,]  -0.006634876
## [3,]   0.761401955
```

```
x0 = c(0, 0, 1)
x0 %*% beta_hat
```

```
##           [,1]
## [1,] 0.761402
```

With this in mind, confidence intervals for the individual $\hat{\beta}_j$ are actually a special case of a confidence interval for mean response.

Prediction Intervals

As with SLR, creating prediction intervals involves one slight change to the standard error to account for the fact that we are now considering an observation, instead of a mean.

Here we use $\hat{y}(x_0)$ to estimate Y_0, a new observation of Y at the predictor vector x_0.

$$\begin{aligned}\hat{y}(x_0) &= x_0^\top \hat{\beta} \\ &= \hat{\beta}_0 + \hat{\beta}_1 x_{01} + \hat{\beta}_2 x_{02} + \cdots + \hat{\beta}_{p-1} x_{0(p-1)}\end{aligned}$$

$$\begin{aligned}\mathrm{E}[\hat{y}(x_0)] &= x_0^\top \beta \\ &= \beta_0 + \beta_1 x_{01} + \beta_2 x_{02} + \cdots + \beta_{p-1} x_{0(p-1)}\end{aligned}$$

As we did with SLR, we need to account for the additional variability of an observation about its mean.

$$\mathrm{SE}[\hat{y}(x_0) + \epsilon] = s_e \sqrt{1 + x_0^\top \left(X^\top X\right)^{-1} x_0}$$

Then we arrive at our updated prediction interval for MLR.

$$\hat{y}(x_0) \pm t_{\alpha/2, n-p} \cdot s_e \sqrt{1 + x_0^\top \left(X^\top X\right)^{-1} x_0}$$

```
new_cars
```

```
##     wt year
## 1 3500   76
## 2 5000   81
```

```
predict(mpg_model, newdata = new_cars, interval = "prediction", level = 0.99)
```

```
##        fit       lwr      upr
## 1 20.00684 11.108294 28.90539
## 2 13.86154  4.848751 22.87432
```

Significance of Regression

The decomposition of variation that we had seen in SLR still holds for MLR.

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

That is,

$$\mathrm{SST} = \mathrm{SSE} + \mathrm{SSReg}.$$

This means that, we can still calculate R^2 in the same manner as before, which `R` continues to do automatically.

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

```
## [1] 0.8082355
```

The interpretation changes slightly as compared to SLR. In this MLR case, we say that 80.82% for the observed variation in miles per gallon is explained by the linear relationship with the two predictor variables, weight and year.

In multiple regression, the significance of regression test is

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_{p-1} = 0.$$

Here, we see that the null hypothesis sets all of the β_j equal to 0, *except* the intercept, β_0. We could then say that the null model, or "model under the null hypothesis" is

$$Y_i = \beta_0 + \epsilon_i.$$

This is a model where the *regression* is insignificant. **None** of the predictors have a significant linear relationship with the response. Notationally, we will denote the fitted values of this model as $\hat{y}_{0i}$, which in this case happens to be:

$$\hat{y}_{0i} = \bar{y}.$$

The alternative hypothesis here is that at least one of the β_j from the null hypothesis is not 0.

$$H_1 : \text{At least one of } \beta_j \neq 0, j = 1, 2, \cdots, (p-1)$$

We could then say that the full model, or "model under the alternative hypothesis" is

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_{(p-1)} x_{i(p-1)} + \epsilon_i$$

This is a model where the regression is significant. **At least one** of the predictors has a significant linear relationship with the response. There is some linear relationship between y and the predictors, $x_1, x_2, \ldots, x_{p-1}$.

We will denote the fitted values of this model as $\hat{y}_{1i}$.

To develop the F test for the significance of the regression, we will arrange the variance decomposition into an ANOVA table.

Source	Sum of Squares	Degrees of Freedom	Mean Square	F
Regression	$\sum_{i=1}^{n}(\hat{y}_{1i} - \bar{y})^2$	$p-1$	SSReg/$(p-1)$	MSReg/MSE
Error	$\sum_{i=1}^{n}(y_i - \hat{y}_{1i})^2$	$n-p$	SSE/$(n-p)$	
Total	$\sum_{i=1}^{n}(y_i - \bar{y})^2$	$n-1$		

In summary, the F statistic is

$$F = \frac{\sum_{i=1}^{n}(\hat{y}_{1i} - \bar{y})^2/(p-1)}{\sum_{i=1}^{n}(y_i - \hat{y}_{1i})^2/(n-p)},$$

and the p-value is calculated as

$$P(F_{p-1,n-p} > F)$$

since we reject for large values of F. A large value of the statistic corresponds to a large portion of the variance being explained by the regression. Here $F_{p-1,n-p}$ represents a random variable which follows an F distribution with $p-1$ and $n-p$ degrees of freedom.

To perform this test in `R`, we first explicitly specify the two models in `R` and save the results in different variables. We then use `anova()` to compare the two models, giving `anova()` the null model first and the alternative (full) model second. (Specifying the full model first will result in the same p-value, but some nonsensical intermediate values.)

In this case,

- H_0: $Y_i = \beta_0 + \epsilon_i$
- H_1: $Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i$

That is, in the null model, we use neither of the predictors, whereas in the full (alternative) model, at least one of the predictors is useful.

```
null_mpg_model = lm(mpg ~ 1, data = autompg)
full_mpg_model = lm(mpg ~ wt + year, data = autompg)
anova(null_mpg_model, full_mpg_model)
```

```
## Analysis of Variance Table
##
## Model 1: mpg ~ 1
## Model 2: mpg ~ wt + year
##   Res.Df     RSS Df Sum of Sq      F    Pr(>F)
## 1    389 23761.7
## 2    387  4556.6  2     19205 815.55 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

First, notice that `R` does not display the results in the same manner as the table above. More important than the layout of the table are its contents. We see that the value of the F statistic is 815.55, and the p-value is extremely low, so we reject the null hypothesis at any reasonable α and say that the regression is significant. At least one of `wt` or `year` has a useful linear relationship with `mpg`.

```
summary(mpg_model)
```

```
##
## Call:
## lm(formula = mpg ~ wt + year, data = autompg)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.852 -2.292 -0.100  2.039 14.325
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.464e+01  4.023e+00  -3.638 0.000312 ***
## wt          -6.635e-03  2.149e-04 -30.881  < 2e-16 ***
## year         7.614e-01  4.973e-02  15.312  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.431 on 387 degrees of freedom
```

```
## Multiple R-squared:  0.8082, Adjusted R-squared:  0.8072
## F-statistic: 815.6 on 2 and 387 DF,  p-value: < 2.2e-16
```

Notice that the value reported in the row for `F-statistic` is indeed the F test statistic for the significance of regression test, and additionally it reports the two relevant degrees of freedom.

Also, note that none of the individual t-tests are equivalent to the F-test as they were in SLR. This equivalence only holds for SLR because the individual test for β_1 is the same as testing for all non-intercept parameters, since there is only one.

We can also verify the sums of squares and degrees of freedom directly in `R`. You should match these to the table from `R` and use this to match `R`'s output to the written table above.

```
# SSReg
sum((fitted(full_mpg_model) - fitted(null_mpg_model)) ^ 2)
```

```
## [1] 19205.03
```

```
# SSE
sum(resid(full_mpg_model) ^ 2)
```

```
## [1] 4556.646
```

```
# SST
sum(resid(null_mpg_model) ^ 2)
```

```
## [1] 23761.67
```

```
# Degrees of Freedom: Regression
length(coef(full_mpg_model)) - length(coef(null_mpg_model))
```

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

```
# Degrees of Freedom: Error
length(resid(full_mpg_model)) - length(coef(full_mpg_model))
```

```
## [1] 387
```

```
# Degrees of Freedom: Total
length(resid(null_mpg_model)) - length(coef(null_mpg_model))
```

```
## [1] 389
```

Nested Models

The significance of regression test is actually a special case of testing what we will call **nested models**. More generally we can compare two models, where one model is "nested" inside the other, meaning one model contains a subset of the predictors from only the larger model.

Consider the following full model,

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_{(p-1)} x_{i(p-1)} + \epsilon_i$$

This model has $p - 1$ predictors, for a total of p β-parameters. We will denote the fitted values of this model as $\hat{y}_{1i}$.

Let the null model be

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_{(q-1)} x_{i(q-1)} + \epsilon_i$$

where $q < p$. This model has $q - 1$ predictors, for a total of q β-parameters. We will denote the fitted values of this model as $\hat{y}_{0i}$.

The difference between these two models can be codified by the null hypothesis of a test.

$$H_0 : \beta_q = \beta_{q+1} = \cdots = \beta_{p-1} = 0.$$

Specifically, the β-parameters from the full model that are not in the null model are zero. The resulting model, which is nested, is the null model.

We can then perform this test using an F-test, which is the result of the following ANOVA table.

Source	Sum of Squares	Degrees of Freedom	Mean Square	F
Diff	$\sum_{i=1}^{n}(\hat{y}_{1i} - \hat{y}_{0i})^2$	$p - q$	$\text{SSD}/(p - q)$	MSD/MSE
Full	$\sum_{i=1}^{n}(y_i - \hat{y}_{1i})^2$	$n - p$	$\text{SSE}/(n - p)$	
Null	$\sum_{i=1}^{n}(y_i - \hat{y}_{0i})^2$	$n - q$		

$$F = \frac{\sum_{i=1}^{n}(\hat{y}_{1i} - \hat{y}_{0i})^2/(p - q)}{\sum_{i=1}^{n}(y_i - \hat{y}_{1i})^2/(n - p)}.$$

Notice that the row for "Diff" compares the sum of the squared differences of the fitted values. The degrees of freedom is then the difference of the number of β-parameters estimated between the two models.

For example, the `autompg` dataset has a number of additional variables that we have yet to use.

```
names(autompg)
```

```
## [1] "mpg"  "cyl"  "disp" "hp"   "wt"   "acc"  "year"
```

We'll continue to use `mpg` as the response, but now we will consider two different models.

- Full: `mpg ~ wt + year + cyl + disp + hp + acc`
- Null: `mpg ~ wt + year`

Note that these are nested models, as the null model contains a subset of the predictors from the full model, and no additional predictors. Both models have an intercept β_0 as well as a coefficient in front of each of the predictors. We could then write the null hypothesis for comparing these two models as,

$$H_0 : \beta_{\texttt{cyl}} = \beta_{\texttt{disp}} = \beta_{\texttt{hp}} = \beta_{\texttt{acc}} = 0$$

The alternative is simply that at least one of the β_j from the null is not 0.

To perform this test in `R` we first define both models, then give them to the `anova()` commands.

```
null_mpg_model = lm(mpg ~ wt + year, data = autompg)
#full_mpg_model = lm(mpg ~ wt + year + cyl + disp + hp + acc, data = autompg)
full_mpg_model = lm(mpg ~ ., data = autompg)
anova(null_mpg_model, full_mpg_model)
```

```
## Analysis of Variance Table
##
## Model 1: mpg ~ wt + year
## Model 2: mpg ~ cyl + disp + hp + wt + acc + year
##   Res.Df    RSS Df Sum of Sq      F Pr(>F)
## 1    387 4556.6
## 2    383 4530.5  4     26.18 0.5533 0.6967
```

Here we have used the formula `mpg ~ .` to define to full model. This is the same as the commented out line. Specifically, this is a common shortcut in `R` which reads, "model `mpg` as the response with each of the remaining variables in the data frame as predictors."

Here we see that the value of the F statistic is 0.553, and the p-value is very large, so we fail to reject the null hypothesis at any reasonable α and say that none of `cyl`, `disp`, `hp`, and `acc` are significant with `wt` and `year` already in the model.

Again, we verify the sums of squares and degrees of freedom directly in `R`. You should match these to the table from `R`, and use this to match `R`'s output to the written table above.

```
# SSDiff
sum((fitted(full_mpg_model) - fitted(null_mpg_model)) ^ 2)
```

```
## [1] 26.17981
```

```
# SSE (For Full)
sum(resid(full_mpg_model) ^ 2)
```

```
## [1] 4530.466
```

```
# SST (For Null)
sum(resid(null_mpg_model) ^ 2)
```

```
## [1] 4556.646
```

```
# Degrees of Freedom: Diff
length(coef(full_mpg_model)) - length(coef(null_mpg_model))
```

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

```
# Degrees of Freedom: Full
length(resid(full_mpg_model)) - length(coef(full_mpg_model))
```

```
## [1] 383
```

```
# Degrees of Freedom: Null
length(resid(null_mpg_model)) - length(coef(null_mpg_model))
```

```
## [1] 387
```

Simulation

Since we ignored the derivation of certain results, we will again use simulation to convince ourselves of some of the above results. In particular, we will simulate samples of size `n = 100` from the model

$$Y_i = 5 + -2x_{i1} + 6x_{i2} + \epsilon_i, \qquad i = 1, 2, \ldots, n$$

where $\epsilon_i \sim N(0, \sigma^2 = 16)$. Here we have two predictors, so $p = 3$.

```
set.seed(1337)
n = 100 # sample size
p = 3

beta_0 = 5
beta_1 = -2
beta_2 = 6
sigma  = 4
```

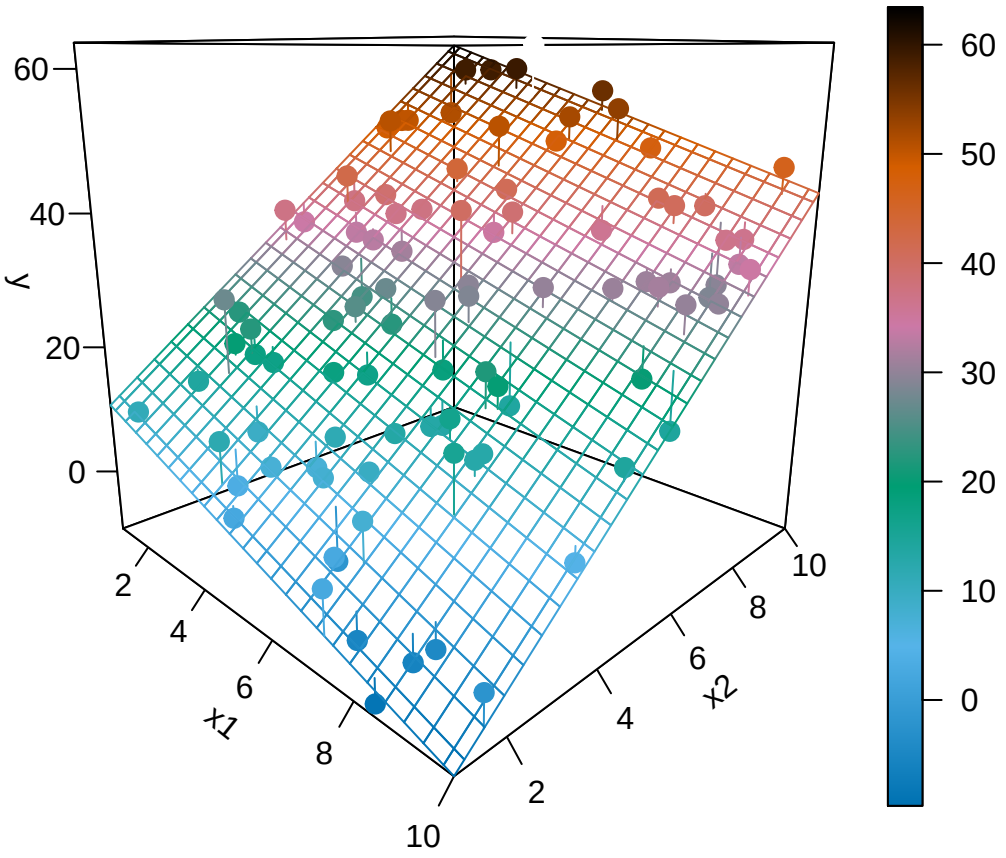
As is the norm with regression, the x values are considered fixed and known quantities, so we will simulate those first, and they remain the same for the rest of the simulation study. Also note we create an `x0` which is all `1`, which we need to create our `X` matrix. If you look at the matrix formulation of regression, this unit vector of all `1`s is a "predictor" that puts the intercept into the model. We also calculate the `C` matrix for later use.

```
x0 = rep(1, n)
x1 = sample(seq(1, 10, length = n))
x2 = sample(seq(1, 10, length = n))
X = cbind(x0, x1, x2)
C = solve(t(X) %*% X)
```

We then simulate the response according the model above. Lastly, we place the two predictors and response into a data frame. Note that we do **not** place `x0` in the data frame. This is a result of `R` adding an intercept by default.

```
eps      = rnorm(n, mean = 0, sd = sigma)
y        = beta_0 + beta_1 * x1 + beta_2 * x2 + eps
sim_data = data.frame(x1, x2, y)
```

Plotting this data and fitting the regression produces the following plot.



We then calculate

$$\hat{\beta} = \left(X^\top X\right)^{-1} X^\top y.$$

```
(beta_hat = C %*% t(X) %*% y)
```

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

```
## x0  7.290735
## x1 -2.282176
## x2  5.843424
```

Notice that these values are the same as the coefficients found using `lm()` in `R`.

```
coef(lm(y ~ x1 + x2, data = sim_data))
```

```
## (Intercept)          x1          x2
##    7.290735   -2.282176    5.843424
```

Also, these values are close to what we would expect.

```
c(beta_0, beta_1, beta_2)
```

```
## [1]  5 -2  6
```

We then calculated the fitted values in order to calculate s_e, which we see is the same as the `sigma` which is returned by `summary()`.

```
y_hat = X %*% beta_hat
(s_e = sqrt(sum((y - y_hat) ^ 2) / (n - p)))
```

```
## [1] 4.294307
```

```
summary(lm(y ~ x1 + x2, data = sim_data))$sigma
```

```
## [1] 4.294307
```

So far so good. Everything checks out. Now we will finally simulate from this model repeatedly in order to obtain an empirical distribution of $\hat{\beta}_2$.

We expect $\hat{\beta}_2$ to follow a normal distribution,

$$\hat{\beta}_2 \sim N\left(\beta_2, \sigma^2 C_{22}\right).$$

In this case,

$$\hat{\beta}_2 \sim N\left(\mu = 6, \sigma^2 = 16 \times 0.0014534 = 0.0232549\right).$$

$$\hat{\beta}_2 \sim N\left(\mu = 6, \sigma^2 = 0.0232549\right).$$

Note that C_{22} corresponds to the element in the **third** row and **third** column since β_2 is the **third** parameter in the model and because `R` is indexed starting at `1`. However, we index the C matrix starting at `0` to match the diagonal elements to the corresponding β_j.

```
C[3, 3]
```

```
## [1] 0.00145343
```

```
C[2 + 1, 2 + 1]
```

```
## [1] 0.00145343
```

```
sigma ^ 2 * C[2 + 1, 2 + 1]
```

```
## [1] 0.02325487
```

We now perform the simulation a large number of times. Each time, we update the `y` variable in the data frame, leaving the `x` variables the same. We then fit a model, and store $\hat{\beta}_2$.

```
num_sims = 10000
beta_hat_2 = rep(0, num_sims)
for(i in 1:num_sims) {
  eps           = rnorm(n, mean = 0 , sd = sigma)
  sim_data$y    = beta_0 * x0 + beta_1 * x1 + beta_2 * x2 + eps
  fit           = lm(y ~ x1 + x2, data = sim_data)
  beta_hat_2[i] = coef(fit)[3]
}
```

We then see that the mean of the simulated values is close to the true value of β_2.

```
mean(beta_hat_2)
```

```
## [1] 5.999723
```

```
beta_2
```

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

We also see that the variance of the simulated values is close to the true variance of $\hat{\beta}_2$.

$$\text{Var}[\hat{\beta}_2] = \sigma^2 \cdot C_{22} = 16 \times 0.0014534 = 0.0232549$$

```
var(beta_hat_2)
```

```
## [1] 0.02343408
```

```
sigma ^ 2 * C[2 + 1, 2 + 1]
```

```
## [1] 0.02325487
```

The standard deviations found from the simulated data and the parent population are also very close.

```
sd(beta_hat_2)
```

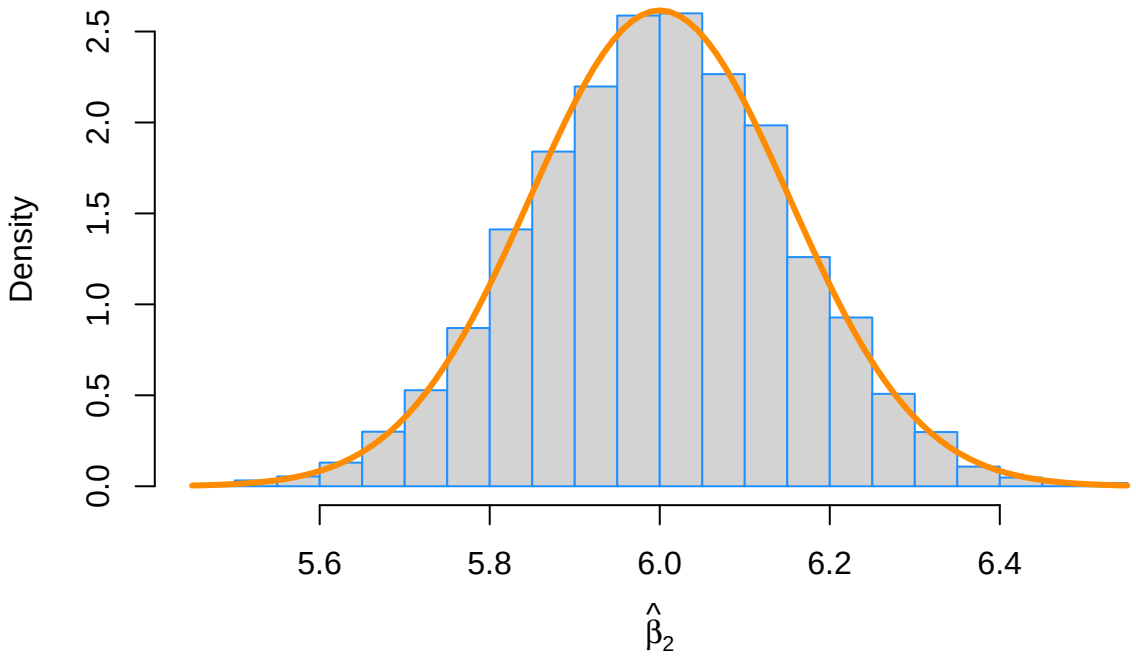
```
## [1] 0.1530819
```

```
sqrt(sigma ^ 2 * C[2 + 1, 2 + 1])
```

```
## [1] 0.1524955
```

Lastly, we plot a histogram of the *simulated values*, and overlay the *true distribution*.

```
hist(beta_hat_2, prob = TRUE, breaks = 20,
     xlab = expression(hat(beta)[2]), main = "", border = "dodgerblue")
curve(dnorm(x, mean = beta_2, sd = sqrt(sigma ^ 2 * C[2 + 1, 2 + 1])),
      col = "darkorange", add = TRUE, lwd = 3)
```



This looks good! The simulation-based histogram appears to be Normal with mean 6 and spread of about 0.15 as you measure from center to inflection point. That matches really well with the sampling distribution of $\hat{\beta}_2 \sim N\left(\mu = 6, \sigma^2 = 0.0232549\right)$.

One last check, we verify the $68 - 95 - 99.7$ rule.

```
sd_bh2 = sqrt(sigma ^ 2 * C[2 + 1, 2 + 1])
# We expect these to be: 0.68, 0.95, 0.997
mean(beta_2 - 1 * sd_bh2 < beta_hat_2 & beta_hat_2 < beta_2 + 1 * sd_bh2)
```

```
## [1] 0.6807
```

```
mean(beta_2 - 2 * sd_bh2 < beta_hat_2 & beta_hat_2 < beta_2 + 2 * sd_bh2)
```

```
## [1] 0.9529
```

```
mean(beta_2 - 3 * sd_bh2 < beta_hat_2 & beta_hat_2 < beta_2 + 3 * sd_bh2)
```

```
## [1] 0.9967
```