An Introductory Text on Using R As a Statistical Computing Software

For Use in the Biomedical and Life Sciences

Morgan F. Breitmeyer

Harvard University
Cambridge, MA
August 25, 2017

Contents

Chapter 0

Introduction to R

Contents

This text is a companion to *Introductory Statistics for the Life and Biomedical Sciences*; while the main text focuses on a conceputal introduction to the use of statistics in the life sciences, this supplement provides an analytical and computational introduction using the statistical computing language R. An understanding of the basics of R, how it stores, utilizes, and processes data, and how to analyze these results will be built.

0.1 What is R?

R is an open-source statistical software that allows users to import, transform, and analyze data in order to draw conclusions. In this case, it will be used to perform analyses of biomedical data, and this text will go through the steps to do so from installation to drawing conclusions.

0.1.1 Installation of R

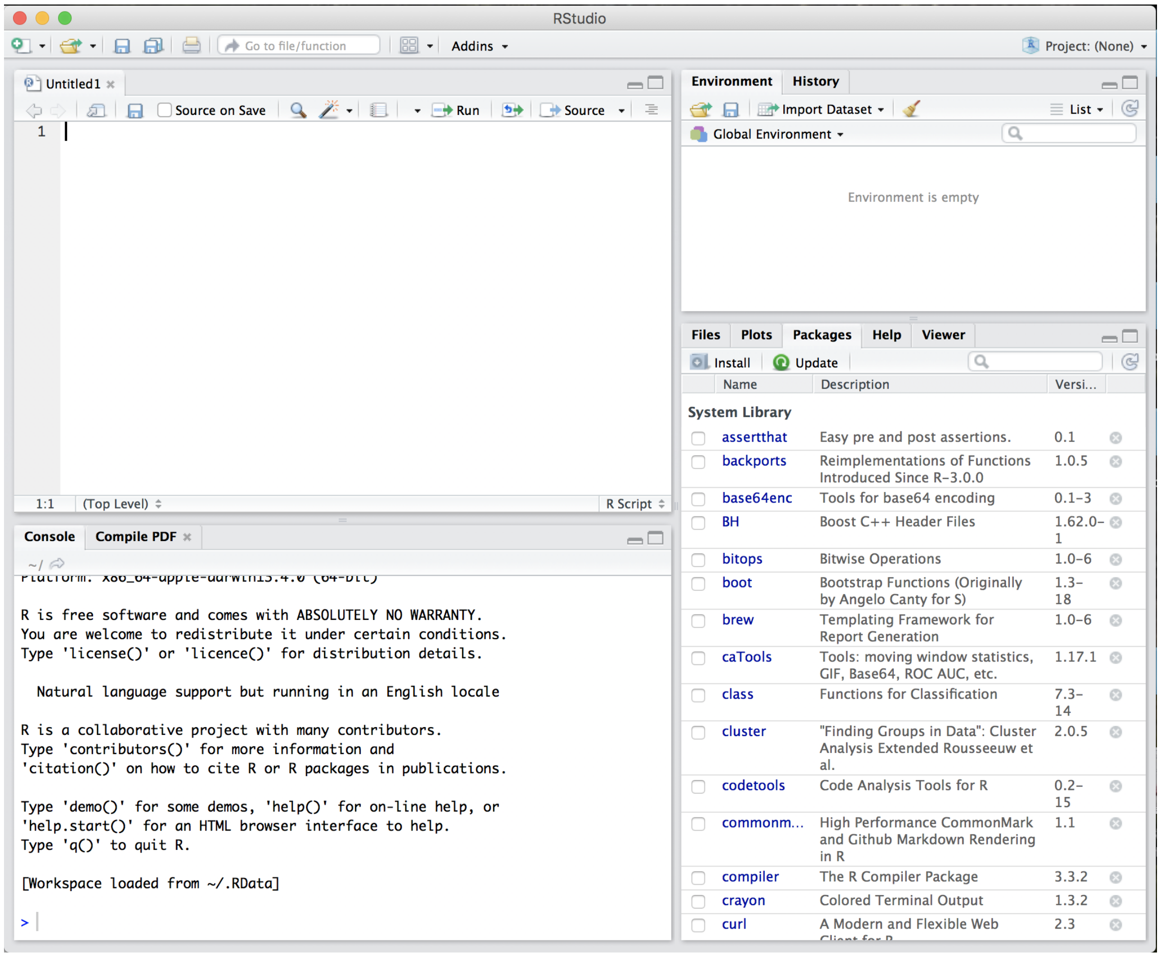
R is freely available on the internet, and it is recommended to download it from `http://cran.us.r-project.org/`, where there are up to date versions for Windows, Mac OS X, and Linux. This link will provide instructions for a complete download.

0.1.2 Introduction to RStudio

RStudio is a platform that makes using R significantly easier, providing a visual framework for working. It can be downloaded from `https://www.rstudio.com/products/rstudio/download/`, by scrolling to the bottom and selecting the appropriate operating system from *Installers* under *Installers for Supported Platforms*.

RStudio is not necessary in order to use R, but it is highly recommended for organizational and accesibility reasons.

Upon opening RStudio, something similar to Figure 1 should appear. The four pane window environment is standard and provides easy access to many of the most commonly utilized tools. Each corner contains one pane, and each pane can include several tabs. The four panes shown in Figure 1 demonstrate a common starting setup environment, but different versions of Rstudio may look a bit different. Furthermore, panes can be moved around or closed. Therefore, this section will go through the various panes, their elements and uses, and how to access them.



```
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

>
```

Name	Description	Versi...
System Library		
assertthat	Easy pre and post assertions.	0.1
backports	Reimplementations of Functions Introduced Since R–3.0.0	1.0.5
base64enc	Tools for base64 encoding	0.1–3
BH	Boost C++ Header Files	1.62.0–1
bitops	Bitwise Operations	1.0–6
boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3–18
brew	Templating Framework for Report Generation	1.0–6
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.17.1
class	Functions for Classification	7.3–14
cluster	"Finding Groups in Data": Cluster Analysis Extended Rousseeuw et al.	2.0.5
codetools	Code Analysis Tools for R	0.2–15
commonm...	High Performance CommonMark and Github Markdown Rendering in R	1.1
compiler	The R Compiler Package	3.3.2
crayon	Colored Terminal Output	1.3.2
curl	A Modern and Flexible Web	2.3

Figure 1: The default RStudio layout.

The Script Editor

The top left pane in Figure 1 shows the script editor, used to edit a R script file. Think of this pane as similar to any text editor, providing a working space for the code being utilized and a means to save that for later. We recommend that all work is done in a R script file so no information is lost. If a script editor is not visible, a new R script file can be created by going to *File* > *New File* > *R Script*.

In order to run a command in the script editor, you must place your curser on the line you want to run and type command + Return on Mac or Ctrl + Enter on Windows or Linux. Alternatively, at the top of the script editor pane is the run button, which can be clicked after highlighting the code to be run. Try this for yourself by opening a script file and running the following example. If successful, you shall see the result of this simple calculation show up in the bottom left pane. One of the most basic capabilties of R is as a calculator as shown in this example below. The light grey box indicates a section of R code, where the purple text is code that has been inputted and the second line, as indicated by the hashtags and [1] is output in response to the command in the first line.

```
5+3

## [1] 8
```

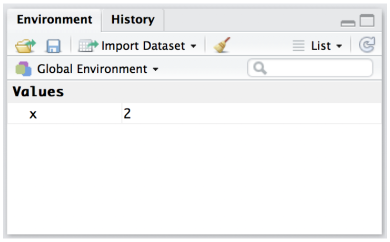
The Console

The bottom left pane in Figure 1 is home to the console tab. This is the machinery in R that does the actual computing. When commands are run in the script editor, they and their output show up in the console, so the console is an important pane to always have in your RStudio environment. If the console is not visible, it is likely minimized and can be reopened by going to *View > Move Focus to Console*.

The Environment Tab

The top right pane in Figure 1 hosts the Environment tab, which is the home of all stored information in R, such as datasets and variables. The creation of any variables will show up here. For example, upon running the following command in the script editor, a variable called *x* should come up under Values in the Environment tab as shown in Figure 2. The equals sign in the command below indicates that the variable of name x is taking on the numerical value 2. A variable created in this way is stored in R for later use and can later be called upon by referencing its variable name, *x*.

```
x = 2
```



```
x    2
```

Figure 2: An example of how the global environment stores variables.

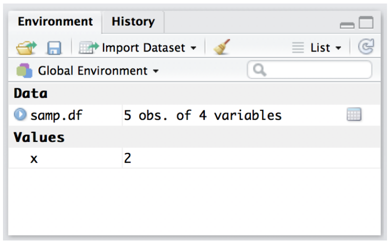
To access a variable once created, the variable name can be called upon in the script editor and various operations can be performed to it. For example, the below command uses the above created variable to get another result. More examples of this will be shown in Section **??**.

```
x*2

## [1] 4
```

If the Environment tab is not visible, the upper right pane is likely minimized and can be reopened by going to *View > Show Environment*.

The Files Tab

In the lower right pane of Figure 1, several tabs can be seen. The Files tab is a directory of all the files on the local computer and can be used to access other files, such as saved datasets or previously created R scripts. Navigating betweeen folders in this tab is similar to doing so on your computer. This is where datsets with the file type *Rdata* or *Rda* can be loaded into the R environment.[1] Upon locating a dataset by navigating to its location on your computer, it can be loaded into the environment by clicking on it in this Files tab. Doing this will cause the dataset to show up in the Environment tab in the same manner as described in the previous section. For example, if the dataset called *samp_df.Rda* is loaded from the Files tab, the global environment should looks something like that seen in Figure 3, showing both the variable *x* and the dataset *samp.df*.



```
Data
samp.df     5 obs. of 4 variables
Values
x           2
```

Figure 3: An example of how the global environment stores datasets and variables.

If the Files tab is not visible, the lower right pane is likely minimized and can be reopened by going to *View > Show Files*.

The Plots Tab

The next tab seen is the Plots tab, which is where plots will be displayed when created in a R script file. Section 1.4.4 will go into more detail as to how to create plots, but just note that this is where they will show up.

The Packages Tab

Because R is an open-source software, anyone can creates packages that contain functions, data, or other useful programs. Packages are then shared and built upon to make the knowledge base of the community greater. The Packages tab is where these packages are accessible in RStudio. Many packages come standard with the installation of RStudio and can be seen here. By checking the box to the left of a package, all of its contents will be available for use. A package has been created to accompany this book, and it can be included for use in the packages tab. Section 0.5 will discuss this specific package in further detail.

The Help Tab

The Help Tab provides basic helpful information on most features of R and RStudio. It can be searched, as well as accessed by typing a question mark next to any function or package in the script file. For example, the following command will pull up in the Help tab all the information about the mean function.

```
?mean
```

[1] A data file that is not in the format *Rdata* must be imported rather than just loaded. This can be done in the Environment tab using the button *Import Dataset* or from the script editor using the command *load()* with the dataset name in quotation marks inside the parentheses. Insert reference here for an example of doing this

The Viewer Tab

0.2 Working with Data in R

R is built to work with many types of data, which can hold different pieces of information. From large datasets to small lists of names, R is designed to handle data relative to its purpose and input form. Several examples will be presented to illustrate many of these different means of handling data.

0.2.1 Datatypes

Numerical

Categorical

Checking the Datatype

0.2.2 Vectors

Vectors are one dimensional arrays used to store lists of information. A few examples of a vector could be a grocery list, a list of ages of people in a room, or a guest list. Vectors can be created using a concatenation function, *c()*. The values intended to be in the list are separated by commas inside the parentheses. For example, a vector of numbers 1 through 5 can be made as follows. Note that the first line shows this method of creating a list, while the second line assigns the variable *num.vector* to be equal to the vector. The third line calls on the variable just created and simply prints it.

```
c(1,2,3,4,5)

## [1] 1 2 3 4 5

num.vector = c(1,2,3,4,5)
num.vector

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

The same numerical vector can also be created using a colon, which implies the numbers between the end values. For example, the following command creates the same vector as above, but without the concatenation command.

```
num.vector2 = 1:5
num.vector2

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

Vectors can also be used for characters or strings, meaning a group of characters forming a word. The following is an example of a list of characters, where each of the characters must be individually surrounded by quotation marks in order to signal to R that it is a character and not a variable.

```
alph.vector = c("a", "b", "c", "d", "e")
alph.vector

## [1] "a" "b" "c" "d" "e"
```

Operating with Vectors

Vectors can be used for computational purposes by applying many mathematical functions. R applies computation to vectors pairwise, meaning that each element is paired with the corresponding element in the opposite vector and the computation applied accordingly. Note that vectors should be the same length in order to apply such computations. For example, addition, multiplication, and squaring are done pairwise as follows,

```
a = c(1,2,3,4)
b = c(5,6,7,8)
a+b

## [1]  6  8 10 12

a*b

## [1]  5 12 21 32

a^2

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

Furthermore, a single numerical computation can be applied to a vector, by applying the single computation to each element in the vector. For example, multiplying a vector by 2, as in the following example, individually multiplies every element in the vector by 2.

```
2*a

## [1] 2 4 6 8
```

Note that attempting to apply such mathematical operations to a vector of strings will not produce results. An error like the following will appear,

```
2*alph.vector

## Error in 2 * alph.vector: non-numeric argument to binary operator
```

Another use of the concatenating function *c()* is to combine multiple vectors into a single vector. This is done as follows,

```
c(a,b)

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

It is reccomended that you try out these and other examples for yourself in order to fully appreciate how this works.

Accessing Elements of a Vector

Once a vector has been created, it is helpful to be able to pull out certain parts of it. Vectors are indexed, meaning that each element in the vector corresponds to a location identified by number, numbered 1 through the length of the matrix. To access an element in the vector, brackets are used, which signals to R that the element at the location specified within the brackets is wanted. For example, to obtain the first element in the vector above *b*, the correct command is *b[1]*, as shown below

```
b[1]

## [1] 5
```

In order to obtain more than one element in a vector, the notation is more specific. A vector of index locations must be passed into the brackets, such as in the following two examples,

```
b[1:3]

## [1] 5 6 7

b[c(1,3)]

## [1] 5 7
```

If every element in a vector is wanted except for one or two, a negative sign can be applied to the vector contained inside the brackets. For example, the following removes the second and third elements of the list *b*.

```
b[-c(2,3)]

## [1] 5 8
```

0.2.3 Matrices

A matrix is similar to a vector in that it stores information except that it is a two dimensional array. The function *matrix()* can be used to create two dimensional matrices. The input of this function is the following

- A vector containing all the matrix entries
- A specification of *byrow = TRUE* which fills the matrix firstly left to right and then top to bottom or a specification of *byrow = FALSE* which fills columns top to bottom first moving from left to right
- A specification of *nrow =* which outlines how many rows are in the matrix

To go through a basic example,

```
mat1 = matrix(1:8, byrow = TRUE, nrow = 4)
mat1

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

If you change the *byrow* specification, you get the following

```
mat2 = matrix(1:8, byrow = FALSE, nrow = 4)
mat2

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

Just like with vectors, mathematical computations can be applied to matrices. Again, the computations are applied pairwase unless a single numerical value is given which applies to all elements. Some examples are as follows,

```
mat1+mat2

##      [,1] [,2]
## [1,]    2    7
## [2,]    5   10
## [3,]    8   13
## [4,]   11   16

mat1*mat2

##      [,1] [,2]
## [1,]    1   10
## [2,]    6   24
## [3,]   15   42
## [4,]   28   64

2*mat1

##      [,1] [,2]
## [1,]    2    4
## [2,]    6    8
## [3,]   10   12
## [4,]   14   16
```

Matrices can become significantly more complicated, and in such structures, form the basis for much statistical computation. Consider the following table,

Name	Weight	Gender	Age
John	210	Male	22
Suzie	140	Female	45
Mary	120	Female	35
Bob	180	Male	50
Dave	160	Male	70

There are two ways to create a matrix from this information: create a long list of values as used above or create several individual vectors and combine them into a matrix. The second method is laid out here,

```
names  = c("John", "Suzie", "Mary", "Bob", "Dave")
weights = c(210, 140, 120, 180, 160)
genders = c("Male", "Female", "Female", "Male", "Male")
ages = c(22, 45, 35, 50, 70)

samp.matrix = matrix(c(names, weights, genders, ages), byrow = FALSE, nrow = 5)
samp.matrix

##      [,1]    [,2]  [,3]     [,4]
## [1,] "John"  "210" "Male"   "22"
## [2,] "Suzie" "140" "Female" "45"
## [3,] "Mary"  "120" "Female" "35"
## [4,] "Bob"   "180" "Male"   "50"
## [5,] "Dave"  "160" "Male"   "70"
```

Notice how the *c()* function is required to concatenate the several vectors into one long vector, as well as the specification of *nrow()*, so as not to get a single long list of values. Try this out for yourself, perhaps changing values for *nrow* or *byrow* to see how it changes the output.

When this matrix was created, the variable names that helped to identify the purpose of each column were lost. Therefore, it may be desirable to add column names to the matrix. This can be done using the function *colnames()*, which is applied to the variable that represents the matrix and takes in a list of strings as input. The following example shows how this can be done.

```
colnames(samp.matrix) = c("Name", "Weight", "Gender", "Age")
samp.matrix

##      Name    Weight Gender   Age
## [1,] "John"  "210"  "Male"   "22"
## [2,] "Suzie" "140"  "Female" "45"
## [3,] "Mary"  "120"  "Female" "35"
## [4,] "Bob"   "180"  "Male"   "50"
## [5,] "Dave"  "160"  "Male"   "70"
```

Accessing Elements of a Matrix

Because matrices are two dimensional, accessing elements inside them requires two dimensions. As with vectors, bracket notation is used with a comma in the middle indicating the distinction between rows and columns. Inside the brackets, the order is rows comma columns. The best way to see this is through examples. Using the *samp.matrix* created just above, the item in the first row and first column can be pulled as follows,

```
samp.matrix[1,1]

##   Name
## "John"
```

If instead, the entire first row was desired, this could be done by putting a vector of indeces wanted for the columns such as this

```
samp.matrix[1,1:4]

##   Name Weight Gender    Age
## "John"  "210" "Male"   "22"
```

Alternatively, if the column section after the comma is left blank, this signals to R that all columns are desired. The result would be the same as above,

```
samp.matrix[1,]

##   Name Weight Gender    Age
## "John"  "210" "Male"   "22"
```

Combining Matrices

Often times, it is useful to combine two matrices or a matrix and a vector into one larger matrix. This can be done using the function *cbind()* or *rbind()*. These work similarly to the concatenation function *c()*, but with multiple dimensions. *cbind()* combines along the column dimension, while *rbind()* does so along the rows. The function takes as input the elements to be combined in the order of choice, separated by commas. For example, in the example below, an extra column representing smoking status is added to the matrix.

```
Smoker = c("Yes", "No", "Yes", "No", "No")
samp.matrix2 = cbind(samp.matrix, Smoker)
samp.matrix2

##      Name    Weight Gender   Age  Smoker
## [1,] "John"  "210"  "Male"   "22" "Yes"
## [2,] "Suzie" "140"  "Female" "45" "No"
## [3,] "Mary"  "120"  "Female" "35" "Yes"
## [4,] "Bob"   "180"  "Male"   "50" "No"
## [5,] "Dave"  "160"  "Male"   "70" "No"
```

Similarly, an extra row could be added to the matrix representing another individual instead using *rbind()* such as the following,

```
samp.matrix3 = rbind(samp.matrix2, c("Lucy", 130, "Female", 18, "No"))
samp.matrix3

##      Name    Weight Gender   Age  Smoker
## [1,] "John"  "210"  "Male"   "22" "Yes"
## [2,] "Suzie" "140"  "Female" "45" "No"
## [3,] "Mary"  "120"  "Female" "35" "Yes"
## [4,] "Bob"   "180"  "Male"   "50" "No"
## [5,] "Dave"  "160"  "Male"   "70" "No"
## [6,] "Lucy"  "130"  "Female" "18" "No"
```

0.2.4 Dataframes

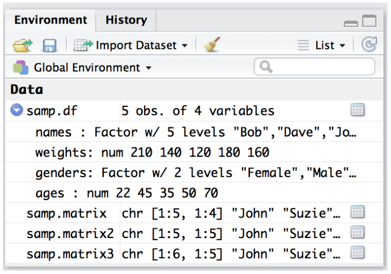
Another method that R uses to store data is called a dataframe, which is similar to a matrix, with slight differences. It is a two dimensional array just like a matrix, but the important difference is that it can handle more than a single data type. Looking at the output of the matrix created above, it is noticable that every value has quotation marks on it in the output. This indicates that R is viewing every value of that matrix as a character, even the numbers. This is problematic in case computations on the numerical values were desired, such as calculating the mean weight of all individuals. If you tried to do that, R would return an error. Therefore, it is a better idea to store 2 dimensional data that is not a uniform datatype in a dataframe.

```
samp.matrix

##      Name    Weight Gender   Age
## [1,] "John"  "210"  "Male"   "22"
## [2,] "Suzie" "140"  "Female" "45"
## [3,] "Mary"  "120"  "Female" "35"
## [4,] "Bob"   "180"  "Male"   "50"
## [5,] "Dave"  "160"  "Male"   "70"
```

In order to create a dataframe in R, the function *data.frame()* is used. As input, it takes a list of vectors that are the desired columns in the dataframe. For example, a dataframe that presents the same information as the matrix *samp.matrix* can be created as follows,

```
samp.df = data.frame(names, weights, genders, ages)
samp.df

##   names weights genders ages
## 1  John     210    Male   22
## 2 Suzie     140  Female   45
## 3  Mary     120  Female   35
## 4   Bob     180    Male   50
## 5  Dave     160    Male   70
```

Notice how none of the values have quotation marks on them anymore. This begs two questions: whether the strings are still strings and whether the numerical values are now indeed numerical rather than strings. There are two ways to determine this. The easiest is to look in the Environment tab and to find the *samp.df* element. Clicking on the little blue triangle on the left will reveal something like the following image in Figure 4. Notice how under *samp.df*, each column can be seen as an individual row, listed by its column name. Next to the columns *names* and *genders* is the datatype "Factor", which is a desired string classification. Furthermore, the columns *weights* and *ages* are indeed numerical.



```
samp.df       5 obs. of 4 variables
  names : Factor w/ 5 levels "Bob","Dave","Jo…
  weights: num 210 140 120 180 160
  genders: Factor w/ 2 levels "Female","Male"…
  ages : num 22 45 35 50 70
samp.matrix  chr [1:5, 1:4] "John" "Suzie"…
samp.matrix2 chr [1:5, 1:5] "John" "Suzie"…
samp.matrix3 chr [1:6, 1:5] "John" "Suzie"…
```

Figure 4: An example of how the global environment displays important information about a dataframe.

Accessing Elements of a Dataframe

Accessing elements of a dataframe can be a bit trickier than with vectors or matrices. As with matrices, two dimensional indeces can be used with brackets to access individual elements, rows, or columns. For example, the first element and the first row can be retrieved respectively as follows,

```
samp.df[1,1]

## [1] John
## Levels: Bob Dave John Mary Suzie

samp.df[1,]

##   names weights genders ages
## 1  John     210    Male   22
```

Note that if you put a one dimensional list inside the brackets, R will default to pulling the column with that index. For example, the following pulls the first column.

```
samp.df[1]

##   names
## 1  John
## 2 Suzie
## 3  Mary
## 4   Bob
## 5  Dave
```

It must be noted that dataframes maintain column vector names upon creation, and this is an important thing to know when trying to access elements. With dataframes, the notation *dataframe$variable_name* can be used to access an entire column, where the dollar sign signifies that you are looking for the column with that variable name that follows. For example, the following returns the *weights* column of the dataframe.

```
samp.df$weights

## [1] 210 140 120 180 160
```

This command returns a vector of the weights, and as discussed in Section 0.2.2, a specific item in that vector can be accessed using a one dimensional vector inside the brackets. For example, the following returns the first element in the column *weights*.

```
samp.df$weights[1]

## [1] 210
```

Altering Dataframes

When working with dataframes, it is often reccomended that you create a new column if you want to alter the data in some form. For example, if you wanted to convert the weights given to kg, you could add another column to the dataframe to do so. In order to add a new column, you simply call *dataframe$new_variable_name* and set it equal to something in the same way that you define a variable. For example, the following command adds this new column of weights in kilograms.

```
samp.df$weights_kg = samp.df$weights/2.2
samp.df

##   names weights genders ages weights_kg
## 1  John     210    Male   22   95.45455
## 2 Suzie     140  Female   45   63.63636
## 3  Mary     120  Female   35   54.54545
## 4   Bob     180    Male   50   81.81818
## 5  Dave     160    Male   70   72.72727
```

The functions *cbind()* and *rbind()* can be applied in the same way to dataframes as matrices. It is reccomended that you try this out for yourself.

0.3 Functions

0.4 Some Programming Background

0.5 The OIBioStat Package

All the datasets used in the text can be accessed by downloading the OIBiostat package from R. Run the following command to download the package:

Each time the package is needed, run the following command:

```
require(OIBioStat)
```

To access a dataset in the package, simply run the `data()` command, which will load it into the R environment. To see this has been done, it will pop up in the top right of RStudio in the pane labeled *Environment* under *Data*. For example, a dataset called `swim` is in the package and can be loaded as follows,

```
data(swim)
```

0.6 Exercises

Chapter 1

Introduction to Basic Techniques

Contents

This chapter introduces basic commands for manipulating datasets, including how to calculate numerical summaries, create tables from data, and make graphical plots.

1.1 Case study: preventing peanut allergies

The `LEAP` dataset contains the results of the "Learning Early About Peanut Allergy" (LEAP) study, an experiment conducted to assess whether early exposure to peanut products reduces the probability of peanut allergies developing. To access the documentation associated with the dataset, run the following command:

```
help(LEAP)
```

A file will appear in the Help pane that provides some basic information about the dataset:

- **Description:** A general overview of the dataset.
- **Usage:** Instructions for how to load the dataset.

- **Format:** The names and descriptions of each variable in the dataset, including information about variable type and measurement units.
- **Details:** Additional details about the conditions under which the data were collected.
- **Source**: Information about where the data originates from.

To view the dataset itself, run:

```
View(LEAP)
```

In the main console pane, a window will appear that shows the entire dataset. The variable names are displayed across the top, as the names of the columns. Data for each case in the study are contained within the rows. Note that the farthest left column shows the **indices**, which are computer-generated values that allow specific rows in the dataset to be accessed. These can be used to view a specific portion of the data.

For example, to print out data contained in the first 5 rows and first 3 columns of the data, use the following syntax:

```
LEAP[1:5,1:3]

##   participant.ID    treatment.group age.months
## 1    LEAP_100522 Peanut Consumption     6.0780
## 2    LEAP_103358 Peanut Consumption     7.5893
## 3    LEAP_105069   Peanut Avoidance     5.9795
## 4    LEAP_105328 Peanut Consumption     7.0308
## 5    LEAP_106377   Peanut Avoidance     6.4066
```

The bracket notation after the dataset name implies location; the syntax `dataset[rows,columns]` specifies rows 1 through 5 and columns 1 through 6. To access only the first 5 rows, but all the columns, leave an empty space where the columns would usually be specified:

```
## note the space (or lack of text after the comma)
LEAP[1:5, ]

##   participant.ID    treatment.group age.months    sex primary.ethnicity
## 1    LEAP_100522 Peanut Consumption     6.0780 Female             Black
## 2    LEAP_103358 Peanut Consumption     7.5893 Female             White
## 3    LEAP_105069   Peanut Avoidance     5.9795   Male             White
## 4    LEAP_105328 Peanut Consumption     7.0308 Female             White
## 5    LEAP_106377   Peanut Avoidance     6.4066   Male             White
##   overall.V60.outcome
## 1            PASS OFC
## 2            PASS OFC
## 3            PASS OFC
## 4            PASS OFC
## 5            PASS OFC
```

Alternatively, since there are 6 columns in the dataset, the command `LEAP[1:5,1:6]` would also achieve the same result. This is a common theme in R – there can be several ways to accomplish a desired result.

OI Biostat Table 1.1 shows the participant ID, treatment group, and overall outcome for five patients. It is not specified in the main text, but the data are specifically from rows 1, 2, 3, 529, and 530. The command `c()` can be used to bind the row numbers into a list, as well as to create a list of the desired columns. Columns can be referred to by name, instead of by number:

```
## OI Biostat Table 1.1
LEAP[c(1, 2, 3, 529, 530),c("participant.ID", "treatment.group",
                         "overall.V60.outcome")]

##     participant.ID    treatment.group overall.V60.outcome
## 1      LEAP_100522 Peanut Consumption            PASS OFC
## 2      LEAP_103358 Peanut Consumption            PASS OFC
## 3      LEAP_105069   Peanut Avoidance            PASS OFC
## 639    LEAP_994047   Peanut Avoidance            PASS OFC
## 640    LEAP_997608 Peanut Consumption            PASS OFC
```

Two-way summary tables organize the data according to two variables and display the number of counts matching each combination of variable categories. The following code corresponds to *OI Biostat* Table 1.2, which groups participants into categories based on treatment group and overall outcome. In the `table()` command, the first variable specifies the rows and the second variable specifies the columns. The addition of the `addmargins()` command prints the sums of the rows and columns on the sides of the table.

```
## Table 1.2
table(LEAP$treatment.group, LEAP$overall.V60.outcome)

##
##                        FAIL OFC PASS OFC
##   Peanut Avoidance           36      227
##   Peanut Consumption          5      262

addmargins(table(LEAP$treatment.group, LEAP$overall.V60.outcome))

##
##                        FAIL OFC PASS OFC Sum
##   Peanut Avoidance           36      227 263
##   Peanut Consumption          5      262 267
##   Sum                        41      489 530
```

1.2 Data Basics

Entire datasets can be partitioned into data frames using bracket notation. For example, *OI Biostat* Table 1.3 shows a data frame consisting of rows 1-3 and 150 (and all columns) from the `frog.altitude` dataset. The data frame can then be given a specific name, such as `frog.df`, and directly called on for later operations.

```
## Table 1.3
frog.df = frog.altitude.data[c(1:3, 150),]
frog.df

##     altitude latitude clutch.size body.size clutch.volume egg.size
## 1   3,462.00    34.82    181.9701  3.630781      177.8279 1.949845
## 2   3,462.00    34.82    269.1535  3.630781      257.0396 1.949845
## 3   3,462.00    34.82    158.4893  3.715352      151.3561 1.949845
## 150 2,597.00    34.05    537.0318        NA      776.2471 2.238721
```

Similarly, matrix notation can be used to create *OI Biostat* Table 1.5.

```
## Table 1.5
famuss[c(1,2,3,595),c( "sex", "age", "race", "height", "weight", "actn3.r577x",
                       "ndrm.ch")]

##         sex age      race height weight actn3.r577x ndrm.ch
## 1    Female  27 Caucasian   65.0    199          CC    40.0
## 2      Male  36 Caucasian   71.7    189          CT    25.0
## 3    Female  24 Caucasian   65.0    134          CT    40.0
## 1348 Female  30 Caucasian   64.0    134          CC    43.8
```

1.4 Numerical Data

Numerical summaries can be quickly and easily calculated using R. The `summary()` command is one way to access several numerical summaries at once, including the minimum and maximum values of a variable:

```
summary(frog.altitude.data$clutch.volume)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   151.4   609.6   831.8   882.5  1096.0  2630.0
```

1.4.1 Measures of center: mean and median

The **mean** of a numerical value is the sum of all observations divided by the number of observations, where $x_1, x_2, \dots, x_n$ represent the n observed values:

$$\overline{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

This calculation can be completed in R the same way as a hand calculation is done:

```
# identify n, the number of observations in the data
n = length(frog.altitude.data$clutch.volume)
n

## [1] 431

# calculate the sum of all observations and divide by n
sum(frog.altitude.data$clutch.volume)/n

## [1] 882.474
```

Alternatively, the R function `mean()` can be directly applied to the variable of interest:

```
x.bar = mean(frog.altitude.data$clutch.volume)
x.bar

## [1] 882.474

## round to the first decimal
round(x.bar, 1)

## [1] 882.5
```

To identify the **median** value, use the following command:

```
median(frog.altitude.data$clutch.volume)

## [1] 831.7638
```

1.4.2 Measures of spread: standard deviation and variance

The distance of a single observation from the mean is its **deviation**. The following command produces the deviations for the 1^{st}, 2^{nd}, 3^{rd}, and 431^{th} observations in the `clutch.volume` variable.

```
frog.altitude.data$clutch.volume[c(1,2,3,431)]-x.bar

## [1] -704.64604 -625.43440 -731.11786   50.78032
```

The sample **standard deviation** is computed as the square root of the **variance**, which is the sum of squared deviations divided by the number of observations minus 1.

$$s = \sqrt{\frac{(x_1 - \overline{x})^2 + (x_2 - \overline{x})^2 + \cdots + (x_n - \overline{x})^2}{n-1}}$$

The following steps illustrate how to calculate the variance and standard deviation using the formula:

```
# calculate all deviations
dev = frog.altitude.data$clutch.volume - x.bar

# sum the squares of the deviations and divide by n - 1
var = (sum(dev^2))/(n-1)
var

## [1] 143680.9

# take the square root of the variance
sd = sqrt(var)
sd

## [1] 379.0527
```

Alternatively, use the R functions `var()` and `sd()`:

```
var(frog.altitude.data$clutch.volume)

## [1] 143680.9

sd(frog.altitude.data$clutch.volume)

## [1] 379.0527
```

Variability can also be measured using the **interquartile range (IQR)**, which equals the third quartile (the 75^{th} percentile) minus the first quartile (the 25^{th} percentile).

```r
IQR(frog.altitude.data$clutch.volume)

## [1] 486.9009
```

1.4.3 Robust statistics

In the `frog.altitude` dataset, there are four extreme values for clutch volume that are larger than 2,000 mm^3. To illustrate how the summary statistics are influenced by extreme values, *OI Biostat* Table 1.15 shows summary statistics for the data without the four largest observations.

The subset of the data with values less than 2,000 mm^3 can be pulled out by first specifying a logical condition, which assigns `TRUE` or `FALSE` to each entry.

```r
# logical condition
less.than.2000 = frog.altitude.data$clutch.volume<= 2000

# view the first 5 values
less.than.2000[1:5]

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

Bracket notation can then be used to calculate summary statistics specifically for the values in `clutch.volume` that satisfy the logical condition.

```r
## robust estimates
median(frog.altitude.data$clutch.volume[less.than.2000])

## [1] 831.7638

IQR(frog.altitude.data$clutch.volume[less.than.2000])

## [1] 493.9186

## non-robust estimates
mean(frog.altitude.data$clutch.volume[less.than.2000])

## [1] 867.9425

sd(frog.altitude.data$clutch.volume[less.than.2000])

## [1] 349.1596
```

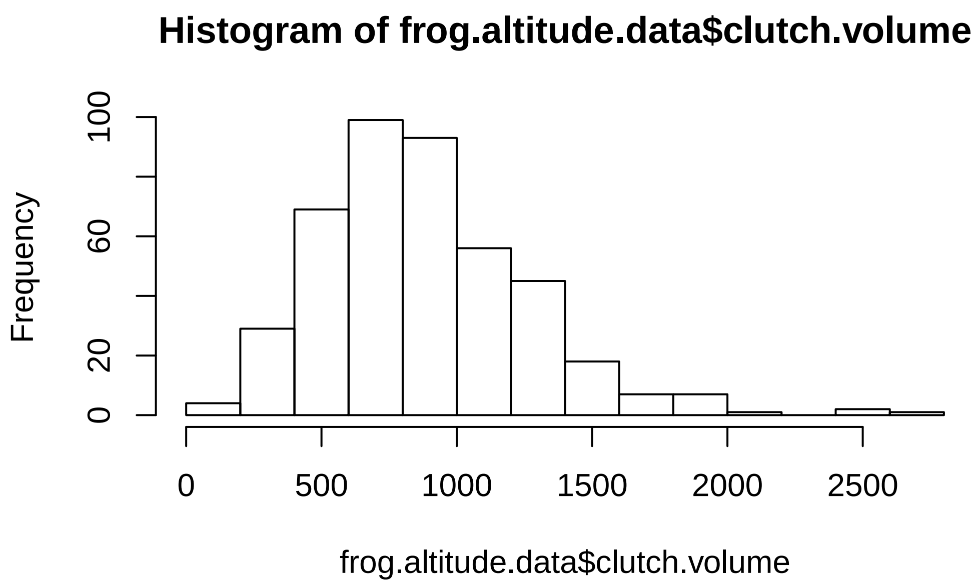
1.4.4 Visualizing distributions of data

While most numerical summaries can be calculated by hand, R is essential for creating graphical summaries.

Histograms

A **histogram** provides a view of data density. Observations are sorted into different bins based on their value, and the histogram shows the number of observations in each bin. The following command produces a histogram of the `clutch.volume` variable.

```r
hist(frog.altitude.data$clutch.volume)
```



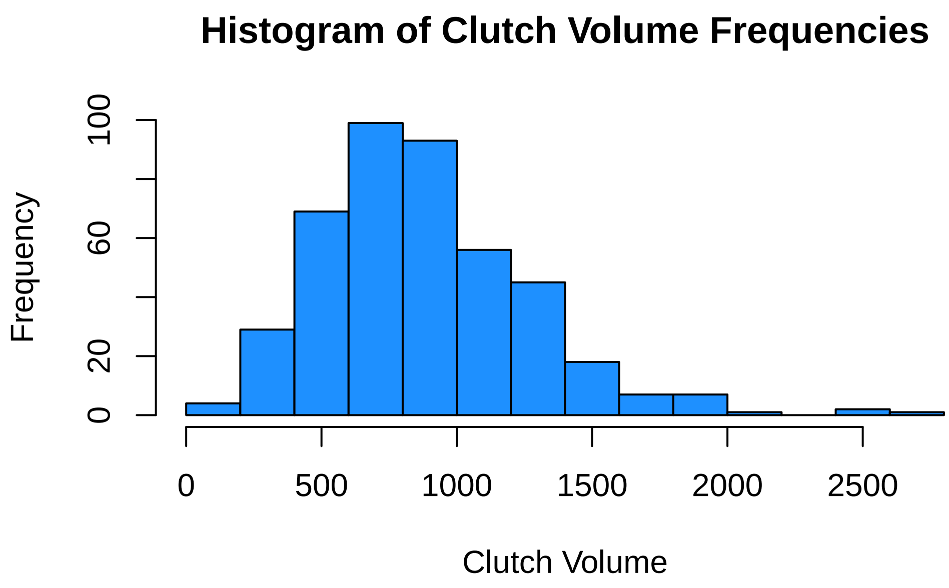
The `hist()` function takes several arguments:

- `x`: variable of interest
- `breaks`: number of bins
- `col`: color of the bars, enclosed in `""`
- `xlab`: x-axis label, enclosed in `""`
- `ylab`: y-axis label, enclosed in `""`
- `xlim`: range of values for the x-axis, in the form `c(lower bound, upper bound)`
- `ylim`: range of values for the y-axis, in the form `c(lower bound, upper bound)`
- `main`: main title of the plot, enclosed in `""`
- `plot`: if `TRUE` (default), a histogram is plotted; otherwise, data about the histogram is returned

As seen above, not all arguments must be specified; only the `x` argument is necessary. When options are not specified, R uses the default options, such as the default color of white for the histogram bars.

The following command reproduces *OI Biostat* Table 1.16 and Figure 1.17.

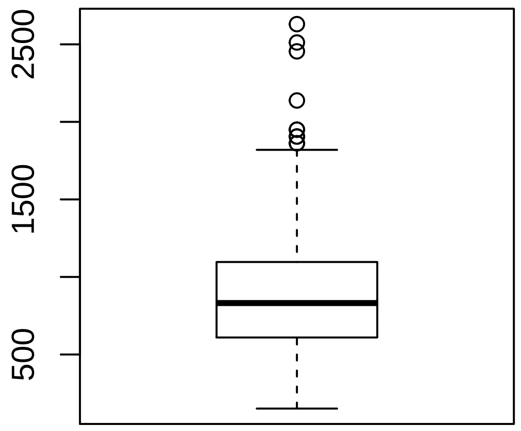
```r
## Table 1.16
hist(frog.altitude.data$clutch.volume, breaks = 14, plot = FALSE)$counts

##  [1]  4 29 69 99 93 56 45 18  7  7  1  0  2  1

## Figure 1.17
hist(x = frog.altitude.data$clutch.volume, breaks = 14, col = "dodgerblue",
     xlab = "Clutch Volume", ylab = "Frequency", ylim = c(0, 100),
     main = "Histogram of Clutch Volume Frequencies")
```



Boxplots

A **boxplot** uses five statistics to summarize a dataset, in addition to showing unusual observations. The following command produces a boxplot of the `clutch.volume` variable.

```
boxplot(frog.altitude.data$clutch.volume)
```

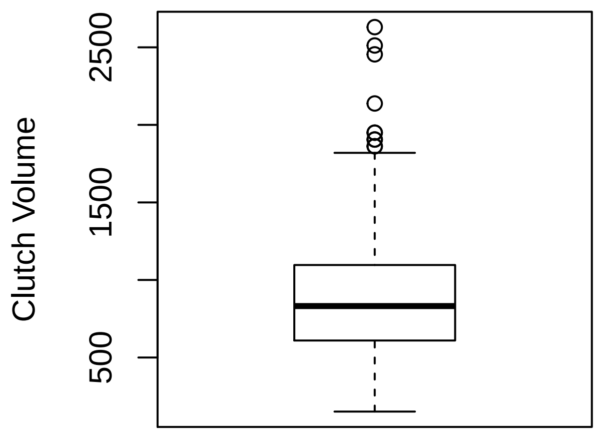


Like the `hist()` function, the `boxplot()` function also takes several arguments that allow for certain parameters to be specified:

- `x`: variable of interest
- `axes`: if `TRUE`, numbers are shown on the axes
- `col`: color of the boxplot, enclosed in ""
- `xlab`: *x*-axis label, enclosed in ""
- `ylab`: *y*-axis label, enclosed in ""
- `xlim`: range of values for the *x*-axis, in the form `c(lower bound, upper bound)`
- `ylim`: range of values for the *y*-axis, in the form `c(lower bound, upper bound)`
- `main`: main title of the plot, enclosed in ""

The following command reproduces a simplified version of *OI Biostat* Figure 1.19.

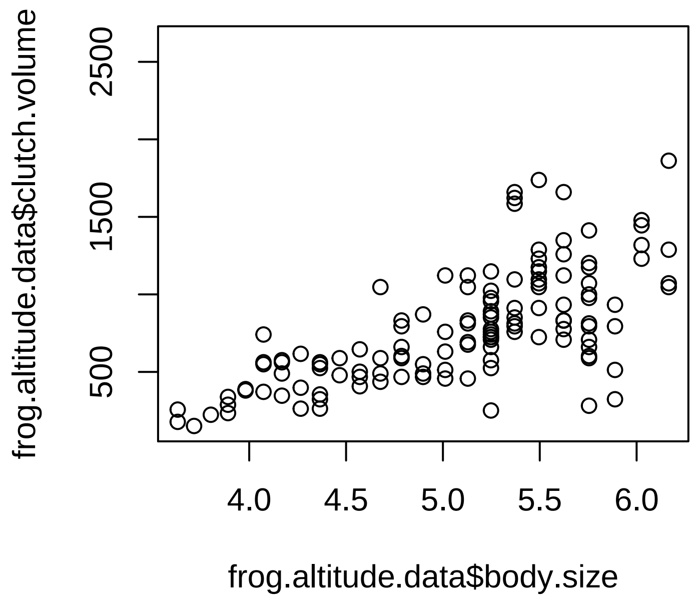
```
## Figure 1.19
boxplot(x = frog.altitude.data$clutch.volume, ylab = 'Clutch Volume', axes = TRUE,
        ylim = range(frog.altitude.data$clutch.volume))
```



1.4.5 Scatterplots and correlation

Scatterplots can be used to visualize the relationship between two numerical variables. In the `plot()` command, either a comma or a tilde can be used between the variable names; i.e., `plot(x,y)` versus `plot(y ~ x)`.

```
plot(frog.altitude.data$body.size, frog.altitude.data$clutch.volume)
```



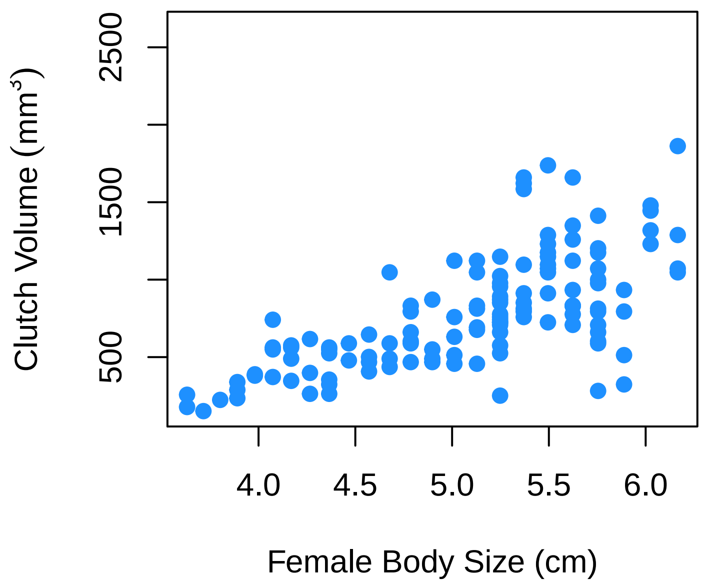
The `plot()` command takes the following arguments:

- `x`: variable defining the x-coordinates
- `y`: variable defining the y-coordinates
- `col`: color of the dots, enclosed in `""`
- `type`: by default, data points are marked with dots; other options include `"l"` which draws a line chart, or `"b"` which includes both dots and lines
- `xlab`: x-axis label, enclosed in `""`
- `ylab`: y-axis label, enclosed in `""`
- `xlim`: range of values for the x-axis, in the form `c(lower bound, upper bound)`
- `ylim`: range of values for the y-axis, in the form `c(lower bound, upper bound)`
- `main`: main title of the plot, enclosed in `""`

The *plot* command has an interesting feature that you can either specify your x and y variables by running $plot(x, y)$ or by running $plot(y\ x)$. Either command will give the same result.

The following command reproduces a simplified version of *OI Biostat* Figure 1.20. The additional argument pch changes the appearance of the dots to filled dots, which is specified by 19.

```
## Figure 1.20
plot(frog.altitude.data$clutch.volume~frog.altitude.data$body.size, col = "dodgerblue",
     pch = 19, xlab = "Female Body Size (cm)", ylab = expression("Clutch Volume" ~ (mm^3)))
```



Simplified versions of *OI Biostat* Figures 1.21, 1.22, and 1.23 can also be reproduced using `plot()`.

1.4.6 Correlation

Correlation is a numerical measure of the strength of a linear relationship. The formula for correlation uses the pairing of the two variables, just as the scatterplot does in a graph, so the data used in calculating correlation is a set of n ordered pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

The correlation between two variables x and y is given by:

$$r = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s_x} \right) \left(\frac{y_i - \overline{y}}{s_y} \right)$$

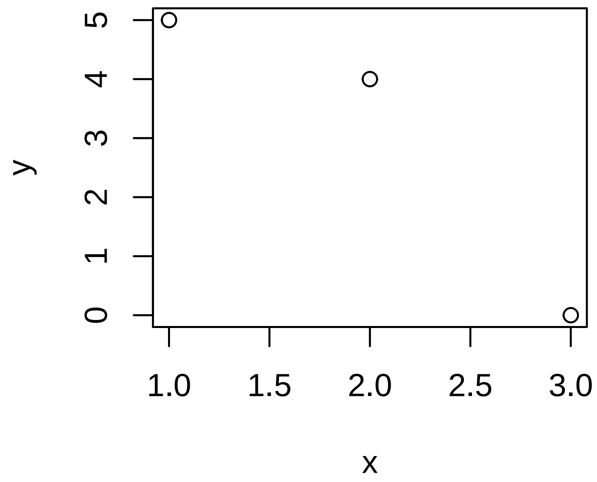
where $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ are the n paired values of x and y, and s_x and s_y are the sample standard deviations of the x and y variables, respectively.

The following illustrates the calculations for finding the correlation coefficient of (1,5), (2, 4), and (3,0), as shown in *OI Biostat* Example 1.13.

```
# define variables
x = c(1, 2, 3)
y = c(5, 4, 0)

# calculate sample means
x.bar = mean(x)
y.bar = mean(y)

# calculate sample sd's
sd.x = sd(x)
sd.y = sd(y)

# calculate products and sum of products
x.component = (x - x.bar)/(sd.x)
y.component = (y - y.bar)/(sd.y)
products = x.component * y.component
products.sum = sum(products)

# divide by n - 1
n = length(x)   ## n also equals length(y)
r = products.sum / (n - 1)
r

## [1] -0.9449112
```

It is much easier to use the `cor()` function to find correlation.

```
## Figure 1.22
cor(x,y)

## [1] -0.9449112

plot(x,y)
```



The correlation can also be calculated for the income versus life expectancy data plotted in *OI Biostat* Figure 1.23. In this case, the command includes the argument `use = "complete.obs"` to allow for the computation to disregard any missing values in the dataset.

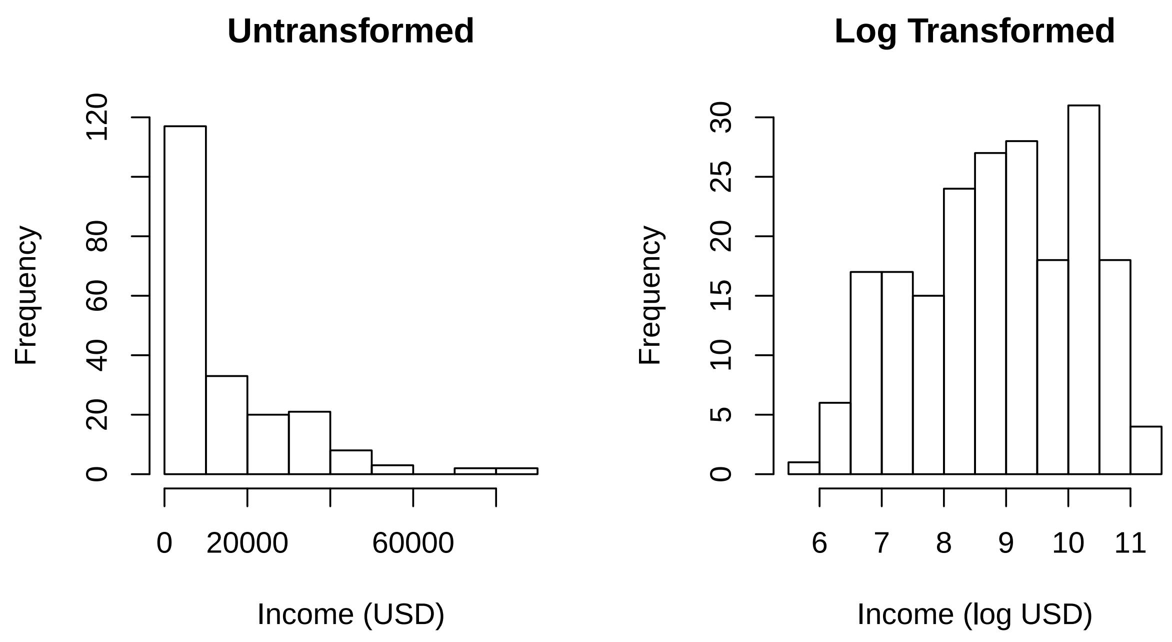
```
cor(life.expectancy.income$income, life.expectancy.income$life.expectancy,
    use = "complete.obs")

## [1] 0.6308783
```

1.4.7 Transforming Data

A **transformation** is a rescaling of data using a function. The figure below shows the original plot of income data as well as the plot of the log-transformed data. Note that the `log` command in R computes natural logarithms.

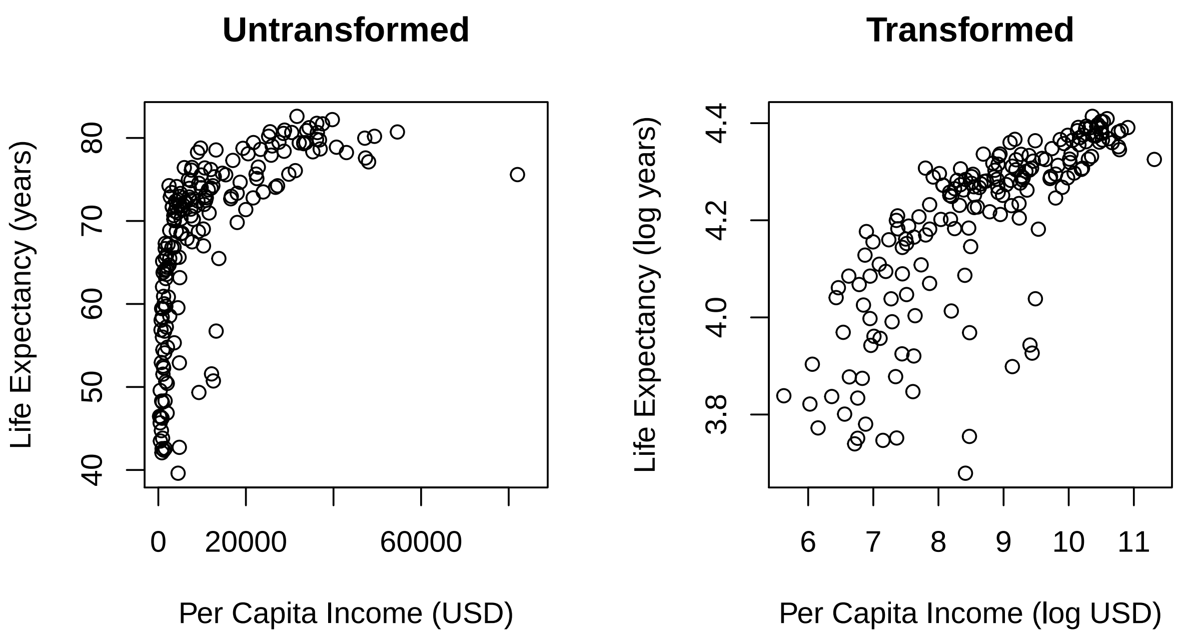
The function `par()` creates partitions in the graphing output. In this case, specifying `mfrow = c(1,2)` produces 1 row and 2 columns.

```r
## Figure 1.26
par(mfrow = c(1, 2))  ## this line allows two plots to print at the same time
hist(life.expectancy.income$income, breaks = 12, xlab = "Income (USD)",
    ylab = "Frequency", ylim = c(0, 120), main = "Untransformed")

hist(log(life.expectancy.income$income), breaks = 12, xlab = "Income (log USD)",
    ylab = "Frequency", ylim = c(0, 30), main = "Log Transformed")
```



Transformations can also be applied to both variables when exploring a relationship. The following figure shows simplified versions of *OI Biostat* Figures 1.23 and 1.27.

```r
## Figure 1.23 and Figure 1.27
par(mfrow = c(1, 2))
plot(life.expectancy.income$income, life.expectancy.income$life.expectancy,
    ylab = "Life Expectancy (years)", xlab = "Per Capita Income (USD)",
    main = "Untransformed")
plot(log(life.expectancy.income$income), log(life.expectancy.income$life.expectancy),
    ylab = "Life Expectancy (log years)", xlab = "Per Capita Income (log USD)",
    main = "Transformed")
```



1.5 Categorical Data

1.5.1 Contingency Tables

A **frequency table** shows the counts for each category within a variable. The following `table()` command produces a frequency table for the `actn3.r577x` variable.

```
## Table 1.28
addmargins(table(famuss$actn3.r577x))

##
##  CC  CT  TT Sum
## 173 261 161 595
```

A **contingency table** summarizes data for two categorical variables, such as race and genotype in *OI Biostat* Table 1.29.

```
## Table 1.29
addmargins(table(famuss$race, famuss$actn3.r577x))

##
##                CC  CT  TT Sum
##   African Am   16   6   5  27
##   Asian        21  18  16  55
##   Caucasian   125 216 126 467
##   Hispanic      4  10   9  23
##   Other         7  11   5  23
##   Sum         173 261 161 595
```

OI Biostat Table 1.30 shows a contingency table with row proportions, computed as the counts divided by their row totals. The `prop.table()` command produces a table with row proportions, as specified by the 1 in the argument.

```
## Table 1.30
counts.table = table(famuss$race, famuss$actn3.r577x)
row.prop.table = prop.table(counts.table, 1)
row.prop.table

##
##                     CC        CT        TT
##   African Am 0.5925926 0.2222222 0.1851852
##   Asian      0.3818182 0.3272727 0.2909091
##   Caucasian  0.2676660 0.4625268 0.2698073
##   Hispanic   0.1739130 0.4347826 0.3913043
##   Other      0.3043478 0.4782609 0.2173913
```

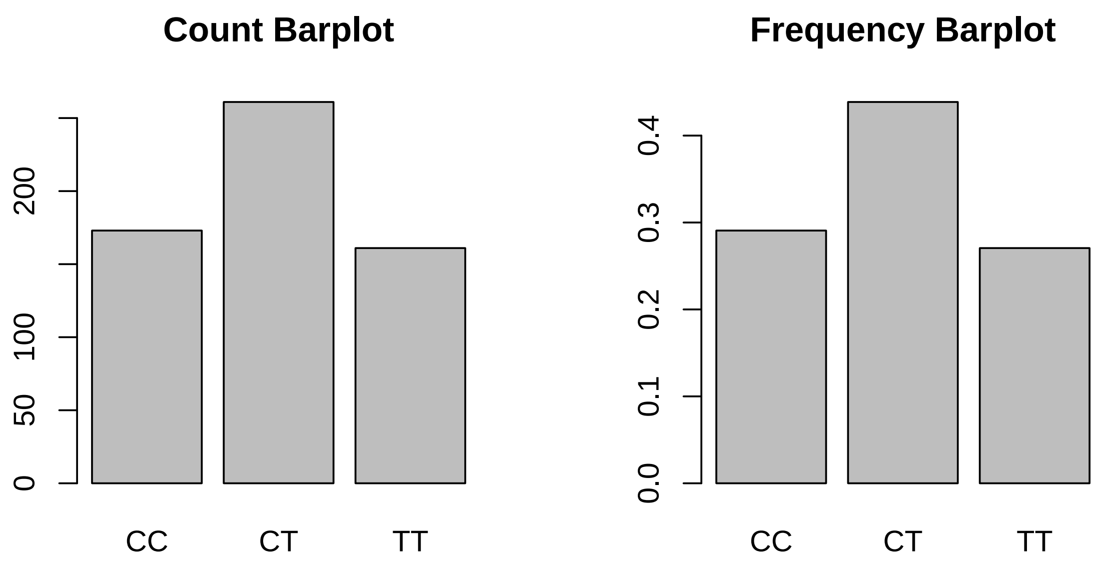
Alternatively, a contingency table can be created with column proportions by changing the 1 in `prop.table()` to a 2.

```
## Table 1.31
counts.table = table(famuss$race, famuss$actn3.r577x)
col.prop.table = prop.table(counts.table, 2)
col.prop.table

##
##                      CC         CT         TT
##   African Am 0.09248555 0.02298851 0.03105590
##   Asian      0.12138728 0.06896552 0.09937888
##   Caucasian  0.72254335 0.82758621 0.78260870
##   Hispanic   0.02312139 0.03831418 0.05590062
##   Other      0.04046243 0.04214559 0.03105590
```

1.5.2 Bar Plots

A **bar plot** is a common way to display a single categorical variable, either from count data or from proportions. The `barplot()` command requires values to be input in a table format. In the below example, the `table()` command is nested within the `barplot()` command.

```
## Figure 1.32
par(mfrow = c(1, 2))
barplot(table(famuss$actn3.r577x), main = "Count Barplot")
barplot(table(famuss$actn3.r577x)/sum(table(famuss$actn3.r577x)), main = "Frequency Barplot")  ## frequency ba
```

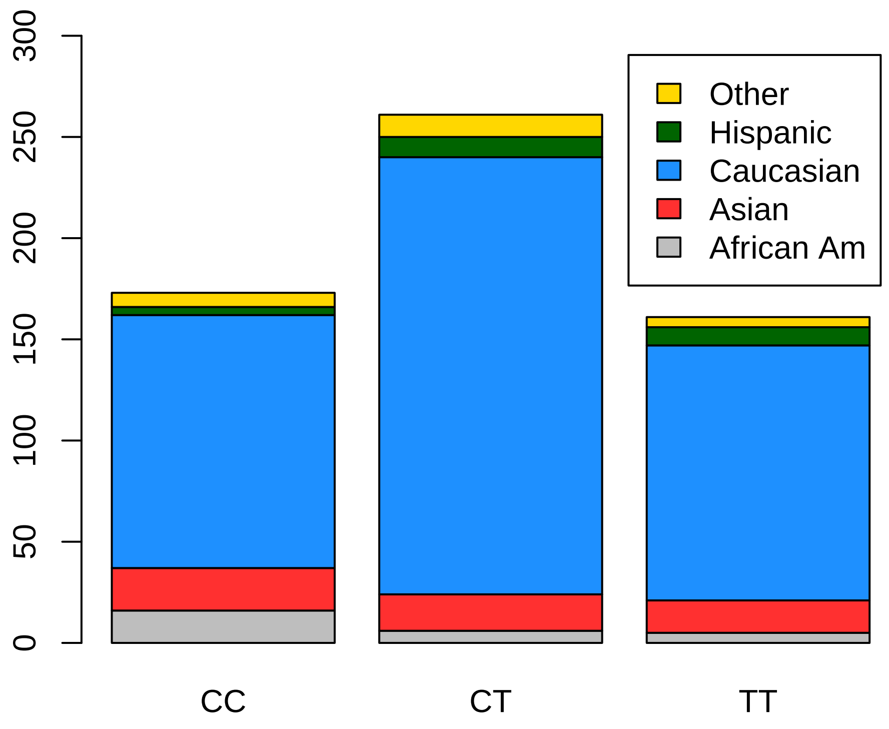

Count Barplot
0
50
100
200
CC
CT
TT
Frequency Barplot
0.0
0.1
0.2
0.3
0.4
CC
CT
TT

1.5.3 Segmented Bar Plot

A **segmented bar plot** draws from a contingency table to display information about two categorical variables. The following code reproduces *OI Biostat* Figure 1.33a, in which a bar plot was created using the `actn3.r577x` variable, with each group divided by the levels of race. The simplified version of the plot uses the default greyscale shading; it is also possible to specify a list of colors using `c()`.

The argument `legend` can be used to specify whether the row names or the column names are used for the legend.

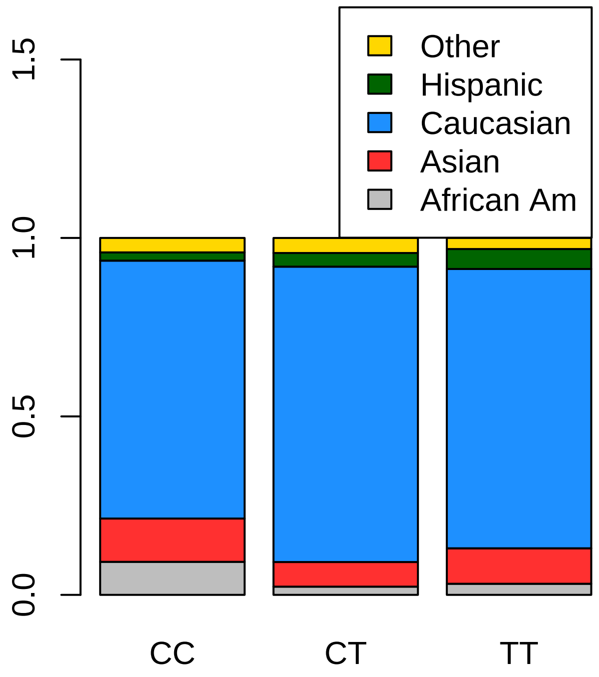
```
## Figure 1.31a First, create a table of the data that is
## sorted
genotype.race = matrix(table(famuss$actn3.r577x, famuss$race),
    ncol = 3, byrow = T)

# Second, change the column and row names on the table
colnames(genotype.race) = c("CC", "CT", "TT")
rownames(genotype.race) = c("African Am", "Asian", "Caucasian",
    "Hispanic", "Other")

# Third, plot the barplot where colors are specified
barplot(genotype.race, col = c("gray", "firebrick1", "dodgerblue",
    "darkgreen", "gold"), ylim = c(0, 300), width = 2, legend = rownames(counts.table))
```



300
250
200
150
100
50
0
Other
Hispanic
Caucasian
Asian
African Am
CC
CT
TT

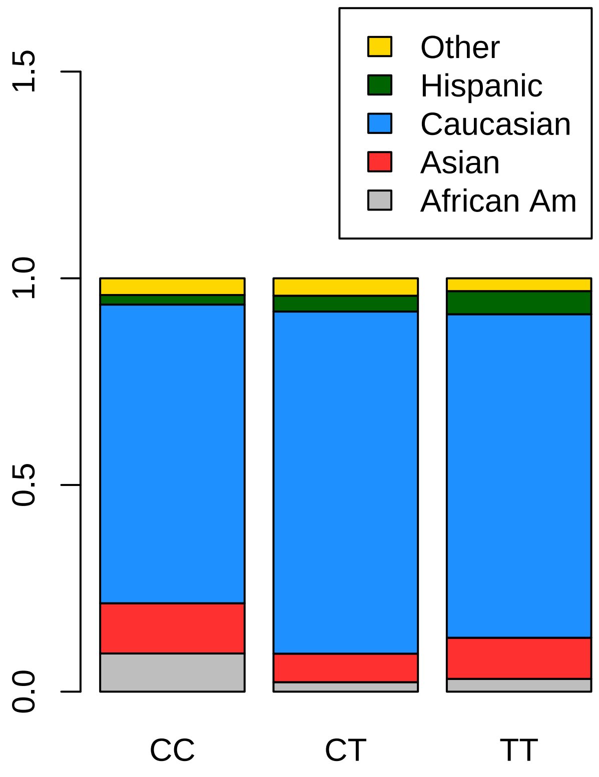
A **standardized segmented bar plot** uses proportions to scale the data, and draws from a contingency table with proportions. Setting `ylim` from 0 to 1.7 allows for enough empty space on the plot so that the legend does not overlap the bars.

```r
## Figure 1.33b
counts.table = (table(famuss$race, famuss$actn3.r577x))
row.prop.table = prop.table(counts.table, 2)

barplot(row.prop.table, col = c("gray", "firebrick1", "dodgerblue",
    "darkgreen", "gold"), ylim = c(0, 1.7), legend = rownames(counts.table))
```



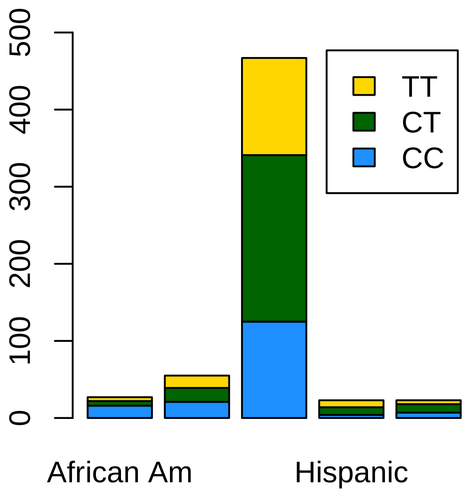
```r
## Figure 1.31b First, create table of proportions
prop.genotype.race <- prop.table(genotype.race, 2)

# Second, change the column and row names on the table
colnames(prop.genotype.race) = c("CC", "CT", "TT")
rownames(prop.genotype.race) = c("African Am", "Asian", "Caucasian",
    "Hispanic", "Other")

# Third, plot the output
```

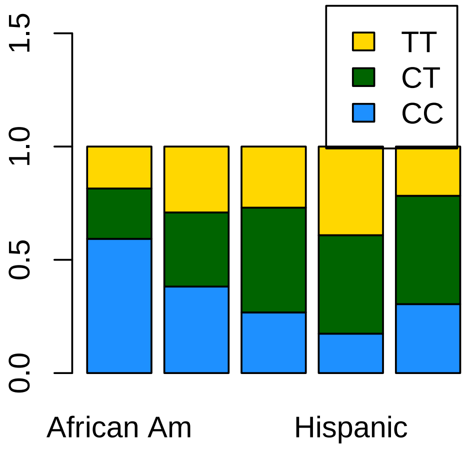
```r
barplot(prop.genotype.race, col = c("gray", "firebrick1", "dodgerblue",
    "darkgreen", "gold"), ylim = c(0, 1.7), width = 2, legend = rownames(counts.table))
```



In *OI Biostat* Figure 1.34, the data from the contingency table are organized differently, with each bar representing a level of race. To make this change, reverse the order of the variables in `table()`

```r
## Figure 1.34 Plotting two graphs next to each other
par(mfrow = (c(1, 2)))

# Setting up the table and changing column/row names
race.genotype = matrix(table(famuss$race, famuss$actn3.r577x),
    ncol = 5, byrow = T)
colnames(race.genotype) = c("African Am", "Asian", "Caucasian",
    "Hispanic", "Other")
rownames(race.genotype) = c("CC", "CT", "TT")

# Creating segmented bar plot (Figure 1.34a)
barplot(race.genotype, col = c("dodgerblue", "darkgreen", "gold"),
    ylim = c(0, 500), width = 2, legend = rownames(race.genotype))

# Creating standardized segmented bar plot (Figure 1.34b)
prop.race.genotype <- prop.table(race.genotype, 2)
barplot(prop.race.genotype, col = c("dodgerblue", "darkgreen",
    "gold"), ylim = c(0, 1.7), width = 2, legend = rownames(race.genotype))
```



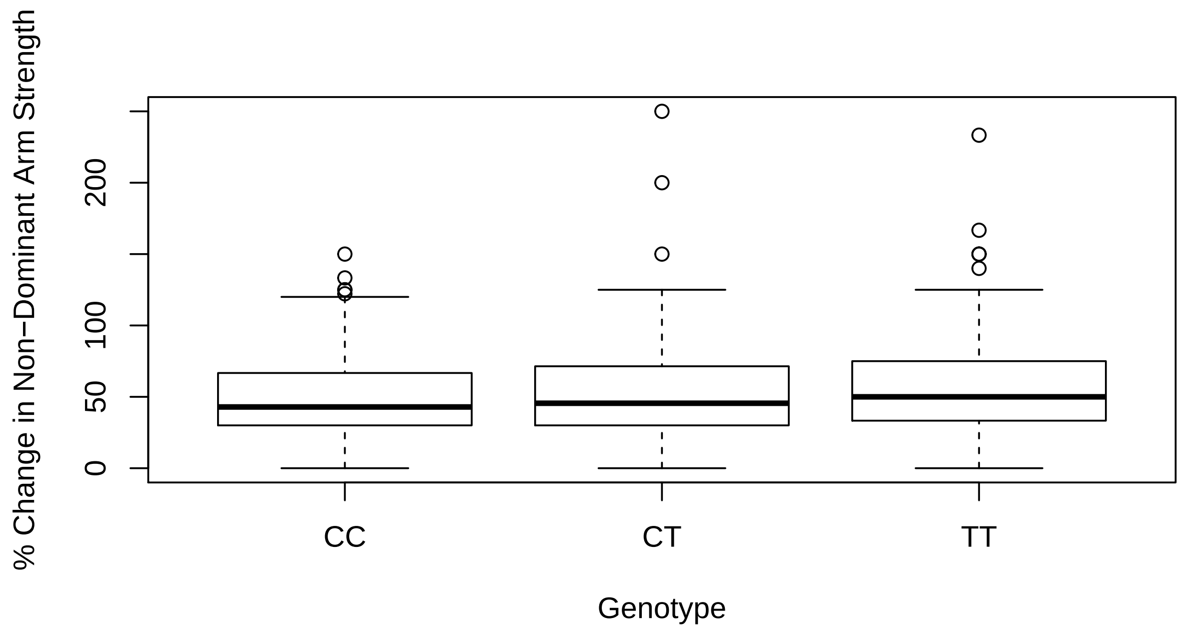


1.5.4 Comparing numerical data across groups

Side-by-side boxplots

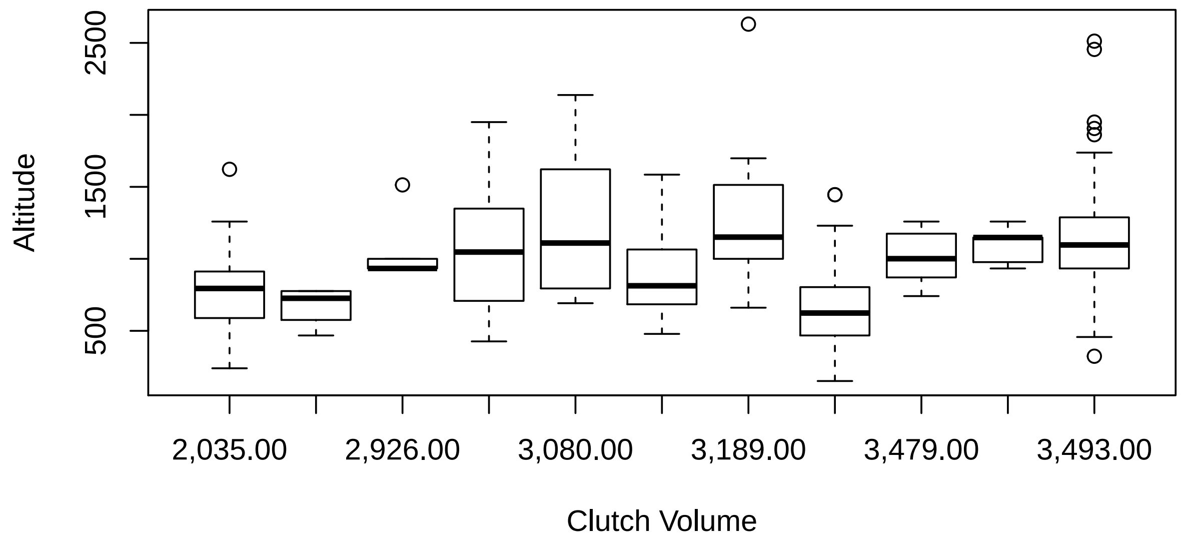
The **side-by-side** boxplot is a useful tool for comparing the distribution of numerical data across categories. *OI Biostat* Figure 1.35a shows a side-by-side boxplot for percent change in non-dominant arm strength grouped by genotype. The response variable y comes before the tilde and the explanatory variable x; in this case, the response variable is percent change in strength.

```
## Figure 1.35(a)
boxplot(famuss$ndrm.ch ~ famuss$actn3.r577x,
    ylab = "% Change in Non-Dominant Arm Strength",
    xlab = "Genotype")
```



OI Biostat Figure 1.36 shows a larger side-by-side boxplot which compares the distribution of frog clutch sizes for different altitudes.

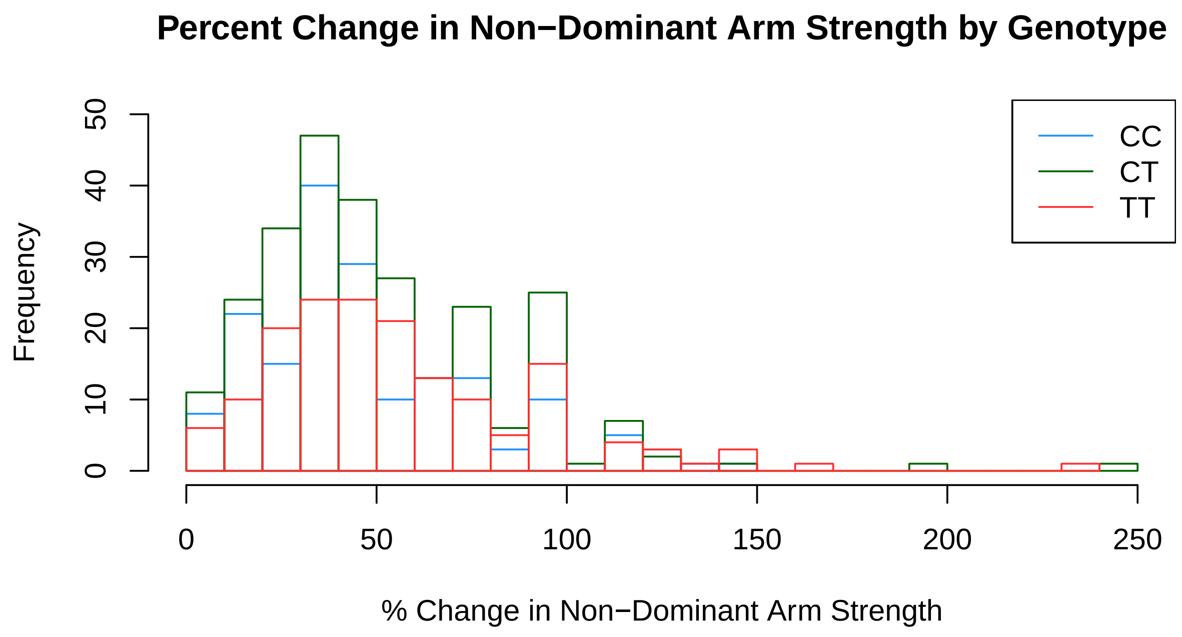
```r
## Figure 1.36
boxplot(frog.altitude.data$clutch.volume ~ frog.altitude.data$altitude,
    xlab = "Clutch Volume", ylab = "Altitude")
```



Hollow Histograms

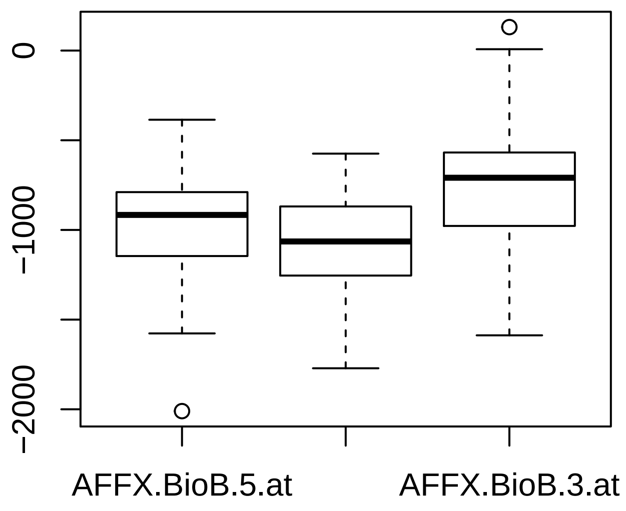
A **hollow histogram** plots the outlines of histograms for each group onto the same axes. To plot a hollow histogram, specify any axis limits and labels in the command for the first histogram; the subsequent histograms require the argument `add = T` in order for them to be overlaid on top of the first plot.

```
## Figure 1.35(b)
hist(famuss$ndrm.ch[famuss$actn3.r577x == "CC"], breaks = 20, border = "dodgerblue",
     xlab = "% Change in Non-Dominant Arm Strength", ylim = c(0,50), xlim = c(0,250),
     main = "Percent Change in Non-Dominant Arm Strength by Genotype")
hist(famuss$ndrm.ch[famuss$actn3.r577x == "CT"], breaks = 20, border = "darkgreen",
     add = T)
hist(famuss$ndrm.ch[famuss$actn3.r577x == "TT"], breaks = 20, border = "firebrick1",
     add = T)

legend('topright',
       col = c("dodgerblue", "darkgreen", "firebrick1"),
       lwd = c(1, 1, 1),      ## line width
       legend = c('CC', 'CT', 'TT'))    ##legend labels
```



1.6 Genomic Data

```
boxplot(Golub[, 7:9])
```


0
−1000
−2000
AFFX.BioB.5.at
AFFX.BioB.3.at

Chapter 2

Using R for Probabilities

Contents

This chapter focuses on the material covered in *OI Biostat* Chapter 2, which focuses on the idea of probability and how to solve for various types of probabilities. This chapter will work through several examples, illustrating how R can be used to solve propability problems. Once again, it will be shown that there are several different computational methods that can be used to answer the same question.

2.1 Simple Probability

2.1.1 Using R as a Calculator

For the most basic probability problems, R can be used similarly to a calculator. Variable values can be stored and then used for calculations. An example of this is *OI Biostat* Example 2.24, which is walked through below.

> ***OI Biostat* Example 2.24.** Mandatory drug testing in the workplace is common practice for certain professions, such as air traffic controllers and transportation workers. A false positive in a drug screening test occurs when the test incorrectly indicates that a screened person is an illegal drug user. Suppose a mandatory drug test has a false positive rate of 1.2% (i.e., has probability 0.012 of indicating that an employee is using illegal drugs when that is not the case). Given 150 employees who are in reality drug free, what is the probability that at least one will (falsely) test positive? Assume that the outcome of one drug test has no effect on the others.

```r
## Example 2.24
false.positive = 0.012
true.negative = 1-false.positive
```

```
one.positive = 1-true.negative^150
one.positive

## [1] 0.836491
```

2.1.2 Through Simulation

A powerful method for calculating probabilities in R is called **simulation** and consists of simulating a large population with some consistent known parameters so that population statistics can be measured.

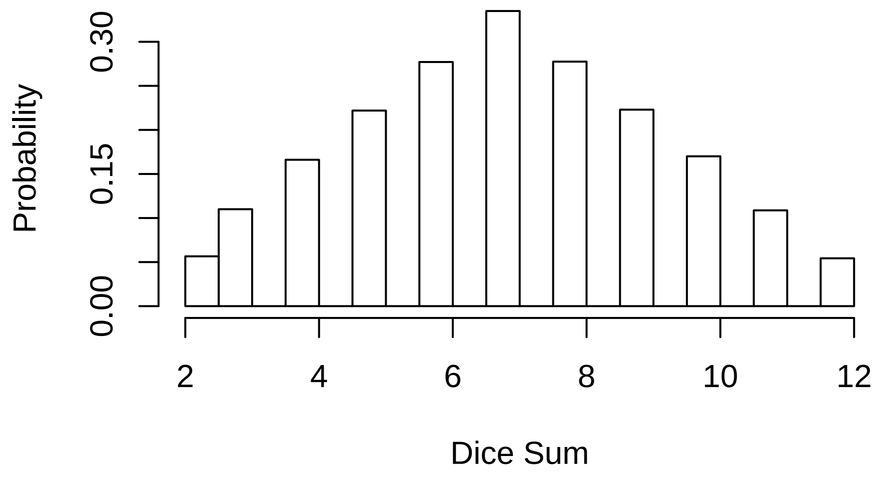
The steps for a simulation process are as follows,

1. Define the parameters of the simulation: population size.
2. In order for the results of the simulation to be reproducible, it is necessary to use `set.seed()` to associate the particular set of results with a "seed". Any integer can be used as the seed. Different seeds will produce a different set of results.
3. Using the `vector()` command, create one empty list that will be used to store the results of the simulation.
4. Simulate rolling each of the dice using the `sample()` command, which takes a sample of a specified size from a list `x`. In this context, the goal is to sample the integers between `1` and `6` 100,000 times, where the result represents the face of the die that shows up. This is done in combination with a `for` loop. The loop allows for this process to be repeated for each of the 100,000 individuals being simulated. The first line sets up the loop, with a variable `ii` that starts at 1 and finishes at `population.size`, which was specified earlier as 100,000. This allows for each dice roll to be performed one at a time.
5. Look at your final results of the simulated population distribution, and if desired, calculate population statistics.

Dice Rolling Example

For example, simulation can be used to obtain the probability distribution of the sum of two dice seen in *OI Biostat* Figure 2.7. Note that these probabilities can be calculatd directly as well, as shown in *OI Biostat*.

```
## 1. Set parameters
population.size = 1e+05

## 2. Set Seed
set.seed(2016)

## 3. Create empty list
dice.sums = vector("numeric", population.size)

## 4. Simulate 'rolling the dice'
for (ii in 1:population.size) {
    dice.1 = sample(1:6, size = 1)
    dice.2 = sample(1:6, size = 1)
    dice.sums[ii] = dice.1 + dice.2
}

## 5. Plot results
hist(dice.sums, freq = FALSE, ylab = "Probability", xlab = "Dice Sum", main = "")
```



Drug Testing Example

Another example of a probability that can be calculated using simulation is *OI Biostat* Example 2.24, which was calculated directly above. In the `sample` command here, a result of 0 indicates a negative test result while a result of 1 indicates a positive test, and the inclusion of `replace = TRUE` ensures that each individual's test is independent of the others.

```
## 1. Set parameters
simulation.size = 100000
n = 150
false.positive = 0.012

## 2. Set Seed
set.seed(2016)

## 3. Create empty list
test.results = vector("numeric", population.size)

## 4. Simulate "rolling the dice"
for(ii in 1:population.size){
  ## Simulate the 150 employee's test results
  employees = sample(c(0,1), size = n, prob = c(1-false.positive, false.positive), replace = TRUE)
  ## Count how many employees tested positive
  test.results[ii] = sum(employees)
}

## 5. Obtain results
mean(test.results>=1)

## [1] 0.83584
```

2.2 Conditional Probability

What are the chances that a woman with a positive mamogram has breast cancer? This question can be rephrased as the conditional probability that a woman has breast cancer, given that her mammogram is abnormal, otherwise known as the **positive predictive value** of a mammogram. Two methods discussed in this chapter, using Bayes' Theorem and creating a contingency table, are also explained in *OI Biostat*. This chapter will also cover an additional approach: modeling the problem scenario by running a simulation.

> ***OI Biostat* Example 2.37.** In Canada, about 0.35% of women over 40 will develop breast cancer in any given year. A common screening test for cancer is the mammogram, but it is not perfect. In about 11% of patients with breast cancer, the test gives a **false negative**: it indicates a woman does not have breast cancer when she does have breast cancer. Similarly, the test gives a **false positive** in 7% of patients who do not have breast cancer: it indicates these patients have breast cancer when they actually do not. If a randomly selected woman over 40 is tested for breast cancer using a mammogram and the test is positive – that is, the test suggests the woman has cancer – what is the probability she has breast cancer?

2.2.1 Bayes' Theorem

Bayes' Theorem states that the conditional probability $P(A_1|B)$ can be identified as the following fraction:

$$\frac{P(A_1 \text{and } B)}{P(B)} = \frac{P(B|A_1)P(A_1)}{P(B|A_1)P(A_1) + P(B|A_2)P(A_2) + \cdots + P(B|A_k)P(A_k)}$$

where A_2, A_3, ..., and A_k represent all other possible outcomes of the first variable.

The expression can also be written in terms of diagnostic testing language, where D = {has disease}, D^c = {does not have disease}, T^+ = {positive test result}, and T^- = {negative test result}.

$$\begin{aligned} P(D|T^+) &= \frac{P(D \text{ and } T^+)}{P(T^+)} \\ &= \frac{P(T^+|D) \times P(D)}{[P(T^+|D) \times P(D)] + [P(T^+|D^c) \times P(D^c)]} \\ \text{PPV} &= \frac{\text{sensitivity } \times \text{ prevalence}}{[\text{sensitivity } \times \text{ prevalence}] + [(1 \text{ - specificity}) \times (1 \text{ - prevalence})]} \end{aligned}$$

R can be used to store values for prevalence, sensitivity, and specificity so that calculations are less tedious. Recall that the **sensitivity** is the probability of a positive test result when disease is present, which is the complement of a false negative. The **specificity** is the probability of a negative test result when disease is absent, which is the complement of a false positive. With this in mind, solving for the **positive predictive value (PPV)** is quite easy in R for the breast cancer example as follows.

```
prevalence = 0.0035
sensitivity = 1 - 0.11
specificity = 1 - 0.07

ppv.num = (sensitivity*prevalence)  ## numerator
ppv.den = ppv.num + ((1-specificity)*(1-prevalence))  ## denominator
ppv = ppv.num / ppv.den

ppv

## [1] 0.04274736
```

2.2.2 Contingency Table

The PPV can also be calculated by constructing a two-way contingency table for a hypothetical population and calculating conditional probabilities by conditioning on rows or columns. While this method results in an estimate of PPV, using a large enough population size such as 100,000 produces an empirical estimate that is very close to the exact value found through using Bayes' Theorem.

	D+	D-	Total
T+			
T-			
Total			100,000

First, calculate the expected number of disease cases and non-disease cases in the population:

```
population.size = 100000
expected.cases = prevalence * population.size
expected.cases

## [1] 350

expected.noncases = (1 - prevalence) * population.size
expected.noncases

## [1] 99650
```

	D+	D-	Total
T+			
T-			
Total	350	99,650	100,000

Next, calculate the expected number of cases of true positives and the expected number of cases of false positives:

```
expected.true.positives = expected.cases * sensitivity
expected.true.positives

## [1] 311.5

expected.false.positives = expected.noncases * (1 - specificity)
expected.false.positives

## [1] 6975.5

total.expected.positives = expected.true.positives + expected.false.positives
total.expected.positives

## [1] 7287
```

	D+	D-	Total
T+	311.5	6,975.5	7,287
T-			
Total	350	99,650	100,000

Finally, calculate the positive predictive value:

```
ppv = expected.true.positives/total.expected.positives
ppv

## [1] 0.04274736
```

2.2.3 Simulation

The final method for solving this conditional probability problem in R is through **simulation** and involves the process of simulating 100,000 individuals who each have the same probability of having the disease. Afterwards, using the known specificity and sensitivity of the diagnostic test, individuals can be assigned a test result of either positve or negative as if a test had been performed. This results in a simulated dataset of 100,000 individuals that each have a disease status and test result. This is a more complex version of the simulation performed above, so many steps are parallel to the process outlined above.

1. Define the parameters of the simulation: disease prevalence, test sensitivity, test specificity, and hypothetical population size.

2. Set the seed using `set.seed()`.

3. Using the `vector()` command, create two empty lists that will be used to store the results of the simulation: one will store disease status and the other will store test outcome. The results will be in the form of numbers; specifically, either `0` or `1`.

4. Assign disease status by using the `sample()` command, which takes a sample of a specified size from a list `x`. In this context, the goal is to sample between `0` and `1` 100,000 times, where `0` represents an individual without breast cancer and `1` an individual with breast cancer. The argument `prob` defines the probability that either number is sampled; a `1` should be sampled with the same probability as the prevalence, since the prevalence indicates how many members of the population have disease. Similarly, a `0` should be sampled with probability `(1 - prevalence)`. The argument `replace = TRUE` allows for sampling with replacement; in other words, allowing the numbers `0` and `1` to be assigned multiple times.

5. Assign test result by using the `sample()` command in combination with a `for` loop and conditional statements. In this step, a test result is assigned with different probability depending on whether the disease status is a `0` or a `1`; this relates to the sensitivity and specificity of the diagnostic test. The loop allows for this process to be repeated for each of the 100,000 individuals being simulated.

   - The first line sets up the for loop that cycles `ii` from 1 to the population size of 100,000.
   - Two `if` statements are in the loop which direct R to take a different action depending on the value of `disease.status`. The double equals signs == imply a conditional statement, allowing `if(disease.status[ii] == 1)` to make the statement "For this loop, if disease status is equal to 1, then do the following...".
   - Depending on disease status, R will execute one of the two `sample()` functions in the loop. One sample function has probabilities weighted based on sensitivity and the other has probabilities defined by specificity.

6. Make calculations using the two lists, `disease.status` and `disease.outcome`, which are now filled with values. Since the value 1 was assigned to a positive test result and to an individual with disease, the `sum()` command can be used to determine the total number of individuals with disease and the number of positive test results.

```r
## 1. Set parameters
prevalence = 0.0035
sensitivity = 1 - 0.11
specificity = 1 - 0.07
population.size = 100000

## 2. Set seed
set.seed(2016)

## 3. Create empty lists
disease.status = vector("numeric", population.size)
test.outcome = vector("numeric", population.size)

## 4. Assign disease status
disease.status = sample(x = c(0,1), size = population.size,
                        prob = c(1 - prevalence, prevalence),  ## matches order of x
                        replace = TRUE)

## 5. Assign test result
for (ii in 1:population.size) {
  if (disease.status[ii] == 1) {  ## note the ==
    test.outcome[ii] = sample(c(0, 1), size = 1, ## test results assigned 1 at a time
                              prob = c(1 - sensitivity, sensitivity))}
  if (disease.status[ii] == 0) {  ## note the ==
    test.outcome[ii] = sample(c(0, 1), size = 1, ## test results assigned 1 at a time
                              prob = c(specificity, 1 - specificity))}

}

## 6. Calculate ppv
num.disease = sum(disease.status)  ## total number of individuals with disease
num.pos.test = sum(test.outcome)   ## total number of positive tests

# Identify the number of individuals with disease who tested positive
num.true.pos = sum(test.outcome[disease.status == 1])

# Calculate ppv
ppv = num.true.pos / num.pos.test
ppv

## [1] 0.04273973
```

Chapter 3

Distributions

Contents

Chapter 3 presents the idea of several known and commonly used distributions. R is a powerful source for accessing and using these distributions.

3.1 Normal Distribution

We have learned that the **normal distribution** can be written as

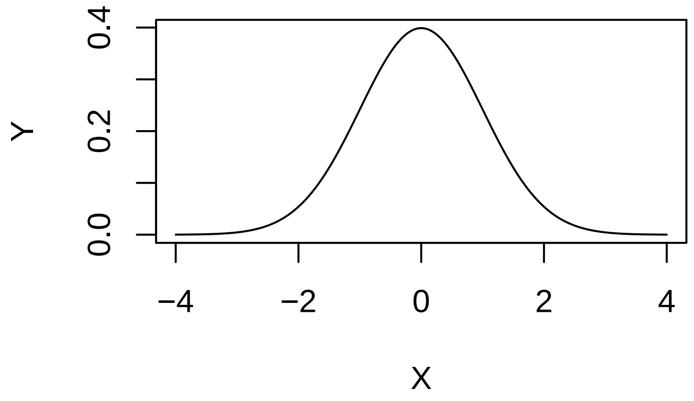
$$N(\mu, \sigma)$$

where μ is the mean of the distribution and σ is the standard deviation. Using this idea, a standard normal distribution can be created in one of two ways:

3.1.1 Direct Plot

There is a function in R called the `dnorm()` function, which, for a given posible value of the distribution, returns the probability of the distribution holding that value.

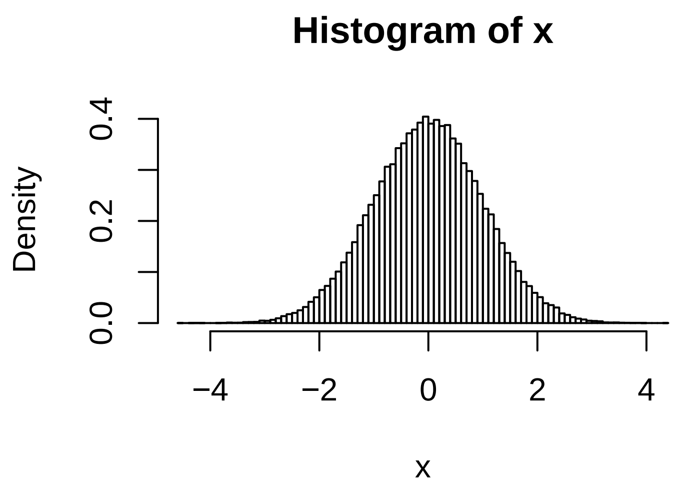
```r
## getting a list of values to evaluate distribution
X = seq(-4, 4, 0.01)
## getting normal distribution value of the given X values
Y = dnorm(X, mean = 0, sd = 1)
## plotting results
plot(X,Y, type = "l")
```



3.1.2 Through Simulation

The second method involves sampling randomly from the known normal distribution to simulate the probabilities that we drew directly in the above method. In this case, the function `rnorm()` takes n number of random samples from the normal distribution with the given mean and standard deviation. In the `hist()` command, the specification that `freq = FALSE` should be included to make the plot reflect percentages rather than frequency counts.

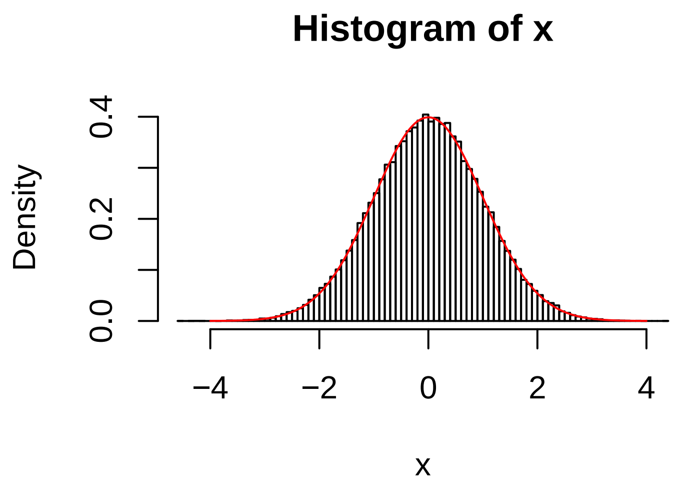
```r
set.seed(1)
x <- rnorm(n = 100000, mean = 0, sd = 1)
hist(x, breaks = 100, freq = FALSE)
```



3.1.3 A Comparison

Plotting both methods on the same graphs is useful for comparison as follows.

```r
hist(x, freq = FALSE, breaks = 100)
lines(X,Y, col = "red")
```



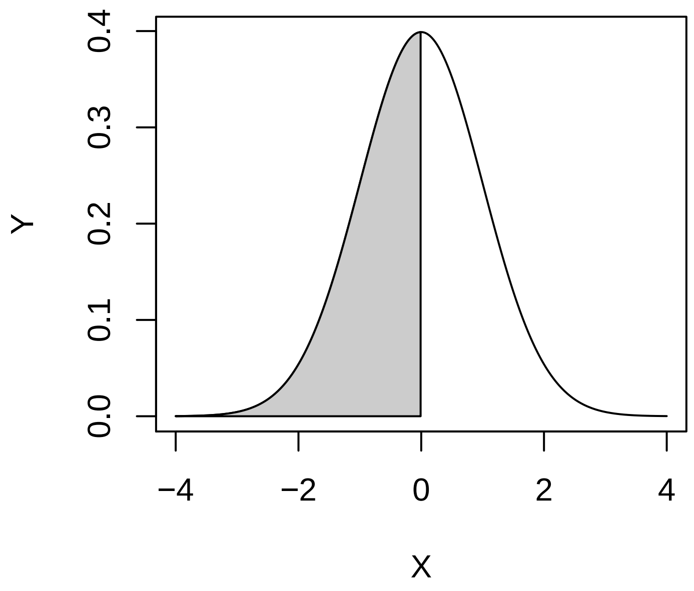
Note that the two show very similar results. For practical purposes, `dnorm()` is typically used as it only requires one command and is the asymptotic result of simulations, rather than an approximation.

3.1.4 Probability Functions

For the normal distribution, there are two more functions that are very useful, `pnorm()` and `qnorm()`.

pnorm

The function `pnorm()` tells the percentage of the normal distribution that is less than or equal to the value that is put in. This corresponds to the gray area in the below plot and can be written as $pnorm(x, mean = \mu, sd = \sigma)$ which is equivalent to $P(X < x)$ where $X \sim N(\mu, \sigma)$ and x is the point of interest.



For some example scenarios, where the goal is to find a proportion of the normal distribution given x or z, the following configurations of `pnorm()` can be used:

- $pnorm(z) = P(Z \leq z)$
- $pnorm(z, lower.tail = FALSE) = P(Z > z)$
- $pnorm(x, \mu, \sigma) = P(X \leq x)$ where $X = \sigma Z + \mu$
- $pnorm(x, \mu, \sigma, lower.tail = FALSE) = P(X > x)$ where $X = \sigma Z + \mu$

Several examples that are shown in the text can be evaluated as follows. Keep note that for every question, there are multiple methods to arrive at the same answer. The two main methods involve approximating to a standard normal distribution and allowing R to do all the work for you.

```
## Example 3.5
pnorm(q = 1)

## [1] 0.8413447
## Example 3.7
# using the approximation to the standard normal
pnorm(q = 0.43, lower.tail = FALSE)

## [1] 0.3335978

1 - pnorm(q = 0.43)

## [1] 0.3335978

# using the actual normal distribution
pnorm(q = 1630, mean = 1500, sd = 300, lower.tail = FALSE)

## [1] 0.3323863

1 - pnorm(q = 1630, mean = 1500, sd = 300)

## [1] 0.3323863
```

```
## Example 3.10 using standard normal approximation
pnorm(1.21, lower.tail = TRUE) - pnorm(-0.3, lower.tail = TRUE)

## [1] 0.504772

# using actual distribution
pnorm(q = 74, mean = 70, sd = 3.3, lower.tail = TRUE) - pnorm(q = 69, mean = 70,
    sd = 3.3, lower.tail = TRUE)

## [1] 0.5063336
```

qnorm

On the other hand, the `qnorm()` function is the inverse function, instead giving the X value such that the given percentage of the distribution is less than or equal to that value. This function takes a known probability and returns a value on the normal distribution that is corresponding.

Some sample configurations of using `qnorm()` can be as follows,

- $qnorm(p) = P(Z \leq z)$
- $qnorm(p, lower.tail = FALSE) = P(Z > z)$
- $qnorm(p, \mu, \sigma) = P(X \leq x)$ where $X = \sigma Z + \mu$
- $qnorm(p, \mu, \sigma, lower.tail = FALSE) = P(X > x)$ where $X = \sigma Z + \mu$

Working through a Example 3.12 from *OI Biostat*,

```
## Example 3.12
# using standard normal
70 + 3.3*qnorm(0.4)

## [1] 69.16395
```

```
# one step function
qnorm(p = .4, mean = 70, sd = 3.3)

## [1] 69.16395
```

3.2 Evaluating the Normal Approximation

It is of interest to test the normality of a distriubtion in order to determine if approximation with a normal distribution is appropriate. There are two methods in R that can be used to test this. The first involves plotting the data with a histogram and then superimposing a line on top with the corresponding mean and standard deviation of the data. The more closely that the histogram matches the line, the more appropriate a normal approximation would be.

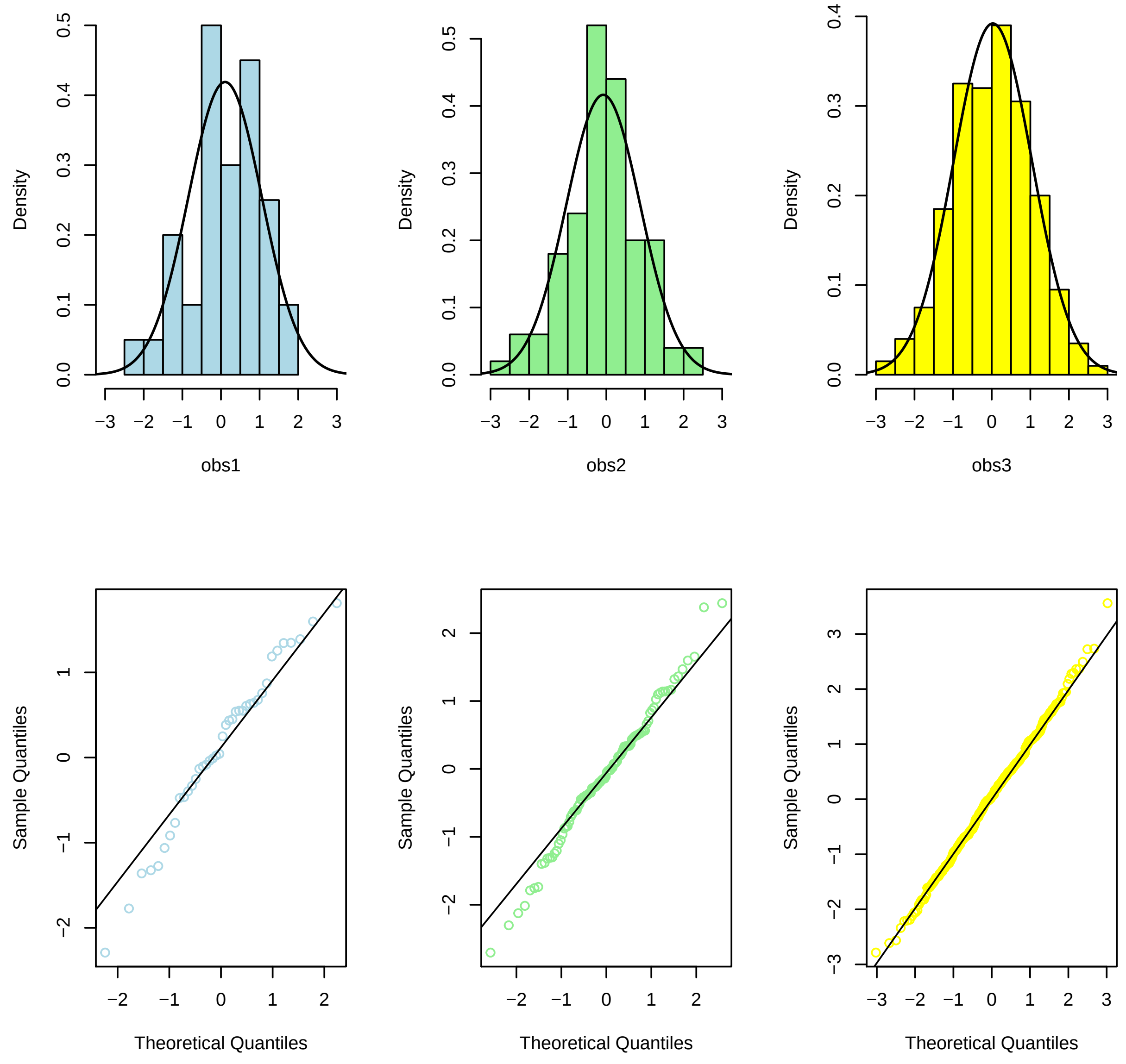
The second method utilizes the function `qqnorm()`, which plots the residual differences between the observed data and the theoretical normal distribution that would match. A line can be plotted on top of this plot, using `qqline()`, which demonstrates how a perfect match would appear. Deviation of the dots on the plot from the line shows lack of fit for the normal approimation.

Both of these methods can be seen below in this recreation of *OI Biostat* Figure 3.10 from the main text. The rows show the first and second methods respectively, while the columns show a progressively better fit to the normal from left to right.

```
obs1 <- c(-0.01, 1.39, 1.598, 0.383, -0.084, -0.475, -0.105, -1.062, 0.607,
    0.539, 0.871, 0.45, -0.039, 1.256, 0.55, -0.333, -0.252, 1.187, -0.916,
    0.677, -1.324, -1.773, 1.813, 0.023, -2.291, -1.361, 0.642, 0.249, -0.132,
    1.345, 0.629, -1.274, 0.435, 0.043, 0.55, -0.465, 0.756, -0.396, -0.767,
    1.348)

obs2 <- c(-0.78, -0.552, -0.027, 0.33, -0.964, 0.865, -0.112, 0.075, -0.392,
    0.365, -1.738, 0.491, -0.245, 0.436, 0.016, -0.202, 1.322, 0.515, 0.333,
    -0.28, -0.843, 0.181, -0.284, -0.409, 0.542, 0.117, -0.194, -0.415, 1.362,
    0.826, 1.099, -1.243, -0.265, -0.387, 0.901, 0.706, -0.353, -1.05, 0.081,
    -1.102, -2.705, -0.142, -0.608, 0.661, -0.616, 1.025, -1.384, 0.337, 1.14,
    0.523, -0.662, 0.19, 1.468, 2.38, -0.351, -2.125, 1.141, 1.149, -0.448,
    1.166, -0.269, -0.145, -1.319, -0.445, 0.34, -1.789, -0.626, -1.302, 2.441,
    -2.016, 0.333, -0.019, 0.457, -0.706, 0.236, 0.496, -0.02, -0.228, -1.756,
    -1.309, 0.564, 1.597, -0.172, -0.369, 0.478, -0.854, -0.52, -0.045, 1.654,
    1.122, -0.155, 0.281, 0.205, 0.096, -2.303, -1.399, -0.877, -1.205, 0.02,
    0.563)

obs3 <- c(0.673, -0.538, -0.703, -0.004, -1.395, -0.354, 0.415, 1.097, 0.74,
    0.657, -0.759, 0.415, 2.094, -0.662, -0.161, 0.293, 0.057, -1.487, -1.822,
    1.192, 2.186, -0.26, 0.453, -0.267, -0.049, -1.075, -0.959, -2.338, 0.512,
    2.365, 0.56, -0.812, 0.363, -0.731, -0.644, -0.448, -0.024, -0.024, -1.133,
    -0.692, 1.233, 0.566, 1.109, -2.215, 0.494, 0.011, -2.785, -0.59, -0.895,
    -1.084, -0.848, -0.129, 1.132, -0.851, -0.419, -0.232, -0.789, -2.018, 1.302,
    -1.276, -0.592, 1.578, 0.474, -0.437, 0.3, 0.145, 0.263, -2.189, -0.265,
    -0.02, 0.85, 1.523, 0.938, 0.651, -1.866, -2.202, 0.083, -0.816, 1.082,
    1.448, -1.563, -0.145, -1.168, 1.633, -0.472, 0.173, -1.592, 0.623, -0.674,
    -1.336, -0.059, 0.209, 0.152, -0.345, -0.686, 2.494, -0.616, 0.615, -0.718,
    1.748, 0.011, -0.936, -0.196, 0.839, -0.099, 0.216, -0.036, -0.687, 1.126,
    -0.024, -0.239, 1.475, 1.548, -1.254, -1.513, 0.635, 0.482, 1.068, -0.814,
```

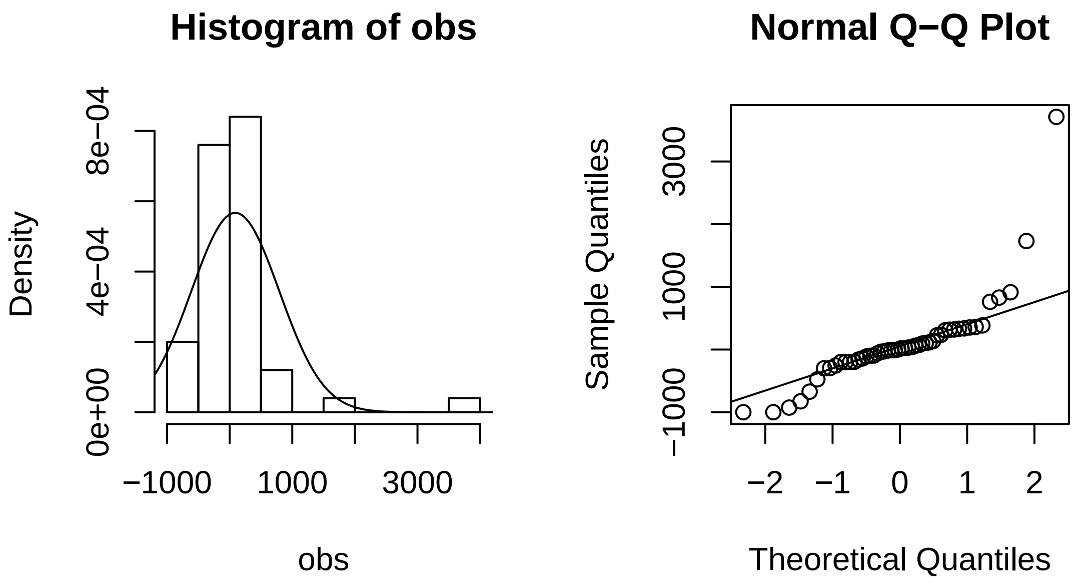
```r
    1.171, -0.509, -0.733, -0.32, 0.05, -0.359, -0.22, 0.269, 1.581, -0.967,
    1.725, 0.322, -0.176, -0.655, 2.362, -0.004, -1.209, 0.622, -1.125, 1.767,
    -0.053, 0.066, 0.049, 0.45, 0.302, -0.172, -1.237, 0.072, -1.007, 0.312,
    -0.647, -0.759, 0.753, -1.179, 0.984, 1.947, -0.653, -0.34, -0.669, -0.066,
    -1.774, -1.409, -0.875, -0.225, -0.775, -0.657, 0.812, 2.278, 0.25, 1.206,
    -0.646, -0.693, -0.545, -1.44, -1.548, -0.924, 0.408, 1.135, 1.173, 1.472,
    -0.578, -0.04, 0.422, -0.214, 0.983, -1.605, -0.6, -0.641, -0.501, 0.795,
    0.542, -1.471, 0.185, 0.504, 3.559, 0.487, 0.406, 0.318, 1.485, 0.217, 0.409,
    -0.005, -0.351, -0.932, 1.504, 0.528, -2.061, -1.405, -0.256, 2.293, 0.385,
    0.363, 0.928, 1.455, -0.317, 0.804, -1.358, 1.137, -1.072, 0.015, -0.905,
    1.768, -0.562, -1.268, 0.284, -0.952, -1.163, -0.352, 0.507, 0.194, 0.579,
    0.345, 1.171, -1.009, 0.622, -1.311, 0.407, 0.277, -0.191, -1.417, 0.089,
    -1.607, 0.012, -1.355, 0.267, 2.723, -1.16, -2.613, 0.161, -0.371, -0.331,
    0.726, 1.389, 1.111, -0.911, -0.74, -0.818, 1.667, 0.714, -0.262, 0.045,
    0.009, -0.022, -0.508, -1.423, 0.229, 0.765, -0.549, 0.587, 0.183, -0.091,
    0.501, 0.452, 1.927, -0.237, 0.697, 0.181, -1.044, 0.605, 1.178, 1.047,
    1.405, 1.686, -0.382, 1.217, 0.499, 0.271, 0.662, 0.562, 0.528, 0.684, -0.751,
    1.843, 1.063, -1.828, 1.345, 0.077, 0.943, 1.048, 1.637, -0.535, 0.664,
    0.433, -0.559, -0.141, 0.663, 0.777, 1.442, -0.685, -0.451, 0.265, 0.727,
    -1.206, 0.339, -0.32, 0.079, -0.052, 0.097, 0.827, -2.121, 0.57, -2.563,
    0.663, -1.115, 0.176, 0.786, 0.783, 1.068, -1.734, 1.255, 0.79, -0.721,
    -0.028, 0.514, -0.963, -2.052, -1.195, 0.091, 0.187, 0.61, -1.61, -0.066,
    2.733, -0.853, -1.175, 1.079, -0.58, 0.033, 1.213, 0.367, -0.567, -0.107,
    0.188, -0.091, -0.932, -0.11, -1.312, 0.968, 0.698, 1.089, 0.695, 1.309,
    1.017, 0.677, 0.471, -1.524, -1.82, 0.256, 0.397, -0.489, 1.734, -0.297,
    0.075, 0.533, 0.165, 0.814, -1.915, -0.332, 1.035, -0.858, 1.07, 0.532,
    -0.016, 1.932, -0.564, -0.018, -0.542, 1.048, 0.759, 1.575, -1.263, -0.667,
    -1.195, -0.179, 0.337, 0.257, 0.356, 0.375, -1.326, 0.509, 0.975)

par(mfrow = c(2, 3))

hist(obs1, breaks = 12, xlim = c(-3, 3), freq = FALSE, col = "light blue", main = " ")
x1 <- seq(min(obs1) - 2, max(obs1) + 2, 0.01)
y1 <- dnorm(x1, mean(obs1), sd(obs1))
lines(x1, y1, lwd = 1.5)

hist(obs2, breaks = 12, xlim = c(-3, 3), freq = FALSE, col = "light green",
    main = " ")
x2 <- seq(min(obs2) - 2, max(obs2) + 2, 0.01)
y2 <- dnorm(x2, mean(obs2), sd(obs2))
lines(x2, y2, lwd = 1.5)

hist(obs3, breaks = 12, xlim = c(-3, 3), freq = FALSE, col = "yellow", main = " ")
x3 <- seq(min(obs3) - 2, max(obs3) + 2, 0.01)
y3 <- dnorm(x3, mean(obs3), sd(obs3))
lines(x3, y3, lwd = 1.5)

qqnorm(obs1, col = "light blue", main = " ")
qqline(obs1)

qqnorm(obs2, col = "light green", main = " ")
```

```r
qqline(obs2)

qqnorm(obs3, col = "yellow", main = " ")
qqline(obs3)
```



```r
par(mfrow = c(1, 2))
obs <- c(-110, -9, -60, 316, -200, -196, 320, -160, 31, 331, 1731, 21, -926,
    -475, 914, -300, -15, 1, -29, 829, 761, 227, -141, -672, 352, 385, 24, 103,
    -826, 95, 115, 39, -9, -1000, -35, -200, -200, 235, 70, 307, 135, 60, -100,
    -295, -1000, 361, -95, 337, 3712, -255)
```

```r
x <- seq(min(obs) - 3000, max(obs) + 3000, 1)
y <- dnorm(x, mean(obs), sd(obs))

hist(obs, freq = FALSE)
lines(x, y)

qqnorm(obs)
qqline(obs)
```



3.3 Binomial Distribution

A random variable which follows a **binomial distribution** can be written as

$$X \sim Binom(n, p)$$

where n represents the sample size and p represents the probability of success.

The functions seen above for the normal distribution can be similarly used for the binomial as follows where the necessary arguments are `size` and `prob`, corresponding respectively to n and p in our above representation of the binomial.

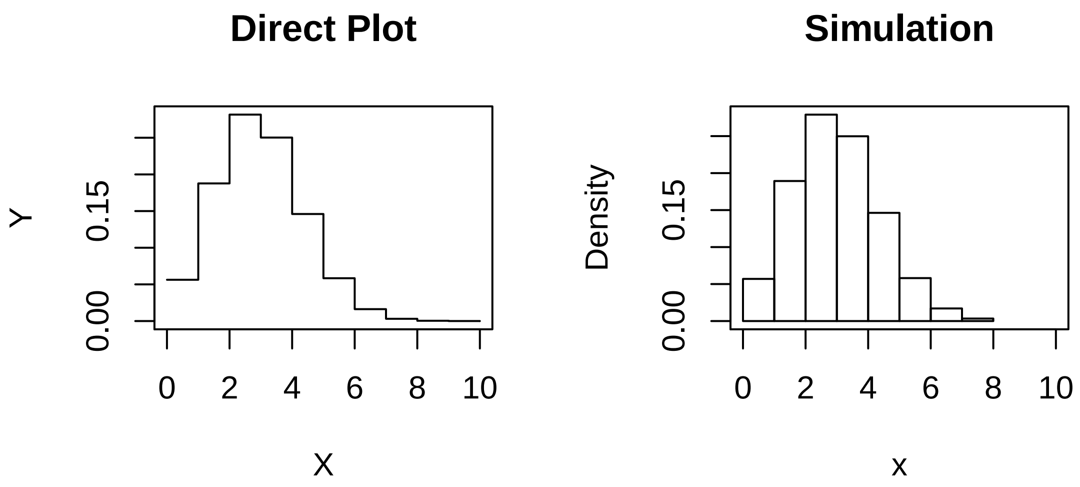
- `dbinom()`
- `rbinom()`
- `pbinom()`
- `qbinom()`

Similar to the above example with the normal distribution, these functions can be used in multiple ways to work with the binomial distribution.

3.3.1 Accessing the Binomial Distribution

As above, the binomial distribution can be accessed directly with `dbinom()` as follows:

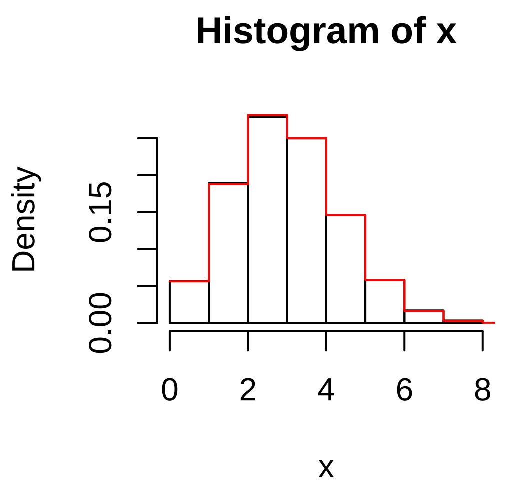
```r
## Splitting the graphics window into two panes
par(mfrow=c(1,2))
## Directly plotting the binomial distribution
 # getting a list of values to evaluate distribution
X = seq(0, 10, 1)
 # getting normal distribution value of the given X values
Y = dbinom(X, size = 10, prob = .25)
 # plotting results
plot(X,Y, type = "s", main = "Direct Plot", xlim = c(0,10))

## Instead Using Simulation
set.seed(1)
x <- rbinom(n = 100000, size = 10, prob = .25)
hist(x, breaks = 10, freq = FALSE,right = FALSE, main = "Simulation", xlim = c(0,10))
box()
```



3.3.2 Comparison of the Methods

Superimposing the graphs is another useful mechanism for direct comparison of the two methods.

```r
hist(x, breaks = 10, freq = FALSE,right = FALSE)
lines(X,Y, type = "s", col = "red")
```



3.3.3 Continuous and Discrete Distributions

Using the binomial distribution highlights an important feature of distributions that must be considered: whether the random variable is **discrete** or **continuous**. A discrete distribution can only take on specified values, typically the integers. The binomial is a great example of this - it can be any integer greater than or equal to zero. For example, the possible values of a binomial are {0, 1, 2, 3, ...}.

On the other hand, a continuous distribution can take on any real number within a specified range. For example, looking at the standard normal distribution, a list of some possible values could be {0, 0.1, 0.01, 0.001, 0.0001, ...}. This is a limited list of the infinite list of possible values that a continuous distribution could take on.

The reason the classification of a random variable as discrete or continous is important is that it affects the implementation of probability functions in R. Let's walk through an example to see the difference. Consider a normally distributed random variable, X, and a binomially distributed random variable, Y.

$$X \sim N(0,1) \qquad Y \sim Binom(15,0.5)$$

If the goal is to determine the probability of being less than 2 for each of these two distributions, this would be equivalent to $P(X < 2)$ and $P(Y < 2)$. However, for the binomial distribution $P(Y < 2) = P(Y \leq 1) = P(Y = 0) + P(Y = 1)$ because there is a discrete set of possible values that Y can take on. Y can only take on the integer values, and less than 2, that just leaves 0 and 1, whereas the normal distribution can take on any non-integer value in that interval, which is an infinite number of values. For the normal distribution, solving for $P(Y < 2)$ is mathematically equivalent to $P(Y \leq 2)$ because it is continuous.

To solve for these probabilities, use `pnorm()` and `pbinom()`, noting that for the discrete binomial case, these functions automatically give the less than or equal to probability, not just the less than probability. The correct commands can be seen through the following examples from the text.

3.3.4 Examples of the Binomial

To summarize, the binomial probaility functions can be used as follows,

- $dbinom(x,n,p) = P(X = x)$
- $pbinom(x,n,p) = P(X \leq x)$

- $pbinom(x, n, p, lower.tail = FALSE) = P(X > x)$

```
## Example 3.19
dbinom(1, size = 4, prob = 0.35)

## [1] 0.384475
```

```
## Example 3.23
dbinom(3, size = 8, prob = 0.35)

## [1] 0.2785858
```

```
## Example 3.24
pbinom(3, size = 8, prob = 0.35, lower.tail = TRUE)

## [1] 0.7063994
```

```
## Example 3.27
pnorm(59, mean = (400*0.2), sd = sqrt(400*0.2*(1-0.2)), lower.tail = TRUE)

## [1] 0.004332448

pbinom(59, prob = .2, size = 400)

## [1] 0.004111644
```

3.4 Poisson Distribution

Similarly, the functions for the **poisson distribution** are the following where the necessary argument is `lambda`

- $dpoisson(x, lambda) = P(X = x)$
- $ppois(x, lambda) = P(X \leq x)$
- $ppois(x, lambda, lower.tail = FALSE) = P(X > x)$

3.4.1 Example of the Poisson

```
## Example 3.28
dpois(x = 2, lambda = 4.4*7)

## [1] 1.99435e-11
```

3.5 Geometric Distribution

The same pattern can be applied for the **geometric distribution** to get the following where the necessary argument is `prob`

- $dgeom(x, prob) = P(X = x)$
- $pgeom(x, prob) = P(X \leq x)$

3.5.1 Examples of the Geometric

Note here that using the dgeom function requires an input of $k-1$, because this function returns the probability of k failures before the first success, ending with a total of k turns.

```
## Example 3.31
dgeom(x = 1-1, prob = 0.35)

## [1] 0.35

dgeom(x = 2-1, prob = 0.35)

## [1] 0.2275

dgeom(x = 3-1, prob = 0.35)

## [1] 0.147875
```

```
## Example 3.35
pgeom(q = (4-1), prob = 0.35)

## [1] 0.8214938
```

3.6 Negative Binomial Distribution

Finally, for the **negative binomial distribution**, the necessary arguments are `prob` and `size` and can be implemented as follows,

- $dnbinom(x,n,p) = P(X = x)$
- $pnbinom(x,n,p) = P(X \leq x)$

3.6.1 Example of the Negative Binomial

```
## Example 3.38
dnbinom(x = 4, size = 6, prob = 0.8)

## [1] 0.05284823
```

Chapter 4

Foundations for Inference

Contents

Taking a Sample of a Dataset

The main text talks through the importance of using a sample of data as a reflection of the population as a whole. The two steps for doing this are as follows

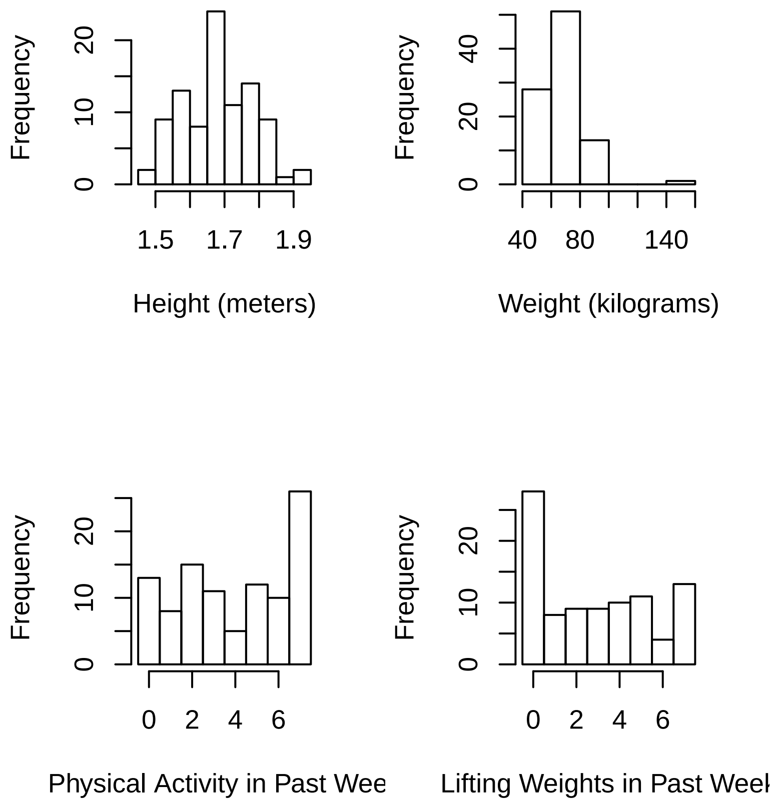
1. Of the total number of datapoints you have (equal to the length of the dataset), select a random sampling of them, of size n. This is done with the sample() command below. x is a list of the row indeces, and sample() returns a random list of values in that list of length n.

2. Taking the output of the sample() command, pair these indeces with the dataset to get the corresponding rows (or columns).

```r
set.seed(5011)
# Step 1: Take a sample of the row indeces
indeces = sample(x = (1:nrow(yrbss)), size = 100, replace = FALSE)
# Step 2: Pull those corresponding individuals from the dataset
yrbss.sample = yrbss[indeces,]
```

Standard histograms plots of some of the variables on this dataset can be plotted.

```r
## Figure 4.4
par(mfrow = c(2, 2))
hist(yrbss.sample$height, xlab = "Height (meters)", main = "", breaks = 15)
hist(yrbss.sample$weight, xlab = "Weight (kilograms)", main = "", breaks = 5)
hist(yrbss.sample$physically.active.7d, xlab = "Physical Activity in Past Week",
    main = "", breaks = -1:7 + 0.5)
hist(yrbss.sample$strength.training.7d, xlab = "Lifting Weights in Past Week",
    main = "", breaks = -1:7 + 0.5)
```



4.1 Variability in Estimates

The sample parameters can be calculated as well as the population parameters. These correspond to the data seen in Table 4.5 in the main text.

```
mean(yrbss.sample$physically.active.7d, na.rm = TRUE)

## [1] 3.93

mean(yrbss$physically.active.7d, na.rm = TRUE)

## [1] 3.903005

median(yrbss.sample$physically.active.7d, na.rm = TRUE)

## [1] 4

median(yrbss$physically.active.7d, na.rm = TRUE)

## [1] 4

sd(yrbss.sample$physically.active.7d, na.rm = TRUE)
```

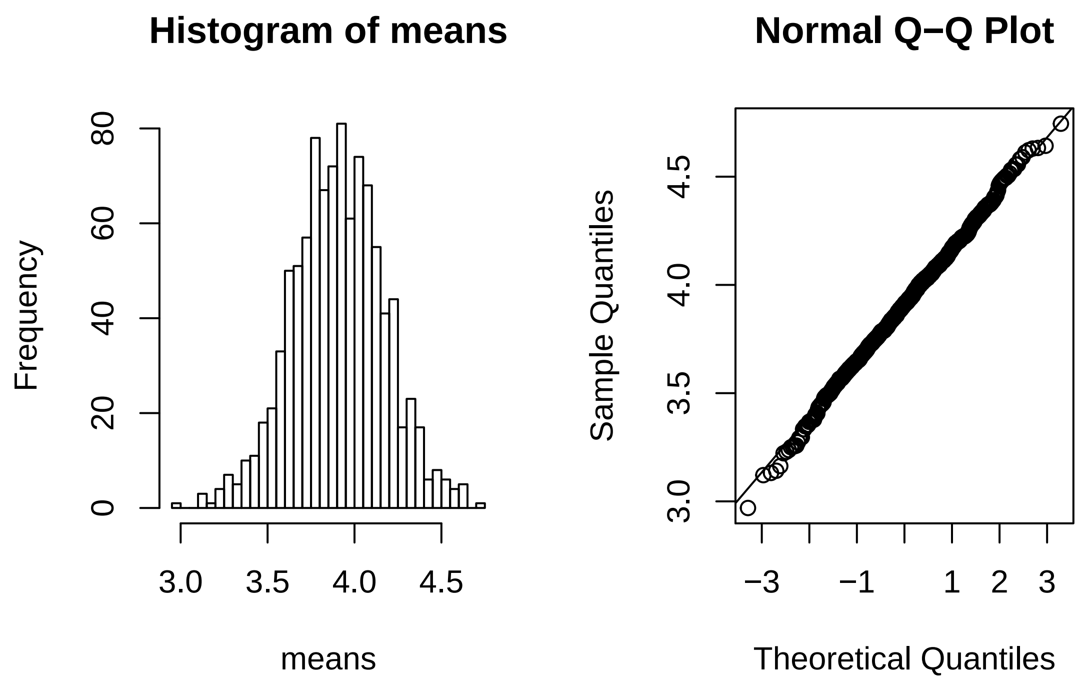
```
## [1] 2.535625

sd(yrbss$physically.active.7d, na.rm = TRUE)

## [1] 2.564105
```

4.1.1 Sampling Distribution for the Mean

The concept of a sampling distribution highlights the fact that sampling is a random process, and every sample is likely to be quite different than any other. For this reason, sampling distributions are created, which represent the accumulated information of a large number of random samples.

The following steps can be used to create a **sampling distribution of the sample mean**.

```
## Figure 4.7
# Step 1: Create an empty list to put values in
means = rep(NA, 1000)

# Step 2: Use a "for" loop to collect 1000 samples
for(ii in 1:1000){
  # Step 2a: Get the random sample
  indeces = sample(x = (1:nrow(yrbss)), size = 100, replace = FALSE)
  sample = yrbss[indeces,]
  # Step 2b: Take the mean of the sample and store it
  means[ii] = mean(sample$physically.active.7d, na.rm = TRUE)
}

# Step 3: Plot your results
par(mfrow = c(1,2))
hist(means, breaks = 30)
qqnorm(means)
qqline(means)
```



As discussed in chapter 3, the above plots are used to determine the normality of the sample. The left plot is just a histogram, which can be inspected visually to see that it approximates a normal distribution. The plot on the right is a more powerful tool for determining the normality - the dots represent deviation of data points from a theoretical normal distribution, as represented by the line on the plot. The above plot shows a fairly normal distribution because of how closely the dots match the line.

The more samples are taken, the more accurately will the distribution be depicted. Figure 4.8 from *OI Biostat* demonstrates this below

```r
## Figure 4.8
means = rep(NA, 100000)
for(ii in 1:100000){
  indeces = sample(x = (1:nrow(yrbss)), size = 100, replace = FALSE)
  sample = yrbss[indeces,]
  means[ii] = mean(sample$physically.active.7d, na.rm = TRUE)
}

par(mfrow = c(1,2))
hist(means, breaks = 30)
qqnorm(means)
qqline(means)
```

4.2 Confidence Intervals

The first formula introduced in the text for calculating confidence intervals is as follows,

$$\text{point estimate } \pm 1.96 \cdot \text{SE}$$

```
## Example 4.3
xbar = mean(yrbss.sample$physically.active.7d, na.rm = TRUE)
std.error = sd(yrbss.sample$physically.active.7d, na.rm = TRUE)/sqrt(length(yrbss.sample$physically.active.7d
ci = c(xbar - 1.96*std.error, xbar + 1.96*std.error)
ci

## [1] 3.433018 4.426982
```

To generalize this formula, a standard normal distribution can be used to obtain z^*, giving the following

$$\bar{x} \pm z^* \cdot \text{SE}$$

R can be used to calculate z^* and then using that value, to solve for the confidence interval. A key point here is that we want the middle 95% of the distribution, which divides the remaining 5% between the two tails, equivalent to 2.5% being in each tail.

```
## Example 4.3 (version 2)
perc = .95
z = qnorm(p = perc + (1-perc)/2, lower.tail = TRUE)
z

## [1] 1.959964

ci = c(xbar - 1.96*std.error, xbar + 1.96*std.error)
ci

## [1] 3.433018 4.426982
```

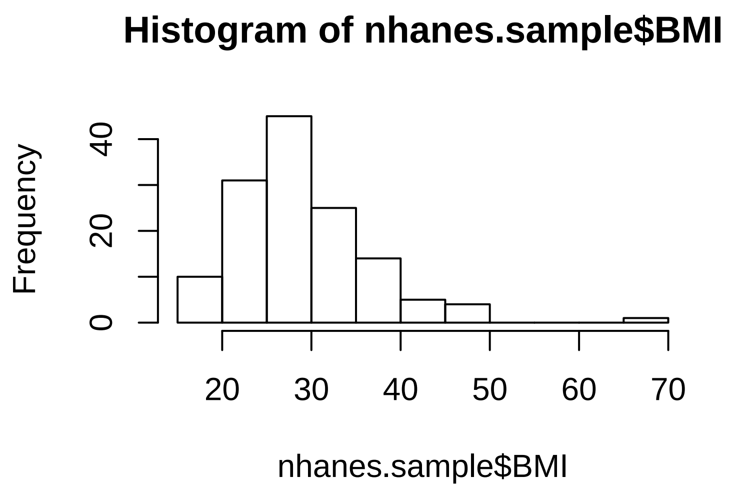
```
## Example 4.6
perc = .99
z = qnorm(p = perc + (1-perc)/2, lower.tail = TRUE)
z

## [1] 2.575829
```

```
## Example 4.8
ci = c(xbar - z*std.error, xbar + z*std.error)
ci

## [1] 3.276866 4.583134
```

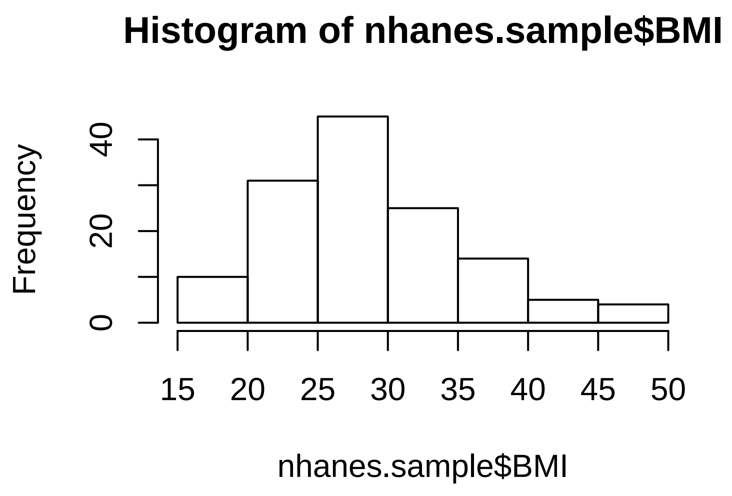
Example 4.10 from *OI Biostat* is an excellent example of how to clean up data. *OI Biostat* restricts the sample to adults over 21 with reported BMI values. The steps below show how to find the individuals who are children or who have missing data and how to remove them from the sample. Often times, when using a sample of a large population, a process similar to this one must be used. Missing data can be problematic for analyses of the data, so understanding how to clean up the data appropriately is quite important.

```
## Example 4.10
# Collect the sample of size 200
```

```r
set.seed(5011)
indeces = sample(1:length(NHANES$ID), size = 200)
nhanes.sample = NHANES[indeces,]

# First remove the children from the sample
children = which(nhanes.sample$Age < 21)  #Find children
nhanes.sample = nhanes.sample[-children, ]       #Remove them from the sample

hist(nhanes.sample$BMI, breaks = "FD") ## This gives Figure 4.10
```



```r
# Locate where the outlier is occuring
x = which(nhanes.sample$BMI == max(nhanes.sample$BMI, na.rm = TRUE))
# Remove the outlier
nhanes.sample = nhanes.sample[-x,]
# Plot again to confirm that it was removed
hist(nhanes.sample$BMI)
```



```
# Calculate sample statistics and confidence interval
xbar = mean(nhanes.sample$BMI, na.rm = TRUE)
n = length(nhanes.sample$BMI)
s = sd(nhanes.sample$BMI, na.rm=TRUE)
se = s/sqrt(n)
perc = .95
z = qnorm(p = perc + (1-perc)/2, lower.tail = TRUE)
ci = c(xbar - z*se, xbar + z*se)
ci

## [1] 27.66076 29.94282
```

4.3 Hypothesis Testing

If you have calculated your test statistic by hand, R can easily be used to get the p-value using the function `pnorm()` as in Chapter 3. The example that is worked through in *OI Biostat* can be completed as follows. Note that `lower.tail = FALSE` because the alternative hypothesis here is

$$H_A : \mu_{bmi} > 21.7$$

which implies that the upper tail must be considered.

```
mu = 21.7
t = (xbar - mu)/(s/sqrt(n))
t

## [1] 12.19889

pnorm(t, lower.tail = FALSE)

## [1] 1.575536e-34
```

In R, all of the steps of hypothesis testing can be done with one single function, `t.test()`. This gives the test statistic, the p-value, and the confidence interval. The function takes the data as an argument, *x*, the null hypothesis value, as *mu*, and the alternative hypothesis type, as *alternative*, which can be `"less"`, `"greater"`, or `"two.sided"`.

Example 4.13 from *OI Biostat*, shown below, gives a comparison of doing the method by hand, as well as using the function.

```
## Example 4.13
set.seed(5011)
indeces = sample(1:length(NHANES$ID), size = 200)
nhanes.sample = NHANES[indeces,]

# First remove the children from the sample
children = which(nhanes.sample$Age < 21)  #Find children
nhanes.sample = nhanes.sample[-children, ]

# Method 1
xbar = mean(nhanes.sample$SleepHrsNight, na.rm = TRUE)
s = sd(nhanes.sample$SleepHrsNight, na.rm = TRUE)
mu = 7
n = length(nhanes.sample$SleepHrsNight)
t = (xbar - mu)/(s/sqrt(n))
t

## [1] -0.864111

p = pnorm(t, lower.tail = TRUE)
p

## [1] 0.1937634

# Method 2
t.test(x = nhanes.sample$SleepHrsNight, mu = mu, alternative = "less")

##
##  One Sample t-test
##
## data:  nhanes.sample$SleepHrsNight
## t = -0.86411, df = 134, p-value = 0.1945
## alternative hypothesis: true mean is less than 7
## 95 percent confidence interval:
##      -Inf 7.095073
## sample estimates:
## mean of x
##  6.896296
```

Chapter 5

Inference For Numerical Data

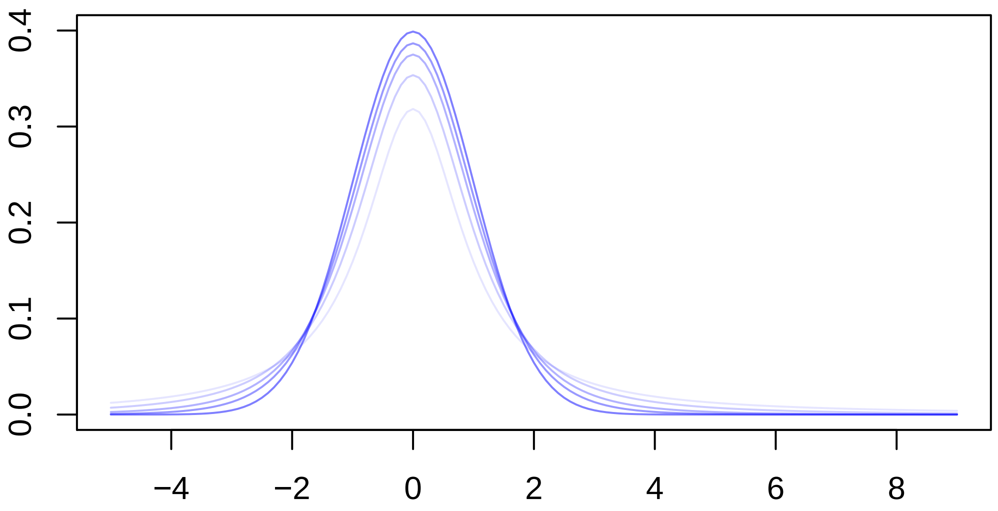
Contents

5.1 The t-distribution

The main text introduces the **t-distribution** and explains under what circumstances it should be used. Using it in in R is very similar to many concepts seen through Chapter 4 thus far.

The following plot shows how the t-distribution approximates the normal distribution as the degrees of freedom grow. The use of the function `dt()` parallels the use of `dnorm()` as has been used in previous chapters.

```r
## Figure 5.2 Getting the x-axis values
x = seq(-5, 9, 0.1)

# Finding the corresponding distribution values for several distributions
t1 = dt(x, df = 1)
t2 = dt(x, df = 2)
t4 = dt(x, df = 4)
t8 = dt(x, df = 8)
norm = dnorm(x)

# Plotting the results, with custom colors
plot(x, t1, type = "l", ylim = c(0, 0.4), xlab = "", ylab = "", col = rgb(0,
    0, 1, alpha = 0.1))
lines(x, t2, col = rgb(0, 0, 1, alpha = 0.2))
lines(x, t4, col = rgb(0, 0, 1, alpha = 0.3))
lines(x, t8, col = rgb(0, 0, 1, alpha = 0.4))
lines(x, norm, col = rgb(0, 0, 1, alpha = 0.5))
```



5.1.1 The Alternative to t-Tables

The main text talks through the use of a **t-table**, but this text will go through examples of using software to do the same calculations. In R, the command of interests here are pt() to get the percentage of the distribution above or below a certain value and qt() to get distribution values for which a certain percentage of the distribution falls above. The command pt() takes values inside the t-table to return the values on the x-axis of the table, whereas qt() returns values within the table when given values along the upper x-axis.

An example of a one-sided calculation using the t-distribution can be viewed in Example 5.1 from *OI Biostat* shown below. Note the use of the stipulation lower.tail = TRUE which ensures that the calculation is of values in the lower tail, i.e. those values which are less than -2.1.

```r
## Example 5.1
pt(-2.1, df = 18, lower.tail = TRUE)

## [1] 0.0250452
```

Example 5.2 from *OI Biostat* shows a two sided calculation using the t-distribution. In order to do this in R, each tail can be calculated separately and then the two values summed. Alternatively, the symmetry of the t-distribution can be utilized to calculate one tail and multiply it by two.

```r
## Example 5.2
# Method 1: calculate each tail and sum
pt(1.65, df = 20, lower.tail = FALSE) + pt(-1.65, df = 20, lower.tail = TRUE)
```

```
## [1] 0.1145608

# Method 2: calculate one tail and multiply by 2
2*pt(1.65, df = 20, lower.tail = FALSE)

## [1] 0.1145608
```

5.1.2 Confidence Intervals

As in Chapter 4, confidence intervals using the t-distribution can be set up in the same way with a t^* value instead of a z^* value.

```
## Example 5.3
n = 19
xbar = 4.4
s = 2.3
std.error = s/sqrt(n)
perc = .95
t.star = qt(p = perc + (1-perc)/2, df = n-1, lower.tail = TRUE)
ci = c(xbar - t.star*std.error, xbar + t.star*std.error)
ci

## [1] 3.291435 5.508565
```

```
## Example 5.5
mu = 0.5
n = 15
xbar = 0.287
s = 0.069
std.error = s/sqrt(n)
t = (xbar - mu)/std.error
pt(t, df = n-1)

## [1] 4.903809e-09
```

5.2 Paired Data

When working with paired data, several methods can be used to measure the effect. The first method is to calculate by hand differences between the pairs and then to perform a standard t-test as follows.

```
## Example 5.6
diff = swim$wet.suit.velocity - swim$swim.suit.velocity
t.test(diff, mu = 0, alternative = "two.sided")

##
##  One Sample t-test
##
## data:  diff
## t = 3.7019, df = 11, p-value = 0.00349
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.02534062 0.09965938
## sample estimates:
## mean of x
##    0.0625
```

The second method involves using a **paired t-test**, which performs the same operation without the direct calculation of the differences being required first. It can be seen that the output of this test is the exact same as the test above. This is a nice example of R taking out a lot of the computational work.

```
t.test(swim$wet.suit.velocity, swim$swim.suit.velocity, paired = TRUE)

##
##  Paired t-test
##
## data:  swim$wet.suit.velocity and swim$swim.suit.velocity
## t = 3.7019, df = 11, p-value = 0.00349
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.02534062 0.09965938
## sample estimates:
## mean of the differences
##                  0.0625
```

5.3 Difference of Two Means

5.3.1 Hypothesis Tests for a Different in Means

In order to perform a **two-sample t-test**, there are again two methods. The first involves dividing the data into two samples (unless the data has already come separated) and performing a t-test. The second has R do the entire process. Note that for this method, the tilde signifies the grouping variable. So in this case, the weight data is grouped by the variable smoke.

```
## Example 5.11
# Method 1: divide the data by hand
smoker = baby.smoke$weight[baby.smoke$smoke == "smoker"]
non.smoker = baby.smoke$weight[baby.smoke$smoke == "nonsmoker"]
t.test(non.smoker, smoker)

##
##  Welch Two Sample t-test
##
## data:  non.smoker and smoker
## t = 1.4967, df = 89.277, p-value = 0.138
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.1311663  0.9321663
## sample estimates:
## mean of x mean of y
##    7.1795    6.7790
```

```
# Method 2: R does it all
t.test(baby.smoke$weight~baby.smoke$smoke)

##
##  Welch Two Sample t-test
##
## data:  baby.smoke$weight by baby.smoke$smoke
## t = 1.4967, df = 89.277, p-value = 0.138
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.1311663  0.9321663
## sample estimates:
## mean in group nonsmoker    mean in group smoker
##                  7.1795                  6.7790
```

5.3.2 Pooled t-test

Performing what is known as a **pooled t-test** is very simple in R. The procedure is the same as a two-sample t-test, with the addition of the clause `var.equal = TRUE` in the `t.test()` command. This is because a pooled t-test relies on the assumption of nearly equal variance between the two samples. For example, performing the same test as above but using a pooled t-test would look like the following,

```
t.test(baby.smoke$weight~baby.smoke$smoke, var.equal = TRUE)

##
##  Two Sample t-test
##
## data:  baby.smoke$weight by baby.smoke$smoke
## t = 1.5517, df = 148, p-value = 0.1229
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.1095531  0.9105531
## sample estimates:
## mean in group nonsmoker    mean in group smoker
##                  7.1795                  6.7790
```

5.5 ANOVA

The main text walks through the arithmetic procedures of the ANOVA test and calculating the F-statistic. R makes this process much simpler, such that the entire calculation can be performed with just one line of code. The command for this is aov(), which takes two variables. The first variable before the tilde is the one for which the means are being compared, while the second variable after the tilde specifies the groups. The command below works out Example 5.22 from *OI Biostat*, getting the same *F* statistic as the hand calculated value.

```
## Example 5.22
summary(aov(famuss$ndrm.ch ~ famuss$actn3.r577x))

##                     Df Sum Sq Mean Sq F value Pr(>F)
## famuss$actn3.r577x   2   7043    3522   3.231 0.0402 *
## Residuals          592 645293    1090
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Chapter 6

Simple Linear Regression

Contents

6.1 Checking Assumptions

Before going into a discussion of linear regression itself, an investigation into the appropriateness of its use must first be conducted. The main text walks through the four assumptions necessary to use an approximation to linearity, which are as follows

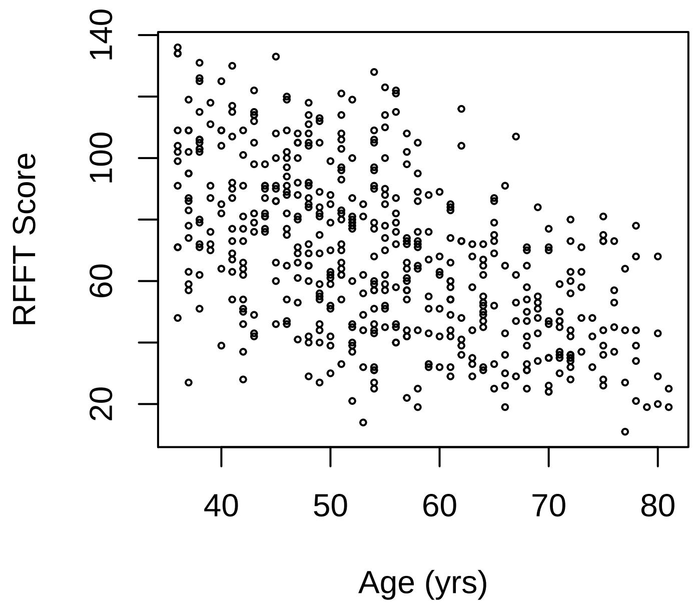
1. Linearity
2. Constant Variability
3. Independent Observations
4. Residuals that are approximately normally distributed

There are various methods to check each of these assumptions, which should be investigated before the use of linear regression. This next section goes through these methods and several examples.

6.1.1 Linearity

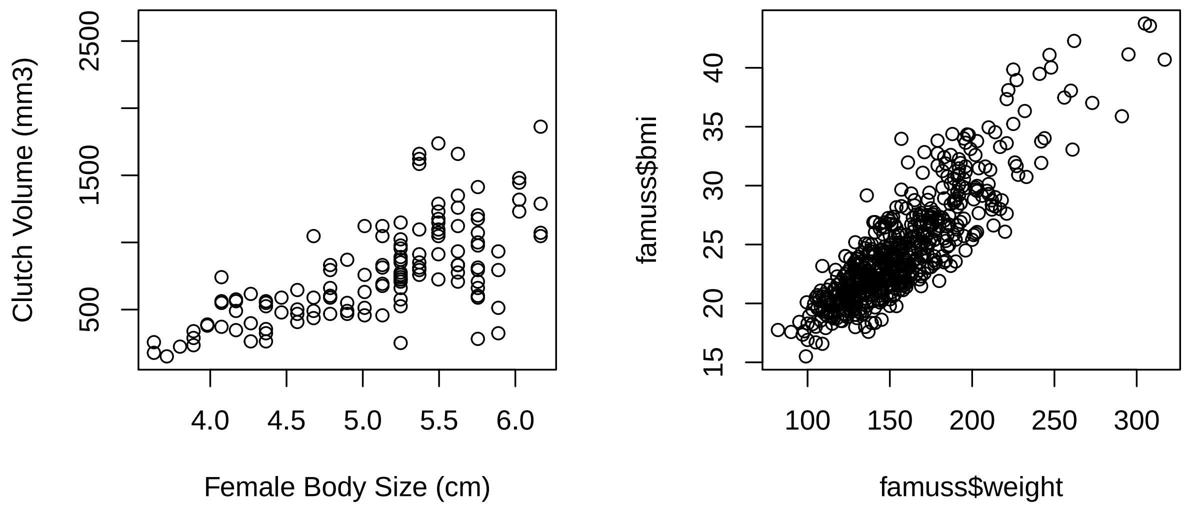
To check for linearity, a simple examination of a scatterplot of the data will suffice. The goal is to look for data such that a straight line can be drawn which appropriately represents the data. The plot of Figure 6.1 from *OI Biostat* below shows that *age* and *RFFT* appear to have a linear relationship. The inclusion of the command `cex = 0.4` simply changes the size of the dots on the scatterplot, making them 40% of their original size.

```r
## Figure 6.1 First collecting the data sample
set.seed(5011)
row.num = sample(1:nrow(statins), 500, replace = FALSE)
statins.samp = statins[row.num, ]

# Then plotting it
plot(statins.samp$RFFT ~ statins.samp$Age, cex = 0.4, ylab = "RFFT Score", xlab = "Age (yrs)",
    main = "")
```



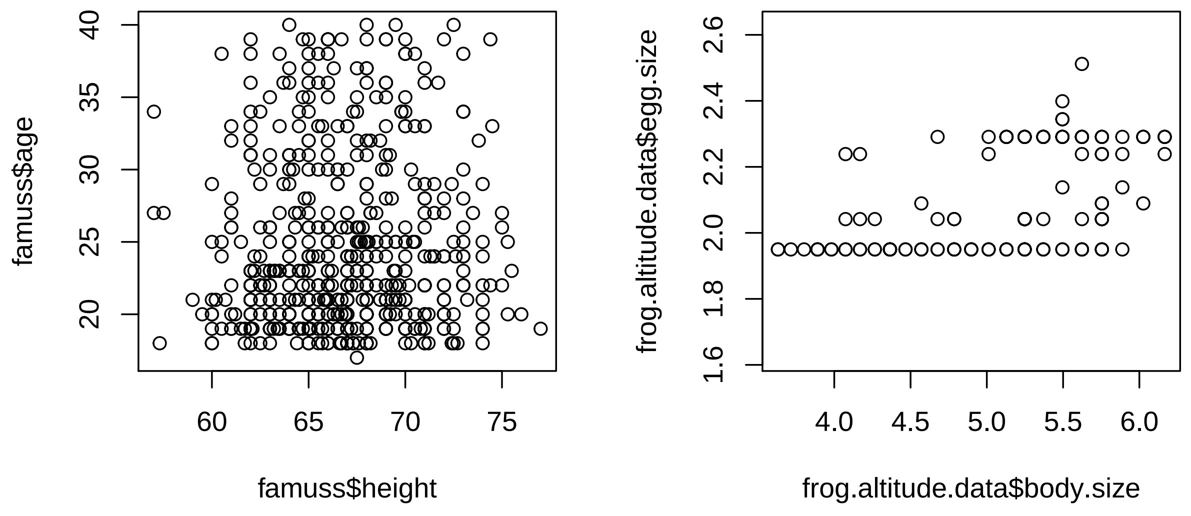
As further examples, the figures below show other linear relatonships.

```r
par(mfrow = c(1, 2))
## Figure 6.2
plot(frog.altitude.data$clutch.volume ~ frog.altitude.data$body.size, ylab = "Clutch Volume (mm3)",
    xlab = "Female Body Size (cm)")

plot(famuss$weight, famuss$bmi)
```



For comparison, the following plot shows data that a linear relationship is not appropriate for. The first example shows data that appears to have no relationship, as indicated by the lack of trend in the scatterplot. The second example shows a pattern that would be difficult to capture with a single straight line.

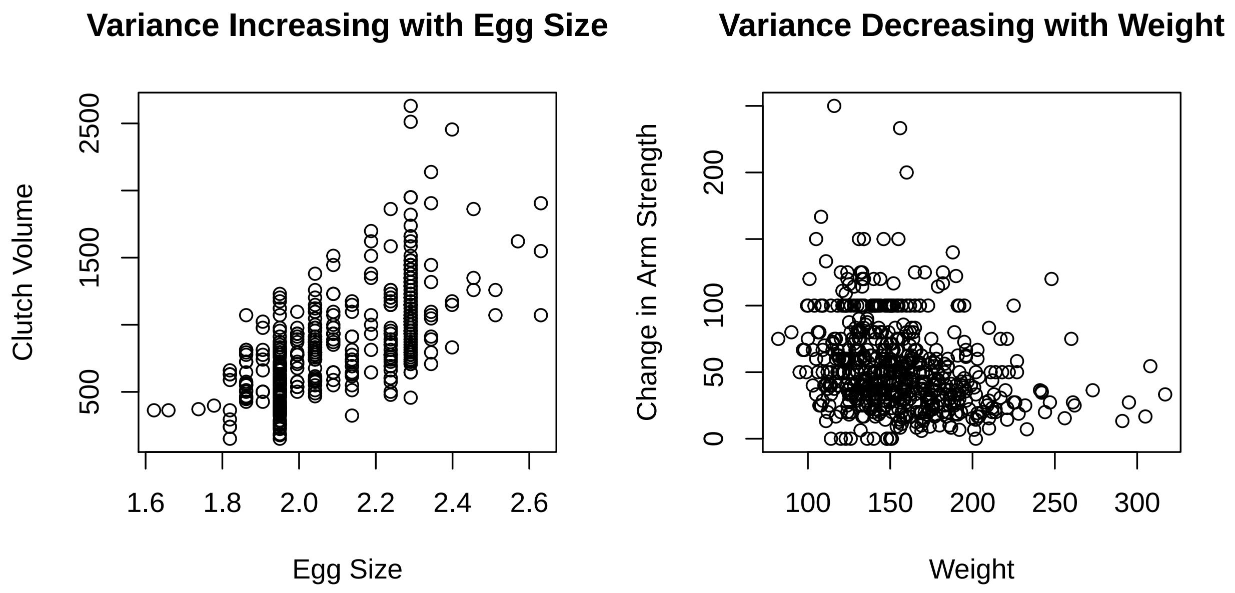
```r
par(mfrow = c(1, 2))
plot(famuss$height, famuss$age)
plot(frog.altitude.data$body.size, frog.altitude.data$egg.size)
```



6.1.2 Constant Variability

To check for **constant variability**, a scatterplot is again a good resource. Both Figure 6.1 and Figure 6.2 above show strong examples of constant variability. For comparison, a few examples of non-constant variability are included here. Notice how in the plot on the left, there is less variability in the clutch volumes with small egg size in comparison to the values with higher egg size. Similarly, there is more variability in change in arm strength for lower weight individuals than for higher weight individuals.

```r
par(mfrow = c(1, 2))
plot(frog.altitude.data$clutch.volume ~ frog.altitude.data$egg.size, main = "Variance Increasing with Egg Size
    xlab = "Egg Size", ylab = "Clutch Volume")
plot(famuss$ndrm.ch ~ famuss$weight, main = "Variance Decreasing with Weight",
    xlab = "Weight", ylab = "Change in Arm Strength")
```



6.1.3 Independence of Observations

The assumption of **independence of observations** is less easily understood through physical study of the data. Instead, the nature of the data and its parameters must be understood. For example, some questions to consider include

- Do the data points depend on each other?
- Is there a known relationship between some or all data points?
- Are the data points from the same individual, region, or time?

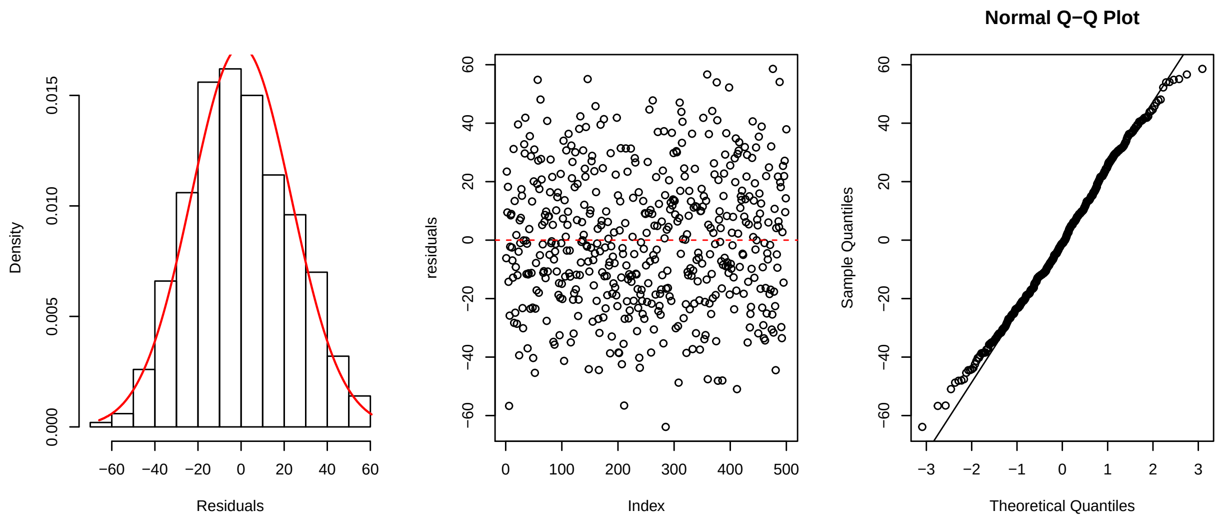
6.1.4 Normally Distributed Residuals

In R, a plot of the residuals can be visualized to determine if this assumption is met. First, to obtain the residuals, the linear regression must be obtained, using the command `lm()`. Then, the residuals of this model can be calculated using the command `resid()` on the linear regression.

These residuals can be visualized in a few ways: as a histogram, as a plot, or using qnorm() and qqline().

Two examples of this procedure are included below. In the first example, the residuals appear very close to normal. This is a strong sign that the assumption is met.

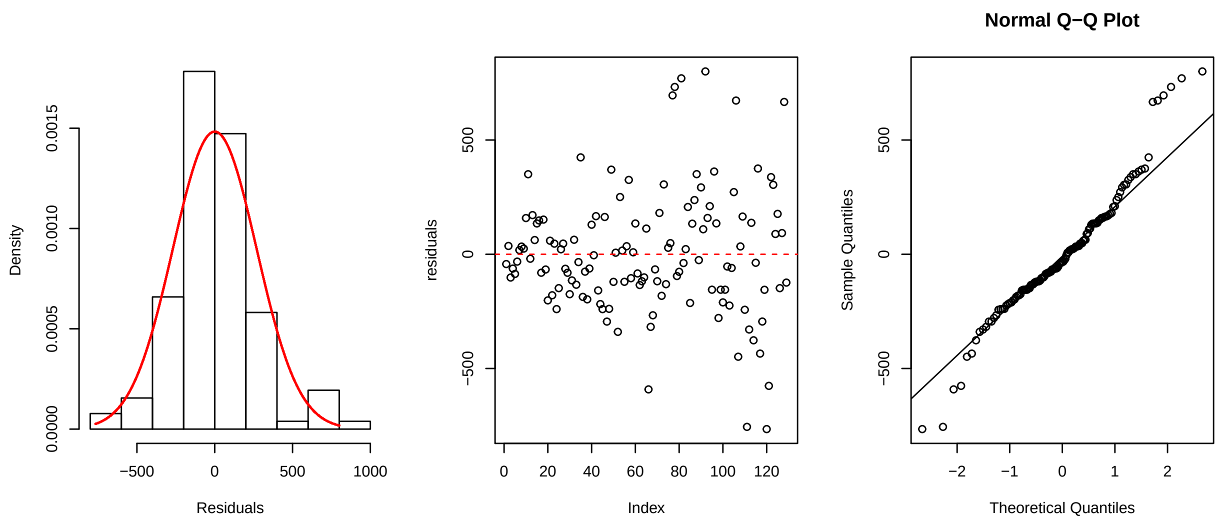
```r
par(mfrow = c(1, 3))
residuals = resid(lm(statins.samp$RFFT ~ statins.samp$Age))

hist(residuals, main = "", xlab = "Residuals", freq = FALSE)
x <- seq(min(residuals) - 2, max(residuals) + 2, 0.01)
y <- dnorm(x, mean(residuals), sd(residuals))
lines(x, y, lwd = 1.5, col = "red")

plot(residuals)
abline(h = 0, col = "red", lty = 2)

qqnorm(residuals)
qqline(residuals)
```



In the second example, the data appears somewhat normal, but with some outliers. This does not appear to be an egregious violation, but is something that should be kept in mind as the results of the linear regression are interpreted.

```r
par(mfrow = c(1, 3))
## Guided Practice 6.1
residuals = resid(lm(frog.altitude.data$clutch.volume ~ frog.altitude.data$body.size))

hist(residuals, main = "", xlab = "Residuals", freq = FALSE)
x <- seq(min(residuals) - 2, max(residuals) + 2, 0.01)
y <- dnorm(x, mean(residuals), sd(residuals))
lines(x, y, lwd = 1.5, col = "red")

plot(residuals)
abline(h = 0, col = "red", lty = 2)
```

```r
qqnorm(residuals)
qqline(residuals)
```



6.2 Creating a Linear Regression

Using R to create a linear regression is quite simple, and several examples will be shown here to illustrate that. As background, there is one command needed for linear regressions in R, `lm()`. This function takes two arguments, the **explanatory variable** and the **response variable** in the form `lm(response ∼ explanatory)` where the symbol between those two variables is a tilde. For example, a regression for the statins data can be calculated as follows

```r
lm(statins.samp$RFFT~statins.samp$Age)

##
## Call:
## lm(formula = statins.samp$RFFT ~ statins.samp$Age)
##
## Coefficients:
##      (Intercept)  statins.samp$Age
##          137.550            -1.261
```

This output implies the following regression formula

$$RFFT = 137.55 - 1.261 \cdot Age$$

To get more information about the regression, the `summary()` command can be used as follows,

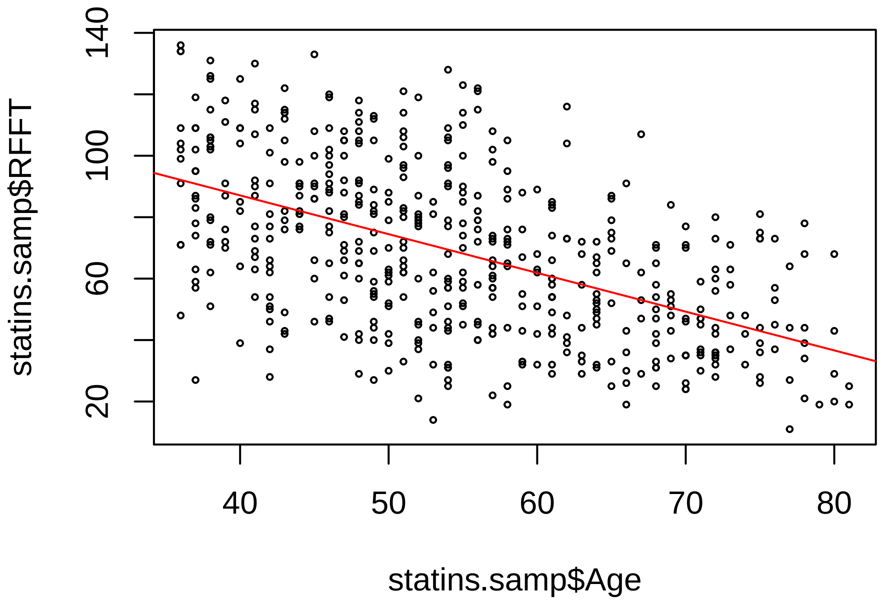
```r
summary(lm(statins.samp$RFFT~statins.samp$Age))

##
## Call:
## lm(formula = statins.samp$RFFT ~ statins.samp$Age)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max
## -63.879 -16.845  -1.095  15.524  58.564
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      137.54972    5.01614   27.42   <2e-16 ***
## statins.samp$Age  -1.26136    0.08953  -14.09   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.19 on 498 degrees of freedom
## Multiple R-squared:  0.285,Adjusted R-squared:  0.2836
## F-statistic: 198.5 on 1 and 498 DF,  p-value: < 2.2e-16
```

Note that the summary of the linear regression is much more useful, providing not only the regression model, but standard errors, p-values, the R^2 value, and a few other useful statistics which will be discussed in greater deal in future chapters.

With this linear regression in mind, it would be helpful to be able to visualize this on the scatterplot. This can be done using the command `abline()`, which takes as its arguments the model itself.

```
plot(statins.samp$RFFT ~ statins.samp$Age, cex = 0.4)
abline(lm(statins.samp$RFFT ~ statins.samp$Age), col = "red")
```

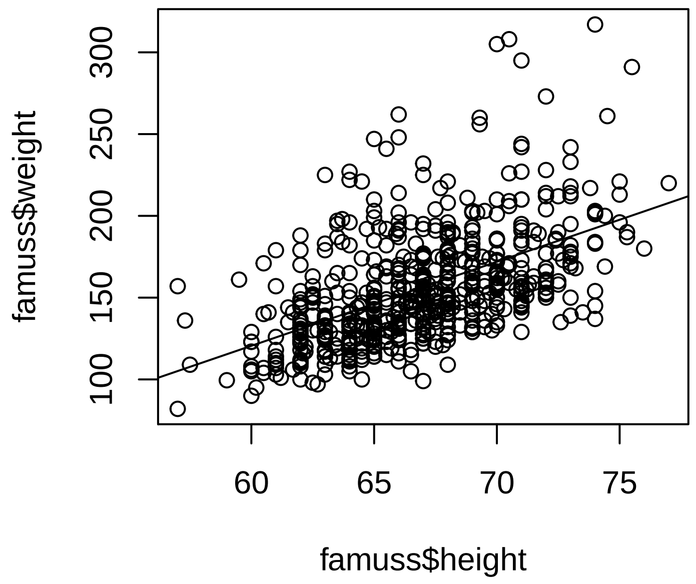


6.2.1 FAMuSS Regression

Performing a similar regression for the FAMuSS dataset between height and weight gives us the following model

$$weight = -187.79 + 5.14 \cdot height$$

```r
plot(famuss$weight ~ famuss$height)
abline(lm(famuss$weight ~ famuss$height))
```



```r
summary(lm(famuss$weight ~ famuss$height))

##
## Call:
## lm(formula = famuss$weight ~ famuss$height)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -57.514 -19.452  -6.959  13.882 133.500
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -187.7908    22.5454  -8.329 5.64e-16 ***
## famuss$height    5.1389     0.3369  15.255  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 29.34 on 593 degrees of freedom
## Multiple R-squared:  0.2818,Adjusted R-squared:  0.2806
## F-statistic: 232.7 on 1 and 593 DF,  p-value: < 2.2e-16
```

6.2.2 WDI Regression with Two Levels

Although the interpretation of such a model can be different, the method of performing a regression on a variable with two levels is the same. The command below illustrates how to obtain Table 6.16.

```
summary(lm(log(wdi.2011$inf.mort)~wdi.2011$sanit.access.factor))

##
## Call:
## lm(formula = log(wdi.2011$inf.mort) ~ wdi.2011$sanit.access.factor)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.7501 -0.4829  0.0440  0.4407  2.0008
##
## Coefficients:
##                                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)                        4.0184     0.1100   36.52   <2e-16 ***
## wdi.2011$sanit.access.factorhigh  -1.6806     0.1322  -12.72   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7781 on 161 degrees of freedom
##   (2 observations deleted due to missingness)
## Multiple R-squared:  0.5011,Adjusted R-squared:  0.498
## F-statistic: 161.7 on 1 and 161 DF,  p-value: < 2.2e-16
```

Furthermore, the two t-tests shown in the text can be computed as follows. Note that the first t-test, which assumes an equal variance among groups, gives the same t-statistic and p-value as the regression above.

```
## Pooled t-test
t.test(log(wdi.2011$inf.mort)~wdi.2011$sanit.access.factor, var.equal = TRUE)

##
##  Two Sample t-test
##
## data:  log(wdi.2011$inf.mort) by wdi.2011$sanit.access.factor
## t = 12.717, df = 161, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  1.419601 1.941570
## sample estimates:
##  mean in group low mean in group high
##           4.018421           2.337835

## Unpooled t-test
t.test(log(wdi.2011$inf.mort)~wdi.2011$sanit.access.factor, var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  log(wdi.2011$inf.mort) by wdi.2011$sanit.access.factor
## t = 12.717, df = 161, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  1.419601 1.941570
## sample estimates:
##  mean in group low mean in group high
##           4.018421           2.337835
```

Chapter 7

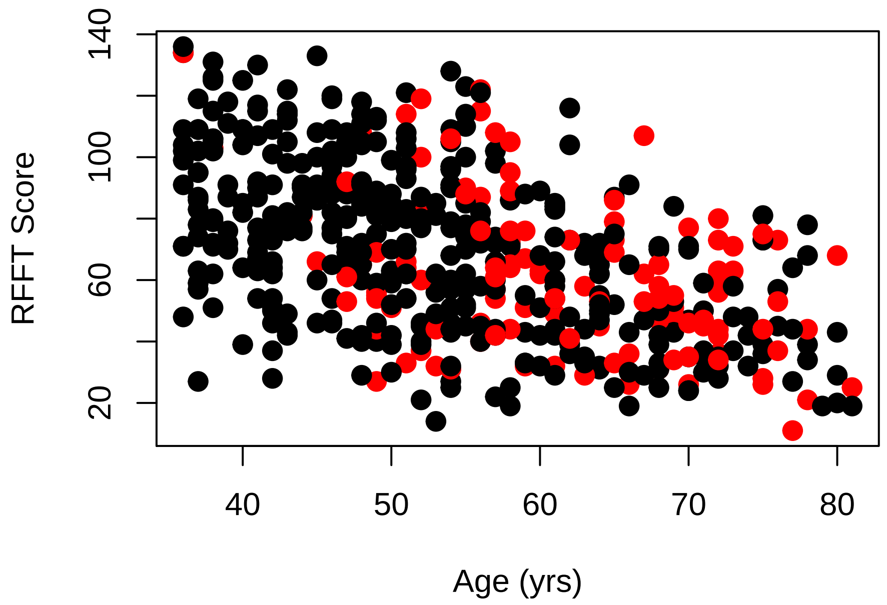
Multiple and Logistic Regression

Contents

In Chapter 7, the dataset statins was introduced when visually understanding the assumptions of simple linear regression. It will again be used in this chapter to understand the idea of multiple linear regression.

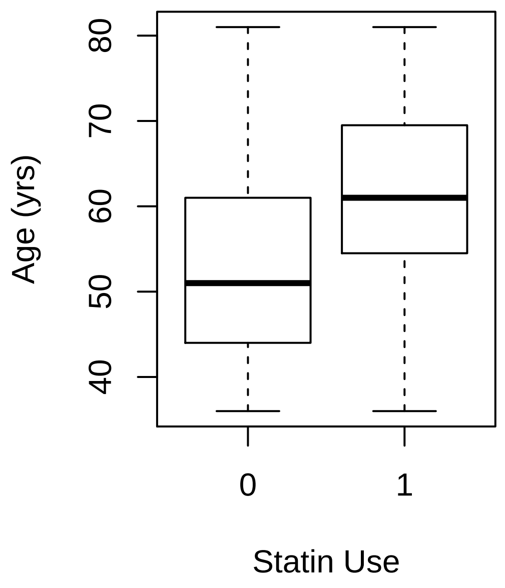
Confounders

```
## Figure 7.1 First collecting the data sample
set.seed(5011)
row.num = sample(1:nrow(statins), 500, replace = FALSE)
statins.samp = statins[row.num, ]

# Make the plot where color is specified by Statin use
plot(statins.samp$Age, statins.samp$RFFT, pch = 19, cex = 1.3, col = as.factor(statins.samp$Statin),
    xlab = "Age (yrs)", ylab = "RFFT Score")
```



OI Biostat walks through the example of statin use as a confounder because it is theoretically associated with both another explanatory variable, in this case age, and with a response variable, here cognitive decline. To test for a confounder, a measure of association can be used such as correlation or a side-by-side boxplot. In this example, the correlations between the variables can be calculated as follows.

```
## Figure 7.4 Visual test of association
boxplot(statins.samp$Age ~ statins.samp$Statin, ylab = "Age (yrs)", xlab = "Statin Use")
```



```
# Numerical test of association
cor(statins.samp$Statin, statins.samp$Age)

## [1] 0.3116923

cor(statins.samp$Statin, statins.samp$RFFT)

## [1] -0.1545881
```

A correlation greater than 0.3 or less than -0.3 is considered signficant. Note that these correlations are not immediately concerning as only one is mildly signficant, but nonetheless, they could be impactful on an investigation and should be accounted for if possible. The conclusion in the main text is thus that a multiple linear regression should be used because it can adjust for confounders. The explanation of how this works is complicated, but the idea is that because multiple linear regression builds a model that reflects the association between explanatory and response variables, the more information included in building this model, the better.

7.2 Simple vs. Multiple Regression

Firstly, the simple linear regression can be performed as would have been done in Chapter 6.

```
summary(lm(statins.samp$RFFT ~ statins.samp$Statin))

##
## Call:
## lm(formula = statins.samp$RFFT ~ statins.samp$Statin)
```

```
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -56.714 -22.714   0.286  18.299  73.339
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)           70.714      1.381  51.212  < 2e-16 ***
## statins.samp$Statin  -10.053      2.879  -3.492 0.000523 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 27.09 on 498 degrees of freedom
## Multiple R-squared:  0.0239,Adjusted R-squared:  0.02194
## F-statistic: 12.19 on 1 and 498 DF,  p-value: 0.0005226
```

This gives the following model,

$$RFFT = 70.714 - 10.053 \cdot Statin$$

Next, the multiple regression will be performed. Note the use of a plus sign after the tilde to include multiple variables.

```
summary(lm(statins.samp$RFFT ~ statins.samp$Statin + statins.samp$Age))

##
## Call:
## lm(formula = statins.samp$RFFT ~ statins.samp$Statin + statins.samp$Age)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -63.855 -16.860  -1.178  15.730  58.751
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         137.8822     5.1221  26.919   <2e-16 ***
## statins.samp$Statin   0.8509     2.5957   0.328    0.743
## statins.samp$Age     -1.2710     0.0943 -13.478   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.21 on 497 degrees of freedom
## Multiple R-squared:  0.2852,Adjusted R-squared:  0.2823
## F-statistic: 99.13 on 2 and 497 DF,  p-value: < 2.2e-16
```

This gives the following model,

$$RFFT = 137.8822 + 0.8509 \cdot Statin - 1.2710 \cdot Age$$

As compared to a model with statin use as the only explanatory variable, it is evident that the coefficients here change quite drastically by including both explanatory variables. This is reflective of the confounding relationship between the two explanatory variables, but will hopefully lead us to more accurate results.

Notice how in this example, the first model shows a negative relationship between statin use and RFFT values, while the multiple regression shows a positive relationship. This is reflective that the `Statin` variable is representative of different things in the two models. In the simple model, the coefficient for statin use represents the difference between the mean of RFFT value for the two groups, statin users and non-users. This calculation can be seen directly from the sample below. However, the coefficient in the multiple regression model is significantly more complicated. It represents the average difference in means for statin users and non-users of the same age.

```
# Calculation of the simple linear regression coefficient
mean(statins.samp$RFFT[statins.samp$Statin == 1]) - mean(statins.samp$RFFT[statins.samp$Statin == 0])

## [1] -10.05342
```

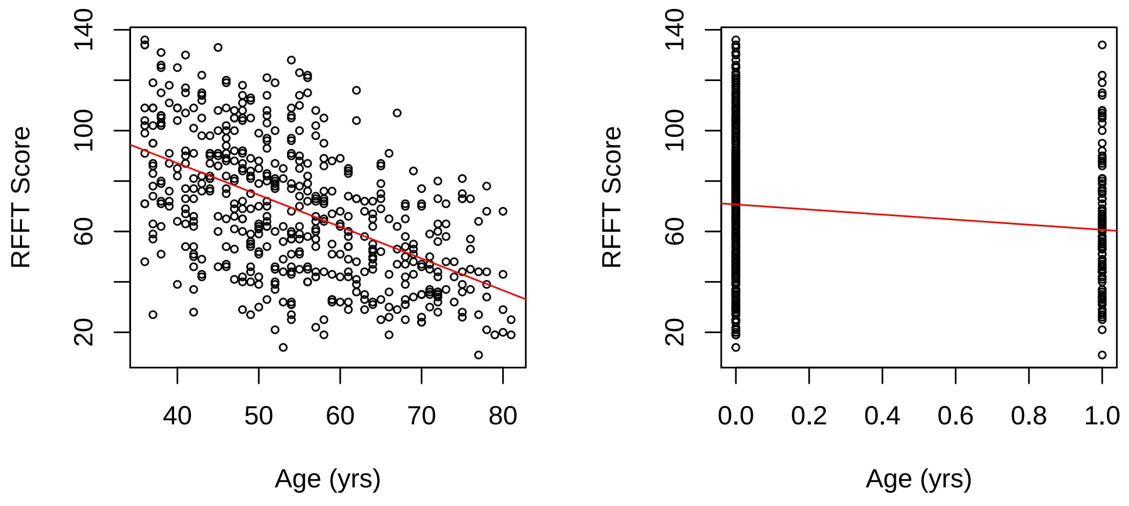
7.3 Evaluating The Fit of a Multiple Regression Model

7.3.1 Assumptions of Linear Regression

The same procedures as outlined in Chapter 6 can be used to test the model assumptions, and will be repeated here.

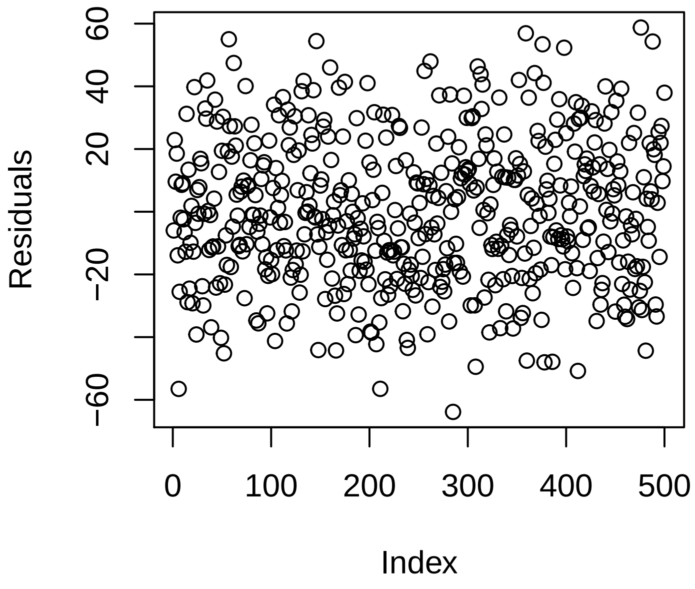
1. **Linearity:** The plots below show that the relationship of the response variable with each explanatory variable is approximately linear.

```
par(mfrow = c(1, 2))
plot(statins.samp$RFFT ~ statins.samp$Age, cex = 0.6, ylab = "RFFT Score", xlab = "Age (yrs)",
    main = "")
abline(lm(statins.samp$RFFT ~ statins.samp$Age), col = "red")
plot(statins.samp$RFFT ~ statins.samp$Statin, cex = 0.6, ylab = "RFFT Score",
    xlab = "Age (yrs)", main = "")
abline(lm(statins.samp$RFFT ~ statins.samp$Statin), col = "red")
```



2. **Constant Variability:** Note that the assumption is approximately constant variance in the residuals, which can be investigated with the following, showing that the residuals do appear to have constant variance.

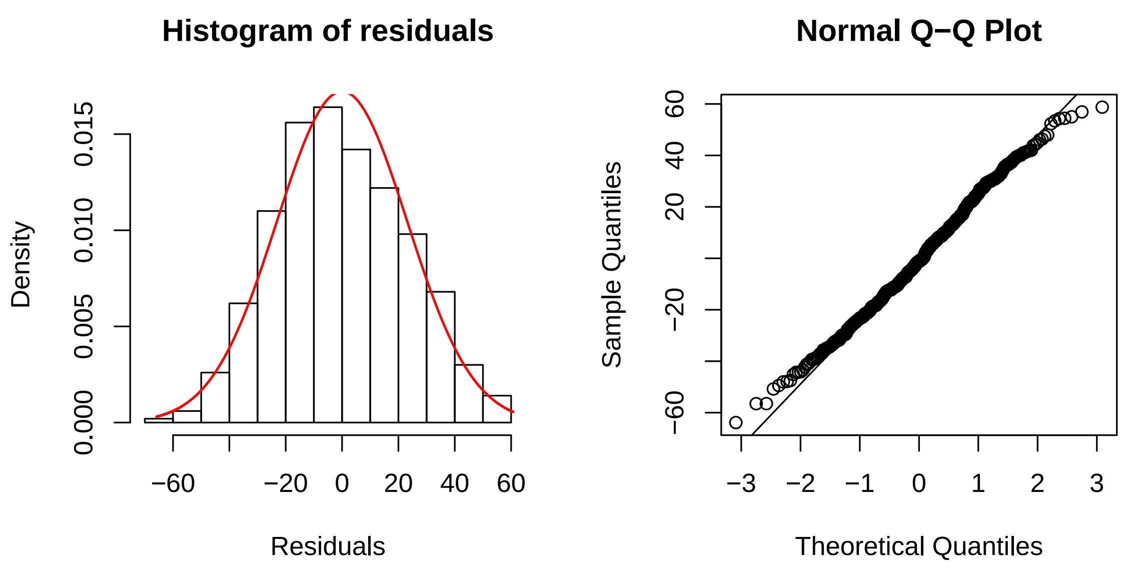
```r
residuals = resid(lm(statins.samp$RFFT ~ statins.samp$Age + statins.samp$Statin))
plot(residuals, ylab = "Residuals")
```



3. **Independent Observations:** This assumption remains dependent on the data collection itself, and based on the parameters of the study, the data is believed to be independent.

4. **Residuals that are approximately normally distributed:** This can be investigated with a historgram or a normal q-q plot as follows. The residuals do appear to be approximately normally distributed.

```r
par(mfrow = c(1, 2))
hist(residuals, xlab = "Residuals", freq = FALSE)
x <- seq(min(residuals) - 2, max(residuals) + 2, 0.01)
y <- dnorm(x, mean(residuals), sd(residuals))
lines(x, y, lwd = 1.5, col = "red")

qqnorm(residuals)
qqline(residuals)
```



7.3.2 Hypothesis Tests and Confidence Intervals

R makes the process of performing hypothesis tests on coefficients of the multiple linear regression model very simple, as it performs all the tests and automatically returns these in the summary of the linear regression. As with simple linear regression, the R output of a multiple regression provides a t-statistic and a corresponding p-value for the hypothesis test of

$$H_0 : \beta_k = 0 \text{vs. } H_A : \beta_k \neq 0$$

Furthermore, the summary output also includes the F-statistic for the hypothesis

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_p = 0$$

In words, this tests if any of the coefficients in combination are useful at predicting the response variable. Note that this tests if any of the coefficients is non-zero, not if all of them are non-zero.

7.5 Categorical Predictors With More Than Two Values

The function `as.factor()` has previously been used to tell R to treat the numerical values of a variable as frivolous, rather representing different categories. In other words, a value of 1 for statin does not imply one unit of statins were used by the individual, rather that any amount of statins were used. Data is often stored with numerical values for ease of manipulation, even if the numerical values do not have significant meaning. In these cases, it is important to utilize the function `as.factor()` to clarify to R whether the numbers have meaning as numbers or as categories, with `as.factor()` implying that they are categories, or factors.

Using this idea on the statin sample data, the following simple regression is obtained, with the reference category being a value of 0 for `Statin` since it is the one that is not printed in the output.

```
summary(lm(statins.samp$RFFT ~ as.factor(statins.samp$Statin)))
```

```
##
## Call:
## lm(formula = statins.samp$RFFT ~ as.factor(statins.samp$Statin))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -56.714 -22.714   0.286  18.299  73.339
##
## Coefficients:
##                                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)                       70.714      1.381  51.212  < 2e-16 ***
## as.factor(statins.samp$Statin)1  -10.053      2.879  -3.492 0.000523 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 27.09 on 498 degrees of freedom
## Multiple R-squared:  0.0239,Adjusted R-squared:  0.02194
## F-statistic: 12.19 on 1 and 498 DF,  p-value: 0.0005226
```

In this example, the regression using `Statin` as either a numerical value or as a categorical variable is the same because of the method by which it was encoded. Using binary values, i.e. 0 and 1, signals to R that a variable is a categorical variable, and this is mathematically equivalent. Where this becomes significantly more complicated is when there are more than two categories because R can no longer make the assumption of categorical variable and will assume non-categorical values with numerical significance.

An example of where this is a problem is as follows,

```
## NOTE, this method is INCORRECT and provided for example,
## so please be careful copying
summary(lm(statins.samp$RFFT ~ statins.samp$Education))

##
## Call:
## lm(formula = statins.samp$RFFT ~ statins.samp$Education)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -56.622 -16.148  -0.885  15.536  62.694
##
## Coefficients:
##                        Estimate Std. Error t value Pr(>|t|)
## (Intercept)              41.148      2.104   19.55   <2e-16 ***
## statins.samp$Education   15.158      1.023   14.81   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 22.85 on 498 degrees of freedom
## Multiple R-squared:  0.3059,Adjusted R-squared:  0.3045
## F-statistic: 219.5 on 1 and 498 DF,  p-value: < 2.2e-16

## Correct method
summary(lm(statins.samp$RFFT ~ as.factor(statins.samp$Education)))
```

```
##
## Call:
## lm(formula = statins.samp$RFFT ~ as.factor(statins.samp$Education))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -55.905 -15.975  -0.905  16.068  63.280
##
## Coefficients:
##                                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)                          40.941      3.203  12.783  < 2e-16
## as.factor(statins.samp$Education)1   14.779      3.686   4.009 7.04e-05
## as.factor(statins.samp$Education)2   32.133      3.763   8.539  < 2e-16
## as.factor(statins.samp$Education)3   44.964      3.684  12.207  < 2e-16
##
## (Intercept)                        ***
## as.factor(statins.samp$Education)1 ***
## as.factor(statins.samp$Education)2 ***
## as.factor(statins.samp$Education)3 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 22.87 on 496 degrees of freedom
## Multiple R-squared:  0.3072,Adjusted R-squared:  0.303
## F-statistic:  73.3 on 3 and 496 DF,  p-value: < 2.2e-16
```

Notice that for an individual with an education level of 2, the first *incorrect* model predicts an RFFT score of 71.464 while the second *correct* model predicts a score of 73.1074. In this example, the difference is not massive but certainly of concern, so use serious caution when working with categorical variables in R.

7.5.1 Using ANOVA For Categorical Variables

When using the ANOVA for categorical variables, it is important to keep in mind that they must again be treated correctly as factors and not as numerical values. An example of the above model treated using an ANOVA is given below.

```
anova(lm(statins.samp$RFFT ~ as.factor(statins.samp$Education)))

## Analysis of Variance Table
##
## Response: statins.samp$RFFT
##                                    Df Sum Sq Mean Sq F value    Pr(>F)
## as.factor(statins.samp$Education)   3 115041   38347  73.304 < 2.2e-16 ***
## Residuals                         496 259469     523
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

7.5.2 Using Dummy Variables

Another method for considering categorical variables is the use of a dummy variable, which is a binary indicator if a value is equal to another. In other words, a model can use several dummy variables, each of which represents a category and is equal to 1 if that individual is in the specified category and 0 otherwise. This can be performed as follow, where the categorical value which is not seprataly specified is the reference variable. Note that this method gives the same results as the linear regression above using `as.factor()`

```
ed.dummy.1 = statins.samp$Education == 1
ed.dummy.2 = statins.samp$Education == 2
ed.dummy.3 = statins.samp$Education == 3
summary(lm(statins.samp$RFFT ~ ed.dummy.1 + ed.dummy.2 + ed.dummy.3))

##
## Call:
## lm(formula = statins.samp$RFFT ~ ed.dummy.1 + ed.dummy.2 + ed.dummy.3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -55.905 -15.975  -0.905  16.068  63.280
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)      40.941      3.203  12.783  < 2e-16 ***
## ed.dummy.1TRUE   14.779      3.686   4.009 7.04e-05 ***
## ed.dummy.2TRUE   32.133      3.763   8.539  < 2e-16 ***
## ed.dummy.3TRUE   44.964      3.684  12.207  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 22.87 on 496 degrees of freedom
## Multiple R-squared:  0.3072,Adjusted R-squared:  0.303
## F-statistic:  73.3 on 3 and 496 DF,  p-value: < 2.2e-16
```

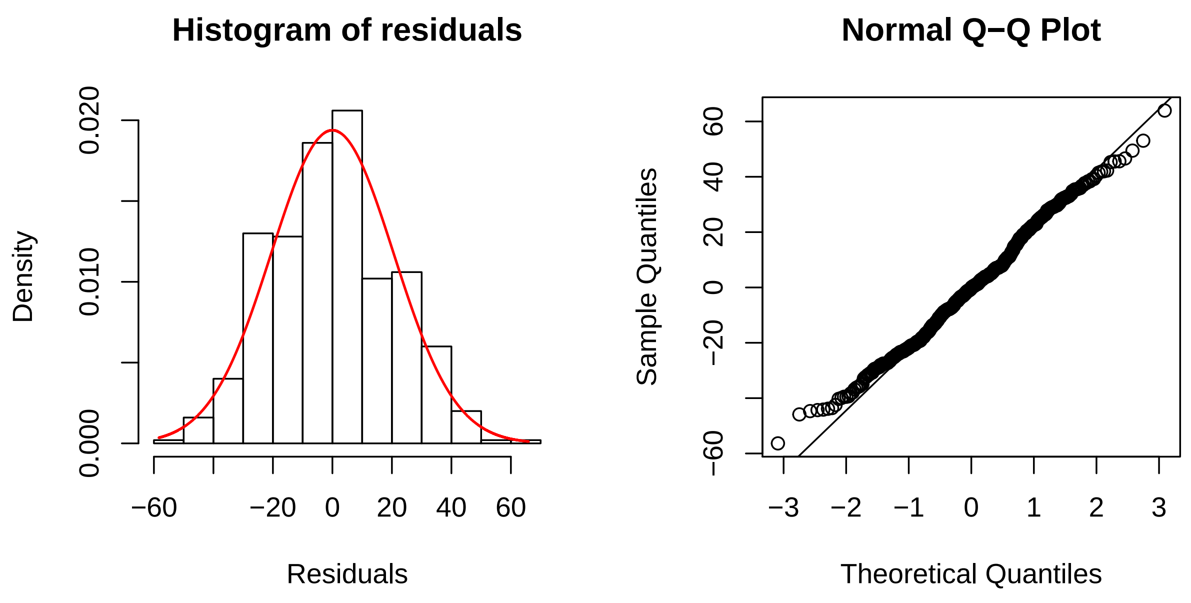
7.6 Analysis of the Statin Dataset

Performing the more complex multiple regression can be done as follows,

```
summary(lm(statins.samp$RFFT ~ as.factor(statins.samp$Statin) + statins.samp$Age + as.factor(statins.samp$Educ

##
## Call:
## lm(formula = statins.samp$RFFT ~ as.factor(statins.samp$Statin) +
##     statins.samp$Age + as.factor(statins.samp$Education) + as.factor(statins.samp$CVD))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -56.348 -15.586  -0.136  13.795  63.935
##
## Coefficients:
##                                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)                          99.03507    6.33012  15.645  < 2e-16
## as.factor(statins.samp$Statin)1       4.69045    2.44802   1.916  0.05594
## statins.samp$Age                     -0.92029    0.09041 -10.179  < 2e-16
## as.factor(statins.samp$Education)1 10.08831    3.37556   2.989  0.00294
## as.factor(statins.samp$Education)2 21.30146    3.57768   5.954 4.98e-09
## as.factor(statins.samp$Education)3 33.12464    3.54710   9.339  < 2e-16
## as.factor(statins.samp$CVD)1        -7.56655    3.65164  -2.072  0.03878
##
## (Intercept)                          ***
## as.factor(statins.samp$Statin)1      .
## statins.samp$Age                     ***
## as.factor(statins.samp$Education)1 **
## as.factor(statins.samp$Education)2 ***
## as.factor(statins.samp$Education)3 ***
## as.factor(statins.samp$CVD)1         *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 20.71 on 493 degrees of freedom
## Multiple R-squared:  0.4355,Adjusted R-squared:  0.4286
## F-statistic: 63.38 on 6 and 493 DF,  p-value: < 2.2e-16
```

Testing the assumptions and evaluating the fit, the model is reasonably good, but by no means perfect. The R^2 value is only 0.4355, which is an improvement, and the plots below show fairly normal residuals. Note that a perfect model would be highly suspcious because of the nature of data analysis, so the goal is to find the best model possible without finding something suspicious.

```
residuals = resid(lm(statins.samp$RFFT ~ as.factor(statins.samp$Statin) + statins.samp$Age +
    as.factor(statins.samp$Education) + as.factor(statins.samp$CVD)))
par(mfrow = c(1, 2))
hist(residuals, xlab = "Residuals", freq = FALSE)
x <- seq(min(residuals) - 2, max(residuals) + 2, 0.01)
y <- dnorm(x, mean(residuals), sd(residuals))
lines(x, y, lwd = 1.5, col = "red")

qqnorm(residuals)
qqline(residuals)
```



7.7 Analysis of the FAMuSS Dataset

Going back to the FAMuSS dataset, a more comprehensive study can now be performed using multiple linear regression to account for various confounders. As a baseline, two different simple linear regressions can be computed as follows, the goal of the following analysis to be to improve upon this simple model.

```
## Simple genotype model - significant
summary(lm(log(famuss$ndrm.ch+5) ~ famuss$actn3.r577x))

##
## Call:
## lm(formula = log(famuss$ndrm.ch + 5) ~ famuss$actn3.r577x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.37691 -0.34090  0.02099  0.39878  1.64371
##
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)           3.81335    0.04722  80.763   <2e-16 ***
## famuss$actn3.r577xCT  0.08420    0.06089   1.383   0.1672
## famuss$actn3.r577xTT  0.17300    0.06801   2.544   0.0112 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.621 on 592 degrees of freedom
## Multiple R-squared:  0.01081,Adjusted R-squared:  0.007471
## F-statistic: 3.236 on 2 and 592 DF,  p-value: 0.04003

## Simple age model - significant
```

```r
summary(lm(log(famuss$ndrm.ch+5) ~ famuss$age))

##
## Call:
## lm(formula = log(famuss$ndrm.ch + 5) ~ famuss$age)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.40718 -0.32241  0.04502  0.39529  1.72844
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.55967    0.10685  42.675  < 2e-16 ***
## famuss$age  -0.02715    0.00426  -6.374 3.69e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6036 on 593 degrees of freedom
## Multiple R-squared:  0.06413,Adjusted R-squared:  0.06255
## F-statistic: 40.63 on 1 and 593 DF,  p-value: 3.691e-10

## Simple bmi model -significant
summary(lm(log(famuss$ndrm.ch+5) ~ famuss$bmi))

##
## Call:
## lm(formula = log(famuss$ndrm.ch + 5) ~ famuss$bmi)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.40374 -0.31402  0.04119  0.40282  1.57997
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.378313   0.137397  31.866  < 2e-16 ***
## famuss$bmi  -0.019721   0.005534  -3.563 0.000396 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6173 on 593 degrees of freedom
## Multiple R-squared:  0.02096,Adjusted R-squared:  0.01931
## F-statistic:  12.7 on 1 and 593 DF,  p-value: 0.0003956

## Simple sex model - significant
summary(lm(log(famuss$ndrm.ch+5) ~ famuss$sex))

##
## Call:
## lm(formula = log(famuss$ndrm.ch + 5) ~ famuss$sex)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.44037 -0.30030  0.07516  0.39285  1.49146
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     4.04981    0.03172 127.685  < 2e-16 ***
## famuss$sexMale -0.37546    0.04973  -7.549 1.66e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5959 on 593 degrees of freedom
## Multiple R-squared:  0.08768,Adjusted R-squared:  0.08615
## F-statistic: 56.99 on 1 and 593 DF,  p-value: 1.657e-13
```

```
## Simple race model - insignificant
summary(lm(log(famuss$ndrm.ch+5) ~ famuss$race))
```

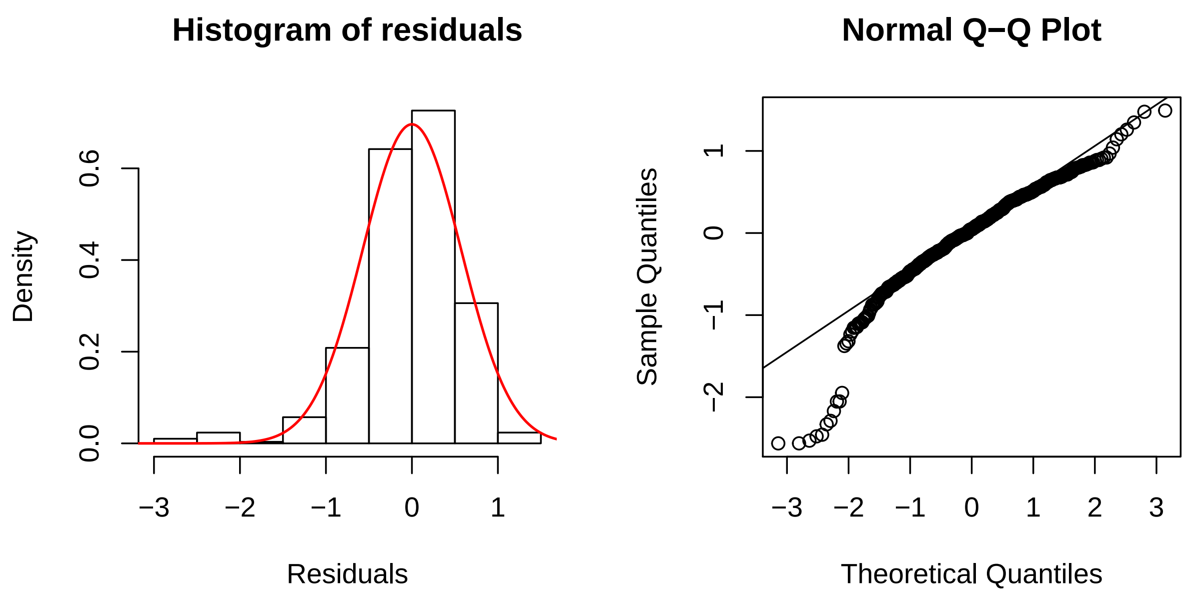
```
##
## Call:
## lm(formula = log(famuss$ndrm.ch + 5) ~ famuss$race)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.31523 -0.29080  0.08267  0.42740  1.61660
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)           3.926241   0.119685  32.805   <2e-16 ***
## famuss$raceAsian     -0.150366   0.146139  -1.029    0.304
## famuss$raceCaucasian -0.001573   0.123096  -0.013    0.990
## famuss$raceHispanic  -0.272195   0.176466  -1.542    0.123
## famuss$raceOther     -0.090221   0.176466  -0.511    0.609
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6219 on 590 degrees of freedom
## Multiple R-squared:  0.0114,Adjusted R-squared:  0.004694
## F-statistic:   1.7 on 4 and 590 DF,  p-value: 0.1483
```

One question that may come up from the following line of code is why the gene actn3.577x, the sex, or the race are not coded as a factor in this regression line, and the simple answer is that the dataset already has these variables coded as a factor. To test if this is the case or not, the function `class()` can be used, which will tell the variable type.

```
class(famuss$actn3.r577x)

## [1] "factor"
```

Now, using the information based on the single regressions, a multiple regression can be built. Both the residuals and the R^2 indicate that this model is far from great. The main text talks through why this inaccuracy could be caused by data collection error and is unlikely to be an indication of fallacy in the methods.

```r
summary(lm(log(famuss$ndrm.ch + 5) ~ famuss$actn3.r577x + famuss$age + famuss$sex +
    famuss$bmi))

## 
## Call:
## lm(formula = log(famuss$ndrm.ch + 5) ~ famuss$actn3.r577x + famuss$age + 
##     famuss$sex + famuss$bmi)
## 
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -2.56248 -0.28173  0.04579  0.39561  1.49186 
## 
## Coefficients:
##                        Estimate Std. Error t value Pr(>|t|)    
## (Intercept)           4.763789   0.156155  30.507  < 2e-16 ***
## famuss$actn3.r577xCT  0.067869   0.056746   1.196   0.2322    
## famuss$actn3.r577xTT  0.139577   0.063286   2.205   0.0278 *  
## famuss$age           -0.023044   0.004169  -5.527 4.90e-08 ***
## famuss$sexMale       -0.348692   0.048436  -7.199 1.86e-12 ***
## famuss$bmi           -0.009430   0.005292  -1.782   0.0753 .  
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.5756 on 589 degrees of freedom
## Multiple R-squared:  0.1546,Adjusted R-squared:  0.1474 
## F-statistic: 21.54 on 5 and 589 DF,  p-value: < 2.2e-16

residuals = resid(lm(log(famuss$ndrm.ch + 5) ~ famuss$actn3.r577x + famuss$age +
    famuss$sex + famuss$bmi))
par(mfrow = c(1, 2))
hist(residuals, xlab = "Residuals", freq = FALSE)
x <- seq(min(residuals) - 2, max(residuals) + 2, 0.01)
y <- dnorm(x, mean(residuals), sd(residuals))
lines(x, y, lwd = 1.5, col = "red")

qqnorm(residuals)
qqline(residuals)
```



7.8 Introduction to Logistic Regression

Section 8.4 in *OI Biostat* introduces logistic regression, a method used to create models with a binary response variable.

Chapter 8

Categorical Data

Contents

8.1 Inference for a Single Proportion

Looking at the example presented in Section 8.1 of *OI Biostat*, a dataset contains 80 cancer patients at the Dana Farber Cancer Institute (DFCI) who survived at least 5 years and 40 patients who did not. R can be used to "create" this dataset and to infer information about the population. The initial statistics calculated are the mean, the standard deviation, and the sample proportion, $\hat{p}$, which can be seen in the code below.

```
x = c(rep(1,80),rep(0,40))
x.bar = mean(x)
std.dev = sd(x)
p.hat = sum(x)/length(x)
p.hat

## [1] 0.6666667
```

8.1.1 Inference Using the Normal Approximation

Following the criteria laid out in *OI Biostat*, the DFCI data can be approximated with a normal distribution. The method for making inferences about the data based on this assumption is laid out as follows.

Confidence Intervals

The same method as discussed in previous chapters can be used to build a confidence interval, following the form

$$\hat{p} \pm z^* \cdot SE_{\hat{p}}$$

An example of this worked out is from *OI Biostat* Example 8.3.

```
## Example 8.3
SE.p.hat = sqrt(p.hat*(1-p.hat)/length(x))
c(p.hat - 1.96*SE.p.hat, p.hat +  1.96*SE.p.hat)

## [1] 0.5823217 0.7510116
```

Hypothesis Testing

Again, the procedures that have been performed before can be used to do a hypothesis test on a proportion. The z-statistic would be

$$z = \frac{\hat{p} - p_0}{SE_{p_0}}$$

Example 8.5 from *OI Biostat* is worked out below. Note that there is a slight difference in the z statistic value because this version does not use an approximation of $\hat{p}$ to 0.67.

```
## Example 8.5
p.0 = 0.6
SE.p.0 = sqrt(p.0*(1-p.0)/length(x))
z = (p.hat - p.0)/SE.p.0
z

## [1] 1.490712

## get the p-value (note that it is a two-sided test)
2*pnorm(z, lower.tail = FALSE)

## [1] 0.1360371
```

8.3 Inference for Two or More Groups

The χ^2 testing method discussed in Section 8.3.2 of *OI Biostat* requires *R* in order to calculate a p-value. The function for doing so is *pchisq()* and like all functions previously discussed that start with ”*p*” such as *pnorm()*, this function takes in a value of the chi-squared distribution and returns a p-value.

```
## Example 8.16
chi_sq = (500-502.6)^2/502.6 + (44425-44422.4)^2/44422.4 + (505-502.4)^2/502.4  + (44405-44407.6)^2/44407.6
pchisq(chi_sq, 1, lower.tail = FALSE)

## [1] 0.8689801
```

The second method for doing this calculation is using the function *chisq.test()*, which takes the contingency table as its input either in matrix or table form. The code below shows how to create a matrix and convert it to a table, but the conversion is not necessary for the chi-squared test.

```
mat = matrix(c(500,44425, 505,44405), ncol = 2, byrow = TRUE)
colnames(mat) = c("Yes", "No")
rownames(mat) = c("Mammogram", "Control")
tbl = as.table(mat)

chisq.test(tbl)
```

```
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  tbl
## X-squared = 0.01748, df = 1, p-value = 0.8948
```

Alternatively, the *chisq.test* function can be used directly on a dataset

```
## Example 8.20
tbl = table(famuss$race ,famuss$actn3.r577x)
chisq.test(tbl)

##
##  Pearson's Chi-squared test
##
## data:  tbl
## X-squared = 19.4, df = 8, p-value = 0.01286
```

8.3.1 Fisher's Exact Test of Independence

The Fisher's Exact Test of Independence can be easily performed in *R* on any matrix or table. The function *fisher.test()* takes in a matrix or table and returns a p-value.

```
## Example 8.24
mat = matrix(c(13,3,4,9,17,12), ncol = 2, byrow = TRUE)
fisher.test(mat, alternative = "two.sided")

##
##  Fisher's Exact Test for Count Data
##
## data:  mat
## p-value = 0.02505
## alternative hypothesis: two.sided
```

8.4 Chi-Square

Using the Chi-Square test for goodness of fit follows the same protocol as above, except some prior data manipulation into a table is required.

```
## Example 8.28
tbl_1 = table(famuss$race)
tbl_2 = as.table(matrix(c(0.128, 0.01, 0.804, 0.058, 1.00), ncol = 5))
colnames(tbl_2) = names(tbl_1)
tbl = rbind(tbl_1, tbl_2)
rownames(tbl) = c("FAMuSS", "US Census")
chisq.test(tbl)

## Warning in chisq.test(tbl): Chi-squared approximation may be incorrect
```

```
##
##  Pearson's Chi-squared test
##
## data:  tbl
## X-squared = 11.112, df = 4, p-value = 0.02534
```

```
## Example 8.29
tbl = as.table(matrix(c(0.25, 0.5, 0.25, 84, 233, 134), ncol = 3))
rownames(tbl) = c("Expected", "Observed")
colnames(tbl) = c("AA", "AB", "BB")
chisq.test(tbl)

## Warning in chisq.test(tbl): Chi-squared approximation may be incorrect

##
##  Pearson's Chi-squared test
##
## data:  tbl
## X-squared = 109.67, df = 2, p-value < 2.2e-16
```