Data Analytics with R

Adam Smith, UCL SoM Rafael P. Greminger, UCL SoM

Table of contents

Preface 4

Getting Started with R 5

Why use R? 5

What does R do? 5

Downloading R and RStudio 6

Installing and Loading Packages 6

1 Introduction to Data Analytics 8

1.1 Loading Data Sets 8

1.1.1 Data formats 8

1.1.2 Reading in Data from R packages 8

1.1.3 Reading in Data from .csv files 9

1.2 The Working Directory 9

1.2.1 Using `setwd()` to change the working directory 9

1.2.2 Using the “Files” tab to change the working directory 10

1.2.3 Open a script to automatically set the working directory 10

1.2.4 Providing full file paths to read in data 12

1.2.5 Uploading data in RStudio/Posit cloud 12

1.3 Inspecting Data 12

1.4 Summary Statistics 13

1.5 Graphical Summaries 16

2 Computing Probabilities 27

2.1 Normal Distribution 27

2.1.1 Probability Density Function (`dnorm`) 28

2.1.2 Cumulative Distribution Function (`pnorm`) 28

2.1.3 Quantile Function (`qnorm`) 30

2.1.4 Random Number Generator (`rnorm`) 31

2.2 Bernoulli and Binomial Distributions 33

2.2.1 Probability Mass Function (`dbinom`) 33

2.2.2 Cumulative Distribution Function (`pbinom`) 34

2.2.3 Quantile Function (`qbinom`) 35

2.2.4 Random Number Generator (`rbinom`) 35

3 Sampling Distributions and the CLT 38
3.1 Sampling Distributions . . . 38
3.2 Central Limit Theorem (CLT) . . . 41

4 Estimation 44
4.1 Point Estimation . . . 44
4.2 Confidence Intervals . . . 45
4.2.1 Types of Confidence Intervals . . . 46
4.2.2 Interpreting confidence intervals . . . 48
4.2.3 Examples . . . 48

5 Hypothesis Tests 52
5.1 Steps of Hypothesis Testing . . . 52
5.2 Connection to Confidence Intervals . . . 56
5.3 Hypothesis Testing in R . . . 58

6 Regression 60
6.1 Linear Regression . . . 60
6.2 Regression Trees . . . 63
6.3 Model Selection . . . 66

7 Classification 69
7.1 k-Nearest Neighbors . . . 70
7.2 Logistic Regression . . . 73
7.3 Classification Trees . . . 76

8 Clustering 78
8.1 k-Means . . . 79
8.2 Hierarchical Clustering . . . 83

Preface

This book is written for use in MSIN0010: Data Analytics I at the UCL School of Management. It is meant to serve as a supplement to lecture and seminar materials and specifically focuses on applications in R.

Last update: October 2023

Getting Started with R

Why use R?

R is a statistical software environment that is widely used by statisticians, social scientists, and data analysts. R is different from "point-and-click" software packages like Microsoft Excel, SPSS, or Tableau in that it requires the user to write code via a command line interface. For this reason, R is also often referred to as a programming language. However, the R environment is much more interactive than other programming languages like C or Java which makes it easier to learn and use.

Here are a few key advantages of R.

- R is **free** and **open source**[1].
- R is available on Windows, Mac OS, and UNIX/Linux.
- R is **flexible**: you can write, modify, save, and share your own code.
- R is **powerful**: you can do everything from making high-quality graphics to running sophisticated statistical machine learning models.
- R is **popular**: there is a large and growing online community of users making it easy to find answers to any problem you run into.

Lastly, learning R is a tangible and highly-valued skill you can put on your CV!

What does R do?

At its core, R (and any other programming language) just translates human-written code into instructions that the computer understands, and then has the computer execute it for us. For example, to assign the number 2 to the variable `a` we just type

```
a <- 2
```

R then translates this instruction into code that looks something like this (which is Assembly code):

[1] https://opensource.org

```
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $1, %eax
    popq    %rbp
    retq
    nopl    (%rax,%rax)
```

Clearly, it is much more convenient to write in R code than in these complicated instructions! Besides, the above will be even further translated into zeros and ones, which would be impossible for us to write instructions in.

Downloading R and RStudio

To download R, go to https://cran.rstudio.com/ and choose "Download R for …" your operating system (Windows, Mac, or Linux). For other questions, see the R FAQ.

After downloading R, you should also download RStudio, which is an integrated development enviroment (IDE) for R. Whereas we could use any basic text editor to write code for R, and IDE like Rstudio provides a much more interactive and user-friendly interface for using R. To download it, go here and download the installer using the link "DOWNLOAD RSTDIO FOR …" for your operating system.

An alternative to run R locally on your computer is to run it in the cloud. By registering on Posit cloud, you can get the Rstudio experience directly in your browser, without the need to install anything on your computer. However, this can only be used while connected to the internet, and the free account has resource limitations (e.g. RAM).

Installing and Loading Packages

As you will see, one of the most attractive features of R is its library of over 10,000 packages. R packages – which are collections of R functions, code, and data sets – allow us to use code written by others in order to use certain data sets, make certain graphs, or run certain models.

For example, in this course we will discuss a variety of regression models including linear regression models and regression trees. While the R function to estimate a linear regression model (called `lm`) is included in "base" R, the function to estimate a regression tree is not. Rather than writing the code ourselves, we can download an R package!

One package that provides code for estimating regression trees is called `rpart`. To use functions within the `rpart` package, we must first install it.

```
install.packages("rpart")
```

Alternatively, you can navigate to the “Packages” tab in RStudio (likely in the lower right panel), click “Install”, and search for **rpart**.

Note: You only need to install a package once! After a package is installed, it will remain installed until you upgrade your version of R/RStudio.

However, in each R session (i.e., each time you open RStudio), you will need to *load the package*.

```
library("rpart")
```

Again, an alternative is to navigate to the “Packages” tab in RStudio, find the package name, and click on the box to the left of the name.

1 Introduction to Data Analytics

In this chapter, we will get acclimated to working with data using a suite of packages in R called the `tidyverse`.[1] If you are interested in a complete introduction to the tidyverse syntax, see R for Data Science. Specifically, for details on data visualization see Chapter 3 - Data Visualization and Chapter 7 - Exploratory Data Analysis.

To use the `tidyverse` packages, we first need to load them in R. Without running this command first, any of the functions used below that are part of `tidyverse` will produce an error.

```
library(tidyverse)
```

1.1 Loading Data Sets

1.1.1 Data formats

Data sets come in different storage formats:

- .csv: Commas separate the values in each row.
- .xls: Excel spreadsheet.
- .txt: Text files
- As part of R package.

R can read in data from most (if not all) of these formats. In our examples, we will use data set from R packages and .csv files.

1.1.2 Reading in Data from R packages

To read in data from an R package, we use the `data()` function. For example, the `ISLR` provides several data sets. To read in the `OJ` data, we simply run `data(OJ)`. As the data is part of a package, don't forget to first load the package.

```
library(ISLR)
data(OJ)
```

[1] https://www.tidyverse.org/

1.1.3 Reading in Data from .csv files

To read in data from .csv files, we will use the `read_csv()` function, which is provided by one of the packages in `tidyverse`.

The following line of code reads in a data set that contains weekly prices, promotional activity, and sales for 20 different brands of beer. The data set comes from many stores within one Chicago grocery retail chain – Dominick's Finer Foods – and spans more than five years of transactions. The complete raw data are publically available from the Kilts Center for Marketing at the University of Chicago.[2]

```
beer <- read_csv("beer.csv")
```

1.2 The Working Directory

For above code to work, the data file `beer.csv` needs to be located in the current **working directory** of R. This is the directory that R uses as a default to look for files. To see what your current working directory is, we can use the `getwd()` function, which is short for "get working directory". The output is the path to the folder that R is currently looking in for files.

```
getwd()
```

```
[1] "C:/Users/Rafael/Documents"
```

1.2.1 Using `setwd()` to change the working directory

We can change the working directory with `setwd("PATH_TO_NEW_WORKING_DIRECTORY")`, where `NEWPATH` is the path to the new working directory. Note that we must use `/` instead of `\` in the path name, otherwise R will not be able to find the folder and give an error.

```
setwd("C:\Users\Rafael\Documents")
```

```
Error: '\U' used without hex digits in character string (<text>:1:11)
```

[2] https://www.chicagobooth.edu/research/kilts/datasets/dominicks

Exercise 1: Change the working directory to your downloads folder, and use the `getwd()` function to confirm that the working directory has changed.

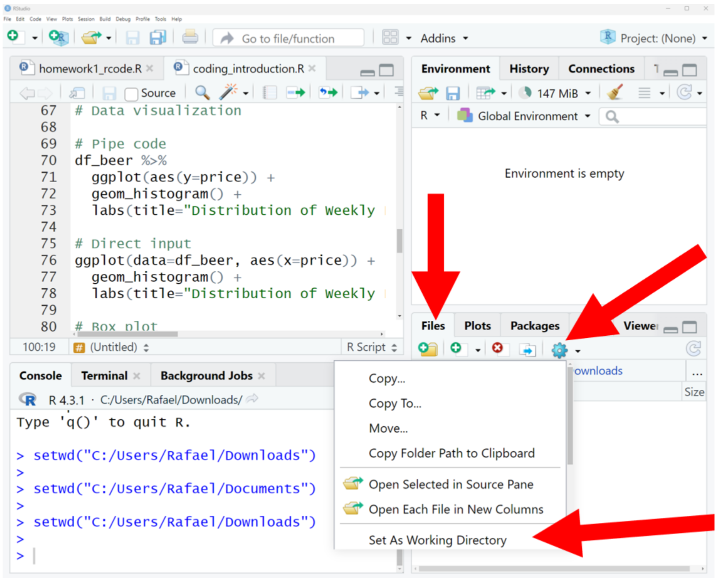
Solution:

```
setwd("C:/Users/Rafael/Downloads")
getwd()
```

```
[1] "C:/Users/Rafael/Downloads"
```

1.2.2 Using the "Files" tab to change the working directory

Another way to change the working directory is to use the "Files" tab in the bottom right panel of RStudio. Navigate to the folder you want to use as your working directory, and click "More" and then "Set As Working Directory".



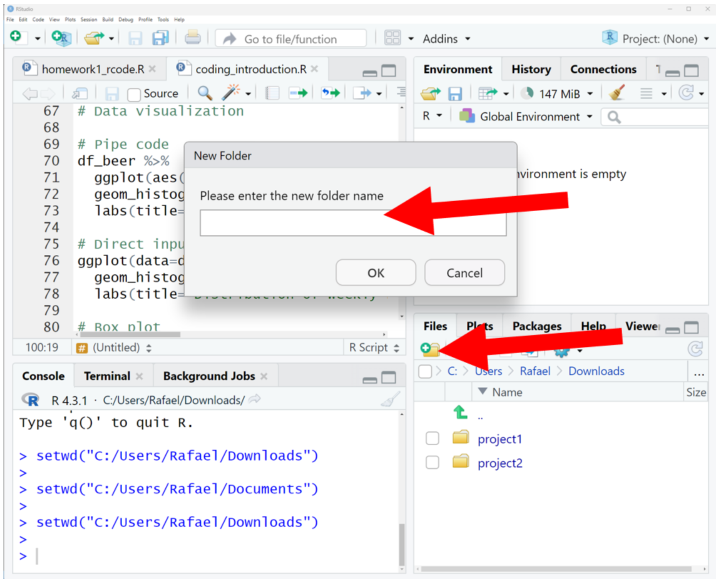
```
# Data visualization

# Pipe code
df_beer %>%
  ggplot(aes(y=price)) +
  geom_histogram() +
  labs(title="Distribution of Weekly

# Direct input
ggplot(data=df_beer, aes(x=price)) +
  geom_histogram() +
  labs(title="Distribution of Weekly

# Box plot
```

```
R 4.3.1 · C:/Users/Rafael/Downloads/
Type 'q()' to quit R.

> setwd("C:/Users/Rafael/Downloads")
>
> setwd("C:/Users/Rafael/Documents")
>
> setwd("C:/Users/Rafael/Downloads")
>
> 
```

1.2.3 Open a script to automatically set the working directory

Another way to set the working directory is to open a .R file with RStudio. RStudio will automatically set the working directory to the folder where the .R file is located. This makes it very convenient to keep all of your code and data files in the same folder.

To make use of this, you should create a new folder for every project (or assignment). You can do this directly in RStudio by going to the "File" tab, navigating to whereever you want the new folder to be, and then clicking “Create a new folder”. This will open a window that allows you to create a new folder and name it.



```
# Data visualization

# Pipe code
df_beer %>%
  ggplot(aes(
  geom_histog
  labs(title=

# Direct inpu
ggplot(data=d
  geom_histog
  labs(title=

# Box plot
```

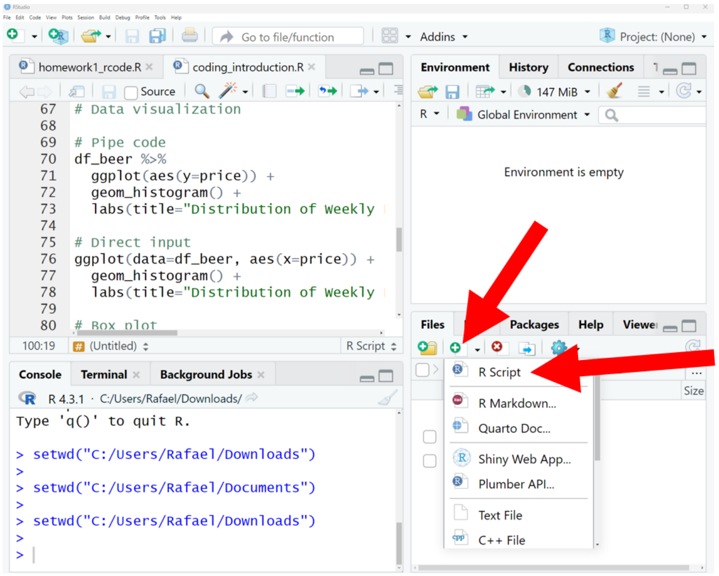
New Folder

Please enter the new folder name

```
R 4.3.1 · C:/Users/Rafael/Downloads/
Type 'q()' to quit R.

> setwd("C:/Users/Rafael/Downloads")
>
> setwd("C:/Users/Rafael/Documents")
>
> setwd("C:/Users/Rafael/Downloads")
>
> |
```

Then, you can create a new .R file by going to the “File” tab, clicking “New File”, and then “R Script”. This will allow you to name and save a new script in the folder. Then, whenever you open this script with a new RStudio session, RStudio will automatically set the working directory to the folder where the script is located.

IMPORTANT: This only works if all instances of RStudio have been closed! If you have another RStudio session open, it will not automatically set the working directory.



```
# Data visualization

# Pipe code
df_beer %>%
  ggplot(aes(y=price)) +
  geom_histogram() +
  labs(title="Distribution of Weekly

# Direct input
ggplot(data=df_beer, aes(x=price)) +
  geom_histogram() +
  labs(title="Distribution of Weekly

# Box plot
```

```
R 4.3.1 · C:/Users/Rafael/Downloads/
Type 'q()' to quit R.

> setwd("C:/Users/Rafael/Downloads")
>
> setwd("C:/Users/Rafael/Documents")
>
> setwd("C:/Users/Rafael/Downloads")
>
> |
```

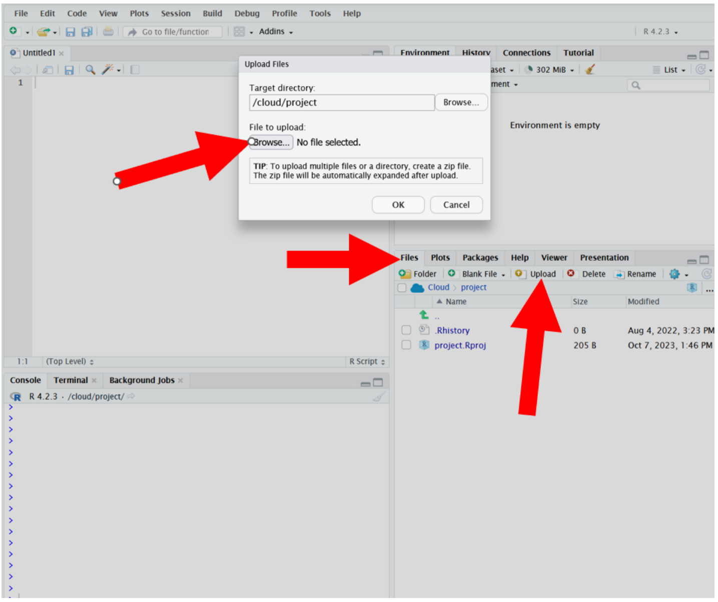
1.2.4 Providing full file paths to read in data

Instead of changing the working directory, we can also specify the full file path to the `beer.csv` data we want to load. We can do this by running `read_csv("PATH/beer.csv")`, where we replace `PATH` with the path to the `beer.csv` file. Note again that we need to use `/` and not `\` in the file path.

```
beer <- read_csv("C:/Users/Rafael/Documents/beer.csv")
```

1.2.5 Uploading data in RStudio/Posit cloud

If you are using RStudio in the cloud, you need to upload the .csv data file to the cloud before you can read it in. To do this, click on the “Upload” button in the “Files” tab, and then select the file you want to upload using the file browser.



Upload Files

Target directory: /cloud/project

File to upload: No file selected.

TIP: To upload multiple files or a directory, create a zip file. The zip file will be automatically expanded after upload.

Environment is empty

Name	Size	Modified
..		
.Rhistory	0 B	Aug 4, 2022, 3:23 PM
project.Rproj	205 B	Oct 7, 2023, 1:46 PM

R 4.2.3 · /cloud/project/

1.3 Inspecting Data

We always want to view the data after importing to make sure all the values were read-in correctly. To inspect the first few lines of a data set, use the `head( )` command.

```
head(beer,3)
```

store	upc	week	move	price	sale	profit	brand	packsize	itemsize	units
86	1820000016	91	23	3.49	NA	19.05	BUDWEISER BEER	6	12	oz
86	1820000784	91	9	3.79	NA	28.23	O’DOUL’S N/A LONGNEC	6	12	oz
86	1820000834	91	9	3.69	NA	22.03	BUDWEISER BEER LONG	6	12	oz
86	1820000987	91	78	3.29	B	5.78	MICHELOB REGULAR BEE	6	12	oz
86	3410000354	91	35	3.69	NA	22.98	MILLER LITE BEER	6	12	oz
86	3410000554	91	12	3.69	NA	22.98	MILLER GENUINE DRAFT	6	12	oz

Statistic	R Function
mean	‘mean( )‘
median	‘median( )‘
variance	‘var( )‘
standard deviation	‘sd( )‘
correlation	‘cor( )‘

We can see that our data set contains 11 different variables (i.e., columns). A brief summary of each variable is provided below.

- `store`: unique store ID number
- `upc`: Universal Product Code
- `week`: week ID number
- `move`: number of units sold
- `price`: retail price in US dollars
- `sale`: indicator of promotional activity
- `profit`: gross profit margin
- `brand`: brand name
- `packsize`: number of items in one package
- `itemsize`: size of items in one package
- `units`: units of items

1.4 Summary Statistics

We can compute summary statistics in the tidyverse by combining the `summarise` operator with any one (or many!) of R’s built-in statistics functions. A few of the most common are listed below.

For example, let’s compute the average price across all products and weeks. For this, we’re going to use the pipe operator `%>%`, which works by putting together a function from the left to right. Below, the dataset `beer` is put as an argument into the `summarise()` function. An alternative way of writing the same function would be `summarise(beer,mean(price))`.

```
beer %>%
  summarise(mean(price))
```

```
# A tibble: 1 x 1
  `mean(price)`
          <dbl>
1          4.28
```

Now suppose we wanted to find the average price for only one brand of beer, say Budweiser. To do this, we can use the `filter( )` operator to select rows in the data that satisfy certain conditions. Here we want Budweiser beers so the condition is that `brand` is equal to `BUDWEISER BEER`, or `brand=="BUDWEISER BEER"`. Note that a double equals sign `==` is always used when writing logical statements to check equality. As above, we again use `%>%` to put the different commands together.

```
beer %>%
  filter(brand=="BUDWEISER BEER") %>%
  summarise(mean(price))
```

```
# A tibble: 1 x 1
  `mean(price)`
          <dbl>
1          3.81
```

To compute summary statistics for multiple brands, we can use the `group_by( )` operator. As the name suggests, this operator tells R to first group by a certain categorical variable, and to compute a summary for each level that the given variable takes on.

```
beer %>%
  group_by(brand) %>%
  summarise(mean(price))
```

```
# A tibble: 19 x 2
   brand                 `mean(price)`
   <chr>                         <dbl>
 1 BECK'S REG BEER NR B           5.88
 2 BERGHOFF REGULAR BEE           3.94
 3 BUDWEISER BEER                 3.81
 4 BUDWEISER BEER LONG            3.75
 5 CORONA EXTRA BEER NR          5.80
 6 HEINEKEN BEER N.R.BT          6.34
 7 LOWENBRAU BEER NR BT          4.05
 8 MICHELOB REGULAR BEE          4.04
 9 MILLER GEN DRFT LNNR          3.69
10 MILLER GEN DRFT LT L          3.69
11 MILLER GENUINE DRAFT          3.78
12 MILLER HIGH LIFE LNN          3.68
13 MILLER LITE BEER              3.82
14 MILLER LITE BEER N.R          3.74
15 MILLER LITE LONGNECK          3.69
16 MILLER SHARP'S N/A L          3.36
17 O'DOUL'S N/A LONGNEC          3.78
18 OLD STYLE BEER                3.68
19 SAMUEL ADAMS LAGER N          5.41
```

We can also easily extend the code above to compute multiple summary statistics across groups.

```
beer %>%
  group_by(brand) %>%
  summarise(mean(price), mean(move))
```

```
# A tibble: 19 x 3
   brand                `mean(price)` `mean(move)`
   <chr>                        <dbl>        <dbl>
 1 BECK'S REG BEER NR B          5.88         18.3
 2 BERGHOFF REGULAR BEE          3.94         15.6
 3 BUDWEISER BEER                3.81         16.3
 4 BUDWEISER BEER LONG           3.75         18.2
 5 CORONA EXTRA BEER NR          5.80         15.4
 6 HEINEKEN BEER N.R.BT          6.34         16.7
 7 LOWENBRAU BEER NR BT          4.05         16.9
 8 MICHELOB REGULAR BEE          4.04         14.2
 9 MILLER GEN DRFT LNNR          3.69         51.0
10 MILLER GEN DRFT LT L          3.69         20.1
11 MILLER GENUINE DRAFT          3.78         16.4
12 MILLER HIGH LIFE LNN          3.68         14.1
13 MILLER LITE BEER              3.82         18.1
14 MILLER LITE BEER N.R          3.74         18.7
15 MILLER LITE LONGNECK          3.69         38.4
16 MILLER SHARP'S N/A L          3.36         11.5
```

Graph	Operator
histogram	‘geom_histogram( )‘
box plot	‘geom_boxplot( )‘
bar plot	‘geom_bar( )‘
line plot	‘geom_line( )‘
scatter plot	‘geom_point( )‘

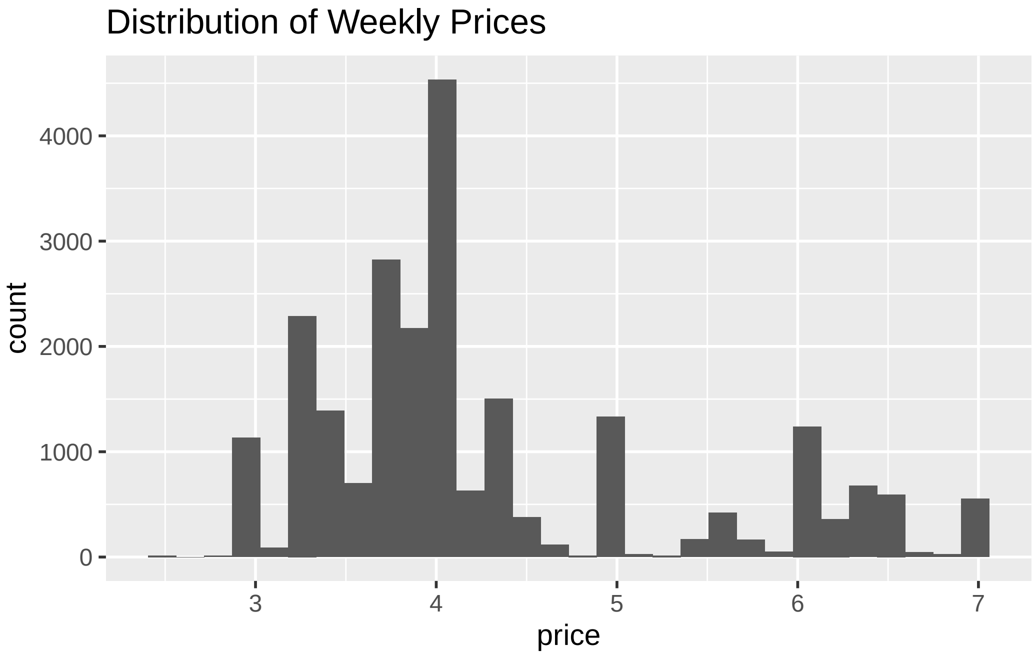
```
17 O'DOUL'S N/A LONGNEC          3.78          12.0
18 OLD STYLE BEER                3.68          13.4
19 SAMUEL ADAMS LAGER N          5.41          20.6
```

1.5 Graphical Summaries

Data visualization is one of the strengths of the tidyverse. A fairly exhaustive list of graph types can be found at https://www.r-graph-gallery.com. For our purposes, we will start with a few of the most commonly used graphs.

Let's start be looking at the distribution of weekly prices across all products.

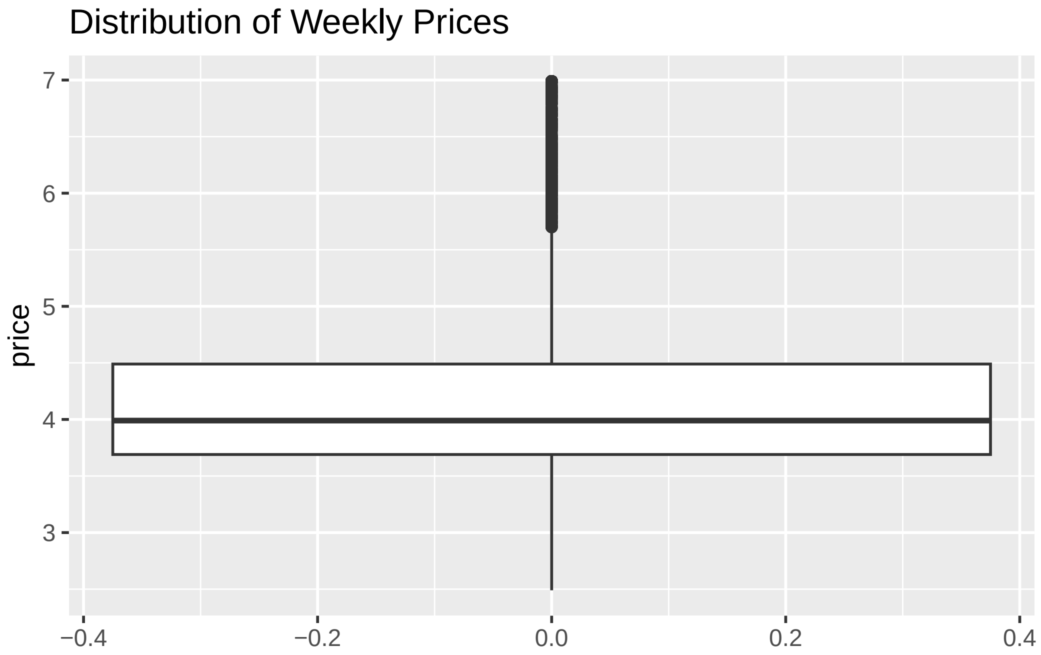
```
beer %>%
  ggplot(aes(x=price)) +
  geom_histogram() +
  labs(title="Distribution of Weekly Prices")
```



This is an example of a **histogram**. The variable on the x-axis (price) is split into different bins, and the y-axis counts the number of observations that fall withiin each bin.

We can also inspect the distribution of a variable like prices using a **boxplot**.

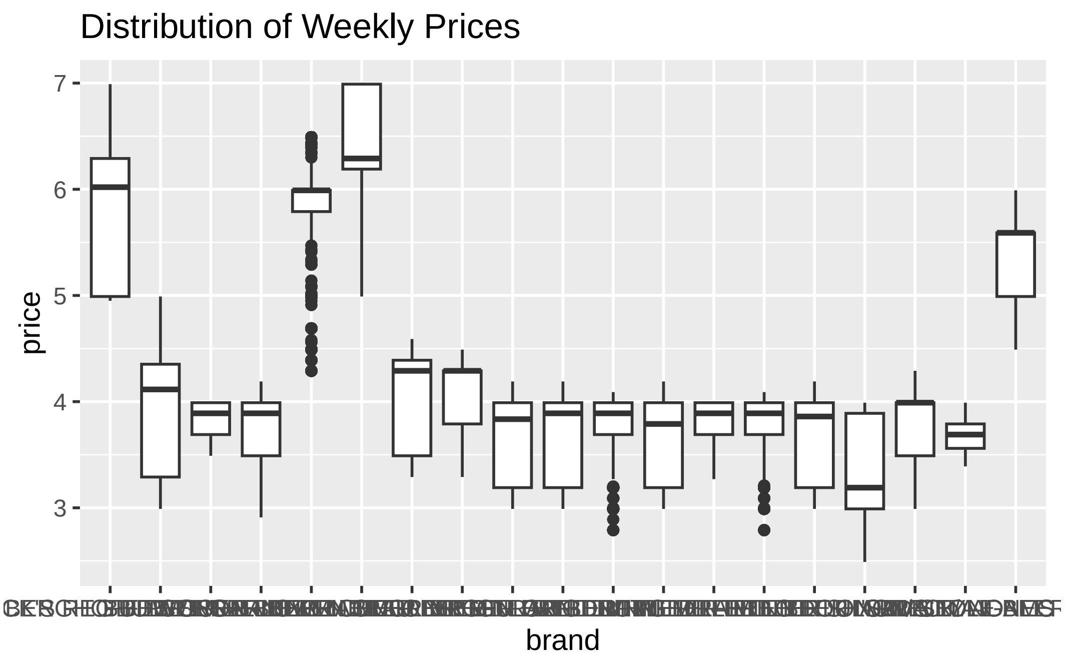
```
beer %>%
  ggplot(aes(y=price)) +
  geom_boxplot() +
  labs(title="Distribution of Weekly Prices")
```



Notice that the focal variable of interest is now on the y-axis. The rectangular box shown in the middle of the plot indicates three key summary statistics: the bottom line is the 25th percentile, the middle line is the 50th percentile (or median), and the top line is the 75th percentile. The vertical line starting around 2.5 and ending around 7 indicates the full range of prices in the data.

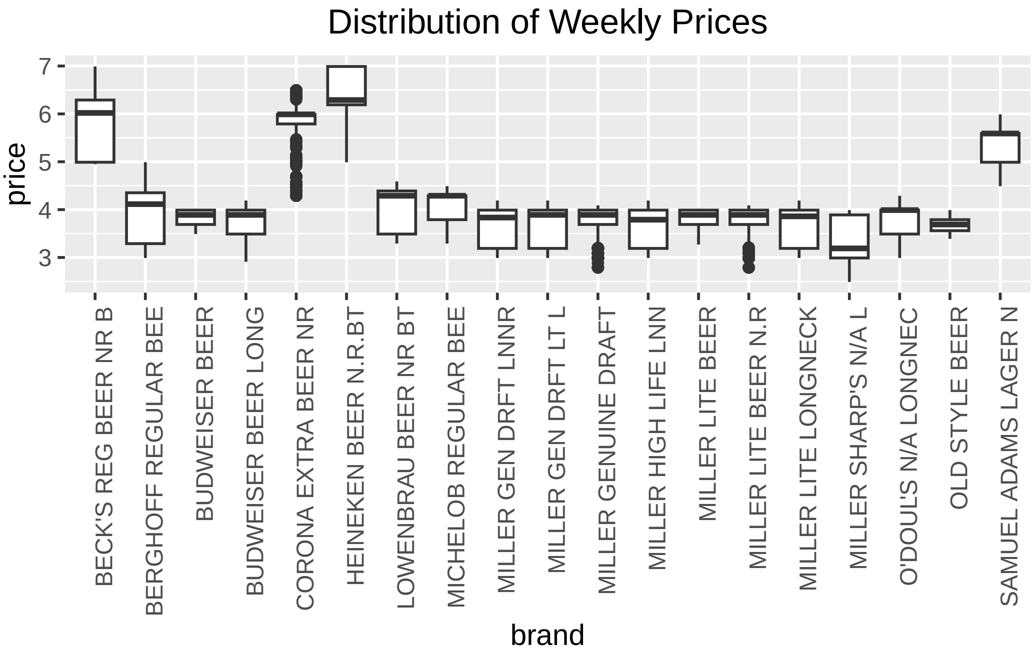
The figure above shows the distribution of prices across all prodcuts. However, we may want to explore whether the distribution of prices is *different across products*. This can be done by defining `x=brand` within the `ggplot( )` function.

```
beer %>%
  ggplot(aes(x=brand, y=price)) +
  geom_boxplot() +
  labs(title="Distribution of Weekly Prices")
```



We have succeeded in creating a box plot for each brand, but the brand labels are impossible to read! As easy fix is to rotate the x-axis labels, which can be controlled using the `theme()` operator. The `theme()` operator is generally what we use to change plot formatting, such as the size of the axis labels or the position of the title.

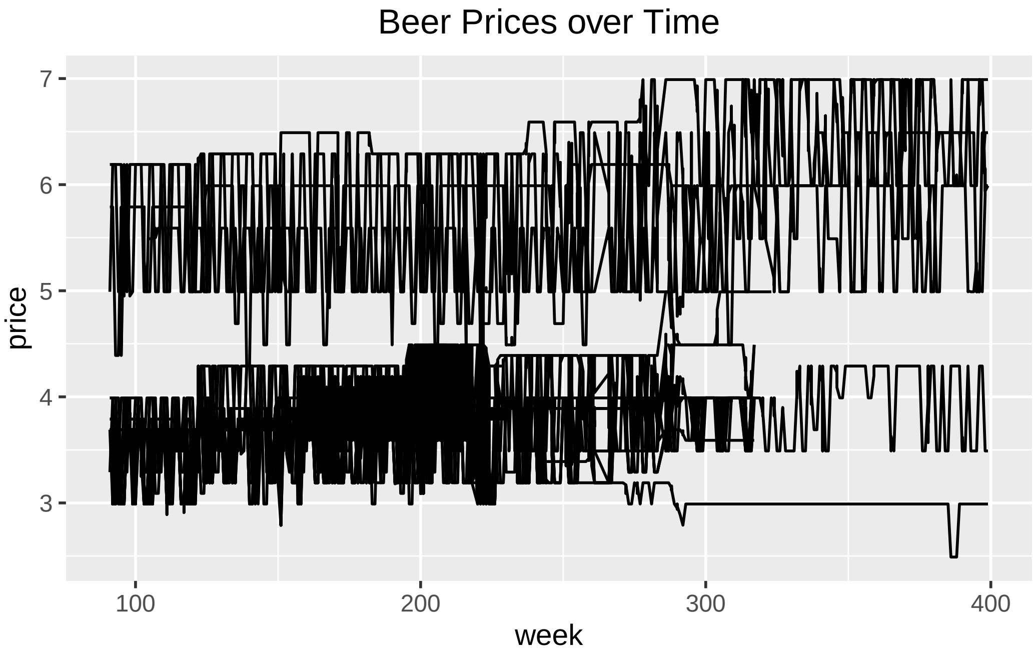
```
beer %>%
  ggplot(aes(x=brand, y=price)) +
  geom_boxplot() +
  labs(title="Distribution of Weekly Prices") +
  theme(axis.text.x = element_text(angle=90, hjust=1),
        plot.title = element_text(hjust=0.5))
```



Much better!

Next, let's explore the variation of prices over time. We can make a time series plot (or line plot) to do this, where we specify `group=brand` so that R makes a separate line for each brand.

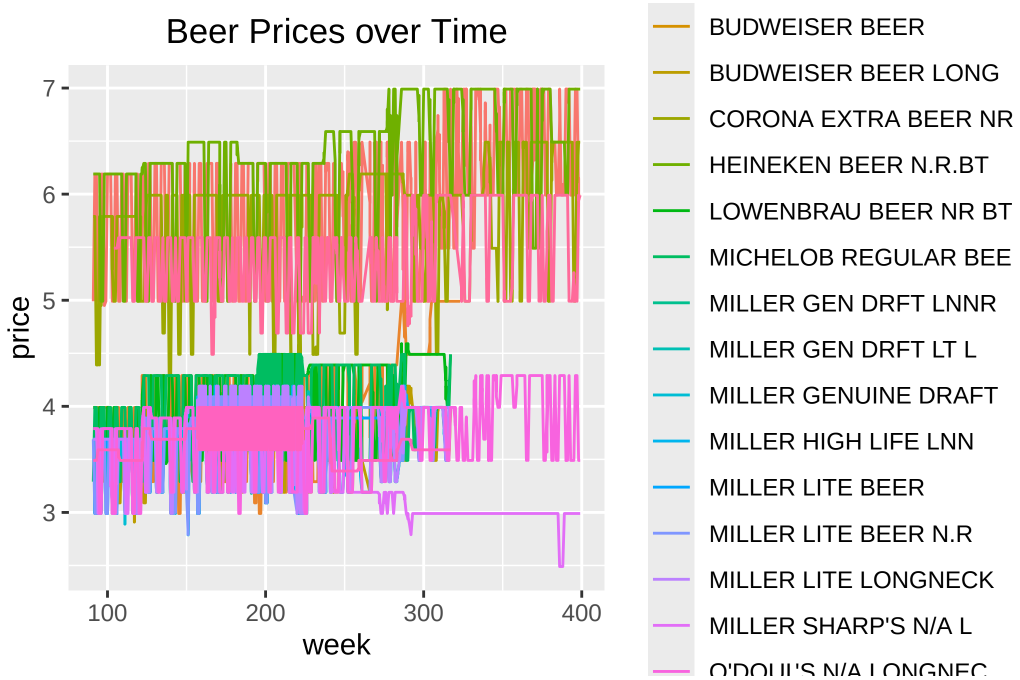
```
beer %>%
  ggplot(aes(x=week, y=price, group=brand)) +
  geom_line() +
  labs(title="Beer Prices over Time") +
  theme(plot.title = element_text(hjust=0.5))
```



This was a good attempt, but the plot is not especially useful! While we do notice that the prices are changing over time, we can't identify the products themselves so we don't know which products are changing more or less than others.

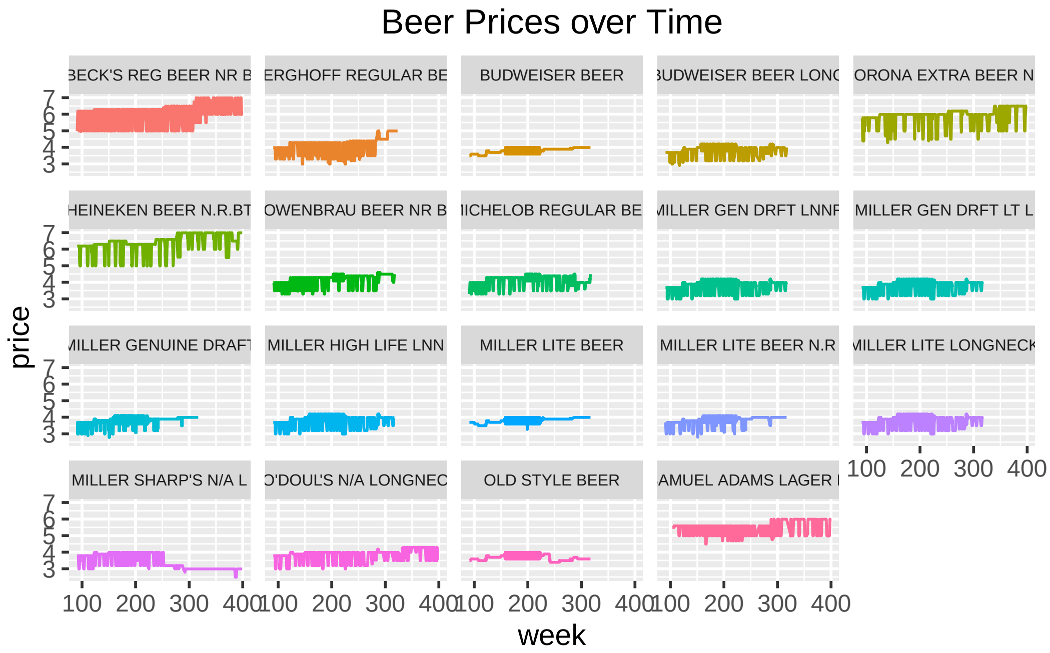
We can fix this in a couple ways. The first thing we will try is to simply add color to the plot above, so that we can identify a product by the color of its line.

```
beer %>%
  ggplot(aes(x=week, y=price, group=brand, color=brand)) +
  geom_line() +
  labs(title="Beer Prices over Time") +
  theme(plot.title = element_text(hjust=0.5))
```



This is better, but it is still hard to identify products because of how much overlap there is in prices. So maybe the best thing to do is create a separate plot for each brand. This can be easily accomplished using `facet_wrap( )`.

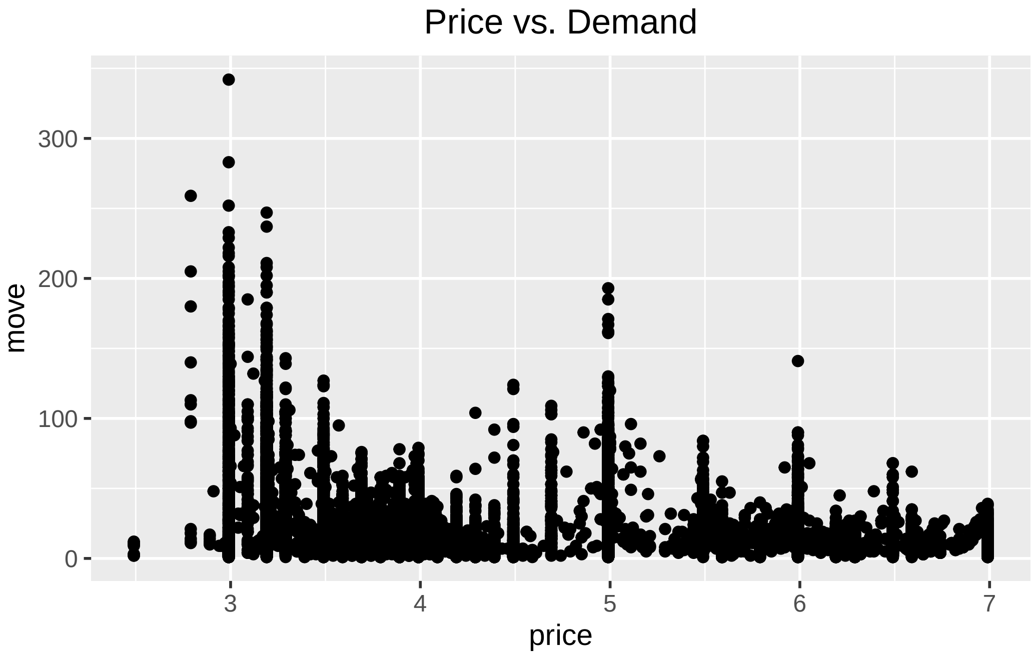
```
beer %>%
  ggplot(aes(x=week,y=price,group=brand,color=brand)) +
  geom_line(show.legend=FALSE) +
  labs(title="Beer Prices over Time") +
  facet_wrap(brand ~ .) +
  theme(plot.title = element_text(hjust=0.5),
        strip.text.x = element_text(size=6))
```



Note that we added `show.legend=FALSE` to `geom_line( )` since we no longer need the color to identify products. We also added an option to `theme( )` to control the size of the text to ensure that the labels are all legible.

Finally, let's explore the relationship between *two variables* like price and demand.

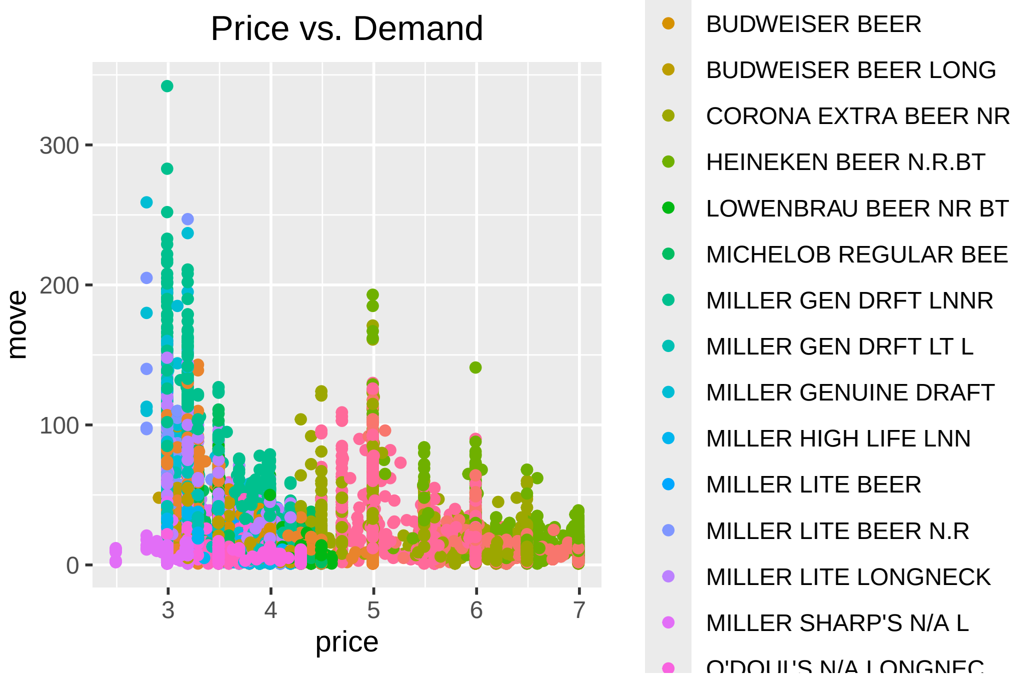
```
beer %>%
  ggplot(aes(x=price, y=move)) +
  geom_point() +
  labs(title="Price vs. Demand") +
  theme(plot.title = element_text(hjust=0.5))
```



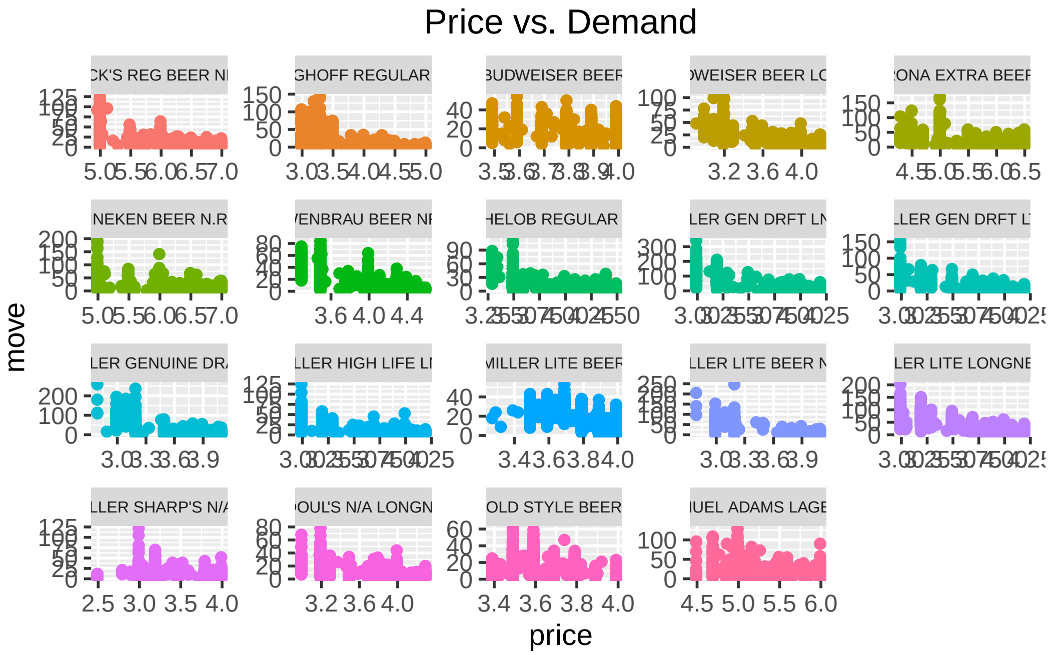
This figure matches our intuition from economics, which is that as price increases, demand seems to fall. We can even imagine a line going through these points – this line would be a demand curve!

As before, it would be interesting to know how the relationship between price and demand changes across products. Let's apply the same techniques above – adding color and using separate plots – to investigate.

```
beer %>%
  ggplot(aes(x=price, y=move, color=brand)) +
  geom_point() +
  labs(title="Price vs. Demand") +
  theme(plot.title = element_text(hjust=0.5))
```



```
beer %>%
  ggplot(aes(x=price, y=move, color=brand)) +
  geom_point(show.legend=FALSE) +
  labs(title="Price vs. Demand") +
  facet_wrap(brand~.,scales="free") +
  theme(plot.title = element_text(hjust=0.5),
        strip.text.x = element_text(size=6))
```



These last two plots indeed show that demand is negatively related to price (as price increases, demand falls) and that the magnitude of this relationship may change across products.

2 Computing Probabilities

There are many common families of probability distributions and we have discussed six so far. The discrete distributions include the discrete Uniform, Bernoulli, and Binomial. The continuous distributions include the continuous Uniform, Normal, and t.

This chapter provides a set of examples to show you how to compute probabilities from a few of these distributions in R.

2.1 Normal Distribution

R has four normal distribution functions: `dnorm( )`, `pnorm( )`, `qnorm( )`, and `rnorm( )`.

`dnorm(x,mean,sd)` probability density function (PDF) - *input*: `x` is the value at which you want to evaluate the normal PDF - *output*: a positive number since the PDF $f(x)$ must be positive - *example*: evaluate $f(x)$

`pnorm(q,mean,sd)` cumulative distribution function (CDF) - *input*: `q` is the value for which you want to find the area below/above - *output*: a probability - *example*: compute $P(X < q)$

`qnorm(p,mean,sd)` quantile function - *input*: `p` is a probability - *output*: a real number since $X \in (-\infty, \infty)$ - *example*: find the value q such that $P(X < q) = p$

`rnorm(n,mean,sd)` random number generator - *input*: `n` is the number of observations you want to generate - *output*: a vector of n real numbers - *example*: generate n independent $N(\mu, \sigma^2)$ random variables

More information is also accessible in R if you type `?dnorm`, `?pnorm`, `?qnorm`, or `?rnorm`.

To learn how to use these functions, we'll start with a few exercises on the **standard normal distribution** which is a normal distribution with mean 0 and standard deviation of 1. We will then move on to the more general $N(\mu, \sigma^2)$ distribution.

2.1.1 Probability Density Function (`dnorm`)

When X is a continuous random variable, we know that $P(X = x) = 0$. Therefore, `dnorm( )` does not return a probability, but rather the *height* of the PDF. Even though the height of the PDF is not a probability, we can still interpret density evaluations as the relatively likelihood of observing a certain value x.

PROBLEM 1: Let $X \sim N(0, 1)$. Is the value $x = 1$ or $x = -0.5$ more likely to occur under this normal distribution?

Solution:

```
dnorm(1, mean=0, sd=1)
```

```
[1] 0.2419707
```

```
dnorm(-0.5, mean=0, sd=1)
```

```
[1] 0.3520653
```

The results show that $x = -0.5$ is more likely, since $f(-0.5) > f(1)$. This should be expected because we know that density function is symmetric and peaks at the mean value which is 0 here. Since $x = -0.5$ is closer to 0 than $x = 1$, it should have higher likelihood under $N(0, 1)$ distribution.

2.1.2 Cumulative Distribution Function (`pnorm`)

The `pnorm( )` function is useful for evaluating probabilities of the form $P(X \leq x)$ or $P(X \geq x)$.

PROBLEM 2: If $X \sim N(0, 1)$, what is $P(X < 0)$?

Solution:

```
pnorm(0, mean=0, sd=1)
```

```
[1] 0.5
```

PROBLEM 3: If $X \sim N(0, 1)$, what is $P(X < 1)$?

Solution:

```
pnorm(1, mean=0, sd=1)
```

```
[1] 0.8413447
```

PROBLEM 4: If $X \sim N(0,1)$, what is $P(X > 1)$?

Solution:

We have two ways of answering this question. First, we can recognize that $P(X > 1) = 1 - P(X \geq 1)$.

```
1-pnorm(1, mean=0, sd=1)
```

```
[1] 0.1586553
```

A second approach is to use the `lower.tail` option within the `pnorm( )` function. When `lower.tail=TRUE` then the `pnorm( )` function returns the probability to the *left* of a given number x and if `lower.tail=FALSE` then `pnorm( )` returns the probability to the *right* of x.

```
pnorm(1, mean=0, sd=1, lower.tail=FALSE)
```

```
[1] 0.1586553
```

PROBLEM 5: If $X \sim N(0,1)$, what is $P(0 < X < 1)$

Solution:

```
pnorm(1, mean=0, sd=1) - pnorm(0, mean=0, sd=1)
```

```
[1] 0.3413447
```

Once we understand how to use the `pnorm( )` function to compute standard normal probabilities, extending the function to compute probabilities of any normal distribution is straightforward. All we have to do is change the `mean` and `sd` arguments.

Remember that the normal functions in R call for the standard deviation σ, NOT the variance σ^2!

PROBLEM 6: If $X \sim N(4,9)$, what is $P(X < 0)$?

Solution:

```
pnorm(0, mean=4, sd=3)
```

```
[1] 0.09121122
```

PROBLEM 7: If $X \sim N(2, 3)$, what is $P(X > 5)$?

Solution:

```
pnorm(5, mean=2, sd=sqrt(3), lower.tail=FALSE)
```

```
[1] 0.04163226
```

2.1.3 Quantile Function (qnorm)

Next, let's use the **qnorm( )** function to find quantiles of the normal distribution.

PROBLEM 8: If $X \sim N(0, 1)$, find the value q such that $P(X < q) = 0.05$.

Solution:

```
qnorm(0.05, mean=0, sd=1)
```

```
[1] -1.644854
```

PROBLEM 9: If $X \sim N(0, 1)$, find the value q such that $P(X > q) = 0.025$. That is, q is the value such that 2.5% of the area under the standard normal PDF is to its right.

Solution:

```
qnorm(0.025, mean=0, sd=1, lower.tail=FALSE)
```

```
[1] 1.959964
```

PROBLEM 10: If $X \sim N(-4, 2)$, find the value q such that $P(X > q) = 0.1$. That is, q is the value such that 10% of the area under the $N(-4, 2)$ PDF is to its right.

Solution:

```
qnorm(0.1, mean=-4, sd=sqrt(2), lower.tail=FALSE)
```

```
[1] -2.187612
```

2.1.4 Random Number Generator (rnorm)

Finally, let's use `rnorm( )` to generate random samples of size n from a normal distribution.

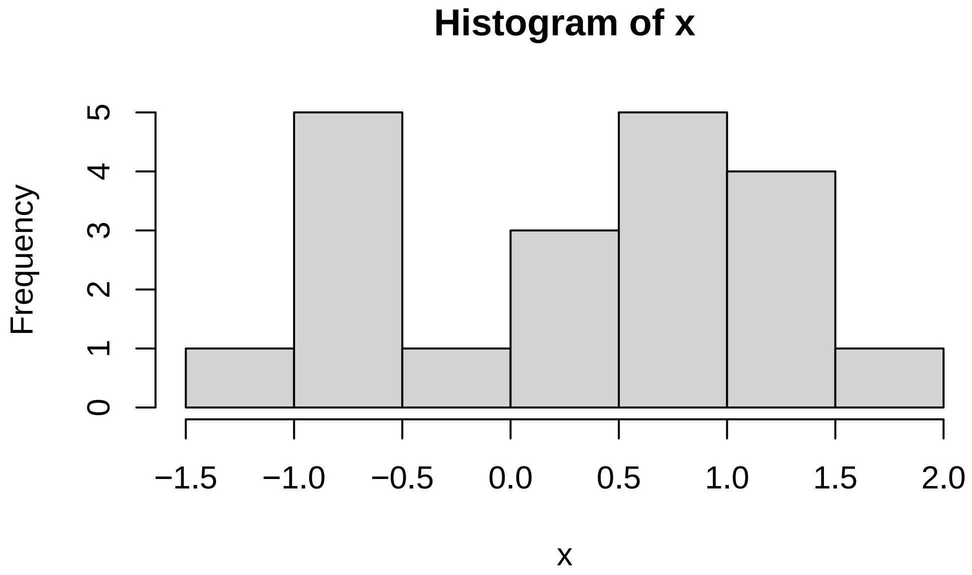
PROBLEM 11: Generate $n = 20$ random variables from a standard normal distribution.

Solution:

```
x = rnorm(20, mean=0, sd=1)
x
```

```
 [1] -0.9749379  0.8227464  1.6623786  0.4182657  0.4470871  0.8727184
 [7]  1.3512166 -1.0328228  1.3204735 -0.6179503  0.7118602  1.1692493
[13]  0.6165703  0.5904384  0.3872705  1.0062132 -0.6629688 -0.5608575
[19] -0.4002011 -0.9912227
```

```
hist(x)
```



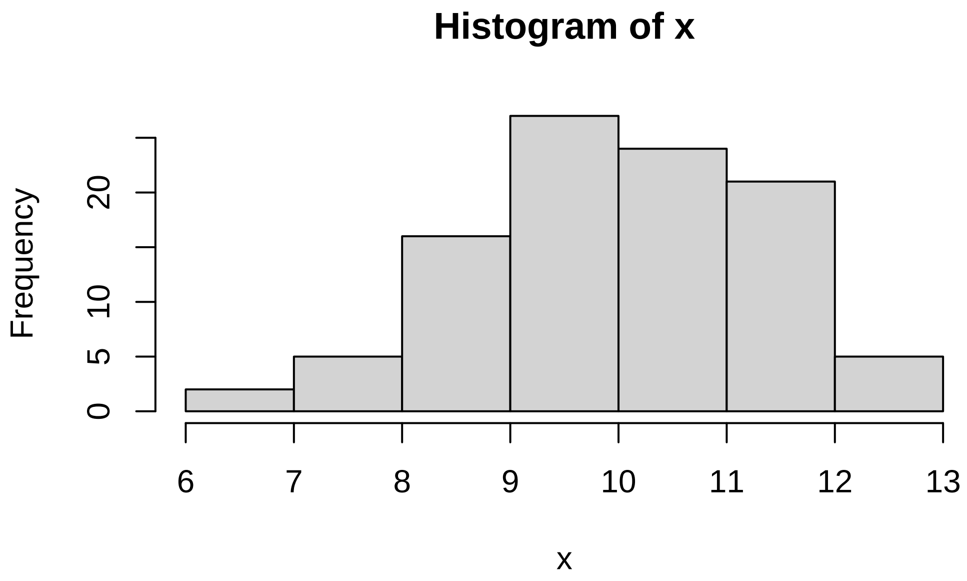
PROBLEM 12: Generate $n = 100$ random variables from a $N(10, 2)$ distribution.

Solution:

```r
x = rnorm(100, mean=10, sd=sqrt(2))
x
```

```
 [1] 10.848575 11.300006  9.948129 11.634975  9.337425 11.097273  8.753547
 [8] 12.039143 10.017215  8.832616 10.070273  9.804772 10.358305 11.802722
[15]  8.576856 11.072486 11.340607  9.113091 11.043841 11.119889 10.853907
[22]  8.359748 12.421762 11.556567 10.760304 11.368390 11.373047  9.037928
[29]  9.087627 10.185939  7.900130  8.883223  9.534408  9.523283  9.689046
[36] 11.177796 10.574653  9.275597  8.330428  8.810345 12.417532 10.014187
[43] 11.814048  8.486352  9.334884 10.939220 11.440567 10.661745 10.550126
[50] 10.470393 11.019471 10.959816 10.348984  9.320937  8.761973  9.410154
[57] 11.024346  6.033355  7.550519  9.782853  8.924229 10.881331  6.556257
[64]  8.815112  9.795822 10.971643  9.727555  9.361634  8.313081 11.054473
[71]  9.485186  8.019866 11.315693 10.230002  7.006249 11.295927  9.646452
[78] 12.177684 10.330485  9.241811 10.666765  8.782117 11.236744  9.011387
[85] 10.287409  9.857368 12.933749  9.291389 11.522342 10.908933  8.598177
[92]  8.427399  9.399232 10.262159  7.857832 10.831886  7.817496  9.962547
[99]  9.776494  9.533845
```

```r
hist(x)
```



2.2 Bernoulli and Binomial Distributions

The Bernoulli and Binomial distributions are intimately related: a Binomial random variable corresponds to the number of successes in n independent Bernoulli trials. For example, consider flipping a coin. Each coin flip can be modelled as a Bernoulli(p) random variable with probability of success (heads) equal to p. If you flipped a coin $n = 10$ times and wanted to model the number of sucesses (heads) in $n = 10$ trials, that would be a Binomial(n, p) random variable.

R has four functions that can be used to compute both Bernoulli and Binomial probabilities: `dbinom( )`, `pbinom( )`, `qbinom( )`, `rbinom( )`.

`dbinom(x,size,prob)` probability mass function (PMF) - *input*: `x` is the number of successes, `size` is the number of trials n, `prob` is the probability of success p - *output*: a probability since $0 \leq P(X = x) \leq 1$ - *example*: evaluate $P(X = x)$

`pbinom(q,size,prob)` probability distribution function (CDF) - *input*: `q` is the value for which you want to find the area below/above, `size` is the number of trials n, `prob` is the probability of success p - *output*: a probability - *example*: evaluate $P(X \leq x)$

`qbinom(p,size,prob)` quantile function
- *input*: `p` is a probability, `size` is the number of trials n, `prob` is the probability of success p - *output*: a positive integer since $X \in \{0, 1, \dots, n\}$ - *example*: find q s.t. $P(X \leq q) = p$

`rbinom(n,size,prob)` random number generator
- *input*: `n` is the number of observations you want to generate, `size` is the number of trials n, `prob` is the probability of success p - *output*: a vector of n positive integers - *example*: generate n independent Binomial(n, p) random variables

Note: These functions correspond to the Bernoulli distribution whenever `size=1`.

More information is also accessible in R if you type `?dbinom`, `?pbinom`, `?qbinom`, or `?rbinom`.

2.2.1 Probability Mass Function (`dbinom`)

PROBLEM 13: If you flip a coin $n = 5$ times and in each flip the probability of heads is $p = 0.5$, what is the chance that you get 2 successes?

Solution:

Here, our random variable X is the number of successes in n independent trials, so $X \sim$ Binomial(n, p) with $n = 5$ and $p = 0.5$.

```
dbinom(2, size=5, prob=0.5)
```

```
[1] 0.3125
```

We can also check our answer using the Binomial probability mass function: $P(X = x) = \binom{n}{x}p^x(1-p)^{n-x}$.

```
choose(5,2)*0.5^2*(1-0.5)^(5-2)
```

```
[1] 0.3125
```

2.2.2 Cumulative Distribution Function (`pbinom`)

PROBLEM 14: If you flip a coin $n = 5$ times and in each flip the probability of heads is $p = 0.5$, what is the chance that you get *at most* 2 successes?

Solution:

Now we want to find $P(X \leq 2)$. We know that $P(X \leq 2) = P(X = 2) + P(X = 1) + P(X = 0)$, so we could again use the `dbinom( )` function.

```
dbinom(2, size=5, prob=0.5) + dbinom(1, size=5, prob=0.5) + dbinom(0, size=5, prob=0.5)
```

```
[1] 0.5
```

The problem is that this approach becomes cumbersome as the number of trials increases. A more efficient approach is to recognize that $P(X \leq 2)$ takes the form of the CDF and use `pnorm( )`.

```
pbinom(2, size=5, prob=0.5)
```

```
[1] 0.5
```

PROBLEM 15: If you flip a coin $n = 100$ times and in each flip the probability of heads is $p = 0.25$, what is the chance that you get *at most* 20 successes?

Solution:

```
pbinom(20, size=100, prob=0.25)
```

```
[1] 0.1488311
```

PROBLEM 16: If you flip a coin $n = 100$ times and in each flip the probability of heads is $p = 0.25$, what is the chance that you get *at least* 20 successes?

Solution:

We have two ways to solve this problem. First, we can write $P(X \geq 20) = 1 - P(X < 20) = 1 - P(X \leq 19)$ where $P(X < 20) = P(X \leq 19)$ since X is discrete.

```
1-pbinom(19, size=100, prob=0.25)
```

```
[1] 0.9004696
```

Alternatively, we can use the `lower.tail=FALSE` option to tell R we want the probability *greater than* x. However, note that this is *strictly greater than*, so we must again remember than $P(X \geq 20) = P(X > 19)$.

```
pbinom(19, size=100, prob=0.25, lower.tail=FALSE)
```

```
[1] 0.9004696
```

2.2.3 Quantile Function (`qbinom`)

PROBLEM 17: Suppose you flip a coin $n = 20$ times where each flip has a probability of heads equal to $p = 0.5$. Find the value q such that the probability of getting at most q successes is equal to 0.25.

Solution:

```
qbinom(0.25, size=20, prob=0.5)
```

```
[1] 8
```

2.2.4 Random Number Generator (`rbinom`)

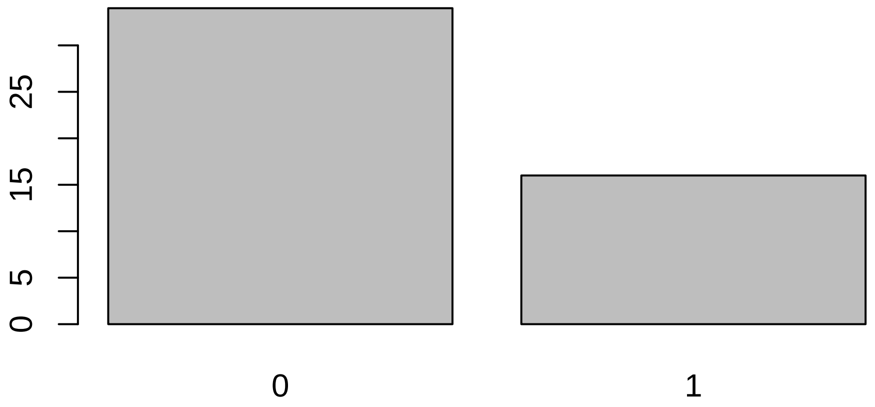
PROBLEM 18: Generate $n = 50$ Bernoulli(p) random variables with $p = 0.2$.

Solution:

```
x = rbinom(50, size=1, prob=0.2)
x
```

```
 [1] 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0
[39] 0 0 1 0 0 1 0 1 1 1 1 1
```

```
barplot(table(x))
```



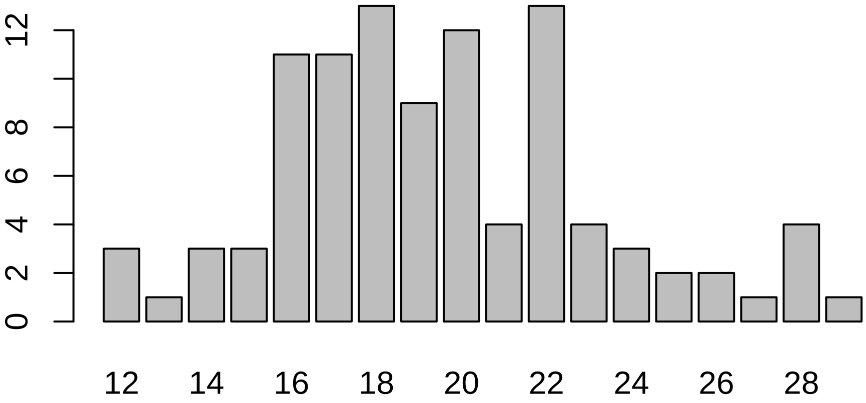
PROBLEM 19: Generate $n = 100$ Binomial(n, p) random variables with $p = 0.4$.

Solution:

```
x = rbinom(100, size=100, prob=0.2)
x
```

```
  [1] 18 22 19 17 16 16 16 18 20 20 25 19 17 16 22 22 15 12 17 18 28 23 20 20 17
 [26] 22 18 18 20 19 16 16 12 13 26 17 22 12 22 17 18 17 26 16 17 20 16 18 18 19
 [51] 18 19 24 14 23 20 22 15 21 20 28 24 19 17 20 19 28 20 16 21 17 20 21 22 18
 [76] 22 16 23 14 30 22 21 19 22 16 17 27 25 14 18 24 18 22 18 23 28 19 22 20 15
```

```
barplot(table(x))
```


0
2
4
6
8
12
12
14
16
18
20
22
24
26
28

3 Sampling Distributions and the CLT

3.1 Sampling Distributions

Sampling distributions are theoretical objects that represent the probability distribution of a statistic (usually the sample mean).

The **sampling distribution of the sample mean** is the theoretical distribution of means that would result from taking all possible samples of size n from the population.

We can build some intuition for what this means in R by simulating this sampling distribution.

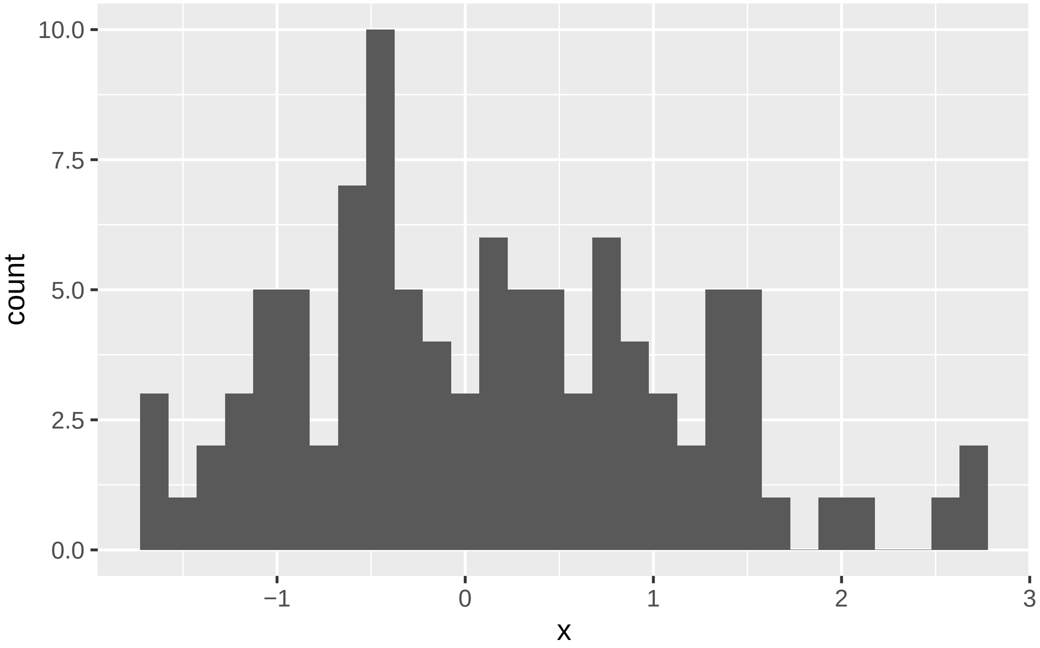
Let's start by assuming that we are sampling from the normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$ such that observations in our sample are IID. A specific sample of size n then is going to consist of realizations $\{x_1, \ldots, x_n\}$, where each x_i is a realization of the random variable $X_i \sim N(0, 1)$.

In R, we now can take a sample of size $n = 100$ by simulating draws from the normal distribution. For this sample, we then also compute the mean defined by $\frac{1}{n}\sum_{i=1}^{n} x_i$ and plot the distribution of the realizations $\{x_1, \ldots, x_n\}$ in a histogram.

```
x = rnorm(n=100, mean=0, sd=1)
mean(x)
```

```
[1] 0.1302831
```

```
ggplot(data.frame(x),aes(x)) + geom_histogram()
```

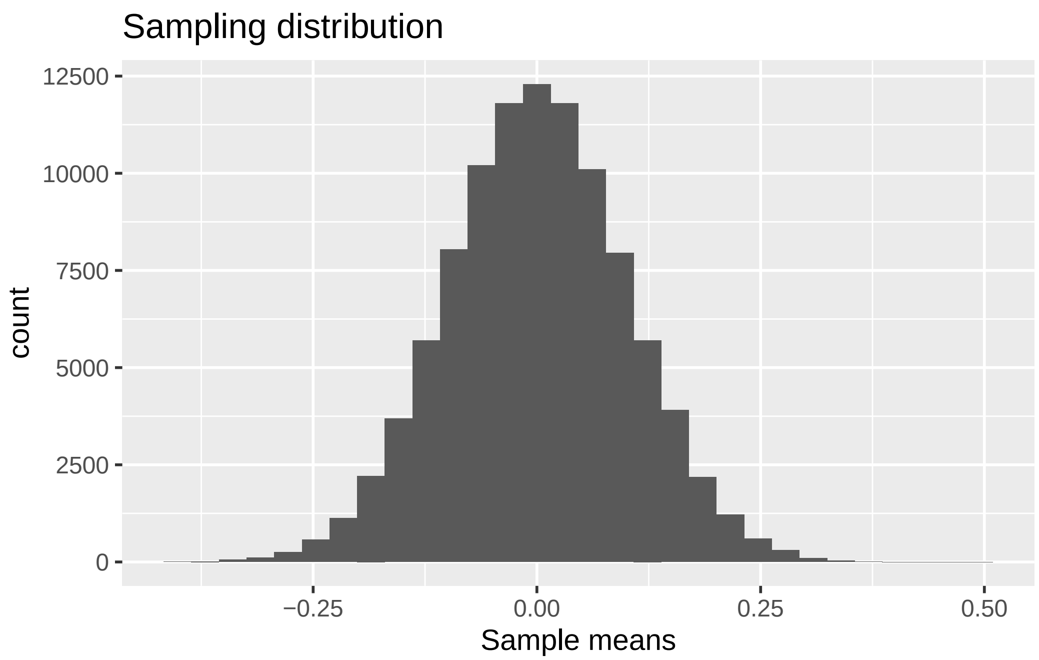


```
# note: to plot the vector with ggplot, we convert x to a data.frame first
```

While we cannot generate all possible samples of size n from this normal distribution to get the sampling distribution, we can many sample and get close to it.

Let $R = 100,000$ be the number of samples we want to generate. The code below now constructs a for loop in R to do the following: (1) generate a random sample of size n from the population; (2) compute the sample mean and store the computed mean in a vector called `xbar`.

```
R = 100000
xbar = double(R)
for(r in 1:R){
    x = rnorm(n=100, mean=0, sd=1) # generate new sample
    xbar[r] = mean(x)              # calculate mean and ad it in vector xbar
}
ggplot(data.frame(xbar),aes(xbar),xtitle="mean in sample") +
  geom_histogram(title="Sampling distribution") +
  ggtitle("Sampling distribution") +
  xlab("Sample means")
```



This plot is the distribution of sample means after taking $R = 100,000$ samples with size $n = 100$ from the population.

Notice that this again looks like a normal distribution. The mean looks the same as in the first histogram within a single sample, but the variance looks much smaller.

In fact, when the population distribution is $N(\mu, \sigma^2)$, then the distribution of $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is $N(\mu, \sigma^2/n)$. In this case, we know that $E(\bar{X}) = E(X) = \mu = 0$ and $Var(\bar{X}) = Var(X)/n = \sigma^2/n = 1/100 = 0.01$.

Let's check to see what the mean and variance are in our approximation of the sampling distribution.

```
mean(xbar)
```

```
[1] 0.0003628248
```

```
var(xbar)
```

```
[1] 0.009943945
```

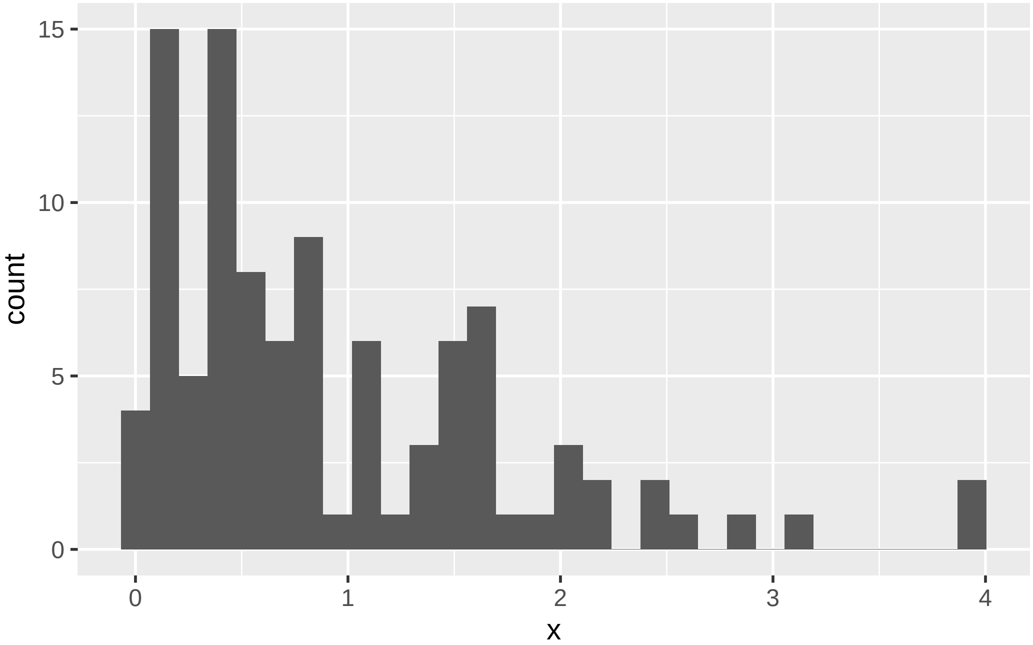
Very close to the true values!

3.2 Central Limit Theorem (CLT)

The next question to consider is what if our population distribution was not normally distributed? What if it was skewed to the right or left?

Let's assume X_i has a Exponential(β) distribution where $\beta = 1$ is a rate parameter. We can again take one individual sample $x_1, \ldots, x_n$.

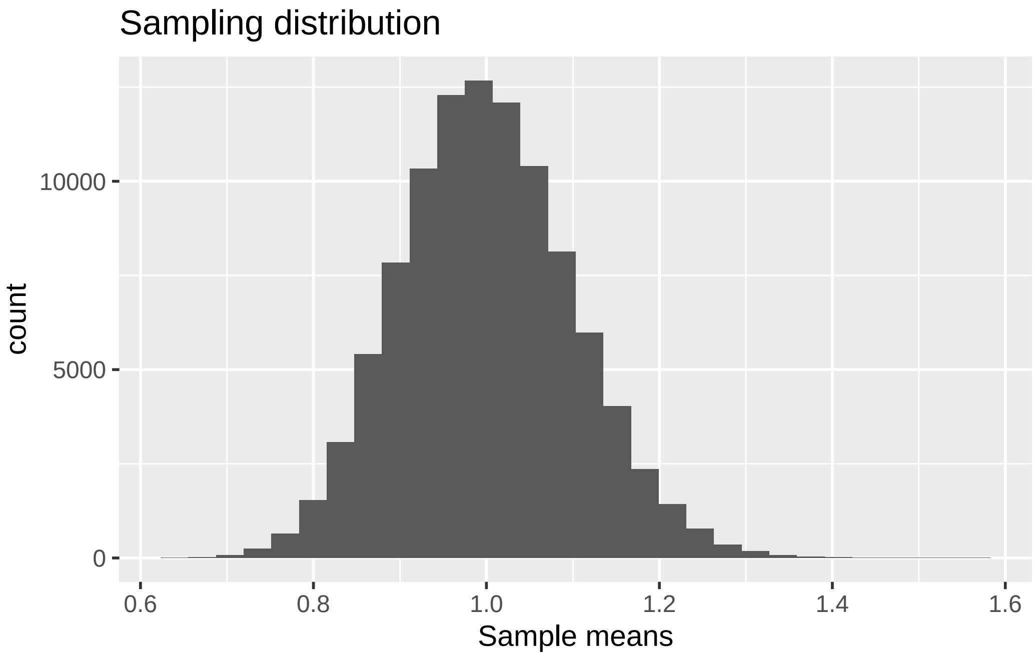
```
x = rexp(n=100, rate=1) # take draws
ggplot(data.frame(x),aes(x)) + geom_histogram()
```



Based on this sample, it appears that distribution is highly asymmetric and skewed to the right.

In this case, should we still expect the sampling distribution of the sample mean to be normal? Let's go through the same exercise as before, i.e. take many samples of size $n = 100$, compute the sample mean for each of the samples, and then plot the histogram of the sample means.

```
R = 100000
xbar = double(R)
for(r in 1:R){
  x = rexp(n=100, rate=1)
  xbar[r] = mean(x)
}
ggplot(data.frame(xbar),aes(xbar),xtitle="mean in sample") +
  geom_histogram(title="Sampling distribution") +
  ggtitle("Sampling distribution") +
  xlab("Sample means")
```



The distribution of sample means again looks normal!

The **Central Limit Theorem** guarantees that this will be the case as the sample size n gets large. Then for any population distribution, we know that the distribution of $\bar{X}$ will be *approximately* normal with mean $E(\bar{X}) = E(X)$ and $Var(\bar{X}) = Var(X)/n$.

Since $X \sim \text{Exponential}(\beta)$ with $\beta = 1$, it can be shown that $E(X) = \beta = 1$ and $Var(X) = \beta^2 = 1$. Therefore, $\bar{X}$ is approximately normal with mean $E(\bar{X}) = E(X) = 1$ and variance $Var(\bar{X}) = Var(X)/n = 1/100 = 0.01$.

We again check these results using our approximate sampling distribution and find consistent answers.

```
mean(xbar)
```

```
[1] 1.000156
```

```
var(xbar)
```

```
[1] 0.009983424
```

4 Estimation

4.1 Point Estimation

Point estimation is all about using one single number (statistic) to estimate a population parameter of interest. Generally, we need to differentiate three different terms:

1. *Parameter*: θ
2. *Estimator*: $\hat{\theta}(X) = g(X_1, \dots, X_n)$
3. *Estimate*: $\hat{\theta}(x) = g(x_1, \dots, x_n)$

θ is the population parameter we want to know. $\hat{\theta}(X)$ is our estimator, which is a function of our random sample. It is itself a random variable because another random sample will lead to a different number. The estimate $\hat{\theta}(x)$ then is our statistic that we compute for given sample; given realizations $(x_1, \dots, x_n)$, this number is not random. Note that we apply the same function $g()$ in the estimator and estimate, but in the estimator the X_i are random variables whereas in the estimate the x_i are realizations.

To highlight the difference between estimate and estimator consider the following example: the population is 1,000,000 consumers and we want to know how many of them purchased our product. Hence, the population parameter we want to know is

$$\theta = \frac{\text{no. purchases in population}}{1,000,000}$$

In principle, we could just call all 1,000,000 consumers and ask, but that would cost way too much time (and time is money)! Instead, we are going to take a *random* sample with n consumers from the population. In this case, our estimator for the mean is

$$\hat{\theta}(X) = g(X_1, \dots, X_n) = \frac{\text{no. purchases in sample}}{n}$$

This is what we now do in R.

```r
# Generate population and calculate population parameter
pop = rbinom(1000000,1,0.2) # generates 1,000,000 consumers that bought (=1) or not(=0)
pop_par = mean(pop)
pop_par
```

```
[1] 0.200342
```

```
# Take random sample and calculate estimate on that sample
x = sample(pop,100,replace=TRUE)    # randomly takes 100 consumers out of pop
estimate = mean(x)
estimate
```

```
[1] 0.17
```

```
# Calculate sampling error
sampling_error = abs(estimate-pop_par)
sampling_error
```

```
[1] 0.030342
```

Both the estimate and sampling error will be different if we take another random sample.

```
x = sample(pop,100,replace=TRUE)
mean(x)
```

```
[1] 0.28
```

4.2 Confidence Intervals

As we just saw, estimators themselves are random variables and subject to variation across different samples. Hence, point estimates are *not enough* to learn about the population parameter we are interested in. Rather than provide a single point estimate, it will be better to provide a *range of plausible values* for the parameter of interest. This is the idea of **confidence intervals**.

A confidence interval for any parameter θ will always take the form:

$$\hat{\theta} \pm (\text{critical value}) \times \text{SD}(\hat{\theta})$$

where $\hat{\theta}$ is an estimator of θ, the critical value is a quantile of a normal or t distribution, and $\text{SD}(\hat{\theta})$ is the standard deviation of the estimator.

Confidence Level $(1-\alpha)$	Critical Value $z_{\alpha/2}$
99%	2.576
95%	1.96
90%	1.645

4.2.1 Types of Confidence Intervals

We will consider four types of $(1-\alpha)100\%$ confidence intervals.

1. Confidence interval for mean μ, data are normally distributed, variance σ^2 is known

$$\bar{x} \pm z_{\alpha/2}\frac{\sigma}{\sqrt{n}}$$

2. Confidence interval for mean μ, data are normally distributed, variance σ^2 is unknown

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

3. Confidence interval for mean μ, data are not normally distributed

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

4. Confidence interval for proportion p, data are binary (0s and 1s)

$$\hat{p} \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

Notice that we can compute the critical values in R. If we need $z_{\alpha/2}$, then we must find the value of a standard normal random variable such that $\alpha/2$ percent of the area is to its right. This is can be found using the normal quantile function `qnorm( )`!

For example, if $\alpha = 0.05$ let's use `qnorm( )` to find $z_{0.025}$.

```
qnorm(0.025, mean=0, sd=1, lower.tail=FALSE)
```

```
[1] 1.959964
```

A summary of the most commonly used normal critical values are provided below.

Similarly, if we need to compute $t_{n-1,\alpha/2}$ with $\alpha = 0.05$ and our data have $n = 50$ observations, we can use the `qt( )` function.

```
qt(0.025, df=50-1, lower.tail=FALSE)
```

```
[1] 2.009575
```

Notice this critical value is larger than $z_{0.025}$ – this comes from the fact that the t-distribution has fatter tails than the normal distribution. But, it also turns out that a t distribution (which only has one parameter ν called the "degrees of freedom") converges to a normal distribution as $\nu \to \infty$. We can formally check this using `qt( )`.

```
qt(0.025, df=50, lower.tail=FALSE)
```

```
[1] 2.008559
```

```
qt(0.025, df=100, lower.tail=FALSE)
```

```
[1] 1.983972
```

```
qt(0.025, df=1000, lower.tail=FALSE)
```

```
[1] 1.962339
```

```
qt(0.025, df=10000, lower.tail=FALSE)
```

```
[1] 1.960201
```

Notice how these values approach $z_{0.025} \approx 1.96$ as the degrees of freedom parameter gets large.

4.2.2 Interpreting confidence intervals

Suppose an analyst tells you that the 95% confidence interval for the population parameter θ is $(0.5, 1.5)$. What can you conclude from this, i.e., what does it mean that we are 95% confident that θ is in this interval?

What the confidence interval tells us is that if we take many samples from the population and construct the confidence interval for each sample, then in approximately 95% of the samples the constructed confidence interval will include the true population parameter. Note that, same as the estimator, the confidence interval depends on the sample and hence is random.

We can check in R whether this holds using the same population we generated earlier. For this, we are now going to construct a for loop that does the following: (1) take a random sample of size $n = 100$ from the population, (2) compute the 95% confidence interval (type 4 above) for the sample, check whether the population parameter lies within the computed confidence interval and store the result in a vector called poppar_in_ci.

```r
R = 1000                                # how many samples to take
poppar_in_ci = double(R)
for(r in 1:R){
    x = sample(pop,100,replace=TRUE)                   # generate new sample
    phat = mean(x)
    s = sqrt(phat*(1-phat)/100)           # see formula for CI of proportion above
    poppar_in_ci[r] = phat -1.96 * s < pop_par &
                        phat + 1.96 * s > pop_par
}
mean(poppar_in_ci)
```

```
[1] 0.933
```

The result of this simulation shows that for approximately 95% of our samples, the constructed confidence interval indeed contains the population parameter value.

Note that in this case, we were sampling from a population we first generated. Often, we take samples just from a hypothetical population, where we assume that the samples are drawn from a distribution that captures the population. For example, in chapter 3 we directly made assumptions on the population distribution and sampled from the respective distributions.

4.2.3 Examples

PROBLEM 1: A wine importer needs to report the average percentage of alcohol in bottles of French wine. From experience with previous kinds of wine, the importer believes that alcolhol percentages are normally distributed and the population standard deviation is 1.2%. The importer randomly samples 60 bottles of the new wine and obtains a sample mean $\bar{x} = 9.3\%$. Give a 90% confidence interval for the average percentage of alcohol in all bottles of the new wine.

Solution:

From the problem, we know the following.

$$\begin{aligned} n &= 60 \\ \sigma &= 1.2 \\ \bar{x} &= 9.3 \\ \alpha &= 0.10 \end{aligned}$$

We must first figure out which type of confidence interval to use. Notice that we are trying to estimate the *average* percentage of alcohol, so our parameter is a mean μ. Moreover, we are told to assume that the data are normally distributed and the population standard deviation σ is known. Thefore, our confidence interval will be of the form:

$$\bar{x} \pm z_{\alpha/2} \frac{\sigma}{\sqrt{n}}.$$

We can now define each object in R and construct the confidence interval.

```r
n = 60
sigma = 1.2
xbar = 9.3
zalpha = qnorm(0.05, mean=0, sd=1, lower.tail=FALSE)
xbar - zalpha*sigma/sqrt(n)
```

```
[1] 9.04518
```

```r
xbar + zalpha*sigma/sqrt(n)
```

```
[1] 9.55482
```

Therefore, we are 90% confident that the true average alcohol content in all new bottles of wine is between 9.05% and 9.55%.

PROBLEM 2: An economist wants to estimate the average amount in checking accounts at banks in a given region. A random sample of 100 accounts gives $\bar{x} = £357.60$ and $s = £140.00$. Give a 95% confidence interval for μ, the average amount in any checking account at a bank in the given region.

Solution:

From the problem, we know the following.

$$\begin{aligned} n &= 100 \\ \bar{x} &= 357.60 \\ s &= 140 \\ \alpha &= 0.5 \end{aligned}$$

Here we are not told whether the data are normally distributed. However, it won't matter because we only have an estimate of σ (remember that among the four types of confidence intervals we considered earlier, there are no differences between case II and III). Therefore, our confidence interval will be of the form:

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

We can again define each object in R and construct the confidence interval.

```
n = 100
xbar = 357.60
s = 140
talpha = qt(0.025, df=n-1, lower.tail=FALSE)
xbar - talpha*s/sqrt(n)
```

```
[1] 329.821
```

```
xbar + talpha*s/sqrt(n)
```

```
[1] 385.379
```

Therefore, we are 95% confident that the true average account checking account value in the given region is between £329.82 and £385.38.

PROBLEM 3: The EuStockMarkets data set in R provides daily closing prices of four major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. Using this data set, produce a 99% confidence interval for the average closing price of the UK FTSE.

Solution:

First, let's load in the data from R.

DAX	SMI	CAC	FTSE
1628.75	1678.1	1772.8	2443.6
1613.63	1688.5	1750.5	2460.2
1606.51	1678.6	1718.0	2448.2
1621.04	1684.1	1708.1	2470.4
1618.16	1686.6	1723.1	2484.7
1610.61	1671.6	1714.3	2466.8

```r
data(EuStockMarkets)
head(EuStockMarkets)
```

Now let's pull the subset of data we care about (i.e., the UK FTSE column).

```r
uk = EuStockMarkets[,4]
```

Notice that we are not told anything about the true distribution of the data. Therefore, our confidence interval will be of the form:

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

Next, let's compute each component necessary to construct the 99% confidence interval.

```r
n = length(uk)
xbar = mean(uk)
s = sd(uk)
talpha = qt(0.005, df=n-1, lower.tail=FALSE)
xbar - talpha*s/sqrt(n)
```

```
[1] 3507.248
```

```r
xbar + talpha*s/sqrt(n)
```

```
[1] 3624.038
```

Therefore, we are 99% confident that the true closing price for the UK FTSE index is between £3,507.25 and £3,624.04.

Finally, notice that we can use the `t.test( )` function to perform the same analysis but with fewer ste

5 Hypothesis Tests

5.1 Steps of Hypothesis Testing

Statistical hypothesis testing provides a rigorous framework for using data to provide evidence for or against claims.

For example, suppose that you are working for a start-up that develops education software for children. You're working on a new software package and are now trying to determine how much to charge. Based on experience and market trends, the leadership team thinks £50 is reasonable. As the data scientist, you are asked to do some research.

The plan is for you to conduct a survey to check how much people would be willing to pay for the software. The leadership team will plan to charge £50 unless there is substantial evidence that people are willing to pay more. Your objective is to use the survey data to determine if the company should re-think the £50 price point.

You design a survey and send it to $n = 30$ potential customers. After everyone has responded, you find that the average willingess to pay in your sample is $\bar{x} = 55.7$ pounds and $s^2 = 64.8$.

```
n = 30
xbar = 55.7
s2 = 64.8
```

Now, what does this mean? We know that we cannot stop here and conclude that people are willing to pay more than £50 because if we had asked a different group of customers, our sample mean could change (and perhaps be lower than £50).

Our approach will be to carry out a hypothesis test to formally decide what to do.

Step 1: State the null and alternative hypotheses

The framework of hypothesis testing requires us to specify two mutually exclusive hypotheses: the null hypothesis H_0 and the alternative hypothesis H_1. Specifically, we should choose H_0 to be the case of "no effect" or "no change" and choose H_1 to be the case of what we want to show.

Here, we are investigating whether people are willing to pay *more than* £50 on average so $\mu > 50$ will constitute the alternative.

$$
\begin{aligned}
H_0 &: \mu \leq 50 \\
H_1 &: \mu > 50
\end{aligned}
$$

Step 2: Choose a test and significance level

To determine which test is appropriate, we must first address the following questions.

1. How many parameters do we have? (one = one-sample test, two = two-sample test)
2. Do we know the population variance? (yes = z-test, no = t-test)

In our case, we only have one parameter μ (the average WTP in the population) and we do not know the population variance, but have an estimate of it in our sample. Therefore, we should use a one-sample t-test.

$$t_{df} = \frac{\bar{X} - \mu}{S/\sqrt{n}}$$

Finally, we will choose $\alpha = 0.01$ so that we are fairly confident that if we detect deviations from £50, they reflect real deviations in the population.

Step 3: Compute the observed test statistic

Since we are using a one-sample t-test, our observed test statistic is:

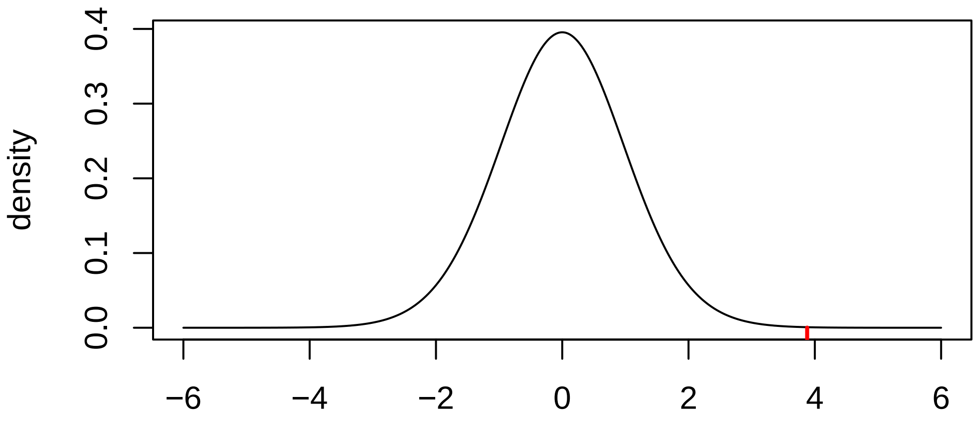
$$t_{obs} = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} = \frac{55.7 - 50}{\sqrt{64.8}/\sqrt{30}} = 3.878$$

```r
t_obs = (xbar - 50)/(sqrt(s2/n))
t_obs
```

```
[1] 3.878359
```

Our observed test statistic provides a measure of "evidence" against the null hypothesis. In particular, we know that under the null hypothesis, the test statistic follows a $t_{df} = t_{n-1} = t_{29}$ distribution. This distribution (plotted below) represents the distribution of sample evidence given that the null is true. Our observed test statistic (the dashed red line) shows that the event we observed ($\bar{x} = 55.7, s^2 = 64.8$), seems fairly unlikely under the null.

Our next task will be to compute this probability more formally.



Step 4: Calculate the p-value

The p-value is the probability of getting sample evidence as or more extreme than what we actually observed given that the null hypothesis is actually true. Remember that the test statistic is our measure of “sample evidence” – as the observed test statistic gets large, that will provide more evidence *against* the null hypothesis.

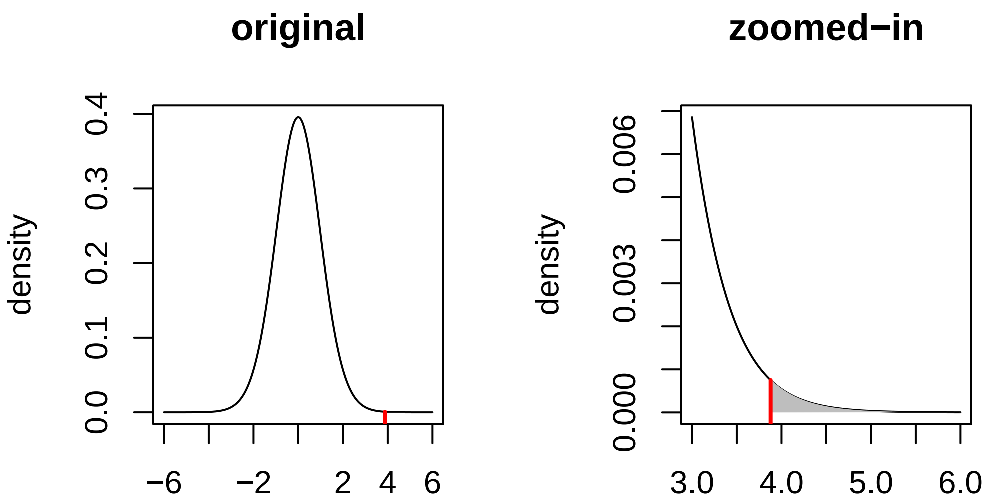
Since we are working with a “greater-than” alternative, our p-value will be

$$\begin{aligned}
\text{p-value} &= P(t_{df} > t_{obs} \mid H_0 \text{ is true}) \\
&= P\left(t_{n-1} > \frac{\bar{x}-\mu_0}{s/\sqrt{n}} \mid \mu \leq 50\right) \\
&= P\left(t_{29} > \frac{55.7-50}{\sqrt{64.8}/\sqrt{30}}\right) \\
&= P\left(t_{29} > 3.878\right) \\
&\approx 0.0003
\end{aligned}$$

```
pt(t_obs,df=n-1,lower.tail=FALSE)
```

```
[1] 0.0002780401
```

Notice that this value just corresponds to the region to the right of the observed test statistic. Since this probability is so small, it is hard to see the shaded area on the original plot. We can therefore create a "zoomed in" plot next to the original.



Step 4: Make a statistical decision and interpret the results

Once the p-value has been calculated, the "decision rule" can be described as follows.

$$\begin{aligned} &\text{if p-value } \leq \alpha \;\; \text{reject } H_0 \\ &\text{if p-value } > \alpha \;\; \text{fail to reject } H_0 \end{aligned}$$

Where does this rule come from? Since α is the maximum p-value at which we reject H_0, then we are ensuring that there is *at most* a $100\alpha\%$ chance of committing a type I error. That is, if we found the p-value to be large, say 40%, then there would be a 40% chance of mistakenly deciding that the true WTP exceeded £50 when it in fact did not. For most problems, an error rate of 40% is too large to tolerate. In the social sciences, we normally choose $\alpha \in \{0.1, 0.05, 0.01\}$ which corresponds to error rates of 10%, 5%, and 1%.

In the context of this problem, we find the p-value is roughly 0.03%. This means that if the true average WTP in the population is less than £50, there is a 0.03% chance that we would have observed sample evidence as or more extreme than what we did observe ($\bar{x} = 55.7, s^2 = 64.8$). This is very small – in fact, much smaller than the 5% error rate we can tolerate. Therefore, we decide to reject the null hypothesis and conclude that it is more likely that the true average WTP in the population exceeds £50.

We can take these results back to the leadership team in our company to convince them that they should consider raising the price.

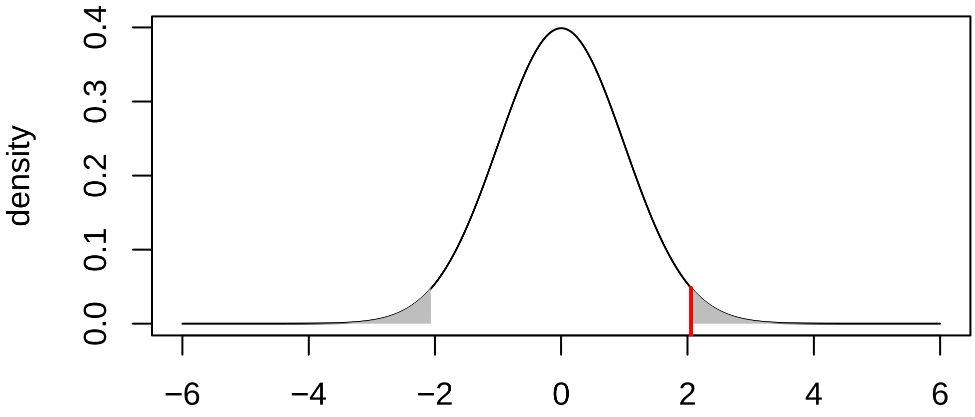
5.2 Connection to Confidence Intervals

There is an intimate connection between hypothesis tests and confidence intervals. We will now go through the details to see why.

To start, remember that our decision rules for hypothesis testing take the following form.

$$\begin{aligned} &\text{if p-value } \le \alpha; \quad \text{reject } H_0 \\ &\text{if p-value } > \alpha; \quad \text{fail to reject } H_0 \end{aligned}$$

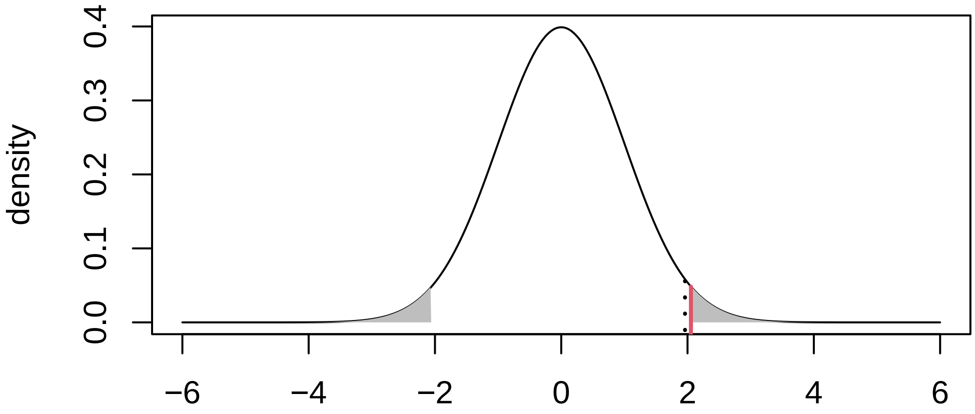
This can also be described visually. Suppose you carry out a two-sided hypothesis test with $\alpha = 0.05$ and compute a test statistic $z_{obs} = 2.054$ and a corresponding p-value equal to 0.04. This corresponds to a *total* area equal to 0.04 in the lower and upper tails of the distribution of the test statistic.



We can also work out what value (on the x-axis) corresponds to an area of $\alpha/2 = 0.05/2 = 0.025$ in the upper tail.

```
qnorm(0.025,lower.tail=FALSE)
```

```
[1] 1.959964
```



Now the dotted black line is at the point $z_{\alpha/2} = 1.96$ – i.e., the value such that the upper tail area is equal to $\alpha/2 = 0.025$. Notice that our shaded area falls to the right of this line, so by our decision rule, we would reject the null.

But, notice that we would reject the null *for any* test statistic (solid red line) that falls to the right of the critical value $z_{\alpha/2}$ (dotted black line).

Therefore, the following would be an equivalent set of decision rules.

$$\begin{aligned} &\text{if } |z_{obs}| \geq z_{\alpha/2} \ \ \text{reject } H_0 \\ &\text{if } |z_{obs}| < z_{\alpha/2} \ \ \text{fail to reject } H_0 \end{aligned}$$

Remember that a confidence interval is a range of plausible values, which we can now formally define as the the range of parameter values that would not be rejected by our hypothesis test. In this case, the "acceptance region" is defined as follows.

$$
\begin{aligned}
|z_{obs}| < z_{\alpha/2} &\implies \left|\frac{\bar{x}-\mu_0}{\sigma/\sqrt{n}}\right| < z_{\alpha/2} \\
&\implies -z_{\alpha/2} < \frac{\bar{x}-\mu_0}{\sigma/\sqrt{n}} < z_{\alpha/2} \\
&\implies -z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}} < \bar{x}-\mu_0 < z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}} \\
&\implies -\bar{x} - z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}} < -\mu_0 < -\bar{x} + z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}} \\
&\implies \bar{x} - z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}} < \mu_0 < \bar{x} + z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}
\end{aligned}
$$

This last line is the exact form of a confidence interval!

5.3 Hypothesis Testing in R

We can use the `t.test( )` function to carry out both one and two-sample t-tests in R. (Note: There are no built-in z-test functions in R because when we work with real data, we *never* know the population variance!)

ONE-SAMPLE t-TEST **`t.test(mydata, alternative, mu, conf.level)`** `mydata`: data on the variable of interest `alternative`: what type of alternative hypothesis is specified? (options: "two.sided", "greater", "less") `mu`: the value of μ under the null hypothesis `conf.level`: confidence level of the test $(1-\alpha)$

TWO-SAMPLE t-TEST **`t.test(mydata1, mydata2, alternative, mu, conf.level)`** `mydata1`: data on the first variable of interest `mydata2`: data on the second variable of interest `alternative`: what type of alternative hypothesis is specified? (options: "two.sided", "greater", "less") `mu`: the value of the difference $\mu_1 - \mu_2$ under the null hypothesis `conf.level`: confidence level of the test $(1-\alpha)$

PROBLEM 1: The EuStockMarkets data set in R provides daily closing prices of four major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. Using this data set, test to see if there are differences in the closing prices of the SMI and CAC indices. Carry out this test at the 5% significance level and do not assume equal variances.

Solution:

```r
# load the data
data(EuStockMarkets)

# create the SMI variable which is the second column of the EuStockMarkets data
SMI = EuStockMarkets[,2]

# create the CAC variable which is the third column of the EuStockMarkets data
CAC = EuStockMarkets[,3]

# execute the two-sample t-test
t.test(SMI, CAC, alternative="two.sided", mu=0, conf.level=0.95)
```

```

	Welch Two Sample t-test

data:  SMI and CAC
t = 28.119, df = 2305.1, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1068.307 1228.484
sample estimates:
mean of x mean of y 
 3376.224  2227.828 
```

We find a p-value much smaller than $\alpha = 0.05$, so we can reject the null and conclude that there are differences in the closing prices between the Swiss SMI and French CAC stock indices.

6 Regression

6.1 Linear Regression

Regression models are useful tools for (1) understanding the relationship between a response variable Y and a set of predictors $X_1, \dots, X_p$ and (2) predicting new responses Y from the predictors $X_1, \dots, X_p$.

We'll start with **linear regression**, which assumes that the relationship between Y and $X_1, \dots, X_p$ is linear.

Let's consider a simple example where we generate data from the following regression model.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$$

To generate data from this model, we first need to set the "true values" for the model parameters $(\beta_0, \beta_1, \beta_2)$, generate the predictor variables (X_1, X_2), and generate the error term (ε).

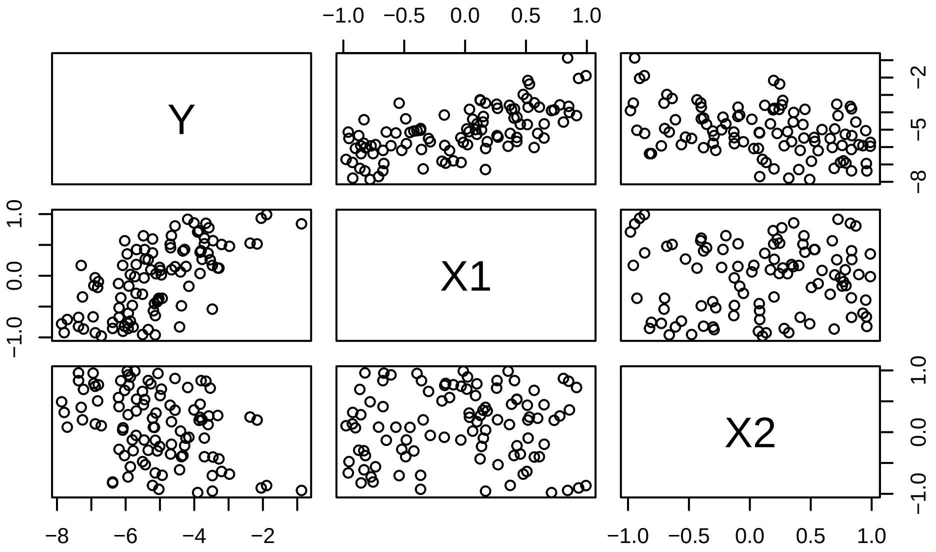
- parameters: $\beta_0 = -5$, $\beta_1 = 2$, $\beta_2 = -1$
- predictor variables: $X_1 \sim Unif(-1, 1)$, $X_2 \sim Unif(-1, 1)$
- error term: $\varepsilon \sim N(0, 1)$

Once we have fixed the true values of the parameters and generated predictor variables and the error term, the regression formula above tells us how to generate the response variable Y.

```
n = 100
beta0 = -5
beta1 = 2
beta2 = -1
X1 = runif(n,min=-1,max=1)
X2 = runif(n,min=-1,max=1)
epsilon = rnorm(n)
Y = beta0 + beta1*X1 + beta2*X2 + epsilon
```

Now let's inspect the data.

```
pairs(Y ~ X1 + X2)
```



As we should expect, we find a positive relationship between Y and X_1, a negative relationship between Y and X_2, and no relationship between X_1 and X_2 (since they are uncorrelated).

Now let's formally estimate the model parameters $(\beta_0, \beta_1, \beta_2)$ using R's built-in linear model function `lm( )`.

```
lm.fit = lm(Y ~ X1 + X2)
summary(lm.fit)
```

```

Call:
lm(formula = Y ~ X1 + X2)

Residuals:
     Min       1Q   Median       3Q      Max 
-2.25655 -0.62299  0.03299  0.57096  2.08334 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept) -4.91360    0.09101 -53.990  < 2e-16 ***
X1           1.66560    0.15890  10.482  < 2e-16 ***
X2          -1.00377    0.15743  -6.376 6.16e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8925 on 97 degrees of freedom
Multiple R-squared:  0.601, Adjusted R-squared:  0.5927
F-statistic: 73.05 on 2 and 97 DF,  p-value: < 2.2e-16
```

Parameter Estimates

First, focus on the "Coefficients" section. Notice that in the first column R reports estimates of our model parameters: $\hat{\beta}_0 = -4.914$, $\hat{\beta}_1 = 1.666$, and $\hat{\beta}_2 = -1.004$. Since we generated this data set, we know the "true" values are $\beta_0 = -5$, $\beta_1 = 2$, and $\beta_2 = -1$. The estimates here are pretty close to the truth. (Remember: the estimates will not exactly equal the true values because we only have a random sample of $n = 100$ observations!)

Interpretation

How should we interpret the estimates? Since $\hat{\beta}_1 = 1.666$, we would say that a one unit increase in X_1 will lead to a 1.666 unit *increase* in Y. Similarly, a one unit increase in X_2 will lead to a 1.004 unit *decrease* in Y. The only way to interpret the intercept is as the value of Y when the X's are all set to zero. In many instances, setting $X = 0$ makes no sense, so we usually focus our attention on the coefficients attached to the predictor variables.

Significance

In the second, third, and fourth columns, R reports the standard error of $\hat{\beta}$ and the t-statistic and p-value corresponding to a (one-sample) test of $H_0 : \beta = 0$ against $H_1 : \beta \neq 0$. The asterisks next to the p-values indicate the levels (e.g., $\alpha = 0.05$, $\alpha = 0.001$) for which we would conclude that the parameter is significantly different from zero. This test is naturally of interest in a regression setting because if $\beta_2 = 0$, for example, then X_2 has no effect on the response Y.

Model Fit

Now look at the last section where it says "Multiple R-squared: 0.601". This value is the R^2 statistic, which measures the percent of the variation in Y that is explained by the predictors. In this case, we find that 60.1% of the variation in Y can be explained by X_1 and X_2. In general, it is difficult to define an absolute scale for what a "good" R^2 value is. In some contexts, 60% may be very high while in others it may be low. It likely depends on how difficult the response variable is to model and predict.

Prediction

Suppose I observed some new values of X_1 and X_2, say $X_1 = 0$ and $X_2 = 0.5$. How can I use the model to **predict** the corresponding value of Y?

I could simply do the calculation by hand:

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 = -4.914 + 1.666(0) - 1.004(0.5) = -5.416$$

where we use the "hat" notation to denote estimates or predicted values.

We can also use built-in prediction tools in R (where any differences would just be due to rounding error).

```
predict(lm.fit, newdata=data.frame(X1=0,X2=0.5))
```

```
        1
-5.415489
```

The first argument of the `predict( )` function is the regression object we created using the `lm( )` function. The second argument is the new set of covariates for which we want to predict a new response Y. (Note: the names of variables in `newdata` must be the same names used in the original data.)

6.2 Regression Trees

A natural question to ask now is what happens if the "true" model that generated our data was not linear? For example, our model could look something like this:

$$Y_i = \beta_0 + \frac{\beta_1 X_{1i}}{\beta_2 + X_{2i}} + \varepsilon_i$$

Here we still have three model parameters $(\beta_0, \beta_1, \beta_2)$, but they enter the regression function in a nonlinear fashion.

If we generate data from this model and then estimate the linear regression model from section 1, what will happen?

```
# generate data
n = 100
beta0 = -5
beta1 = 2
beta2 = -1
X1 = runif(n,min=-1,max=1)
```

```r
X2 = runif(n,min=-1,max=1)
epsilon = rnorm(n)
Y = beta0 + beta1*X1/(beta2+X2) + epsilon

# estimate linear regression model
lm.fit = lm(Y ~ X1 + X2)
summary(lm.fit)
```

```
Call:
lm(formula = Y ~ X1 + X2)

Residuals:
    Min      1Q  Median      3Q     Max 
-47.926  -8.548  -1.961   4.484 165.499 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   -3.815      2.153  -1.772 0.079519 .  
X1           -14.362      3.596  -3.994 0.000126 ***
X2             5.733      3.510   1.634 0.105595    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 21.22 on 97 degrees of freedom
Multiple R-squared:  0.1697,	Adjusted R-squared:  0.1526 
F-statistic: 9.914 on 2 and 97 DF,  p-value: 0.0001209
```

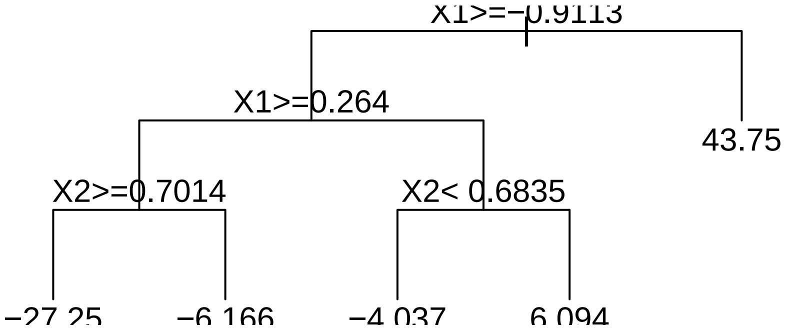
The answer is that we get incorrect estimates of model parameters! (Remember $\beta = -5, \beta_1 = 2, \beta_2 = -1$.)

A more flexible approach to regression modeling is provided by **regression trees**. The idea is to split up the covariate space into homogeneous regions (with respect to the response Y) and then fit simple linear models within each region.

We can use the `rpart` library in R to fit and plot regression trees. You'll actually notice a similar syntax between `lm( )` and `rpart( )`.

```r
library(rpart)

# estimate regression tree
tree.fit = rpart(Y ~ X1 + X2)
```

```
# plot the estimated tree
plot(tree.fit, uniform=TRUE, margin=.05)
text(tree.fit)
```



The output from a regression tree model looks very different from the output of a linear regression model. This is mostly because we had real-valued parameters in the linear model, but have much more complicated parameters in the tree model.

The top node is called the **root node** and indicates the most important variable for predicting Y. Each subsequent node is called an **interior node** until you get to the last node showing a numeric value which is called a **terminal node**.

Tree models should be interpreted as a sequence of decisions for the purposes of making a prediction. Each node will present a logical statement and if that statement is true, we move *down and to the left* whereas if that statement is false, we move *down and to the right*.

For example, if you wanted to predict Y when $X_1 = 0$ and $X_2 = 0.5$, the root note first asks "Is $X_1 \geq -0.9113$?" If yes, then left and if no then right. Here our answer is yes, so we go to the next node to the left and ask "Is $X_1 \geq 0.264$?" Our answer is no so we go to the right and ask $X_2 < 0.6835$?" Our answer is yes so we go to the left. Since this represents the terminal node, we're left with our predition of $\hat{Y} = -4.037$. That is, if $X_1 = 1$ and $X_2 = 9$ then the model predicts $\hat{Y} = -4.037$.

We can also use the `predict( )` function as we did with the linear regression model above.

```
predict(tree.fit, newdata=data.frame(X1=0,X2=0.5))
```

```
        1
-4.036809
```

6.3 Model Selection

Let's see how regression trees compare to linear regression models in terms of out-of-sample prediction. We'll consider two cases:

The true model is a linear model

The true model is a nonlinear model

CASE A: TRUE MODEL IS LINEAR

First, we'll generate a training and test data set from a linear regression model as in section 1. The training data set will be used for estimation and the test data will be used for prediction.

```
n = 100
beta0 = -5
beta1 = 2
beta2 = -1
X1 = runif(n,min=-1,max=1)
X2 = runif(n,min=-1,max=1)
epsilon = rnorm(n)
Y = beta0 + beta1*X1 + beta2*X2 + epsilon
train = data.frame(Y=Y[1:70], X1=X1[1:70], X2=X2[1:70])
test = data.frame(Y=Y[71:100], X1=X1[71:100], X2=X2[71:100])
```

Now let's estimate both the linear regression and regression tree models on the training data.

```
# estimate linear regression model
lm.fit = lm(Y ~ X1 + X2, data=train)

# estimate regression tree model
tree.fit = rpart(Y ~ X1 + X2, data=train)
```

To compare out-of-sample model performance, we'll compute the root mean squared error (RMSE).

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

```r
# linear regression model
lm.predict = predict(lm.fit, newdata=test)
lm.rmse = sqrt(mean((lm.predict-test$Y)^2))
lm.rmse
```

```
[1] 0.9157128
```

```r
# regression tree model
tree.predict = predict(tree.fit, newdata=test)
tree.rmse = sqrt(mean((tree.predict - test$Y)^2))
tree.rmse
```

```
[1] 1.148858
```

In this case the linear regression model has better predictive performance, which is not too surprising because we simulated the data from that model!

CASE B: TRUE MODEL IS NONLINEAR

We will again generate a training and test data set, but now from the nonlinear regression model we used in section 2.

```r
n = 100
beta0 = -5
beta1 = 2
beta2 = -1
X1 = runif(n,min=-1,max=1)
X2 = runif(n,min=-1,max=1)
epsilon = rnorm(n)
Y = beta0 + beta1*X1/(beta2+X2) + epsilon
train = data.frame(Y=Y[1:70], X1=X1[1:70], X2=X2[1:70])
test = data.frame(Y=Y[71:100], X1=X1[71:100], X2=X2[71:100])
```

Let's again estimate both the linear regression and regression tree models on the training data and compute the predictive RMSE.

```r
# linear regression model
lm.fit = lm(Y ~ X1 + X2, data=train)
lm.predict = predict(lm.fit, newdata=test)
lm.rmse = sqrt(mean((lm.predict - test$Y)^2))
lm.rmse
```

```
[1] 18.76132
```

```
# regression tree model
tree.fit = rpart(Y ~ X1 + X2, data=train)
tree.predict = predict(tree.fit, newdata=test)
tree.rmse = sqrt(mean((tree.predict - test$Y)^2))
tree.rmse
```

```
[1] 17.35195
```

Now the regression tree model has better predictive performance (but notice that the linear model still does relatively well!) In general, regression trees suffer from a problem called **overfitting**: the trees learn *too much* from the training data that they don't generalize well to test data. There are ways of correcting for this, and you will learn more about them in Data Analyics II!

7 Classification

Classification shares many similarities with regression: We have a response variable Y and a set of one or more predictors $X_1, \ldots, X_p$. The difference is that for classification problems, the response Y is **discrete**, meaning $Y \in \{1, 2, \ldots, C\}$ where C is the number of classes that Y can take on.

We will focus our attention on **binary** responses $Y \in \{0, 1\}$, but all of the methods we discuss can be extended to the more general case outlined above.

To illustrate classification methods, we will use the Default data in the `ISLR` R library. The data set contains four variables: `default` is an indicator of whether the customer defaulted on their debt, `student` is an indicator of whether the customer is a student, `balance` is the average balance that the customer has remaining on their credit card after making their monthly payment, and `income` is the customer's income.

```r
library(ISLR)

# load data
data(Default)

# inspect first few rows
head(Default)
```

We also need to split up the data into training and test samples in order to measure the predictive accuracy of different approaches to classification.

```r
train = Default[1:7000,]
test = Default[7001:10000,]
```

default	student	balance	income
No	No	729.5265	44361.625
No	Yes	817.1804	12106.135
No	No	1073.5492	31767.139
No	No	529.2506	35704.494
No	No	785.6559	38463.496
No	Yes	919.5885	7491.559

7.1 k-Nearest Neighbors

The k-NN algorithms are built on the following idea: given a new observation X^* for which we want to predict an associated response Y^*, we can find values of X in our data that look similar to X^* and then classify Y^* based on the associated Y's. We will use Euclidean distance is a measure of similarity (which is only defined for real-valued X's).

Let's take a small portion (first 10 rows) of the Default data to work through a simple example. Notice that we will exclude the student variable since it is a categorical rather than numeric variable. We will use the 11th observation as our "test" data X^* that we want to make predictions for.

```
X = Default[1:10,3:4]
Y = Default[1:10,1]
newX = Default[11,3:4]
```

We now need to compute the similarity (i.e., Euclidean distance) between $X^* = (X_1^*, X_2^*)$ and $X_i = (X_{1i}, X_{2i})$ for each $i = 1, \dots, n$.

$$dist(X^*, X_i) = ||X^* - X_i|| = \sqrt{(X_1^* - X_{1i})^2 + (X_2^* - X_{2i})^2}$$

To do this in R, we can take use the apply( ) function. The first argument is the matrix of X variables that we want to cycle through to compare with X^*.

The second argument of the apply( ) function tells R whether we want to perform an operation for each row (=1) of for each column (=2). The last row tells R what function we want to compute. Here, we need to evaluate $dist(X^*, X_i)$ for each row.

```
distance = apply(X,1,function(x)sqrt(sum((x-newX)^2)))
distance
```

```
        1         2         3         4         5         6         7         8 
22502.381  9799.072  9954.126 13843.541 16611.013 14408.889  3144.449  4346.510 
        9        10 
15640.610  7404.195 
```

Notice that the function returns a set of 10 distances. If we wanted to use the 1st-nearest neighbor classifier to predict Y^*, for example, then we would need to find the Y value of X_i for the observation i that has the smallest distance. We can find that value using the which.min( ) function.

```
which.min(distance)
```

```
7
7
```

```
Y[which.min(distance)]
```

```
[1] No
Levels: No Yes
```

Therefore, we would predict $Y^* = No$ having observed X^*.

Now let's go back to the full data set and test the performance of the k-NN classifier. The first thing we should do is standardize the X's since the nearest neighbors algorithm depends on the scale of the covariates.

```
stdtrainX = scale(train[,3:4])
stdtestX = scale(test[,3:4])

summary(stdtrainX)
```

```
    balance              income
 Min.   :-1.72782   Min.   :-2.46318
 1st Qu.:-0.73329   1st Qu.:-0.92073
 Median :-0.03045   Median : 0.08042
 Mean   : 0.00000   Mean   : 0.00000
 3rd Qu.: 0.68581   3rd Qu.: 0.77299
 Max.   : 3.45400   Max.   : 3.00595
```

Now we can use the `knn( )` function in the `class` R library to run the algorithm on the training data and then make predictions for each observation in the test data. The first argument calls for the X's in the training data, the second calls for the X's in the test data (for which we want to predict), the third calls for the Y's in the training data, and the fourth calls for k, the number of nearest neighbors we want to use to make the prediction.

```
library(class)
knn1 = knn(stdtrainX, stdtestX, train$default, k=1)
```

The `knn1` object now contains a vector of predicted Y's for each value of X in the test data. We can then compare the predicted response $\hat{Y}$ to the true response in the test data Y to assess the performance of the classification algorithm. In particular, we will see the fraction of predictions the algorithm gets wrong.

```r
mean(knn1 != test$default)
```

```
[1] 0.04466667
```

In this case, the 1-NN classifier as an error rate of about 4.5% (or equivalently, an accuracy of 95.5%).

We can try increasing k to see if there is any effect on predictive fit.

```r
# 5 nearest neighbors
knn5 = knn(stdtrainX, stdtestX, train$default, k=5)
mean(knn5 != test$default)
```

```
[1] 0.029
```

```r
# 10 nearest neighbors
knn10 = knn(stdtrainX, stdtestX, train$default, k=10)
mean(knn10 != test$default)
```

```
[1] 0.02633333
```

```r
# 50 nearest neighbors
knn50 = knn(stdtrainX, stdtestX, train$default, k=50)
mean(knn50 != test$default)
```

```
[1] 0.024
```

```r
# 100 nearest neighbors
knn100 = knn(stdtrainX, stdtestX, train$default, k=100)
mean(knn100 != test$default)
```

```
[1] 0.02733333
```

We would then likely choose the model that predicts best (i.e., has the lowest error/misclassification rate).

The last object of interest when doing classification is the **confusion matrix**, which allows us to decompose misclassification mistakes into two groups: **false positives** (predict $\hat{Y} = 1$ when $Y = 0$) and **false negatives** (predict $\hat{Y} = 0$ when $Y = 1$).

Let's produce the confusion matrix for the 10-NN classifier.

```
table(knn10,test$default)
```

```
knn10   No  Yes
  No  2889   61
  Yes   18   32
```

```
# false positive rate
18/(18+2889)
```

```
[1] 0.00619195
```

```
# false negative rate
60/(60+33)
```

```
[1] 0.6451613
```

The false negative rate is especially high, which would be concerning given the risks to the lending agency (e.g., bank).

7.2 Logistic Regression

Issues with the k-NN algorithms include the fact they can't accommodate categorical X's, the algorithms aren't based on a formal statistical model so we can't do inference (or learn about how the X's relate to Y), and there is an assumption that all X's matter and matter equally in determining Y.

Our first solution to these problems is **logistic regression**.

Given a response $Y \in \{0,1\}$ and a set of predictors $X_1, \dots, X_P$, the logistic regression model is written as follows.

$$\Pr(Y=1|X) = \frac{\exp(\beta_0 + \beta_1 X_1 + ... + \beta_p X_p)}{1 + \exp(\beta_0 + \beta_1 X_1 + ... + \beta_p X_p)}$$

The intuition for this formula is as follows. If $Y \in \{0,1\}$, then we can assume that $Y \sim \text{Bernoulli}(\theta)$ where $\theta = \Pr(Y=1)$. We can then write down a regression model for θ rather than Y. The only remaining problem is that $\theta \in (0,1)$, so we need to transform the linear regression function $h(X) = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p$) in a way so that it is constrained to be between 0 and 1. The function $e^{h(X)}/(1 + e^{h(X)})$ does just that.

Estimating a logistic regression model in R can be done using the `glm( )` function, which is similar to the `lm( )` command we use to estimate linear regression models.

Let's illustrate with the training sample from the Default data set.

```
glm.fit = glm(default ~ student + balance + income, family="binomial", data=train)
summary(glm.fit)
```

```

Call:
glm(formula = default ~ student + balance + income, family = "binomial",
    data = train)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.101e+01  5.889e-01 -18.704   <2e-16 ***
studentYes  -6.464e-01  2.846e-01  -2.271   0.0231 *
balance      5.829e-03  2.781e-04  20.958   <2e-16 ***
income       4.711e-06  9.875e-06   0.477   0.6333
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2090.7  on 6999  degrees of freedom
Residual deviance: 1109.4  on 6996  degrees of freedom
AIC: 1117.4

Number of Fisher Scoring iterations: 8
```

Notice that we added one more option in the `glm( )` function: `type="binomial"`. This option tells R to use the logistic regression model rather than other types of *generalized linear models*.

The output from the logistic regression model looks fairly similar to that of linear regression models. However, the interpretation of model paramters (and their estimates) changes a bit.

For example, we find that the coefficient on balance is estimated to be about 0.0058, which means that a one dollar increase in balance multiplies the odds of default by exp(0.0058)=1.006. Since this number is greater than 1, we can say that increasing the balance *increases* the odds of default.

To predict responses in the test data, we can use the `predict( )` function in R. We again need to add one option: `type="response"`, which will tell R to return the predicted probabilities that $Y = 1$.

```
glm.probs = predict(glm.fit, newdata=test, type="response")
```

Then we can compute $\hat{Y}$ by using the rule that $\hat{Y} =$ Yes if the predicted probability is greater than 0.5 and $\hat{Y} =$ No otherwise.

```
glm.predict = ifelse(glm.probs>0.5,"Yes","No")
```

Just as before, we can compare the model predictions with the actual Y's in the test data to compute the out-of-sample error (misclassification) rate.

```
mean(glm.predict != test$default)
```

```
[1] 0.024
```

This error rate can be decomposed by producing the associated confusion matrix and computing the false positive and false negative rates.

```
table(glm.predict, test$default)
```

```
glm.predict   No  Yes
        No  2896   61
        Yes   11   32
```

```
# false positive rate
11/(11+2896)
```

```
[1] 0.00378397
```

```
# false negative rate
61/(61+32)
```

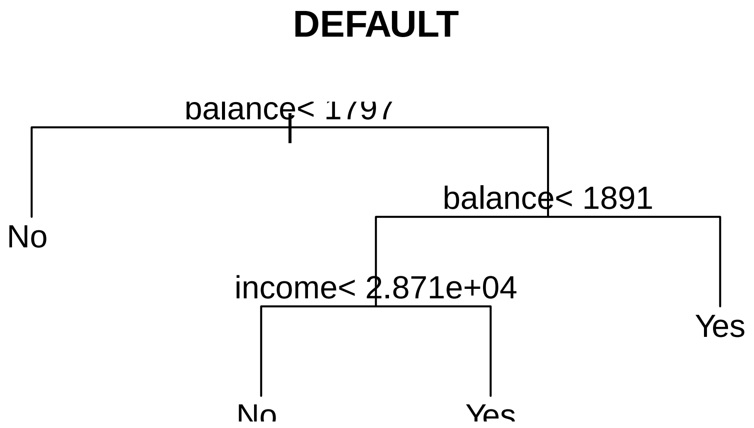
```
[1] 0.655914
```

7.3 Classification Trees

Classification trees offer the same advantages over logistic regression that regression trees do for linear regression. That is, classification trees provide a classification rule that does not assume any form of linearity in the covariates X.

The nice thing is their implimentation in R is nearly identical to that of regression trees.

```
library(rpart)

# estimate regression tree
tree.fit = rpart(default ~ student + balance + income, method="class", data=train)

# plot estimated tree
plot(tree.fit,uniform=TRUE,margin=0.05,main="DEFAULT")
text(tree.fit)
```



We can again use the `predict( )` function to predict the response values for the test data and compute the out-of-sample error (misclassification) rate. We need to specify the `type="class"` option so that the `predict( )` function returns the predicted values $\hat{Y}$.

```
tree.predict = predict(tree.fit, newdata=test, type="class")
mean(tree.predict != test$default)
```

```
[1] 0.027
```

Finally, the error rate can be decomposed by producing the associated confusion matrix and computing the false positive and false negative rates.

```
table(tree.predict, test$default)
```

```
tree.predict   No  Yes
         No  2880   54
         Yes   27   39
```

```
# false positive rate
27/(27+2880)
```

```
[1] 0.009287926
```

```
# false negative rate
54/(54+39)
```

```
[1] 0.5806452
```

8 Clustering

The goal of clustering is to discover groups of similar observations among a set of variables $X_1, \dots, X_p$. Clustering is an example of an **unsupervised learning** method, as we only consider the features $X_1, \dots, X_p$ *without* an associated response Y.

To illustrate clustering methods, we will use the Auto data in the `ISLR` R library. The data set contains information on the gas mileage, number of cylinders, displacement, horsepower, weight, acceleration, year, and origin for 392 vehicles.

```r
library(ISLR)

# load data
data(Auto)
attach(Auto)
```

```
The following object is masked from package:lubridate:

    origin

The following object is masked from package:ggplot2:

    mpg
```

```r
# inspect first few rows
head(Auto)
```

mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
18	8	307	130	3504	12.0	70	1	chevrolet chevelle malibu
15	8	350	165	3693	11.5	70	1	buick skylark 320
18	8	318	150	3436	11.0	70	1	plymouth satellite
16	8	304	150	3433	12.0	70	1	amc rebel sst
17	8	302	140	3449	10.5	70	1	ford torino
15	8	429	198	4341	10.0	70	1	ford galaxie 500

8.1 k-Means

The k-means clustering algorithm uses the variables $X_1, \ldots, X_p$ to partition our observations $1, \ldots, n$ into k non-overlapping groups. The partitioning is done based on the similarity of observations, where similarity is measured using *Euclidean distance*. Consequently, we will need to rescale our data.

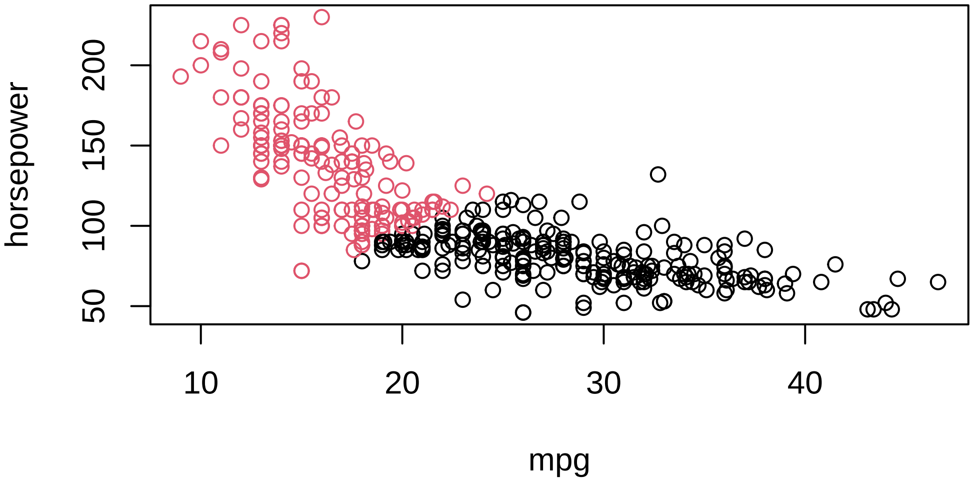
We'll isolate the first seven variables (mpg, cylinders, displacement, horsepower, weight, acceleration, year) and define them as X.

```
# select first seven columns of Auto data
X = Auto[,1:7]

# rescale X's
stdX = scale(X)

# set the name of each row to be the car name stored in the Auto data
rownames(stdX) = Auto$name

# summarize the rescaled data
summary(stdX)
```

```
      mpg            cylinders       displacement      horsepower   
 Min.   :-1.85085   Min.   :-1.4492   Min.   :-1.2080   Min.   :-1.5190  
 1st Qu.:-0.82587   1st Qu.:-0.8629   1st Qu.:-0.8544   1st Qu.:-0.7656  
 Median :-0.08916   Median :-0.8629   Median :-0.4149   Median :-0.2850  
 Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000  
 3rd Qu.: 0.71160   3rd Qu.: 1.4821   3rd Qu.: 0.7773   3rd Qu.: 0.5594  
 Max.   : 2.96657   Max.   : 1.4821   Max.   : 2.4902   Max.   : 3.2613  
     weight        acceleration           year         
 Min.   :-1.6065   Min.   :-2.73349   Min.   :-1.62324  
 1st Qu.:-0.8857   1st Qu.:-0.64024   1st Qu.:-0.80885  
 Median :-0.2049   Median :-0.01498   Median : 0.00554  
 Mean   : 0.0000   Mean   : 0.00000   Mean   : 0.00000  
 3rd Qu.: 0.7501   3rd Qu.: 0.53778   3rd Qu.: 0.81993  
 Max.   : 2.5458   Max.   : 3.35597   Max.   : 1.63432  
```

Let's start by runing the k-means algorithm with $k = 2$ and only using the mpg and horsepower variables.

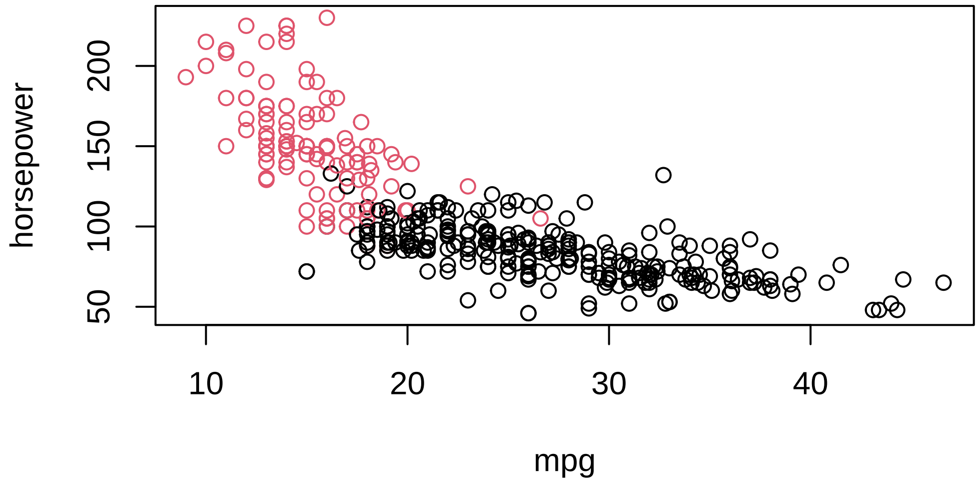
```
# estimate clusters
km2 = kmeans(stdX[,c(1,4)],2)

# plot clusters
plot(mpg, horsepower, col=km2$cluster)
```



We can see how the algorithm divides the observations into two groups: the red observations have a high mpg but lower horsepower, while the black observations have a low mpg but high horsepower.

Now let's try using all seven variables to define the clusters.

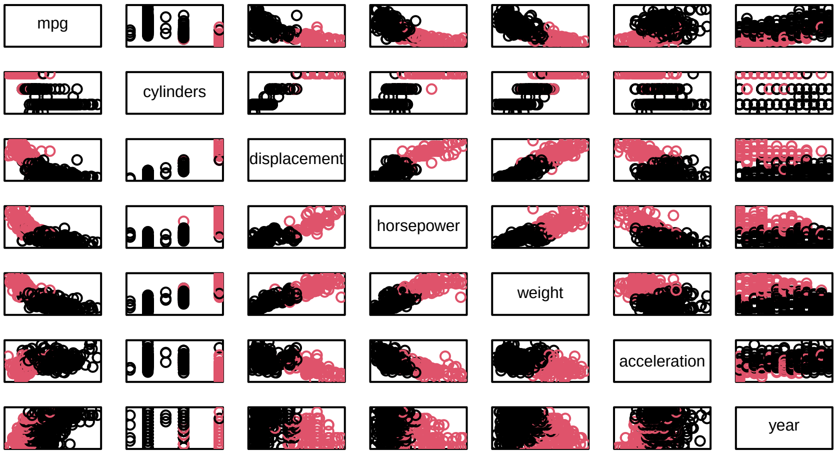
```
# estimate clusters
km2 = kmeans(stdX,2)

# plot clusters pver mpg and horsepower
plot(mpg, horsepower, col=km2$cluster)
```



The plot looks similar: even when we use all variables, the first group of cars (black observations) have a low mpg and high horsepower while the second group (red observations) have a high mpg and low horsepower.

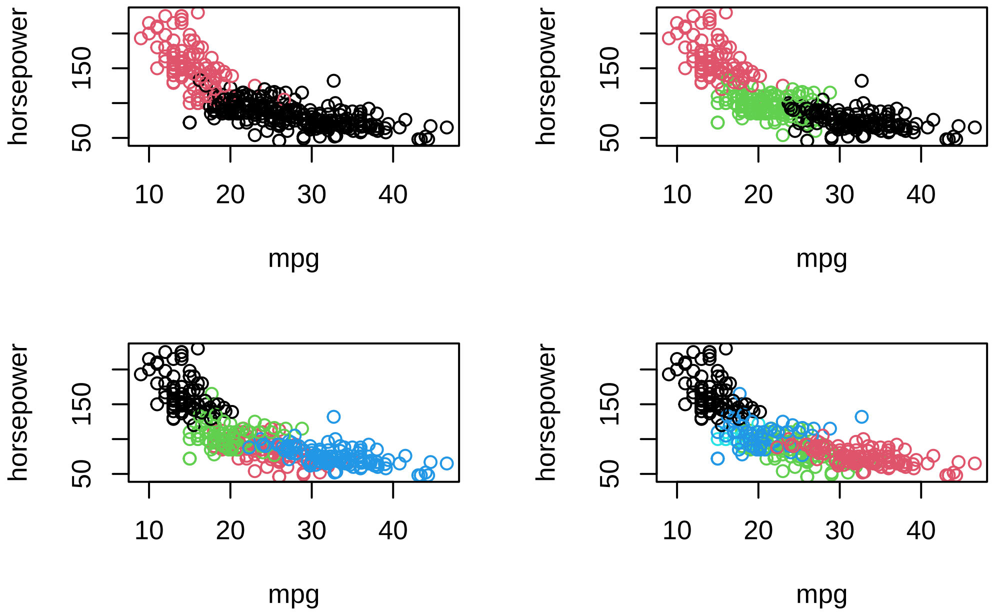
The plot above only shows the clustering solution with respect to two variables (mpg and horsepower). To examine how the clusters are defined over all variables, we can use the `pairs( )` function.

```
# plot clusters over all variables
pairs(stdX, col=km2$cluster, xaxt="n", yaxt="n")
```



Lastly, we can explore clustering solutions for different values of k. For simplicity, we will only examine the clusters for the mpg and horsepower variables.

```
# estimate clusters
km3 = kmeans(stdX,3)
km4 = kmeans(stdX,4)
km5 = kmeans(stdX,5)

# plot clusters over mpg and horsepower
par(mfrow=c(2,2), mar=c(4.1,4.1,2.1,2.1))
plot(mpg, horsepower, col=km2$cluster)
plot(mpg, horsepower, col=km3$cluster)
plot(mpg, horsepower, col=km4$cluster)
plot(mpg, horsepower, col=km5$cluster)
```



As k increases, we get a more granular picture of car segments. However, the problem of interpretation becomes more difficult: What is it that actually differentiates these clusters from each other? We are only plotting the data over two variables, but the other variables also contribute in the determination of cluster assignments.

Moreover, how should we determine an appropriate value of k? Hierachical clustering provides a partial solution.

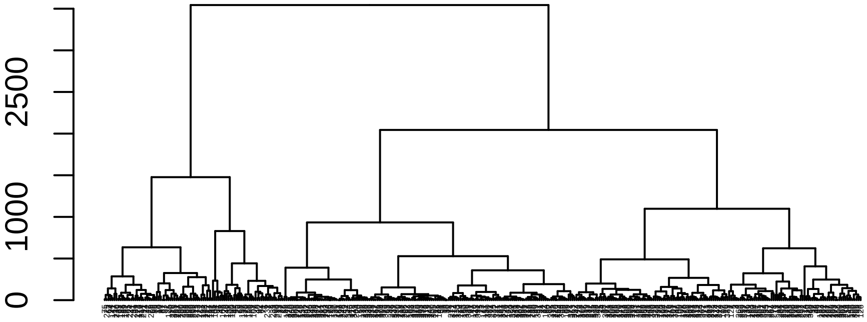
8.2 Hierarchical Clustering

Hierarchical clustering addresses the issue of having to choose the number of clusters k, and instead considers a sequence of clusters from $k = 1, \dots, n$. We'll use the `hclust( )` function and `dendextend` package to fit and plot the output from hierarchical clustering models.

```
library(dendextend)
```

```
# estimate clusters
hc = hclust(dist(X))

# plot clusters
```

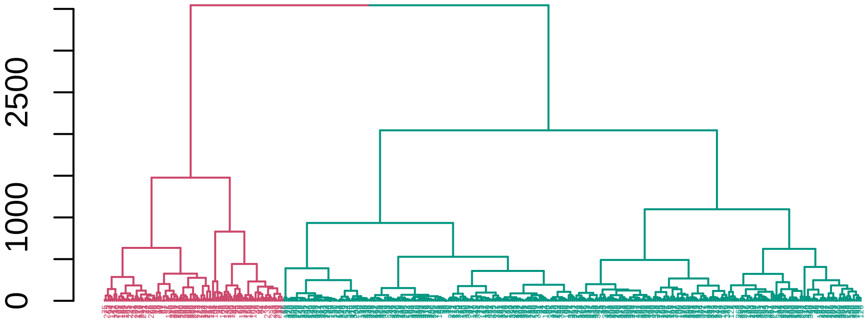
```
dend = as.dendrogram(hc)
labels_cex(dend) = .25
plot(dend)
```



Because we have a few hundred observations, the plot – which called a "dendrogram" – becomes difficult to read and interpret on a small scale (meaning we would need a much larger plotting window!).

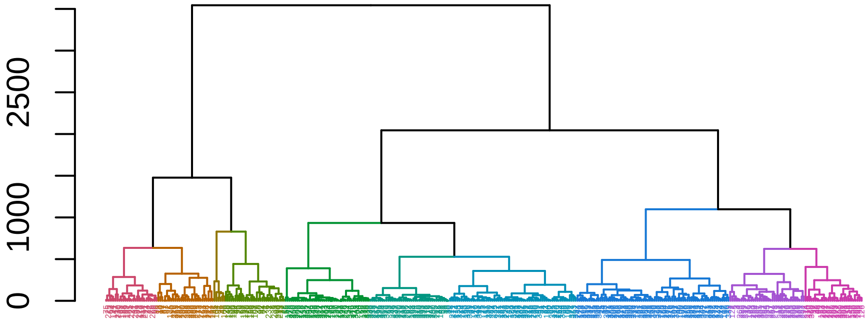
Suppose we were interested in a two group clustering solution. Then we can simply color the dendrogram based on the first split.

```
dend = as.dendrogram(hc)
dend = color_labels(dend,k=2)
dend = color_branches(dend,k=2)
labels_cex(dend) = .25
plot(dend)
```



We can do the same for a larger number of groups, too.

```
dend = as.dendrogram(hc)
dend = color_labels(dend,k=10)
dend = color_branches(dend,k=10)
labels_cex(dend) = .25
plot(dend)
```



Notice that when interpreting a dendrogram, the analyst must still "choose" k, so the problem still hasn't really gone away. The only benefit with hierarchical clustering methods is that the analyst can quickly and easily examine the landscape of clustering solutions to understand how the value of k impacts different clustering solutions.