This page holds a list of key machine learning terms and their definitions.

Introduction

R is a programming language created and developed in 1991 by two statisticians at the University of Auckland, in New Zealand. It officially became free and open-source only in 1995. For its origins, it provides statistical and graphical techniques, linear and non-linear models, techniques for time series, and many other functionalities. Even if Python is the most common in the Data Science field, R is still widely used for specialized purposes, like in financial companies, research, and healthcare.

Examples

These are examples.

Assignment

When we program in R, the entities we work with are called objects [1]. They can be numbers, strings, vectors, matrices, arrays, functions. So, any generic data structure is an object. The assignment operator is `<-`, which combines the characters `<` and `-`. We can visualize the output of the object by calling it:

```
# Assignment
x <- 23
```

A more complex example can be:

```
# A more complex example
x <- 1/1+1*1
y <- x^4
z <- sqrt(y)
x
```

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

```
y
```

```
## [1] 16
```

```
z
```

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

As you can notice, the mathematical operators are the ones you use for the calculator on the computer, so you don't need the effort to remember them. There are also mathematical functions available, like `sqrt`, `abs`, `sin`, `cos`, `tan`, `exp`, and `log`.

Vectors in R Programming

In R, the vectors constitute the simplest data structure. The elements within the vector are all of the same types. To create a vector, we only need the function `c()`:

```
# Create vector
v1 <- c(2,4,6,8)
v1
```

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

We can access the particular element in the vector by `[index]`

```
# Access a particular element
v1[2]
```

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

This function simply concatenates different entities into a vector. There are other ways to create a vector, depending on the purpose. For example, we can be interested in creating a list of consecutive numbers and we don't want to specify them manually. In this case, the syntax is `a:b` , where a and b correspond to the lower and upper extremes of this succession. The same result can be obtained using the function `seq()`

```
# Creating a list of consecutive numbers
1:7
```

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

The function `seq()` can also be applied to create more complex sequences. For example, we can add the argument by the step size and the length of the sequence:

```
# Create list by step size
v4 <- seq(0,1,by=0.1)
v4
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
# Create list by the length of the sequence
v5 <- seq(0,2,len=11)
v5
```

```
##  [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

To repeat the same number more times into a vector, the function rep() can be used:

```
# Repeat the same number more times into a vector
v6 <- rep(2,3)
v6
```

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

```r
v7 <- c(1,rep(2,3),3)
v7
```

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

There are not only numerical vectors. There are also logical vectors and character vectors:

```r
# Logical vector
x <- 1:10
y <- 1:5
l <- x==y
l
```

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

```r
# Character vector
c <- c('a','b','c')
c
```

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

factors in R Programming

factors are specialized vectors used to group elements into categories. There are two types of factors: ordered and unordered. For example, we have the countries of five friends. We can create a factor using the function `factor()`

```r
# Create a factor
states <- c('italy','france','germany','germany','germany')
statesf <- factor(states)
statesf
```

```
## [1] italy   france  germany germany germany
## Levels: france germany italy
```

To check the levels of the factor, the function `levels()` can be applied.

```r
# Check the levels of the factor
levels(statesf)
```

```
## [1] "france"  "germany" "italy"
```

Matrices in R Programming

As you probably know, the matrix is a 2-dimensional array of numbers. It can be built using the function `matrix()`

```r
# Creating a matrix
m1 <- matrix(1:6,nrow=3)
m1
```

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

```r
m2 <- matrix(1:6,ncol=3)
m2
```

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

It can also be interesting combine different vectors into a matrix row-wise or column-wise. This is possible with `rbind()` and `cbind()`:

```r
# Combining vectors into matrix using rbind()
countries <- c('italy','france','germany')
age <- 25:27
rbind(countries,age)
```

```
##           [,1]    [,2]     [,3]
## countries "italy" "france" "germany"
## age       "25"    "26"     "27"
```

```r
# Or using cbind()
cbind(countries,age)
```

```
##      countries age
## [1,] "italy"   "25"
## [2,] "france"  "26"
## [3,] "germany" "27"
```

Arrays in R Programming

Arrays are objects that can have one, two, or more dimensions. When the array is one-dimensional, it coincides with the vector. In the case it's 2D, it's like to use the matrix function. In other words, arrays are useful to build a data structure with more than 2 dimensions.

```r
# Creating an array
a <- array(1:16,dim=c(6,3,2))
a
```

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

```
## [1,]    1    7   13
## [2,]    2    8   14
## [3,]    3    9   15
## [4,]    4   10   16
## [5,]    5   11    1
## [6,]    6   12    2
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    3    9   15
## [2,]    4   10   16
## [3,]    5   11    1
## [4,]    6   12    2
## [5,]    7   13    3
## [6,]    8   14    4
```

list

The list is a ordered collection of objects. For example, it can a collection of vectors, matrices. Differently from vectors, the lists can contain values of different type. They can be build using the function `list()`:

```
# Creating a list
x <- 1:3
y <- c('a','b','c')
l <- list(x,y)
l
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a" "b" "c"
```

Data frames in R Programming

A data frame is very similar to a matrix. It's composed of rows and columns, where the columns are considered vectors. The most relevant difference is that it's easier to filter and select elements. We can build manually the data frame using the function `data.frame()`:

```
# Data frame
countries <- c('italy','france','germany')
age <- 25:27
df <- data.frame(countries,age)
df
```

```
##   countries age
## 1     italy  25
## 2    france  26
## 3   germany  27
```

An alternative is to read the content of a file and assign it to a data frame with the function `read.table()`:

```r
# read.table() function
df <- read.table('titanic.dat')
```

Like in Pandas, there are other functions to read files with different formats. For example, let's read a csv file:

```r
# read.csv() function
df <- read.csv('Data/titanic.csv')
head(df)
```

```
##   PassengerId Survived Pclass
## 1           1        0      3
## 2           2        1      1
## 3           3        1      3
## 4           4        1      1
## 5           5        0      3
## 6           6        0      3
##                                                  Name    Sex Age SibSp Parch
## 1                             Braund, Mr. Owen Harris   male  22     1     0
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38     1     0
## 3                              Heikkinen, Miss. Laina female  26     0     0
## 4        Futrelle, Mrs. Jacques Heath (Lily May Peel) female  35     1     0
## 5                            Allen, Mr. William Henry   male  35     0     0
## 6                                    Moran, Mr. James   male  NA     0     0
##             Ticket    Fare Cabin Embarked
## 1        A/5 21171  7.2500              S
## 2         PC 17599 71.2833   C85        C
## 3 STON/O2. 3101282  7.9250              S
## 4           113803 53.1000  C123        S
## 5           373450  8.0500              S
## 6           330877  8.4583              Q
```

Like in Python, R provides pre-loaded data using the function `data()`:

```r
# Load pre-loaded data
data("mtcars")
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

The function `head()` allows visualizing the first 6 rows of the mtcars dataset, which provides the data regarding fuel consumption and ten characteristics of 32 automobiles.

To check all the information about the dataset, you write this line of code:

```r
# This code not evaluated
help(mtcars)
```

In this way, a window with all the useful information will open. To have an overview of the dataset's structure, the function `str()` can allow having additional insights into the data:

```r
# Structure of the data
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

From the output, it's clear that there are 32 observations and 11 variables/columns. From the second line, there is a row for each variable that shows the type and the content. We show separately the same information using:

- the function `dim()` to look at the dimensions of the data frame
- the function `names()` to see the names of the variables

```r
# Dimensions of the data frame
dim(mtcars)
```

```
## [1] 32 11
```

```r
# Names of the variables
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

The summary statistics of the variables can be obtained through the function `summary()`.

```r
# Summary of the data
summary(mtcars)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
```

```
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

We can access specific columns using the expression `namedataset$namevariable`. If we want to avoid specifying every time the name of the dataset, we need the function `attach()`.

```
# Using $ sign expression
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

```
# Using attach() function
attach(mtcars)
mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

In this way, we attach the data frame to the search path, allowing to refer to the columns with only their names. Once we attached the data frame and we aren't interested anymore to use it, we can do the inverse operation using the function `detach()`.

We can also try to select the first row in the data frame using this syntax:

```
# Select the first row
mtcars[1,]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

Note that the index starts from 1, not from 0! If we want to extract the first columns, it can be done in this way:

```r
# Select the first column
mtcars[,1]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

We can also try to filter the rows using a logical expression:

```r
# Filter with logical expression
mtcars[mpg>20,]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

we can also specify the column while we filter:

```r
# Specify the column while filter
mtcars[mpg>20, 'mpg']
```

```
##  [1] 21.0 21.0 22.8 21.4 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26.0 30.4 21.4
```

`for` and `while` in R Programming

The `for` loop is used to iterate elements over the sequence like in Pandas. The difference is the addition of the parenthesis and curly brackets. It has slightly different syntax:

```r
# for loop syntax
for (var in seq) statement
```

```r
# for loop example
for (i in 1:4)
print(i)     # or {print(i)}
```

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

`while` executes a statement or more statements as long as the condition is true

```
# while loop syntax
while (cond) statement
```

```
# while loop example
i <- 1
while (i < 6)
  {print(i)
   i <- i+1}  # The curly brackets are needed
```

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

`if` statement in R Programming

The syntax of the if statement is similar to the one in Python. As before, the difference is the addition of the parenthesis and curly brackets.

```
# if statement syntax
if (cond1) {statement1} else {statement2}
```

and

```
# if statement syntax
if (cond1) {statement1} else if {statement2} else {statement3}
```

```
# if statement example
for (i in 1:4)
{if (i%%2==0) print('even') else print('odd')}
```

```
## [1] "odd"
## [1] "even"
## [1] "odd"
## [1] "even"
```

If we want to compare two numbers and see which number is greater of the other, we can do it in this way:

```
# if statement example
a <- 10
b <- 2
if (b>a){
  print('b is greater than a')
}else if (a==b){
  print('a and b are equal')
}else{
  print('a is greater than b')
}
```

```
## [1] "a is greater than b"
```

There is also a vectorized version of the if statement, the function **ifelse(condition,a,b)** . It's the equivalent of writing:

```
# ifelse() function syntax
if condition {a} else {b}
```

For example, let's check if a number is positive:

```
# ifelse() function
x <- 3
ifelse(x>=0, 'positive', 'negative')
```

```
## [1] "positive"
```

Function in R Programming

The function is a block of code used to perform an action. It runs only when the function is called. It usually needs parameters, that need to be passed, and returns an output as result. It's defined with this syntax in R:

```
# function  syntax
namefunction <- function(par_1,par_2,...)
  {expression(s)}
```

Let's create a function to calculate the average of a vector:

```
# function example
average <- function(x)
{ val = 0
  for (i in x){val=val+i}
  av = val/length(x)
  av
}

# Execute the function
average(1:3)
```

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

Probability distributions in R Programming

A characteristic of R is that it provides functions to calculate the density, distribution function, quantile function and random generation for different probability distributions. For example, let's consider the normal distribution:

- **dnorm(x)** calculates the value of the density in x
- **pnorm(x)** calculates the value of the cumulative distribution function in x

- **qnorm(p)** calculates the quantile of level p
- **rnorm(n)** generates a sample from a standard normal distribution of n dimension

Now, I show a table with the most known distributions available in R:

R	Distribution	Parameters	Default
norm	normal	mean, sd	0, 1
t	Student's t	df	0, 1
chisq	chi-squared	df	-
f	F	df1, df2	-,-
unif	uniform	min, max	0, 1
exp	exponential	rate	1
gamma	gamma	shape, scale	-, 1
binom	binomial	size, prob	-,-
pois	Poisson	lambda	-

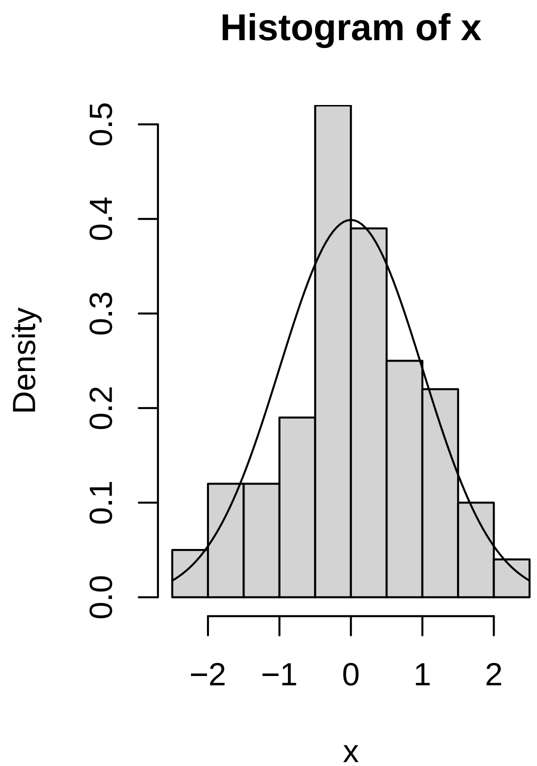
Plotting commands in R Programming

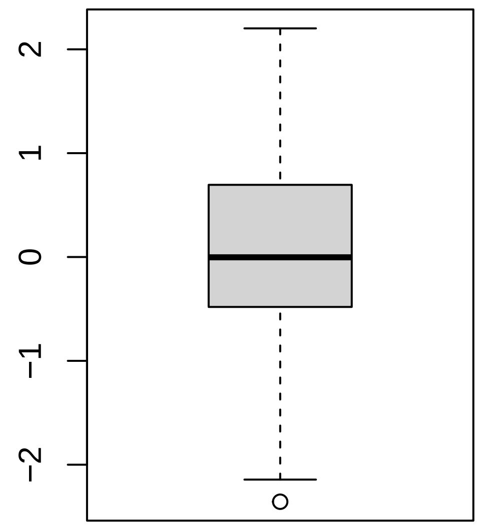
The graphs are very important to get insights into the data. R provides plotting commands to display a huge variety of plots:

- **plot(x)** is the most common function used to produce scatterplots
- **pairs(X)** is used to display multivariate data. It produces a pairwise scatterplot matrix of the variables contained in X.
- **hist(x)** is used to display the histogram
- **box(x)** is used to display the boxplot
- **qqplot(x)** is used to produce the Q-Q plot, useful to check if the distribution analyzed is normal or not.
- **abline(h=y)** and **abline(v=x)** are the most used function to add horizontal and vertical lines in the already built plot
- **curve(expr,add=FALSE)** is used to display a curve, that can be added or not to an already existing graph.
- **par(mfrow=(r,c))** is used put multiple graphs in a single plot. The **mfrow** parameter specifies the number of rows and the number of columns.
- **legend(x,y,legend,...)** is used to specify the legend in the plot at the specified position (x,y)

For example, we can generate a sample with 200 units from a normal distribution. Let's suppose we don't know the distribution and we want to display the histogram and the boxplot:

```r
# Plotting example
x <- rnorm(200)
par(mfrow=c(1,2))
hist(x,ylim=c(0,0.5),prob=TRUE)
curve(dnorm(x),add=TRUE)
boxplot(x)
```





Linear Regression in R

Let's take again the mtcars dataset and let's suppose that we want to perform the linear regression to see the estimated coefficients. As the first trial, I include only one dependent variable, the number of cylinders in the model, which is called linear regression. The syntax of the formula within the function lm is response~terms, where the response is the response variable, while terms refer to one or more dependent variables included in the model.

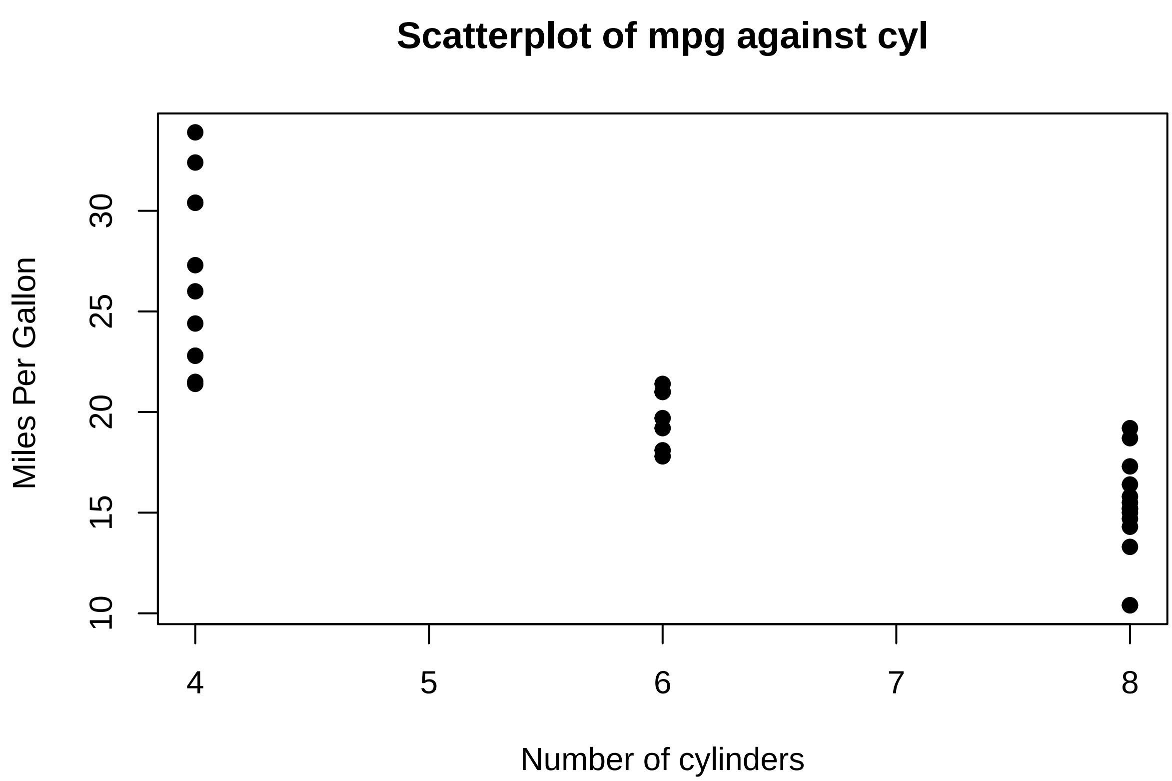
```
# Load the data (the result message has been suppressed)
data(mtcars)
attach(mtcars)
```

```
# Run the regression
lm1 <- lm(mpg~cyl)
lm1$coefficients
```

```
## (Intercept)         cyl
##    37.88458    -2.87579
```

Looking at the parameter of cyl, we can understand that there is negative relationship between the number of cylinders and mpg. To better understand, we can visualize the scatterplot between the two features:

```r
# Scatterplot of mpg against cyl
plot(cyl, mpg, main="Scatterplot of mpg against cyl",
   xlab="Number of cylinders", ylab="Miles Per Gallon ", pch=19)
```



It seems that increasing the number of cylinders lead to a decrease miles/(US) gallon. The most relevant results of the linear model are provided using the function **summary()**.

```r
# summary() function
summary(lm1)
```

```
##
## Call:
## lm(formula = mpg ~ cyl)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.9814 -2.1185  0.2217  1.0717  7.5186
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.8846     2.0738   18.27  < 2e-16 ***
## cyl          -2.8758     0.3224   -8.92 6.11e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 3.206 on 30 degrees of freedom
## Multiple R-squared:  0.7262, Adjusted R-squared:  0.7171
## F-statistic: 79.56 on 1 and 30 DF,  p-value: 6.113e-10
```

It's the summary of the results obtained performing the linear regression model on the data. At the top of the output, we can see the variables included in the model. There are some statistics (minimum, first and third quartiles, median, maximum) regarding the residues of the estimated model. After, there is a table containing the estimated coefficients of the model, where each row corresponds to a coefficient. Each row has the following information:

- the value of the estimated coefficient
- the standard Error
- the observed t-value
- the observed level of significance: in case it's smaller than 0.05, the parameter is significant and, then, there is a linear relationship between that variable and the response variable.

We can see that both coefficients are significant with p-value < 0.05 and R^2 is high, near 1, considering that we only included a variable. `cyl`'s coefficient is negative and, then, indicates the decrease of value for each increase of one unit in `mpg`.

We can also try to include another predictor and include the interaction term between `cyl` and `disp`:

```r
# Regression with additional term and interaction
lm2 <- lm(mpg ~ cyl*disp)
summary(lm2)
```

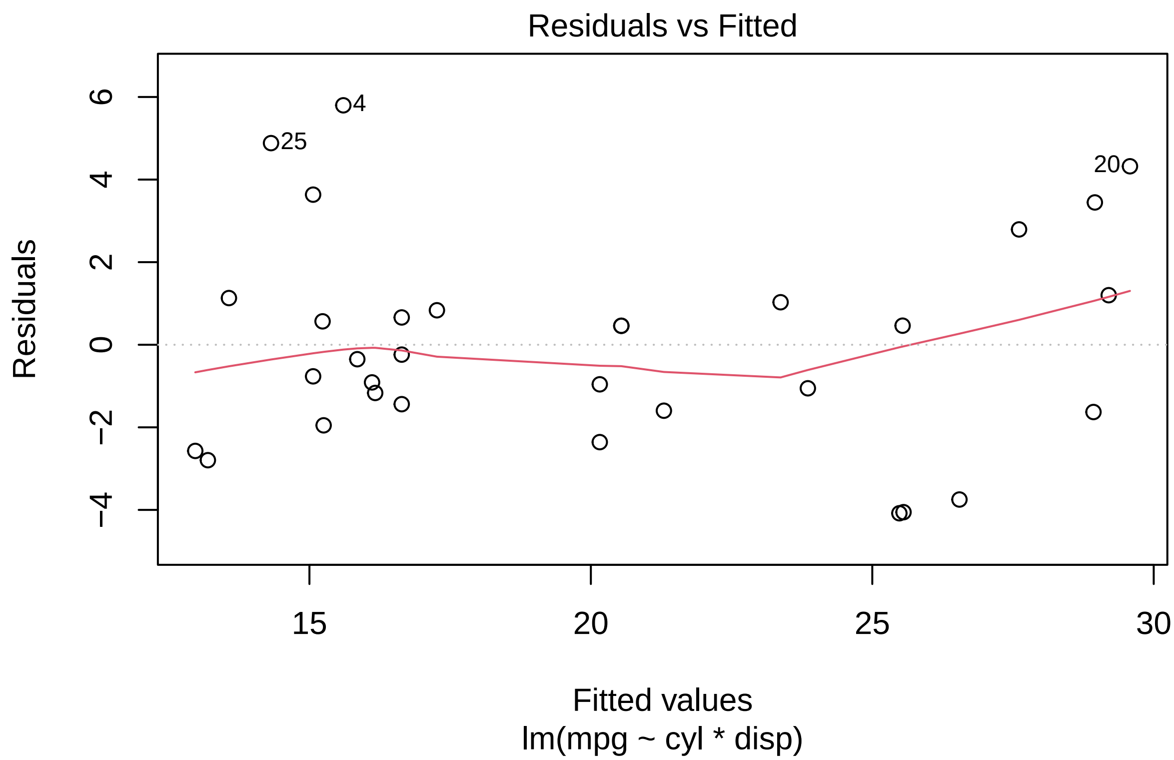
```
##
## Call:
## lm(formula = mpg ~ cyl * disp)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.0809 -1.6054 -0.2948  1.0546  5.7981
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 49.037212   5.004636   9.798 1.51e-10 ***
## cyl         -3.405244   0.840189  -4.053 0.000365 ***
## disp        -0.145526   0.040002  -3.638 0.001099 **
## cyl:disp     0.015854   0.004948   3.204 0.003369 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.66 on 28 degrees of freedom
## Multiple R-squared:  0.8241, Adjusted R-squared:  0.8052
## F-statistic: 43.72 on 3 and 28 DF,  p-value: 1.078e-10
```

The following code will give the same result:

```r
# Alternate coding method
lm3 <- lm(mpg ~ cyl + disp + cyl:disp)
summary(lm3)
```

In the code, I show different syntax formats, that allow reaching the same results. Putting the $*$ between the two features enables to write less code. As before, all the coefficients are significant. Now, the R^2 is higher, equal to 0.8. To evaluate how well the model explains well the behaviour of the data, an efficient way is to display the residuals versus the fitted values, where the residuals are the differences between the true values and the fitted values.

```
plot(lm2, which=1)
```



You can read the documentation of the function `plot.lm` which is the plot function dedicated to `lm`. You can select the graphs that you want to display with argument **`which`**. There is 6 graphs that you can choose. For example, for qqplot & residual plot:

```
# For qqplot and residual plot (The plots suppressed)
plot(lm2, which=c(2,1))
```

The red curve corresponds to the smooth fit to the residuals and has a U-shape, indicating that there are non-linear associations in the data.

After this step, we can finally predict mpg on new data using the fitted model:

```
# Prediction for new data
newdata <- data.frame(mpg=20,cyl=8,disp=150,hp=100,drat=3,wt=2.4,qsec=17,vs=1,am=1,gear=4,carb=2)
predict(lm2,newdata)
```

```
##        1
## 18.99105
```

Remarks

This is document is based on the article Master the basics of R Programming by Eugenia Anello (Published on October 12, 2021 and last modified on November 12, 2021)