R Tutorial

Updated by

Siqi Liang, Yueyun Zhang

July 2020

Updated by

Hilda Ibriga, Jincheng Bai and Qi Wang

July 2016

Originally created by

Hilda Ibriga, Linna Henry, Patricia Wahyu Haumahu, Qi Wang, Yixuan Qiu and Yuying song

March 2016

Statistical Consulting Service

Purdue University

Preface

The goal of the present document is to give a starting point for people who are new to R. It includes the basic R operations such as package installations, data manipulations, data analyses and graphics with R. If you are not able to find the answer to your questions, there are also many other resources you could turn to. We list some of them below.

Mannuals

There are several manuals are distributed with R in https://cran.r-project.org/doc/manuals/r-release/:

- An Introduction to R [R-intro.pdf],
- R Installation and Administration [R-admin.pdf], • R Data Import/Export [R-data.pdf],
- Writing R Extensions [R-exts.pdf],
- R Language Definition [R-lang.pdf].

The files may be in different formats (pdf, html, texi, . . .) depending on the type of installation.

Online Resources

The CRAN Web site (https://cran.r-project.org) hosts several documents, bibliographic resources, and links to other sites. There are also a list of publications (books and articles) about R or statistical methods and some documents and tutorials written by R users.

Table of Contents

Getting and installing R

R is a free and open source statistical software. You can download R from the following website: https://cran.r-project.org/. R is available for Linux, OS X and Windows, just click download R according to your operating system. For now we just need base distribution and you can choose base and click 'Download R x.x.x for …' to get it.

Install packages

The base R comes with numerous basic functions, like *mean()* and *median()*, and procedures, such as *lm()* for linear regression. More advanced functions and procedures will be in packages. In this example, we will be installing the **ggplot2** package, which can be used to create advanced graphics.

The first step is to enter the following command in the R console window:

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

You should see information appear in the R console window as it is installed. If there are no errors, the package is installed. To load the package for use, enter the following in the R console window:

```
library(ggplot2)

## Warning: package 'ggplot2' was built under R version 3.1.3
```

You will often see these warning messages but there is rarely a compatibility issue. To use the functions and procedures in the package, you will need to load the package each time you open R, although you only need to install the package once. '##' is a prompt that indicates the message from the R console.

Getting Help

R provides extensive documentation. For example, entering *?mean* or *help(mean)* at the prompt gives documentation of the function *mean()* in R. Please give it a try.

```
help(c)
```

If you are not sure about the name of the function you are looking for, you can perform a fuzzy search with the apropos function.

```
apropos("nova")

## [1] "anova"        "manova"        "power.anova.test"  "stat.anova"

## [5] "summary.manova"
```

Finally, there is an R specific Internet search engine at http://www.rseek.org for more assistance.

Basic data structure

Vector

A vector is a contiguous collection of objects of the same type. Common types of vectors include logical, integer, double, and character. Here are some examples of vectors:

```
x <- c(3,2,5,6,55)
x

## [1]  3  2  5  6 55

y <- c("a", "d", "e", "hi")
y

## [1] "a"  "d"  "e"  "hi"
```

Here *x* is a vector of integers and *y* is a vector of characters. You can use the function *c()* to create a vector. The letter *c* stands for concatenate. You must separate the values using a comma. You can put as many items as you want inside a vector. Typing the name of a vector as a command prints the vector elements in order. To access one element of the vector, you can use the following code

```
x[3]

## [1] 5
```

This command asks for the third element of the vector *x*.

It is also possible to have a vector of length one (i.e., a named constant).

```
cool <- 4
cool

## [1] 4
```

Here cool is a vector with length one that contains the value 4.

List

A list is an all-encompassing construct that allows one to wrap together objects of all kinds under the same name. Here is an example, with the second element named.

```
junk <- list(c("black", "brown"), e2=22:28, TRUE)
junk

## [[1]]

## [1] "black" "brown"

## $e2

## [1] 22 23 24 25 26 27 28
## [[3]]
## [1] TRUE
```

Elements of a list can be extracted by position or by name, if the latter is available.

```
junk[[2]]

## [1] 22 23 24 25 26 27 28

junk$e2

## [1] 22 23 24 25 26 27 28
```

The following selection returns a list of one element, but not the element itself.

```
junk[2]

## [[1]]
## [1] 22 23 24 25 26 27 28
```

The elements of a list can themselves be lists.

Matrix

A matrix is a two-dimensional data structure in R. Similar to a vector, matrices must contain objects of the same type.

```
mat <- matrix(NA, nrow = 4, ncol = 5)
mat

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

In the preceding example, mat is a matrix with 4 rows and 5 columns. The values inside *mat* are all set to NA. This is essentially an empty (or missing) matrix. You can access a row of a matrix by the following command.

```
mat[row.number,]
```

Where *row.number* is the row you want to look at. You must include a comma after the row number since the matrix is two dimensional. You can access a column of a matrix by having the comma first:

```
mat[,col.number]
```

We can fill the matrix with whatever objects we like, provided all the objects are of the same type. In the following example, we fill in the rows of the matrix mat one at a time. Each row is basically a vector so we can use the same commands we can use to specify a vector.

```
mat[1,] <- c(2,3,7,5,8)
mat[2,] <- rep(4, 5)
mat[3,] <- 33:37
mat[4,] <- seq(23, 31, 2)
mat

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    3    7    5    8
## [2,]    4    4    4    4    4
## [3,]   33   34   35   36   37
## [4,]   23   25   27   29   31
```

For the second row we used the *rep()* function, which stands for repeat. This function takes two numbers; the first number is the number you want to repeat and the second number is how many times you want this number to be repeated. For the third row, we used the: function. This function takes two numbers, the first number before the colon indicates the start and the second number after the colon indicates the end of a numeric sequence. The function, by default, will create a vector that increases by one from the start to the end. For the last row, we used the *seq()* function, which stands for sequence. This function takes 3 arguments. The first argument is the start value, the second argument is the ending value, and the last argument is how much you want to increment by. The *seq()* function will create a vector that starts at the start, ends at the end, and increases by the increment value. You will notice that all four rows are assigned vectors of length 5. This is because we have 5 columns.

You can set the column and row names of a matrix using the functions *colnames()* and *rownames()* respectively. Here is an example:

```
rownames(mat) <- c("r1", "r2", "r3", "r4")
rownames(mat)

## [1] "r1" "r2" "r3" "r4"
```

To look at the contents of one row of a matrix, we state the row number followed by a comma:

```
mat[2,]

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

This shows us the second row. We can also look at a column putting the comma first:

```
mat[,3]

## [1]  7  4 35 27
```

This shows us the third column. You can access one element of a matrix by specifying the element's row and column number:

```
mat[4,5]

## [1] 31
```

This will show us the element in the fourth row and the fifth column.

Data frame

A data frame is a restricted form of list with matrix like features. Elements of a data frame are like columns of a matrix, vectors of the same length, but they usually have names and can be of a mixture of types. Data sets for statistical modeling are typically formulated as data frames. Here is an example of a data frame:

```
df <- data.frame(haircolor = c("red", "black", "blonde"), eyecolor =
c("green", "brown", "blue"), age = c(22, 23, 25), phd = c(T, T, F))
df

##   haircolor eyecolor age   phd
## 1       red    green  22  TRUE
## 2     black    brown  23  TRUE
## 3    blonde     blue  25 FALSE
```

This data frame has three rows and 4 columns. The 4 columns represent different variables. You can access one column of the data frame by the following:

```
df$age

## [1] 22 23 25
```

where df is the name of the data frame and age is the column you want to look at. You can also use the matrix operators to access rows, columns, and elements of a data frame. Here is an example:

```
df[3,]

##   haircolor eyecolor age   phd
## 3    blonde     blue  25 FALSE

df[,2]

## [1] green brown blue
## Levels: blue brown green

df[3,3]

## [1] 25
```

Array

An array is a data structure in R that has 2 or more dimensions. Here is an example of an array

```
my_arr <- array(1:8, c(2,2,2))
my_arr

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

Here we have created an array called my_arr using the function *array()*. The first argument in *array()* is a vector of the numeric values we want in the array. The second argument are the dimensions of my array presented as a vector. You can see in the output, the array is filled with the values 1 to 8, and there are two 2x2 matrices. The first 2 specifies the number of rows, the second 2 specifies the number of columns, and the third 2 specifies the number of matrices. You can access elements of the array with the following code:

```
my_arr[,,1]

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

This returns the first 2x2 matrix.

```
my_arr[1,,1]

## [1] 1 3
```

This returns the first row in the first matrix.

```
my_arr[,1,1]

## [1] 1 2
```

This returns the first column in the first matrix.

Reading and Exporting data

Reading in free formatted data using the built-in module in R Studio

R Studio is a popular IDE for R. It is free and available at https://rstudio.com for both Windows and OS X. Probably the easiest way to import data is using the built-in module in R Studio. It can take care of all the details for you. If you are new to R, it is highly recommended to do that. This tutorial is created based on Version 1.3.959 of R Studio.

In this example, we will import a dataset with the name 'Cars.csv'. The dataset is available on our website:

http://www.stat.purdue.edu/scs/help/Intro_stat_software.html

and you can download it and put it in your working directory.

1. Click *Import Dataset* in the environment panel. The importers are grouped into 3 categories: Text data, Excel data and statistical data.

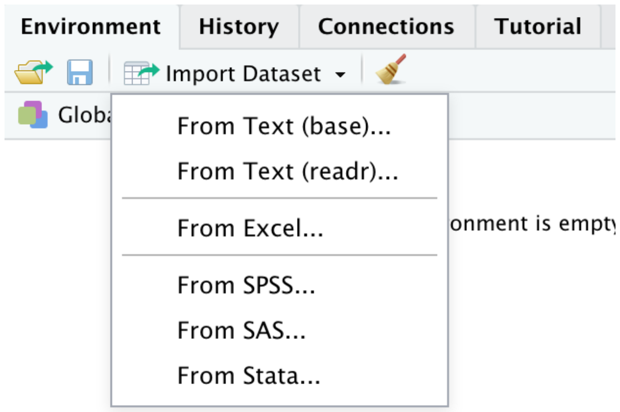
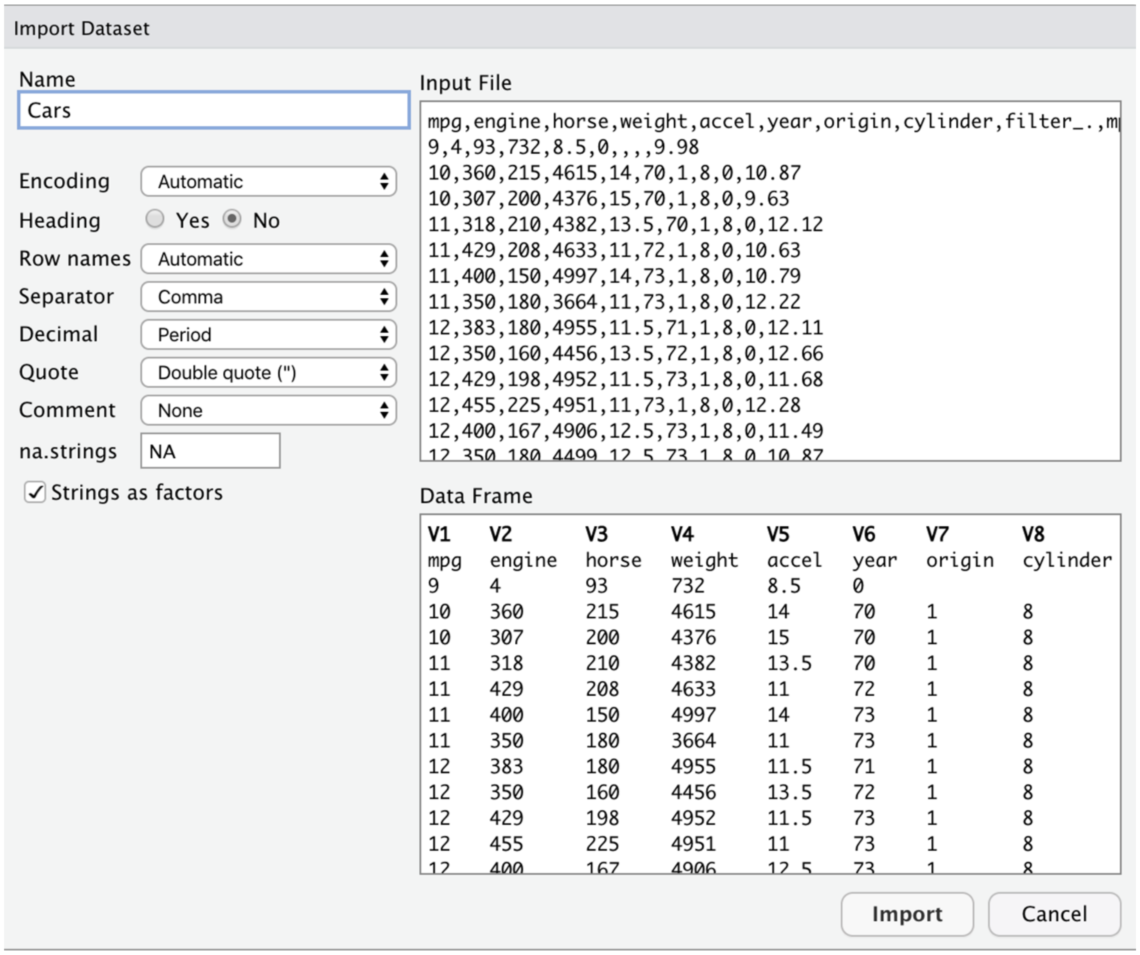


Figure 1

2. Locate the file and a pop-up window will appear.



Name: Cars

Encoding: Automatic
Heading: Yes [✓ No]
Row names: Automatic
Separator: Comma
Decimal: Period
Quote: Double quote (")
Comment: None
na.strings: NA
✓ Strings as factors

Input File

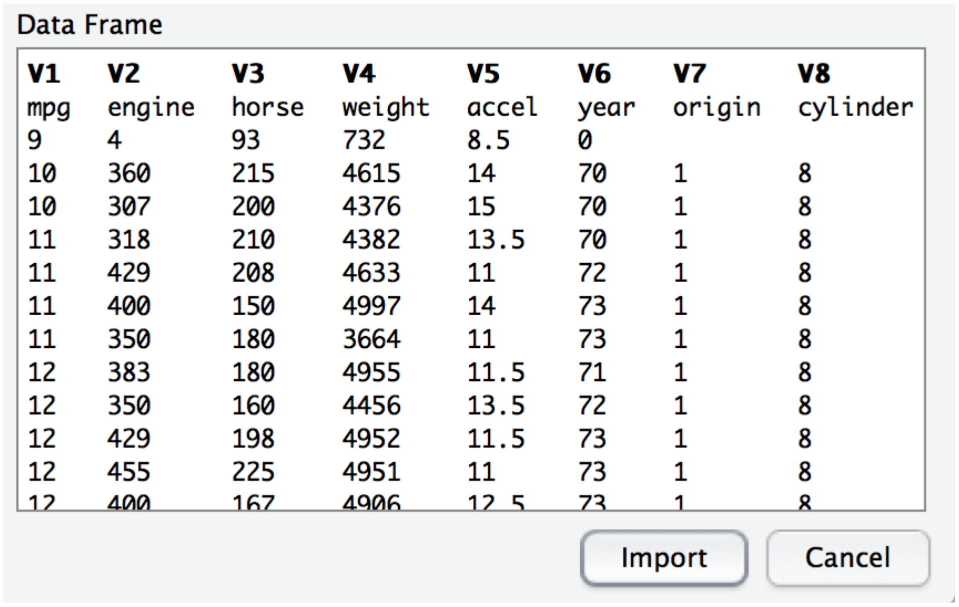
```
mpg,engine,horse,weight,accel,year,origin,cylinder,filter_.,m
9,4,93,732,8.5,0,,,,9.98
10,360,215,4615,14,70,1,8,0,10.87
10,307,200,4376,15,70,1,8,0,9.63
11,318,210,4382,13.5,70,1,8,0,12.12
11,429,208,4633,11,72,1,8,0,10.63
11,400,150,4997,14,73,1,8,0,10.79
11,350,180,3664,11,73,1,8,0,12.22
12,383,180,4955,11.5,71,1,8,0,12.11
12,350,160,4456,13.5,72,1,8,0,12.66
12,429,198,4952,11.5,73,1,8,0,11.68
12,455,225,4951,11,73,1,8,0,12.28
12,400,167,4906,12.5,73,1,8,0,11.49
12,350,180,4499,12.5,73,1,8,0,10.87
```

Data Frame

V1	V2	V3	V4	V5	V6	V7	V8
mpg	engine	horse	weight	accel	year	origin	cylinder
9	4	93	732	8.5	0		
10	360	215	4615	14	70	1	8
10	307	200	4376	15	70	1	8
11	318	210	4382	13.5	70	1	8
11	429	208	4633	11	72	1	8
11	400	150	4997	14	73	1	8
11	350	180	3664	11	73	1	8
12	383	180	4955	11.5	71	1	8
12	350	160	4456	13.5	72	1	8
12	429	198	4952	11.5	73	1	8
12	455	225	4951	11	73	1	8
12	400	167	4906	12.5	73	1	8

Figure 2

3. Specify the name of the data frame of the file imported in the box under '*Name*', the default setting is the file name without its extended name. Click to choose whether there is a headings row in the data file or not. The content of the data frame will show up in the right lower part of the window. Be careful when it looks like this:

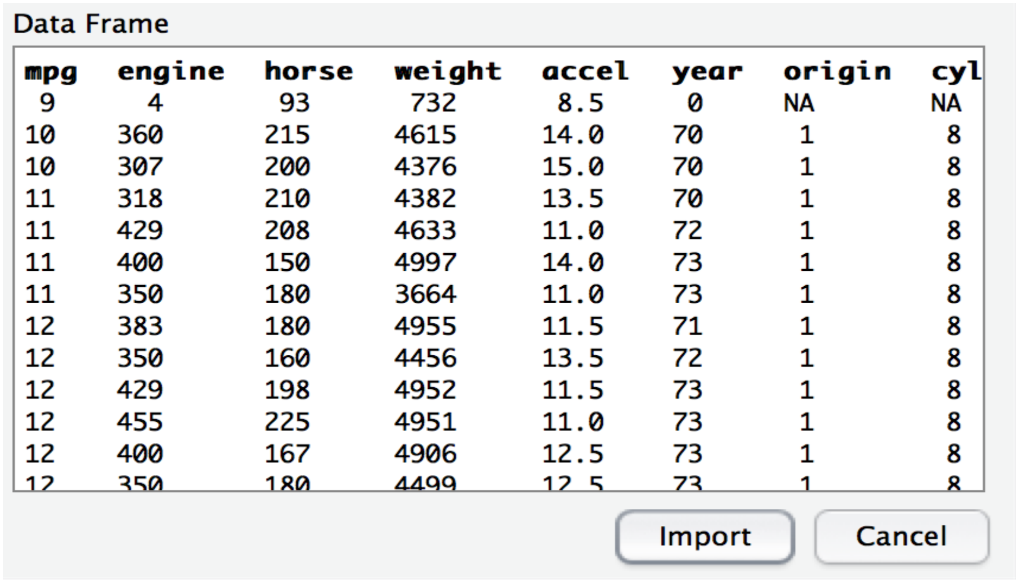

Data Frame

V1	V2	V3	V4	V5	V6	V7	V8
mpg	engine	horse	weight	accel	year	origin	cylinder
9	4	93	732	8.5	0		
10	360	215	4615	14	70	1	8
10	307	200	4376	15	70	1	8
11	318	210	4382	13.5	70	1	8
11	429	208	4633	11	72	1	8
11	400	150	4997	14	73	1	8
11	350	180	3664	11	73	1	8
12	383	180	4955	11.5	71	1	8
12	350	160	4456	13.5	72	1	8
12	429	198	4952	11.5	73	1	8
12	455	225	4951	11	73	1	8
12	400	167	4906	12.5	73	1	8

Figure 3

This suggests there was a header row but the 'Yes' was not selected.

Change the Heading to be 'Yes', the file will load properly. Also it will take care of the missing values which are coded as 'NA' in this document.


Data Frame

mpg	engine	horse	weight	accel	year	origin	cyl
9	4	93	732	8.5	0	NA	NA
10	360	215	4615	14.0	70	1	8
10	307	200	4376	15.0	70	1	8
11	318	210	4382	13.5	70	1	8
11	429	208	4633	11.0	72	1	8
11	400	150	4997	14.0	73	1	8
11	350	180	3664	11.0	73	1	8
12	383	180	4955	11.5	71	1	8
12	350	160	4456	13.5	72	1	8
12	429	198	4952	11.5	73	1	8
12	455	225	4951	11.0	73	1	8
12	400	167	4906	12.5	73	1	8
12	350	180	4499	12.5	73	1	8

Figure 4

Reading in free formatted data from an ASCII file using the read.table function

The *read.table()* function can read in any type of delimited ASCII file. It works pretty much the same as import dataset procedures in R studio. Actually, what this R module does is translate your specifications into function calls. So, *read.table()* is doing that directly.

Here is an example for reading in the Car.csv file we just loaded by using '*Import Dataset*' module.

```
cars <- read.table(file = "Cars.csv", header = TRUE, sep = ",")
```

The data will be stored as a data frame and be assigned to the variable 'cars' on the left hand side of '<-". Three most important argument of *read.table()* are:

- file: Specify the file location, if no specific path is specified, it will look for files in the current working directory. So we can either specify the full path or just the filename if it is in the working directory.
- header: Whether the file's first row is a header row or not, default value is FALSE
- sep: Specifies the separator, default value is ' '.

Exporting files using write.table function

The *write.table()* function outputs data frame. Suppose we made some changes on '*cars*' that we just read in and would like to save somewhere then *write.table()* function can be used . The arguments it takes are similar to *read.table()*. Here is an example:

```
write.table(cars, file = "Cars2.csv", sep = "\t", row.names = FALSE,
            col.names = TRUE)
```

- The first argument is to specify which data frame to be exported.
- file: the path of file to be created.
- sep: separator, the default separator is a blank space, but any separator can be specified in the sep option. In the example we used a *Tab* separator.
- row.names and col.names: whether those names will appear in the output file. The default values are both TRUE.

Basic Statistics

The following commands are commonly used to explore and describe a data set.

Here is a print out the data set for reference:

```
Cars
## 	mpg engine horse weight accel year origin cylinder filter_.  mpg1
## 1    9.0    4.0   93    732   8.5    0     NA      NA      NA  9.98
## 2   10.0  360.0  215   4615  14.0   70     1       8       0 10.87
## 3   10.0  307.0  200   4376  15.0   70     1       8       0  9.63
## 4   11.0  318.0  210   4382  13.5   70     1       8       0 12.12
## 5   11.0  429.0  208   4633  11.0   72     1       8       0 10.63
## .
## .
## .
## 398 46.6   86.0   65   2110  17.9   80     3       4       1 47.93
## 399   NA  133.0  115   3090  17.5   70     2       4       1    NA
## 400   NA  350.0  165   4142  11.5   70     1       8       0    NA
## 401   NA  351.0  153   4034  11.0   70     1       8       0    NA
## 402   NA  383.0  175   4166  10.5   70     1       8       0    NA
## 403   NA  360.0  175   3850  11.0   70     1       8       0    NA
## 404   NA  302.0  140   3353   8.0   70     1       8       0    NA
## 405   NA   97.0   48   1978  20.0   71     2       4       1    NA
## 406   NA  121.0  110   2800  15.4   81     2       4       1    NA
```

There are some missing values, so we need to remove these missing values first.

Removing missing values:

```
cars=cars[-which(is.na(cars),arr.ind=T),]
```

Summary of Statistics

```
#summary(dataset/variable)
summary(cars)

##       mpg            engine           horse            weight
##  Min.   :10.00   Min.   : 68.0   Min.   : 46.0   Min.   :1613
##  1st Qu.:17.50   1st Qu.:105.0   1st Qu.: 75.0   1st Qu.:2220
##  Median :23.00   Median :151.0   Median : 92.0   Median :2790
##  Mean   :23.64   Mean   :193.6   Mean   :103.4   Mean   :2958
##  3rd Qu.:29.00   3rd Qu.:261.0   3rd Qu.:123.5   3rd Qu.:3590
##  Max.   :46.60   Max.   :455.0   Max.   :230.0   Max.   :5140
##      accel            year           origin         cylinder
##  Min.   : 8.00   Min.   :70.00   Min.   :1.000   Min.   :4.000
##  1st Qu.:14.00   1st Qu.:73.00   1st Qu.:1.000   1st Qu.:4.000
##  Median :15.50   Median :76.00   Median :1.000   Median :4.000
##  Mean   :15.57   Mean   :76.03   Mean   :1.567   Mean   :5.462
##  3rd Qu.:17.05   3rd Qu.:79.00   3rd Qu.:2.000   3rd Qu.:8.000
##  Max.   :24.80   Max.   :82.00   Max.   :3.000   Max.   :8.000
##     filter_.            mpg1
##  Min.   :0.0000   Min.   :10.63
##  1st Qu.:0.0000   1st Qu.:17.16
##  Median :1.0000   Median :22.71
##  Mean   :0.7441   Mean   :23.58
##  3rd Qu.:1.0000   3rd Qu.:29.34
##  Max.   :1.0000   Max.   :47.93
```

Data Structure

```
#str(dataset/variable)
str(cars)

## 'data.frame':    379 obs. of  10 variables:
##  $ mpg     : num  10 11 11 11 12 12 12 13 13 13 ...
##  $ engine  : num  360 318 429 400 455 400 350 400 400 350 ...
##  $ horse   : int  215 210 208 150 225 167 180 170 175 165 ...
##  $ weight  : int  4615 4382 4633 4997 4951 4906 4499 4746 5140 ...
##  $ accel   : num  14 13.5 11 14 11 12.5 12.5 12 12 12 ...
##  $ year    : int  70 70 72 73 73 73 73 71 71 72 ...
##  $ origin  : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ cylinder: int  8 8 8 8 8 8 8 8 8 8 ...
##  $ filter_.: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ mpg1    : num  10.9 12.1 10.6 10.8 12.3 ...
```

Mean and Variance

```
#mean(variable,na.rm(removing NA)=True)
mean(cars$mpg)

## [1] 23.64248

#var(variable)
var(cars$mpg)

## [1] 59.8734
```

Number of Observation in the Variable

```
#length(variable)
length(cars$mpg)

## [1] 379
```

Median

```
#median(variable,na.rm=T)
median(cars$mpg)

## [1] 23
```

Quantile

```
#quantile(variable,level,na.rm=T), the argument na.rm==T means any NA
and NaN's are removed from x before the quantiles are computed.
quantile(cars$mpg,0.25)
```

```
##  25%
## 17.5
```

Hypothesis Testing

T-test

The default options of t-test in R are *"x", "y", "alternative", "mu", "paired", "var.equal"* and *"conf.level"*, and their default values are as follows:

```
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
```

One Sample t-test

```
#t.test(variable, mean under null hypothesis)
t.test(cars$mpg,mu=25)

##
##  One Sample t-test
##
## data:  cars$mpg
## t = -3.4155, df = 378, p-value = 0.0007058
## alternative hypothesis: true mean is not equal to 25
## 95 percent confidence interval:
##  22.86096 24.42400
## sample estimates:
## mean of x
##  23.64248
```

Independent Sample t-test (Unequal Variance)

```
#t.test(variable1,variable2)
t.test(cars$mpg,cars$mpg1)

##
##  Welch Two Sample t-test
##
## data:  cars$mpg and cars$mpg1
## t = 0.10196, df = 755.88, p-value = 0.9188
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.052830  1.168186
## sample estimates:
```

```
## mean of x mean of y
##  23.64248  23.58480
```

Independent Sample t-test (Equal Variance)

```
#t.test(variable1,variable2,var.equal=True)
t.test(cars$mpg,cars$mpg1,var.equal=T)

##
##  Two Sample t-test
##
## data:  cars$mpg and cars$mpg1
## t = 0.10196, df = 756, p-value = 0.9188
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.052830  1.168186
## sample estimates:
## mean of x mean of y
##  23.64248  23.58480
```

Paired t-test

```
#t.test(variable1,variable2,paired=T)
t.test(cars$mpg,cars$mpg1,paired=T)

##
##  Paired t-test
##
## data:  cars$mpg and cars$mpg1
## t = 1.0662, df = 378, p-value = 0.287
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.0486872  0.1640434
## sample estimates:
## mean of the differences
##               0.0576781
```

Chi-Squared Tests

We are going to use a made-up dataset in the following example as Cars.csv does not lend itself to a Chi-Squared test. The data set will be a two-way contingency table, where the two factors are Degree (levels are High and Low) and clinic (5 levels from Worse to Marked Improvement), and the response variable is y.

Data input:

```
y1<-c(1,13,16,15,7)
y2<-c(11,53,42,27,11)
```

```
y<-cbind(y1,y2)
dimnames(y)<-list(clinic=c("Worse","Stationary","Slight Improvement",
                          "Moderate Improvement","Marked Improvement"),
                  Degree=(c("High","Low")))


y

##                         Degree
## clinic                    High Low
##   Worse                      1  11
##   Stationary                13  53
##   Slight Improvement        16  42
##   Moderate Improvement      15  27
##   Marked Improvement         7  11
```

Performing the Chi-squared test:

```
chi.test<-chisq.test(y)

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

chi.test

##
##  Pearson's Chi-squared test
##
## data:  y
## X-squared = 6.8807, df = 4, p-value = 0.1423
```

To see the other data produced by chisq.test:

```
names(chi.test)

## [1] "statistic" "parameter" "p.value"   "method"    "data.name"
"observed"
## [7] "expected"  "residuals" "stdres"
```

To get the expected values, for example:

```
chi.test$expected

##                         Degree
## clinic                        High       Low
##   Worse                   3.183673  8.816327
##   Stationary             17.510204 48.489796
##   Slight Improvement     15.387755 42.612245
##   Moderate Improvement 11.142857 30.857143
##   Marked Improvement    4.775510 13.224490
```

Correlation

By default, the correlation function in R is as follows:

```
cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))
#by default it uses Pearson method

#cor(variable1,variable2)
cor(cars$mpg,cars$engine)

## [1] -0.812826

cor(cars$mpg,cars$engine,method="spearman")

## [1] -0.8729739
```

Correlation Significance Test

```
#cor.test(variable1,variable2)
cor.test(cars$mpg,cars$engine)

##
##  Pearson's product-moment correlation
##
## data:  cars$mpg and cars$engine
## t = -27.094, df = 377, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.8444198 -0.7755971
## sample estimates:
##       cor
## -0.812826

cor.test(cars$mpg,cars$engine,method="spearman")

##  Spearman's rank correlation rho
##
## data:  cars$mpg and cars$engine
## S = 16994000, p-value < 2.2e-16
## alternative hypothesis: true rho is not equal to 0
## sample estimates:
##        rho
## -0.8729739

##
```

```
## Warning in cor.test.default(cars$mpg, cars$engine, method =
"spearman"):
## Cannot compute exact p-value with ties
```

Plotting a Scatterplot

```
plot(cars$mpg,cars$engine,xlab="Miles Per Gallon", ylab="Engine
Displacement",
     main="Scatterplot between Miles Per Gallon & Engine Displacement")
```

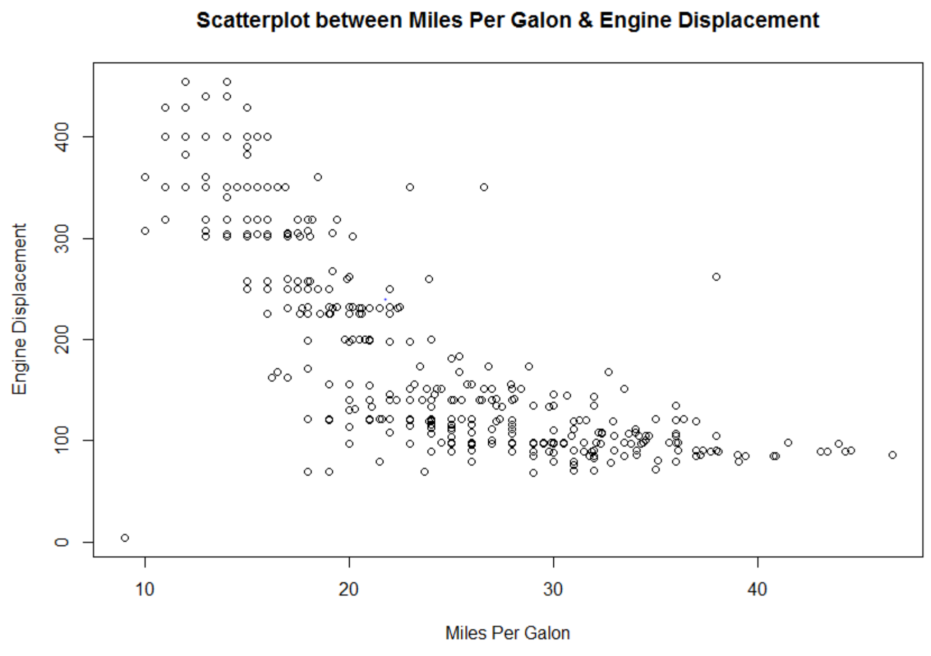


Figure 5

One-Way ANOVA

We will introduce how to perform One-Way ANOVA (analysis of variance) in R based on the dataset **Cars.csv**. The response (dependent) variable is *mpg* and the factor (independent) variable is *origin*. The One-Way ANOVA can be carried out by applying the R function *aov()*.

We import the dataset and name it like *cars* and change the class of the variable *origin* to be a factor (categorical) variable. There are three levels of *origin* and they are 1, 2 and 3, respectively.

```
# Import a csv file using read.csv()

cars <- read.csv("Cars.csv")

cars$origin <- as.factor(cars$origin)

levels(cars$origin)

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

The first step in our analysis is to graphically compare *mpgs* distributions across different *origins*.

```
plot(mpg ~ origin, data=cars, pch=16, cex=0.4)
```

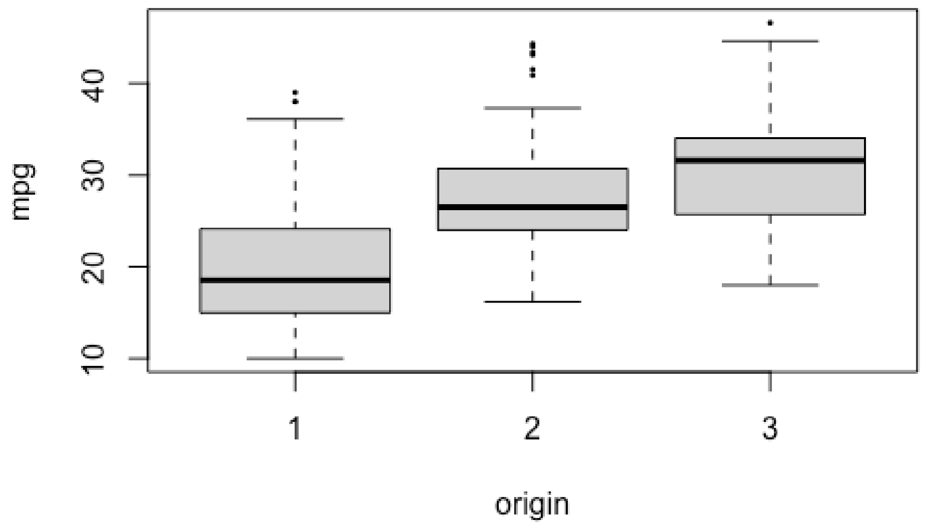


Figure 6

The boxplot shows that the median of *mpg* from origin 1 is smaller than origin 2 and 3.

The R function *aov()* can be used for fitting ANOVA models.

```
results = aov(mpg ~ origin, data=cars)
```

Modeling procedures, such as the *aov()* here and the *lm()* to be discussed later, need a model formula of form say y~x1*x2+x3 specifying the models to be fitted, and the data frame to be worked on can be passed in via the *data=...* argument. In the above fit, the response is `cars$mpg` and the covariate is `cars$origin`, but within the local environment they are simply referred to as `mpg` and `origin`. The fitted model assigned to `results` is a list incomprehensible to the user, but one may extract various results using numerous functions (known as *methods* in R) such as the following.

```
summary(results)
##              Df Sum Sq Mean Sq F value Pr(>F)
## origin        2   7985    3992   97.97 <2e-16 ***
## Residuals   394  16056      41
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 9 observations deleted due to missingness
```

The output of *summary()* shows that the F value is 97.97 with a p-value smaller than 0.05. We reject the null hypothesis that the *mpg* means of all three origins are equal.

Then we will perform the multiple comparisons to see how *mpg* differs in three different *origins* by using R function *pairwise.t.test()*. This function *pairwise.t.test* carries out the pairwise comparisons between group means with corrections for multiple testing.

```
pairwise.t.test(cars$mpg, cars$origin, p.adjust="bonferroni")

## 
##  Pairwise comparisons using t tests with pooled SD 
## 
## data:  cars$mpg and cars$origin 
## 
##   1      2    
## 2 <2e-16 -    
## 3 <2e-16 0.045
## 
## P value adjustment method: bonferroni
```

This result indicates that the *mpg* means of the three origins are significantly different from each other.

Another option for multiple comparisons is Tukey's method by using the R function *TukeyHSD()*. This function creates a set of confidence intervals for the differences between means.

```
results = aov(mpg ~ origin, data=cars)
TukeyHSD(results,conf.level=0.95)

##   Tukey multiple comparisons of means
##     95% family-wise confidence level
## 
## Fit: aov(formula = mpg ~ origin, data = cars)
## 
## $origin
##          diff        lwr       upr     p adj
## 2-1  7.763203 5.73054715  9.795858 0.0000000
## 3-1 10.322407 8.38214860 12.262666 0.0000000
## 3-2  2.559204 0.09398439  5.024424 0.0397888
```

The results also show that all the three differences are significantly different from 0.

Besides, we need to perform the model diagnostics, and we can use the R function *plot()*.

```
plot(results, pch=16, cex=0.4)
```

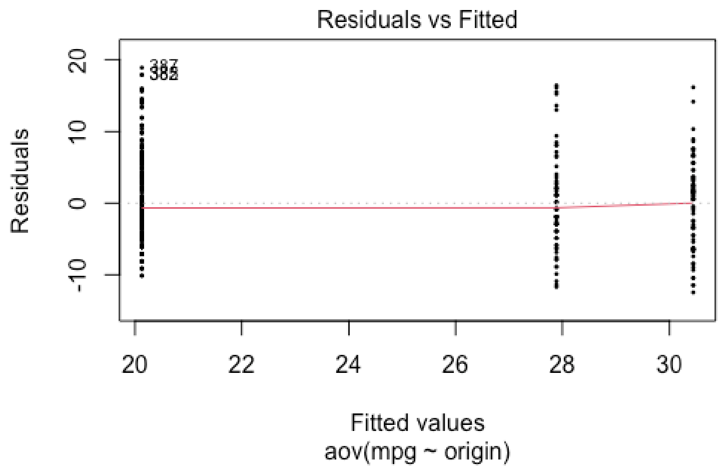


Figure 7

The red line connects the medians of different groups.

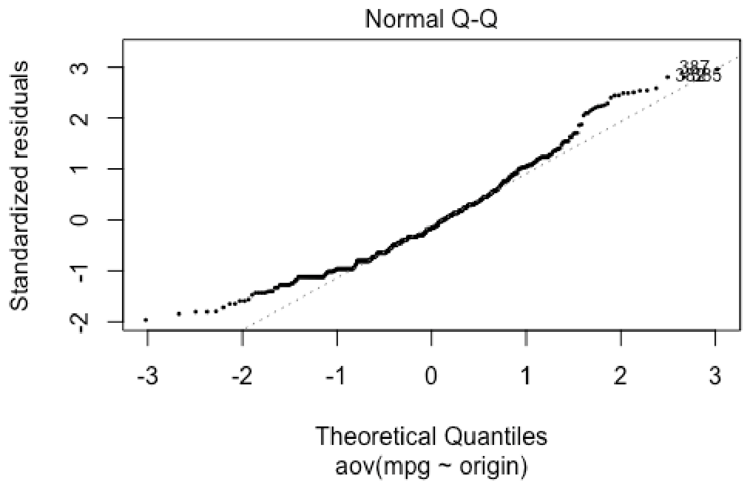


Figure 8

Most of the standardized residuals distribute closely to the straight line. However, we can also observe that a few of them drift away from the line in the beginning and end, which suggests that the distribution of residuals is a little skewed.

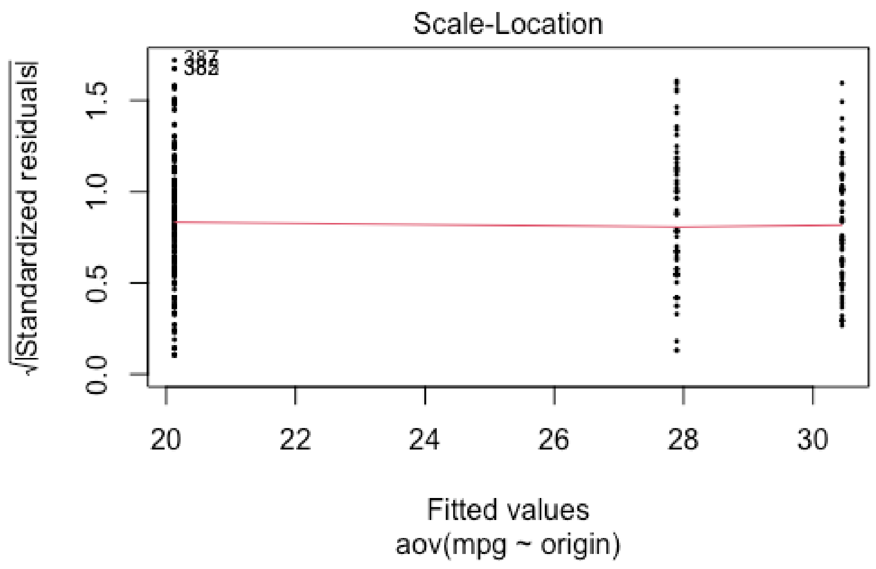


Figure 9

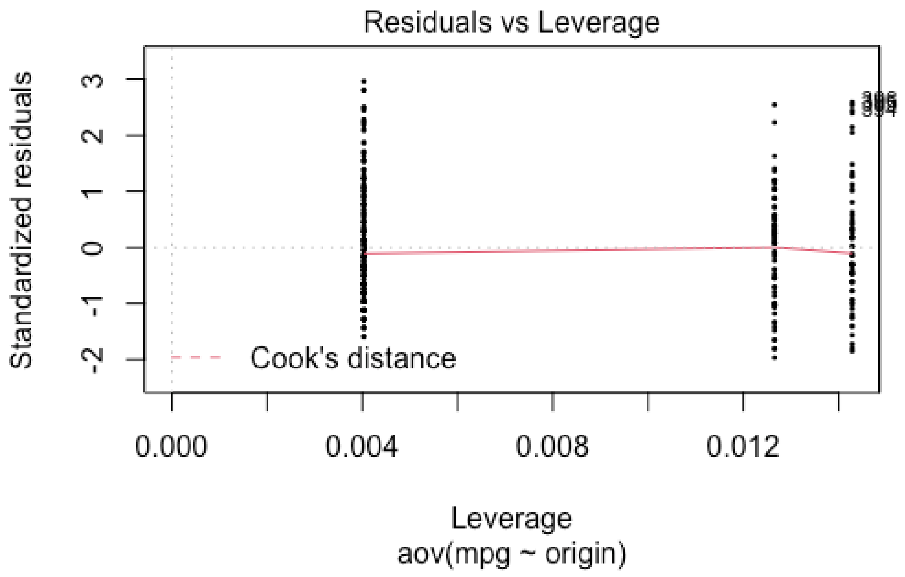


Figure 10

Regression

As before, we remove the missing values first:

```
cars<-cars[-which(is.na(cars),arr.ind=T),]
```

To get the names of variables in the dataset, we can use the R function *names()*:

```
names(cars)

##  [1] "mpg"      "engine"   "horse"    "weight"   "accel"    "year"
##  [7] "origin"   "cylinder" "filter_." "mpg1"
```

Linear Regression

Fit a linear regression model with all variables in the dataset without considering interactions.

```
mod.reg<-lm(mpg~.,data=cars)
summary(mod.reg)

##
## Call:
## lm(formula = mpg ~ ., data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3196 -0.6958  0.0189  0.6254  3.0243
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.5673545  1.5809687  -0.359  0.71990
## engine       0.0025929  0.0024279   1.068  0.28622
## horse        0.0004493  0.0046635   0.096  0.92330
## weight      -0.0006067  0.0002320  -2.615  0.00930 **
## accel        0.0159285  0.0316724   0.503  0.61533
## year         0.0662729  0.0200110   3.312  0.00102 **
## origin       0.0964715  0.0902488   1.069  0.28579
## cylinder    -0.2033509  0.1193718  -1.704  0.08931 .
## filter_.    -0.4577229  0.2929970  -1.562  0.11910
## mpg1         0.9102985  0.0167311  54.407  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.023 on 369 degrees of freedom
## Multiple R-squared:  0.9829, Adjusted R-squared:  0.9825
## F-statistic:  2362 on 9 and 369 DF,  p-value: < 2.2e-16
```

Fit a linear regression model with only several variables in the dataset.

```
mod1<-lm(mpg~engine+horse+weight,data=cars)
summary(mod1)
##
## Call:
## lm(formula = mpg ~ engine + horse + weight, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.4431 -2.7307 -0.3825  2.2516 16.1406
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 44.7774973  1.2369586  36.200  < 2e-16 ***
## engine      -0.0098511  0.0069098  -1.426   0.1548
## horse       -0.0278049  0.0133328  -2.085   0.0377 *
## weight      -0.0055278  0.0007289  -7.584 2.67e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.169 on 375 degrees of freedom
## Multiple R-squared:  0.7121, Adjusted R-squared:  0.7098
## F-statistic: 309.1 on 3 and 375 DF,  p-value: < 2.2e-16
```

Fit a linear regression model with two-way interaction.

```
#engine*horse in R indicates the main effects of engine and horse as
well as their interaction effect.

mod2<-lm(mpg~engine*horse+weight,data=cars)
summary(mod2)

##
## Call:
## lm(formula = mpg ~ engine * horse + weight, data = cars)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.936 -2.216 -0.238  1.871 16.781
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)   5.373e+01  1.568e+00  34.278  < 2e-16 ***
## engine       -7.268e-02  9.879e-03  -7.357 1.20e-12 ***
## horse        -1.796e-01  2.201e-02  -8.161 5.11e-15 ***
## weight       -2.895e-03  7.417e-04  -3.903 0.000113 ***
## engine:horse  4.724e-04  5.686e-05   8.308 1.81e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 3.835 on 374 degrees of freedom
## Multiple R-squared:  0.7569, Adjusted R-squared:  0.7543
## F-statistic: 291.2 on 4 and 374 DF,  p-value: < 2.2e-16
```

Fit a linear regression model with higher order interaction.

```
mod3<-lm(mpg~(engine+horse+weight)^3,data=cars)
summary(mod3)
```

```
##
## Call:
## lm(formula = mpg ~ (engine + horse + weight)^3, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.5411 -2.2346 -0.4175  1.7745 17.3233
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)          6.163e+01  5.560e+00  11.085  < 2e-16 ***
## engine              -1.208e-01  3.819e-02  -3.163  0.00169 **
## horse               -1.871e-01  7.713e-02  -2.426  0.01573 *
## weight              -6.913e-03  2.398e-03  -2.883  0.00417 **
## engine:horse         5.980e-04  2.627e-04   2.277  0.02338 *
## engine:weight        1.982e-05  9.870e-06   2.008  0.04541 *
## horse:weight         1.461e-05  2.680e-05   0.545  0.58582
## engine:horse:weight -7.848e-08  6.647e-08  -1.181  0.23848
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.817 on 371 degrees of freedom
## Multiple R-squared:  0.7612, Adjusted R-squared:  0.7567
## F-statistic:   169 on 7 and 371 DF,  p-value: < 2.2e-16
```

View the contents of model summary.

```
sum=summary(mod.reg)
names(sum)
```

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

Get model coefficients.

```
sum$coefficients
```

```
##                  Estimate  Std. Error     t value      Pr(>|t|)
## (Intercept) -0.5673545304 1.580968723 -0.35886512  7.199012e-01
## engine       0.0025929429 0.002427858  1.06799595  2.862205e-01
## horse        0.0004492931 0.004663497  0.09634253  9.233008e-01
## weight      -0.0006066500 0.000232019 -2.61465725  9.297769e-03
## accel        0.0159284879 0.031672447  0.50291308  6.153254e-01
## year         0.0662728753 0.020011017  3.31181941  1.018461e-03
## origin       0.0964715379 0.090248782  1.06895114  2.857905e-01
## cylinder    -0.2033509048 0.119371772 -1.70350915  8.931478e-02
## filter_.    -0.4577229299 0.292996952 -1.56221055  1.190956e-01
## mpg1         0.9102985190 0.016731122 54.40749836 2.447556e-178
```

Get specific values from the model coefficients (e.g., p-value).

```
sum$coefficients[,4]
```

```
##   (Intercept)        engine         horse        weight
accel
##  7.199012e-01  2.862205e-01  9.233008e-01  9.297769e-03  6.153254e-
01
##          year        origin      cylinder      filter_.
mpg1
##  1.018461e-03  2.857905e-01  8.931478e-02  1.190956e-01 2.447556e-
178
```

Logistic Regression

In the logistic regression, suppose we are interested in how GRE (Graduate Record Exam scores), GPA (grade point average) and prestige of the undergraduate institution affect the admission to graduate school. The response variable admission (Y/N or 1/0) is a binary variable.

Import the dataset remotely.

```
admission <-
read.csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
head(admission)

##   admit gre  gpa rank
## 1     0 380 3.61    3
## 2     1 660 3.67    3
## 3     1 800 4.00    1
## 4     1 640 3.19    4
## 5     0 520 2.93    4
## 6     1 760 3.00    2
```

This dataset has a binary response (dependent) variable called *admit*. There are three predictor variables: *gre, gpa* and *rank*. We will treat the variables *gre* and *gpa* as continuous. The variable *rank* takes on the values 1 through 4. Institutions with a rank of 1 have the highest prestige, while those with a rank of 4 have the lowest. We can get basic descriptions for the entire dataset by R function *summary()*.

```
summary(admission)

##      admit             gre             gpa             rank      
##  Min.   :0.0000   Min.   :220.0   Min.   :2.260   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:520.0   1st Qu.:3.130   1st Qu.:2.000  
##  Median :0.0000   Median :580.0   Median :3.395   Median :2.000  
##  Mean   :0.3175   Mean   :587.7   Mean   :3.390   Mean   :2.485  
##  3rd Qu.:1.0000   3rd Qu.:660.0   3rd Qu.:3.670   3rd Qu.:3.000  
##  Max.   :1.0000   Max.   :800.0   Max.   :4.000   Max.   :4.000
```

First, we convert *rank* to a factor to indicate that *rank* should be treated as a categorical variable.

```
admission$rank <- factor(admission$rank)
levels(admission$rank)

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

Second, we fit a logistic regression model using the R function *glm(),* and R function *summary()* to obtain the estimates of the model.

```
results <- glm(admit ~ gre + gpa + rank, data = admission, family =
"binomial")
summary(results)
```

```
##
## Call:
## glm(formula = admit ~ gre + gpa + rank, family = "binomial",
##     data = admission)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.5802  -0.8848  -0.6382   1.1575   2.1732
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.449548   1.132846  -3.045  0.00233 **

## gre          0.002294   0.001092   2.101  0.03564 *

## gpa          0.777014   0.327484   2.373  0.01766 *

## rank        -0.560031   0.127137  -4.405 1.06e-05 ***


## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 499.98  on 399  degrees of freedom
## Residual deviance: 459.44  on 396  degrees of freedom
## AIC: 467.44
##
## Number of Fisher Scoring iterations: 4
```

The output from *summary()* provides us a lot of useful information. In the 'Coefficients' section, we can check whether the predictor variables are significant or not through the p-value. As we can conclude from the output, *gre*, *gpa* and *rank* are statistically significant. Besides, the logistic regression coefficients give the change in the *log* odds of the outcome for a one unit increase in the predictor variable. We can also calculate the odds ratio.

```
exp(coef(results))
```

```
## (Intercept)         gre         gpa        rank

## 0.03175998     1.00229659  2.17496718  0.57119114
```

Power Calculation

We now take a look at how to conduct power analysis for one sample t-test, two sample t-test, paired t-test and One-Way ANOVA.

The power of a test is the probability of detecting an effect given that the effect exists. In statistical jargon, it is the probability of rejecting the null hypothesis when the alternative hypothesis of interest is true.

Conducting a power analysis is generally done for two of the following reasons:

1) To determine the number of subjects (sample size) needed in order to detect a given effect size at a specified power.
2) To determine the power of the test given an effect size and the sample size available for the study.

In the first case, the power analysis is done before the data collection stage. In the second case, the power analysis is often used when there is a limit to the number of subjects the researcher can use in the study. The power analysis is therefore run in this case in order to determine how likely the study conducted using the limited sample size at hand is to detect an effect. A low power would be a good reason to rethink whether to proceed with the study.

In order to conduct a power analysis, three out of four of the following values need to be specified: (i.e. given three of the quantities below the fourth one can be computed using the power analysis)

- Sample size
- Effect size
- Significance level (Type I error: the probability of finding an effect when the effect is present)
- Power (1- Type II error: The probability of failing to detect an effect when the effect is present)

Important: There is no golden rule for the minimum power required for a study. However, it is important to remember that a larger power is more desirable as it reduces the risk of a Type II error. Scientists often follow the rule proposed by Cohen (1988) in his book "Statistical Power Analysis for the Behavioral Sciences" which states that studies should be designed to allow for a power of at least 0.8. For more information on power and how to determine the minimum effect size of a test please refer to Cohen's book mentioned above.

The R package "**pwr**" allows us to run power analysis for a wide variety of models. The R code below makes the "**pwr**" package available in the R library.

```r
library(pwr) #install.packages("pwr")
```

Power Analysis: One Sample t-test.

In the case of the one sample t-test, the null hypothesis is that the mean of the data of interest is equal to a constant and the alternative could be one of the three options: 1. The mean is greater than the constant (right-tailed hypothesis); 2. The mean is less than the constant (left-tailed hypothesis); 3. The mean is not equal to the constant (two-tailed hypothesis).

The R function for carrying the one sample t-test is provided below where "n" is the total sample size, "d" is the effect size, "sig.level" is the significance level, "power" is the power of the test and "type" is set to be "one.sample".

Specifying any three of the four values will generate the value of the non-specified parameter. For example, we compute the sample size required for a two-tailed One Sample t-test at a 0.05 significance level and a power of 80%. We make an educated guess based on previous studies that the minimum effect size we would like to detect is 0.40.

```
pwr.t.test(n  =  ,  d  =0.40  ,  sig.level  =0.05  ,  power  =  0.8,  type  =
"one.sample",                  alternative          =          "two.sided")
##                         One-sample    t    test    power    calculation
##
##                                                          n    =    51.00945
##                                                              d    =    0.4
##                                                 sig.level     =      0.05
##                                                         power    =    0.8
##      alternative = two.sided
```

The result shows that a total sample size of 51 subjects is required to achieve 80% power at the specified significance level.

In the second example below, we compute the power of a one sample t-test with a two-tailed alternative hypothesis at 0.05 significance level, with a sample size of 30 and a minimum effect size of 0.40.

```
pwr.t.test(n = 30 , d =0.40 , sig.level =0.05 , power = , type =
"one.sample", alternative = "two.sided")
##       One-sample t test power calculation
##
##                n = 30
##                d = 0.4
##        sig.level = 0.05
##            power = 0.5628136
##      alternative = two.sided
```

The result shows that with a total sample size of 30 the power of the test will achieve approximately 56%.

Power Analysis: Two Sample t-test.

A two-sample t-test tests if the means of two different groups are equal. A two-sample t-test is only valid if the data between the two groups are not correlated. In the example below, we determine the total sample size required for detecting an effect size of 0.6 given that we want to achieve a 90% power with a 0.05 significance level.

```
pwr.t.test(n = , d =0.6 , sig.level =0.05 , power = 0.9 , type =
"two.sample",alternative = "two.sided")

##
##      Two-sample t test power calculation
##
##               n = 59.35155
##               d = 0.6
##       sig.level = 0.05
##           power = 0.9
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

The result shows that we need a total sample size of about 60. It means that the two groups will have a sample size of 30 each. We can use the same function in order to find the power of a two-sample t-test with balanced samples given the total sample size by leaving the "power" option empty.

If the samples are unbalanced, that is if the data is such that one of the groups has more samples than the other group then we can use the R function *pwr.t2n.test()* in order to find the power of the two sample t-test as illustrated below.

```
pwr.t2n.test(n1 = 40 , n2=57 , d =0.6 , sig.level = 0.05, power
= ,alternative = "two.sided")

##
##      t test power calculation
##
##              n1 = 40
##              n2 = 57
##               d = 0.6
##       sig.level = 0.05
##           power = 0.821067
##     alternative = two.sided
```

The power in this case is about 82%.

Power Analysis: Paired t-test.

A paired t-test is used to compare the means of two groups when we believe the data between the groups are correlated. For example, the paired t-test can be used to compare the mean responses of an outcome of interest before and after an intervention on the subjects. The following R codes show how to obtain the sample size for a paired t-test to achieve a power of 0.9 at significance level 0.05 with an estimated minimum effect size of 0.8.

```
pwr.t.test(n = , d =0.8 , sig.level =0.05 , power = 0.9 , type =
"paired",alternative= "two.sided")

##      Paired t test power calculation
##
##               n = 18.44623
##               d = 0.8
##       sig.level = 0.05
##           power = 0.9
##     alternative = two.sided
##
## NOTE: n is the number of *pairs*.
```

The result shows that with a sample size of 18 pairs we could achieve 90% power.

Power Analysis: One-Way ANOVA

Power analysis for One-Way ANOVA with *k* levels can be performed with the R function *pwr.anova.test()*. In the example below we determine the sample size for one-way ANOVA with 4 levels, 80% power and effect size of 0.4 at 0.05 significance level. Here "*n*" means the number of samples per level and "*f*" is the effect size.

```
pwr.anova.test(k = 4 , n = , f = 0.4 , sig.level = 0.05 , power = 0.8)
##      Balanced one-way analysis of variance power calculation
##
##               k = 4
##               n = 18.04262
##               f = 0.4
##       sig.level = 0.05
##           power = 0.8
##
## NOTE: n is the number in each group.
```

The result shows that in order to detect a treatment effect size of 0.4 or greater if one exists with an 80% power, each of the four treatment groups should have a sample size of 18. Hence the total sample size for the study would be 18*4= 72. The power for the One-Way ANOVA can be computed if the sample size for each level is known, by leaving the power option in the function empty. Note that in such a case the group sample size should be equal.

Power Curves

A power curve is usually a better option for explaining how the power of the test varies with different sample size and effect size. Below are the codes the user can change so as to generate a power curve for any type of test. In the specific example below, the power curve for a one-way ANOVA is generated with different effect size and sample size at 0.05 significance level.

Note: The user is suggested to read the comments (sentences that come after the "#" symbols) in order to learn how to modify the codes according to their needs.

```
#The code generates a power curve for a One-Way ANOVA model with 4
levels of equal sample size.

#Step1: Load pwr package in R working library.
 library(pwr)

#Step2: Define the range of effect size for generating the power curve
using the R function seq(). In this case we generate an effect size
ranging from 0.1 to 0.6 with intervals of 0.01.

 f <- seq(.1,.6,.01)
 nf <- length(f)

#Step3: Define the range of power acceptable for the study. The code
below generated power which ranges from 0.4 to 0.95 with intervals of
0.1.
 p <- seq(.4,.95,.1)
 np <- length(p)

#Obtain sample sizes.
 samsize <- array(numeric(nf*np), dim=c(nf,np))
 for (i in 1:np){
   for (j in 1:nf){
#You can specify the specific type of power analysis to run here by
replacing the pwr.anova.test() function with the power function
according to your analysis, for example, pwr.t.test() for one-sample t-
test or pwr.t2n.test() for two-sample t-test.
     result <- pwr.anova.test(k= 4, n = , f =  f[j], sig.level = .05,
power = p[i])


     samsize[j,i] <- ceiling(result$n)
   }
 }

#Set up graph
 xrange <- range(f)
 yrange <- round(range(samsize))
 colors <- rainbow(length(p))
 plot(xrange, yrange, type="n",
   xlab="Effect size",
   ylab="Sample Size (n)" )

#Add power curves
 for (i in 1:np){
   lines(f, samsize[,i], type="l", lwd=2, col=colors[i])
 }

#Add annotation (grid lines, title, legend)
 abline(v=0, h=seq(0,yrange[2],50), lty=2, col="grey89")
 abline(h=0, v=seq(xrange[1],xrange[2],.02), lty=2,
    col="grey89")

#Add title for the curve
 title(" Power curve for a One-way Anova, Sig=0.05")
legend("topright", title="Power", as.character(p),
    fill=colors)
```

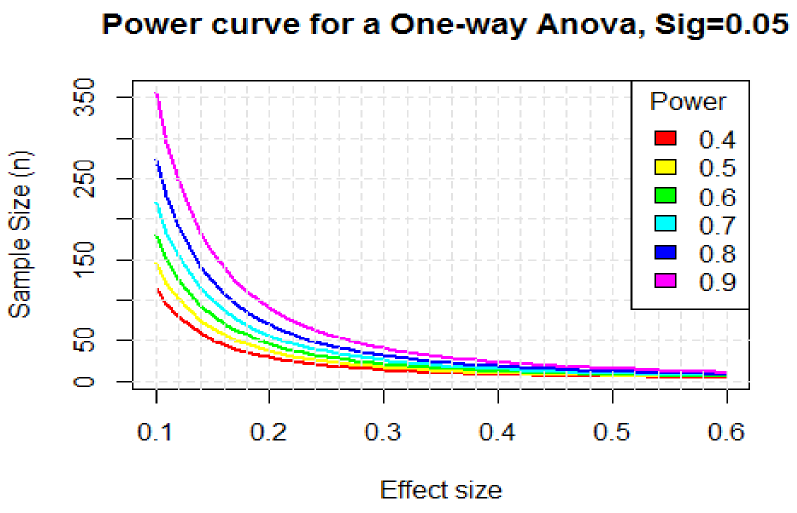


Figure 11

Linear Mixed Models

One or more of the factors may be treated as random effects while performing data analysis. That is, their levels can be thought of as being randomly sampled from a larger population of levels. When one or more random factors exist in a linear model, we call it a Linear Mixed Model (LMM) to state the fact that independent variables are a mixture of fixed factors and random factors.

LMM in R

In R we use the **lme4** package to fit LMM, and the **car** package to do significance test.

Building LMM in R is very similar to building a linear regression via *lm(),* except that we need to declare which variables are random. We will use the car dataset to illustrate how to fit LMM in R.

```
dat = read.csv("Cars.csv")
head(dat, 10)

##    mpg engine horse weight accel year origin cylinder filter  mpg1
## 1    9      4    93    732   8.5    0     NA       NA     NA  9.98
## 2   10    360   215   4615  14.0   70      1        8      0 10.87
## 3   10    307   200   4376  15.0   70      1        8      0  9.63
## 4   11    318   210   4382  13.5   70      1        8      0 12.12
## 5   11    429   208   4633  11.0   72      1        8      0 10.63
## 6   11    400   150   4997  14.0   73      1        8      0 10.79
## 7   11    350   180   3664  11.0   73      1        8      0 12.22
## 8   12    383   180   4955  11.5   71      1        8      0 12.11
## 9   12    350   160   4456  13.5   72      1        8      0 12.66
## 10  12    429   198   4952  11.5   73      1        8      0 11.68
```

We treat *origin, cylinder* and *filter* as categorical variables, so we convert them through R function *as.factor()*.

```
dat$origin = as.factor(dat$origin)
dat$cylinder = as.factor(dat$cylinder)
dat$filter = as.factor(dat$filter)
```

Assume that we want to build a model taking *mpg1* as the response variable, *origin* as fixed factor, and *cylinder* as random factor. It can be performed as followed.

```
library(lme4) #install.packages("lme4")
```

```
## Loading required package: Matrix
model = lmer(mpg1 ~ origin + (1 | cylinder), data = dat, REML = FALSE)
```

The REML parameter controls whether to use the maximum likelihood (ML) or the restricted maximum likelihood (REML) to fit the model. In this example, we choose REML=FALSE to use the ML approach. Then we can use *summary()* to obtain the estimates of model parameters.

```
summary(model)

## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: mpg1 ~ origin + (1 | cylinder)
##    Data: dat
##
##      AIC      BIC   logLik deviance df.resid
##   2373.4   2393.3  -1181.7   2363.4      392
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.4654 -0.6018 -0.1019  0.4598  3.9633
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  cylinder (Intercept) 24.22    4.921
##  Residual             21.54    4.642
## Number of obs: 397, groups:  cylinder, 5
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  20.8890     2.3251   8.984
## origin2       0.7503     0.7459   1.006
## origin3       3.9086     0.7173   5.449
##
## Correlation of Fixed Effects:
##         (Intr) orign2
## origin2 -0.115
## origin3 -0.117  0.437
```

In the "Random effects" section, we can see that the estimated standard deviation of the random factor *cylinder* is 4.921, and the estimated standard deviation of residual is 4.642.

In the "Fixed effects" section, the table gives the estimated fixed effects of different levels of *origin*, where *origin* = 1 is taken to be the baseline level.

Significance Test

The **lme4** package does not provide functions to calculate p-value of fixed effects. Instead, the **car** package has the function *Anova()* working for it.

```
library(car) #install.packages("car")
Anova(model)

## Analysis of Deviance Table (Type II Wald chisquare tests)
##
## Response: mpg1
##         Chisq Df Pr(>Chisq)
## origin 32.038  2  1.104e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The Wald test result shows that *origin* has a significant effect on *mpg1.*

Nested Factors

If we'd like to specify a random factor *g2* nested in a fixed factor *g1,* it can be claimed as *g1 + (1 | g1:g2).* Similarly, if *g1* is also random, it would be *(1 | g1) + (1 | g1:g2),* which can be simplified as *(1 | g1/g2)*. For example, if we treat *filter* as a fixed factor, and *cylinder* is nested in *filter,* the model can be coded as followed.

```
model2 = lmer(mpg1 ~ origin + filter + (1 | filter:cylinder),
              data = dat, REML = FALSE)
summary(model2)

## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: mpg1 ~ origin + filter + (1 | filter:cylinder)
##    Data: dat
##
##      AIC      BIC   logLik deviance df.resid
##   2326.5   2350.3  -1157.3   2314.5      384
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.4593 -0.5981 -0.1003  0.4635  3.9561
##
## Random effects:
##  Groups          Name        Variance Std.Dev.
##  filter:cylinder (Intercept) 10.80    3.287
##  Residual                    21.44    4.630
## Number of obs: 390, groups:  filter:cylinder, 3
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  14.8941     3.3184   4.488
## origin2       0.7022     0.7473   0.940
## origin3       3.9518     0.7184   5.501
## filter1       8.7999     4.0693   2.163
##
## Correlation of Fixed Effects:
##         (Intr) orign2 orign3
## origin2  0.000
## origin3  0.000  0.436
## filter1 -0.815 -0.049 -0.051
```

```
Anova(model2)
```

```
## Analysis of Deviance Table (Type II Wald chisquare tests)
##
## Response: mpg1
##          Chisq Df Pr(>Chisq)
## origin 32.8924  2  7.203e-08 ***
## filter  4.6764  1    0.03058 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Graphics

Create a plot (e.g., histogram).

```
set.seed(1)
hist(rnorm(100))
```

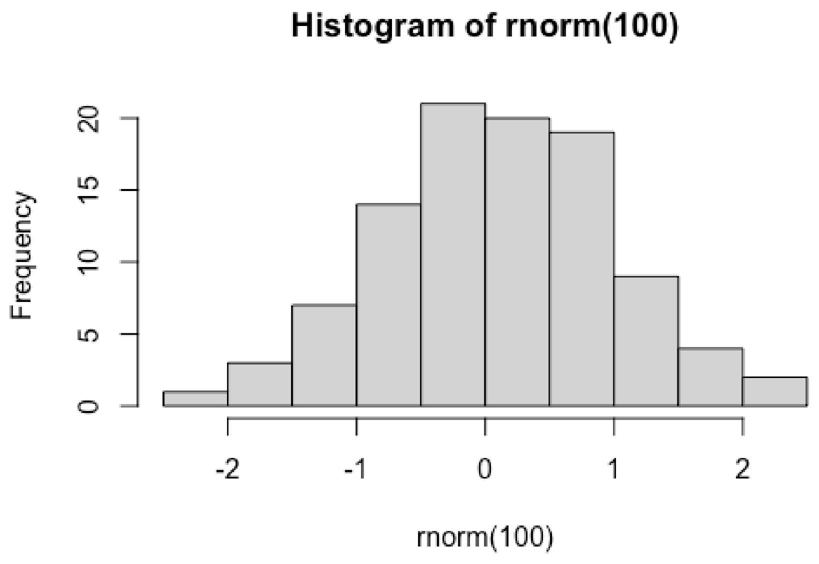


Figure 12