R FOR DATA ANALYSIS

FIRST EDITION

WRITTEN BY
THOMAS PERNET

SHANGHAI

R for Data Science

Thomas Pernet-Coudrier

Contents

Preface

R is a programming language developed by Ross Ihaka and Robert Gentleman in 1993. R possesses an extensive catalogue of statistical and graphical methods. It includes machine learning algorithm, linear regression, time series, statistical inference to name a few. Most of the R libraries are written in R, but for heavy computational task, C, C++ and Fortran codes are preferred.

R is not only entrusted by academic, but many large companies also use R programming language, including Uber, Google, Airbnb, Facebook and so on.

Data analysis with R rotates around a succession of steps; programming, transforming, discovering, modeling and communicate the results

- **Program**: R is a clear and accessible programming tool
- **Transform**: R is made up of a collection of libraries designed specifically for data science
- **Discover**: Investigate the data, refine your hypothesis and analyze them
- **Model**: R provides a wide array of tools to capture the right model for your dta
- **Communicate**: Integrate codes, graphs and outputs to a report with R Markdown or build Shiny apps to share with the world

0.1 What is R used for?

- Statistical inference
- Data analysis
- Machine learning algorithm

Data science is shaping the way companies run the business. Without any doubt, staying away from Artificial Intelligence and Machine Learning is leading the company to the wrong direction. A big question to ask is which tool you should learn to get into the data world. Learning a new language requires some time investment.

They are plenty of tools available in the market to perform data analysis. The picture below depicts the learning curve compared to the business capability a language offers. The negative relationship implies that there is no free lunch. If you want to give the best insight from the data, then you need to spend some time learning the appropriate tool, which is R.

On the top left of the graph, you can see Excel and PowerBI. These two tools are simple to learn but don't offer outstanding business capability, especially in term of modelling. In the middle, you can see Python and SAS. SAS is a dedicated tool to run a statistical analysis for business, but it is not free. SAS is a click and run software. Python, however, is a language with a monotonous learning curve. Python is a fantastic tool to deploy Machine Learning and AI but lacks communication features. With an identical learning scheme, R is a good trade-off between implementation and data analysis.

0.2 R or Python

Python has been developed by Guido van Rossum, a computer guy, circa 1991. Python has influential libraries for math, statistic and Artificial Intelligence. You can think Python as a pure player in Machine Learning. However, Python is not completely mature (yet) for econometrics and communication. Python is the best tool for Machine Learning integration and deployment but not for business analytics.

The good news is R is developed by academics and scientist. It is designed to answer statistical problems, machine learning and generally speaking data science. R is the right tool for data science because of its powerful communication libraries. Besides, R is equipped with many packages to perform time series analysis, panel data and data mining. On the top of that, there are not better tools compared to R.

0.3 About the author

Thomas Pernet-Coudrier is a PhD student in International trade. He studied data analysis and machine learning since many years.

0.4 Copyright

March 2018: First Edition

0.5 Convention Used in This Tutorial

The **R code** will be written inside a grey textbox, like below. You can easily copy and paste the code into Rstudio Console. The **output** is displayed after the character `#`. For instance, we write the code `'Hello, world!`, the output will be `# Hello world!`. The ## means we print an output and the number in the square bracket refers to the numbering of the displayed line. The sentences in green are called **annotation**. We can use `#` inside an R script to add any comment we want. R won't read it during the running time.

```
print('Hello world!')
```

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

```
# This is an annotation. R does not run annotation
```

The description of R function is written inside a text box. It includes the function name and the arguments.

```
function(a, b, c)
Arguments

- a: Explanation of a
- b: Explanation of b
- c: Explanation of c
```

1 Install R

R is a programming language. To use R, we need to install an **Integrated Development Environment** (IDE). **Rstudio** is the Best IDE available as it is user-friendly, open-source and is part of the Anaconda platform.

1.1 Install Anaconda

What is Anaconda?

Anaconda free open source distributing both Python and R programming language. Anaconda is widely used in the scientific community and data scientist to carry out Machine Learning project or data analysis.

Why use Anaconda?

Anaconda will help you to manage all the libraries required for Python or R. For the sake of the tutorial; we will use Rstudio, which is a free and open-source IDE. Rstudio is also available from direct downloading. I recommend you to install Rstudio with Anaconda especially if you are going to use or using Python. Anaconda will install all the required libraries and IDE into one single folder to simplify package management.

To use Rstudio with Anaconda, you need to follow these steps:

- Step 1: Install Anaconda
- Step 2: Install R and Rstudio

Go to Anaconda and Download Anaconda for Python 3.6 for your OS.

By default, Chrome selects the downloading page of your system. In this tutorial installation is done for Mac. If you run on Windows or Linux, download Anaconda 5.1 for Windows installer or Anaconda 5.1 for Linux installer.

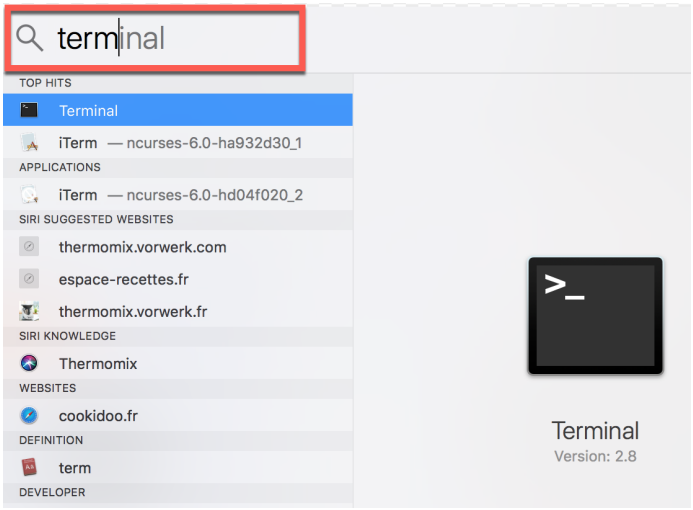
When you are done installing Anaconda, you can proceed to the installation of R and Rstudio. You will use the terminal to perform these two steps.

Open the terminal

Mac user

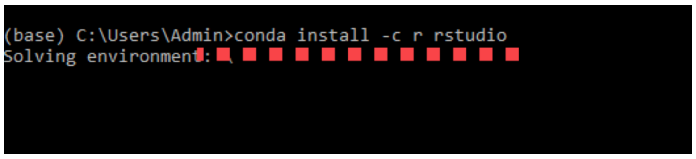
For Mac user, there is two options top open the terminal.

- The shortest way is to use the Spotlight Search and write terminal.
- The second option is to use the shortcut `shift + command + U` and open the terminal.



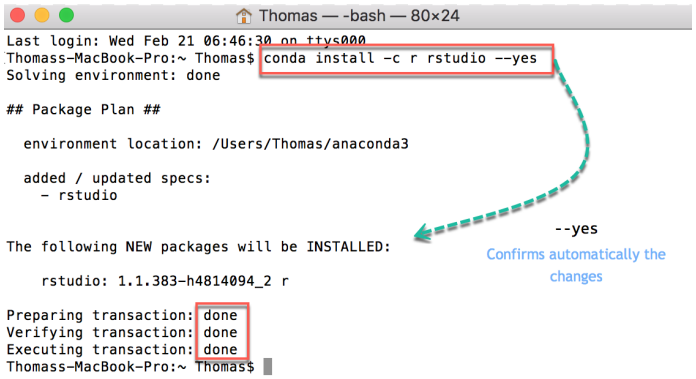
Windows user

- Click on the "Start" button to open the Start menu.
- Open the "All Programs" menu, followed by the "Accessories" option.
- Select the "Command Prompt" option from the "Accessories" menu to open a command-line interface session in a new window on the computer.



1.1.1 Install R and Rstudio

When the setup is done, you are ready to install our first libraries. You recommend you to install all packages and dependencies with anaconda with the conda command in the terminal.

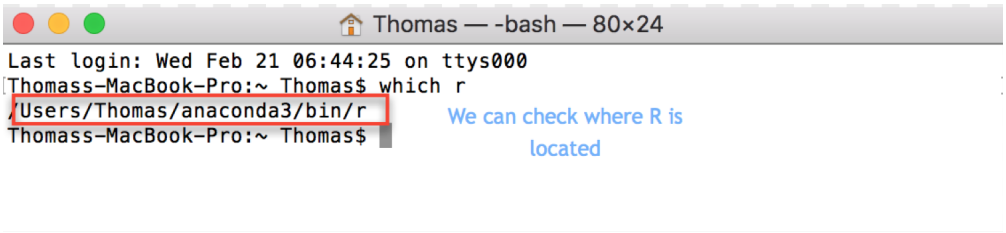


1.1.2 In the terminal

```
conda install r-essentials --yes
conda update r-essentials --yes
conda install -c r rstudio -yes
```

It takes some time to upload all the libraries. Be patient... you are all set.

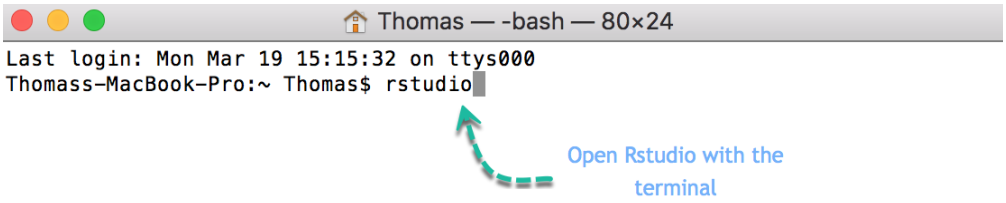
If you want to see where R is located, use which r in the terminal.



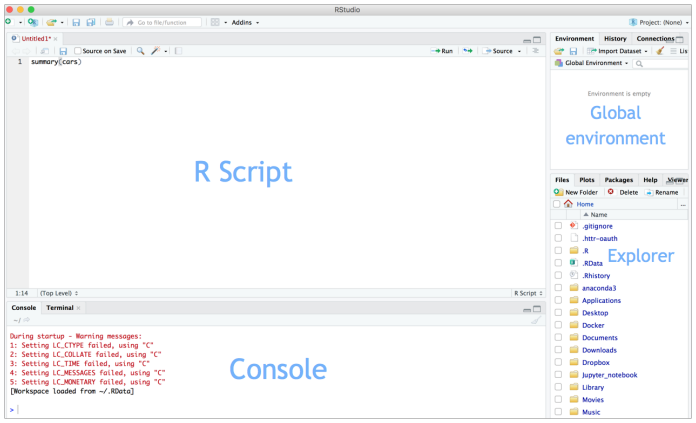
When you run Rstudio, you need to open another terminal window to write the command lines.

1.1.3 Run Rstudio

Directly run the command line from terminal to open Rstudio:



A new window will be opened with Rstudio.



1.1.4 R environment

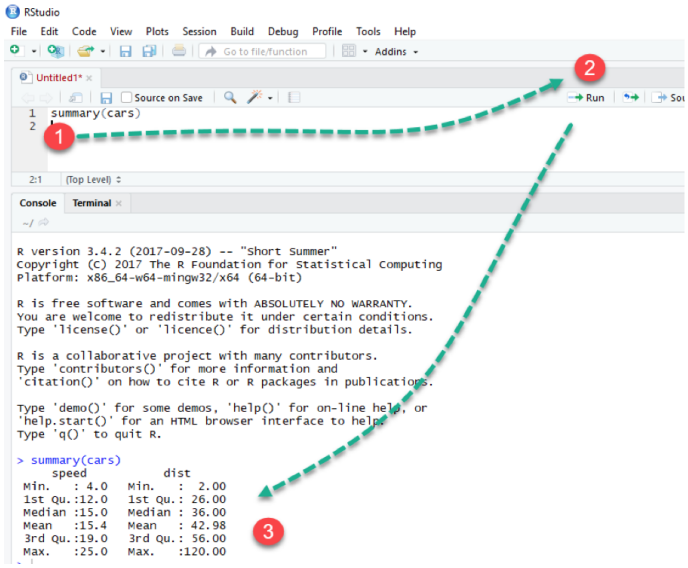
Open Rstudio from the terminal and open a script. Write the following command:

```
## In Rstudio
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
```

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

```
#2. Click Run
#3. Check Output
```

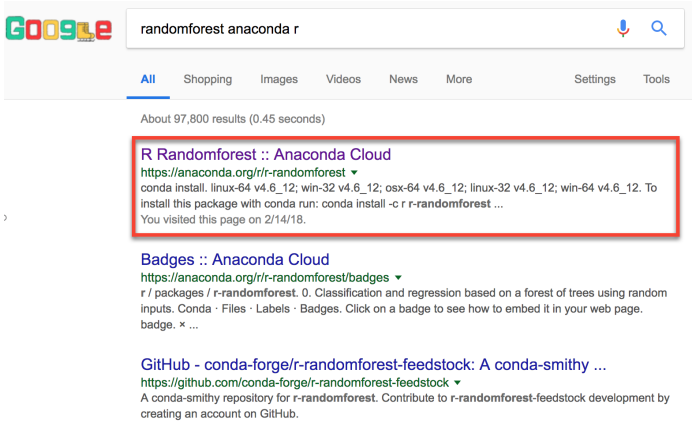


If you can see the summary statistics, it works. You can close Rstudio without saving the files.

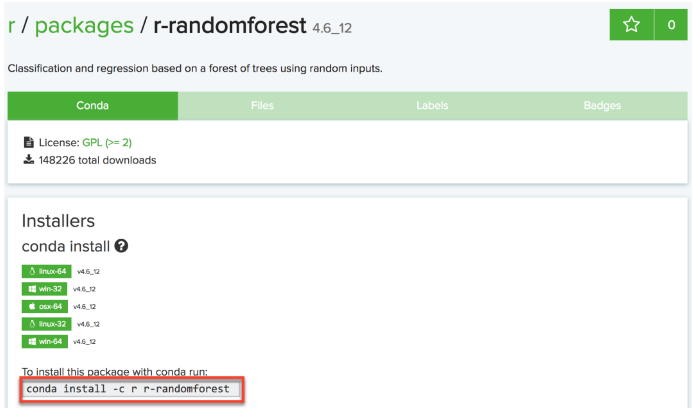
1.1.5 Install package

Install package with anaconda is trivial. You go to your favorite browser, type the name of the library followed by anaconda r.

You choose the link that points to anaconda. You copy and paste the library into the terminal.



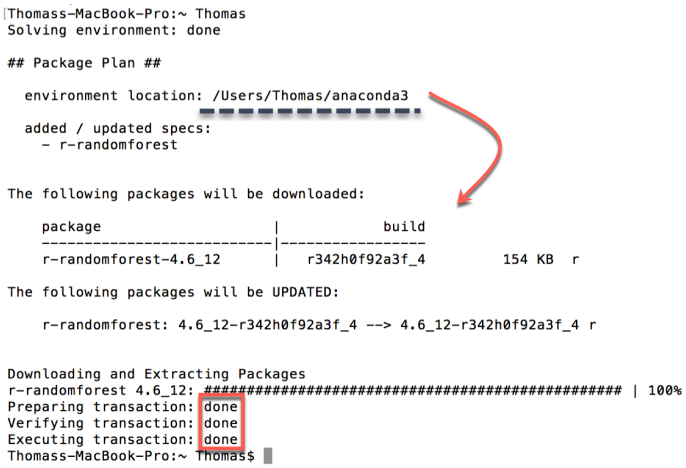
For instance, you need to install randomForest for the tutorial on random forest, you go https://anaconda.org/r/r-randomforest.



Run

```
conda install -c r r-randomforest --yes
```

from the terminal.



The installation is completed.

Note: Thorough this tutorial, you won't need to install many libraries as the most used libraries came with the r-essential conda library. It includes ggplot for the graph and caret for the machine learning project.

1.1.6 Open a library

To run the R function randomForest(), we need to open the library containing the function. In the Rstudio script, we can write

```
library(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

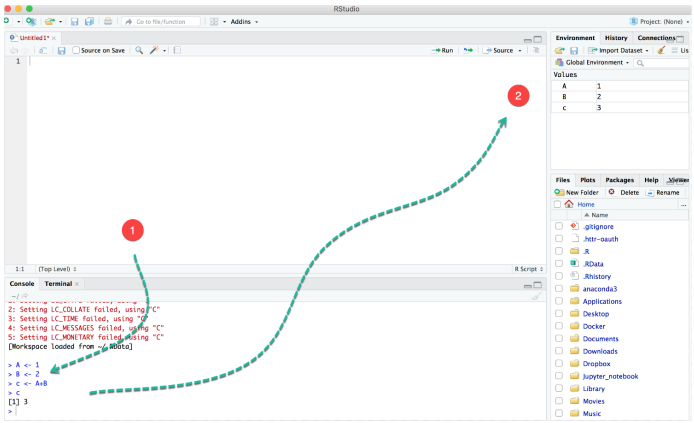
```
## In Rstudio
library(randomForest)
```

Note: Avoid as much as possible to open unnecessary packages. You might ended up creating conflicts between libraries.

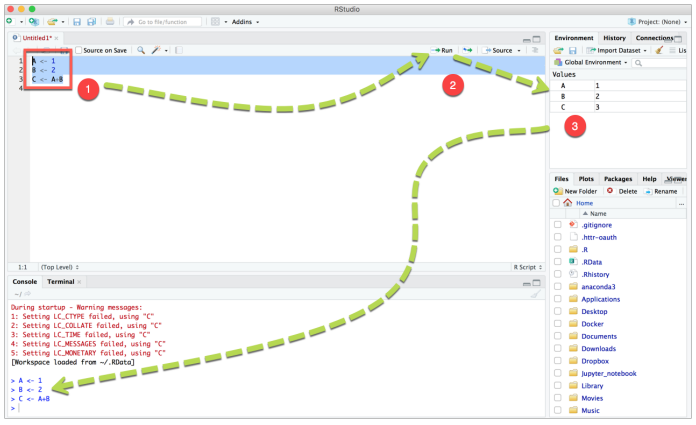
1.1.7 Run R code

We have two ways to run codes in R

- You can run the codes inside the Console. Your data will be stored in the Global Environment but no history is recorded. You won???t be able to replicate the results once R is closed. You need to write the codes all over again. This method is not recommended if you want to replicate or save your codes.

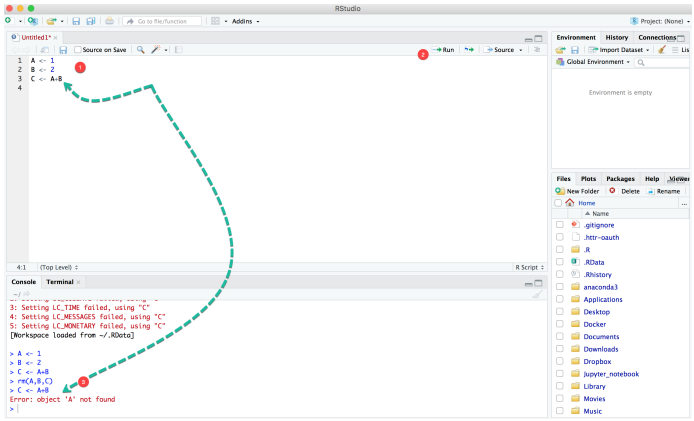


- Write the code in the script. You can write as many lines of codes as we want. To run the code, you select the rows you want to return. Finally click on run. The variable A, B and C are stored in the global environment as values and you can see the output in the Console. You can save your script and open it later. Your results won???t we lost.

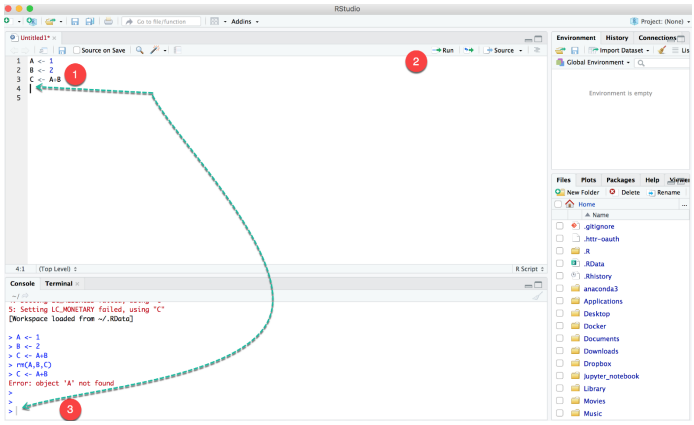


Note: If you point the cursor at the third rows (i.e. `C <- A+B`), the Console displays an error. That???s, you didn???t run the line above C.

In this case, R does not know the value of A and B yet.



In a similar way, if you point the cursor to an empty row and click on run, R return an empty output.



2 Introduction of R framework

The first part of the book introduces you the different types of data and how to manipulate them in the R environment.

2.1 Data type

2.1.1 Basic operations

We will first see the basic arithmetic operations in R. The following operators stand for:

Operator	Description
+	Addition
-	Substraction
*	Multiplication
/	Division
^ or **	Exponentiation

```
# An addition
3 + 4
```

```
## [1] 7
```

```
# A multiplication
3 * 5
```

```
## [1] 15
```

```
 # A division
(5 + 5) / 2
```

```
## [1] 5
```

```
# Exponentiation
2^5
```

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

```
# Modulo
28 %% 6
```

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

2.1.2 Variables

Variables store values and are an important component in programming, especially for a data scientist. A variable can store a number, an object, a statistical result, vector, dataset, a model prediction basically anything R outputs. We can use that variable later simply by calling the name of the variable.

To declare a variable, we need to asssign a variable name. The name should not have space. We can use _ to connect to words.

To add a value to the variable, use <- or '=. Here is the syntax:

```
# First way to declare a variable:  use the <-
name_of_variable <- value
# Second way to declare a variable:  use the =
name_of_variable = value
```

In the command line, we can write the following codes to see what happens:

```
# Print variable x
x <- 42
x
```

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

```
y  <- 10
y
```

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

```
# We call x and y and apply a subtraction
x-y
```

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

2.1.3 Basic data types

R works with numerous data types, including

- Scalars
- Vectors (numerical, character, logical)
- Matrices
- Dataframes
- Lists

For instance, we can enumerate different types of data as follow:

- `4.5` is a decimal value called **numerics**.
- `4` is a natural value called **integers**. Integers are also numerics.
- `TRUE` or `FALSE` are **Boolean** value called **logical**.
- Value inside `" "` or "' are text (string). They are called **characters**.

We can check the type of a variable with the `class` function

```
# Declare variables of different types
# Numeric
numeric <- 28
class(numeric)
```

```
## [1] "numeric"
```

```
# String
character <- "R is Fantastic"
class(character)
```

```
## [1] "character"
```

```
# Boolean
logical <- TRUE
class(logical)
```

```
## [1] "logical"
```

Vectors

A vector is a one-dimeniosnal array. We can create a vector with all the basic data type we learnt before.

The simplest way to build a vector in R, is to use the `c` command.

```
# Numerical store
numeric_vector <- c(1, 10, 49)
numeric_vector
```

```
## [1]  1 10 49
```

```
# Character store
character_vector <- c("a", "b", "c")
character_vector
```

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

```
# Boolean store
boolean_vector <-  c(TRUE, FALSE, TRUE)
boolean_vector
```

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

We can do arithmetic calculations on vectors.

```
# Create the vectors
vect_1 <- c(1, 3, 5)
vect_2 <- c(2, 4, 6)

# Take the sum of A_vector and B_vector
sum_vect <- vect_1 + vect_2

# Print out total_vector
sum_vect
```

```
## [1]  3  7 11
```

In R, it is possible to slice a vector. In some occasion, we are interested in only the first five rows of a vector. We can use the `[1:5]` command to extract the value 1 to 5.

```
# Slice the first five rows of the vector
slice_vector <- c(1,2,3,4,5,6,7,8,9,10)
slice_vector[1:5]
```

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

A shortest way to create a range of value is to use the `:` between two numbers. For instance, from the above example, we can write `c(1:10)` to create a vector of value from one to ten.

```
# Faster way to create adjacent values
c(1:10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

2.1.4 Logical Operators

With logical operators, we want to return values inside the vector based on logical conditions. Following is a detailed list of logical operators available in R

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x	y
x & y	x AND y
isTRUE(x)	Test if X is TRUE

Logical statement in R are wrapped inside the `[]`. We can add a many conditional statement as we like but we need to include them inside parenthesis. We can follow this structure to create a conditional statement:

```
variable_name[(conditional_statement)]
```

with `variable_name` refering to the variable we want to use for the statement. We create the logical statement i.e. `variable_name > 0`. Finally, we use the square bracket to finalize the logical statement. Below, an example of logical statement.

```
# Create a vector from 1 to 10
logical_vector <- c(1:10)
```

In the output above, R reads each value and compares it to the statement `logical_vector>5`. If the value is strickly superior to five, then the condition is `TRUE`, otherwise `FALSE`. R returns a vector of `TRUE` and `FALSE`.

In the example below, we want to extract the values that only meet the condition *is strictly superior to five*. For that, we can wrap the condition inside a square bracket precede by the vector containing the values.

```
# Print value stricly above 5
logical_vector[(logical_vector>5)]
```

```
## [1]  6  7  8  9 10
```

```
# Print 5 and 6
logical_vector[(logical_vector>4) & (logical_vector<7)]
```

```
## [1] 5 6
```

2.1.5 Matrix

A matrix is a 2 dimensional array that has m number of rows and n number of columns. In other words, matrix is a combination of two or more vectors with the same data type.

Note: It is possible to create more than two dimensions arrays with R.

Example of different matrix dimension

- $2 * 2$ matrix

2x2 matrix	column 1	column 2
row 1	1	2
row 2	3	4

- $3 * 3$ matrix

3x3 matrix	column 1	column 2	Column 3
row 1	1	2	3
row 2	4	5	6
row 3	7	8	9

- $5 * 2$ matrix

5x2 matrix	column 1	column 2
row 1	1	2
row 2	3	4
row 3	5	6
row 4	7	8
row 5	9	10

We can create a matrix with the function `matrix()`. This function takes three arguments:

```
matrix(data, nrow, ncol, byrow = FALSE)
Arguments:
```

- data: The collection of elements that R will arrange into the rows and columns of the matrix
- nrow: Number of rows
- ncol: Number of columns
- byrow: The rows are filled from the left to the right. We use `byrow = FALSE` (default values), if we

Let's construct a 5x2 matrix with a sequence of number from 1 to 10. We will create two separate matrix, one with `byrow =TRUE` and one with 'byrow = FALSE to see the difference

```
# Construct a matrix with 5 rows that contain the numbers 1 up to 10 and byrow =  TRUE
matrix_a <-matrix(1:10, byrow = TRUE, nrow = 5)
matrix_a
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
## [4,]    7    8
## [5,]    9   10
```

```
# Print dimension of the matrix with dim()
dim(matrix_a)
```

```
## [1] 5 2
```

```
# Construct a matrix with 5 rows that contain the numbers 1 up to 10 and byrow =  FALSE
matrix_a_1 <-matrix(1:10, byrow = FALSE, nrow = 5)
matrix_a_1
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```
# Print dimension of the matrix with dim()
dim(matrix_a_1)
```

```
## [1] 5 2
```

You can also create a 4x3 matrix using `ncol`.R will create 3 columns and fill the row from top to bottom. Check an example

```
matrix_a_2 <-matrix(1:12, byrow = FALSE, ncol = 3)
matrix_a_2
```

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

```
dim(matrix_a_2)
```

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

You can add a column to a matrix with the `cbind()` command. `cbind()` means column binding. `cbind()` concanates as many matrix or columns as specified. For example, our previous example created a 5x2 matrix. We concanate a third column and verify the dimension is 5x3.

```r
# Concanate c(1:5) to the matrix_a
matrix_b <- cbind(matrix_a, c(1:5))

# Check the dimension
dim(matrix_b)
```

```
## [1] 5 3
```

We can also add more than one column. Let's see the next sequence of number to the `matrix_a_2` matrix. The dimension of the number matrix will be 4x6 with number from 1 to 24.

```r
matrix_a_3 <-matrix(13:24, byrow = FALSE, ncol = 3)
matrix_a_4 <- cbind(matrix_a_2, matrix_a_3)
dim(matrix_a_4)
```

```
## [1] 4 6
```

Note: The number of rows of matrices should be equal for cbind work

`cbind()` concanates columns, `rbind()` appends rows. Let's add one row to our `matrix_b` matrix and verify the dimension is 6x3

```r
# Create a vector of 3 columns
add_row <- c(1:3)

# Append to the matrix
matrix_c <- rbind(matrix_b, add_row)

# Check the dimension
dim(matrix_c)
```

```
## [1] 6 3
```

We can select elements one or many elements from a matrix by using the square brackets `[ ]`. This is where slicing comes into picture.

For example:

- `matrix_c[1,2]` selects the element at the first row and second column.
- `matrix_c[1:3,2:3]` results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3,
- `matrix_c[,1]` selects all elements of the first column.
- `matrix_c[1,]` selects all elements of the first row.

Here is the output you get for the above codes

```r
library(dplyr)
data_frame <- tibble(
  c1 = rnorm(50, 5, 1.5),
  c2 = rnorm(50, 5, 1.5),
  c3 = rnorm(50, 5, 1.5),
  c4 = rnorm(50, 5, 1.5),
  c5 = rnorm(50, 5, 1.5)
)

# return the first value of the first column
data_frame[1:1]
```

```
## # A tibble: 50 x 1
##            c1
##         <dbl>
```

```
##  1  5.6871892
##  2  7.1322875
##  3  2.8822218
##  4  3.8014308
##  5  4.2437962
##  6  5.8238808
##  7  6.1352684
##  8  5.5091652
##  9  6.0366873
## 10 -0.2056563
## # ... with 40 more rows
```

```
# return a matrix with the values on the rows  and 2 and columns 1 and 2.
data_frame[1:2,1:2]
```

```
## # A tibble: 2 x 2
##         c1       c2
##      <dbl>    <dbl>
## 1 5.687189 5.159003
## 2 7.132287 6.304314
```

```
# return the first column with all values.
data_frame[,2]
```

```
## # A tibble: 50 x 1
##          c2
##       <dbl>
##  1 5.159003
##  2 6.304314
##  3 6.792387
##  4 2.460479
##  5 2.565325
##  6 4.216137
##  7 6.460138
##  8 6.724307
##  9 4.816046
## 10 3.286641
## # ... with 40 more rows
```

```
# return the first row.
data_frame[1,]
```

```
## # A tibble: 1 x 5
##         c1       c2       c3       c4       c5
##      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 5.687189 5.159003 5.298086 3.021427 4.650032
```

2.1.6 Categorical and Continuous types of variable

In a dataset, we can distinguish two types of variables: **categorical** and **continuous**.

- In a categorical variable, the value is limited, and usually based on a particular finite group. For example, a categorical variable can be countries, year, gender, occupation.
- A continuous variable, however, can take any values, from integer to decimal. For example, we can have the revenue, price of a share, etc..

Categorical variables

R stores categorical variables into a factor. Let's check the code below to convert a character variable into a factor variable. **Characters** are not supported in machine learning algorithm, and the only way is to convert a string to an integer.

```
factor(x = character(), levels, labels = levels, ordered = is.ordered(x))
Arguments:

- x: A vector of data. Need to be a string or integer, not decimal.
- levels: A vector of possible values taken by x. This argument is optional, and default value is the u
- labels: Add a label to the x data. For example, 1 can take the label `male` while 0, the label `femal
ordered: Determine if the levels should be ordered.
```

Let's create a factor dataframe.

```
# Create gender vector
gender_vector <- c("Male", "Female", "Female", "Male", "Male")
class(gender_vector)
```

```
## [1] "character"
```

```
# Convert gender_vector to a factor
factor_gender_vector <-factor(gender_vector)
class(factor_gender_vector)
```

```
## [1] "factor"
```

It is important to transform a **string** into factor when we perform Machine Learning task. A factor column allows R to transform the matrix into a **one hot encoder**. For instance, in a case of a two dimensions factor, R will create behind the hood 2 columns (i.e. if `male` then the matrix is `[1,0,0,1,1]`, if `female` then `[0,1,1,0,0]`)

There are two types of categorical variables: a **nominal categorical variable** and an **ordinal categorical variable**.

Nominal categorical variable

A categorical variable has several values but the order does not matter. For instance, male or female categorical variable do not have ordering.

```
# Create a color vector
color_vector <- c('blue', 'red', 'green', 'white', 'black', 'yellow')

# Convert the vector to factor
factor_color <- factor(color_vector)
factor_color
```

```
## [1] blue   red    green  white  black  yellow
## Levels: black blue green red white yellow
```

From the `factor_color`, we can't tell any order.

Ordinal categorical variable

Ordinal categorical variables do have a natural ordering. We can specify the order, from the lowest to the highest with `order = TRUE` and highest to lowest with `order = FALSE`.

We can use `summary` to count the values for each factor.

```
# Create Ordinal categorical vector
day_vector <- c('evening', 'morning', 'afternoon', 'midday', 'midnight', 'evening')

# Convert `day_vector` to a factor with ordered level
```

```
factor_day <- factor(day_vector,
                     order = TRUE,
                     levels =c('morning', 'midday', 'afternoon', 'evening', 'midnight'))

# Print the new variable
factor_day
```

```
## [1] evening   morning   afternoon midday    midnight  evening
## Levels: morning < midday < afternoon < evening < midnight
```

```
# Count the number of occurence of each level
summary(factor_day)
```

```
##   morning    midday afternoon   evening  midnight
##         1         1         1         2         1
```

R ordered the level from 'morning' to 'midnight' as specified in the `levels` parenthesis.

We can create an ordering level with label for the column male and female, with 1 refers to `female` and 0 for `male`.

```
set.seed(1)
# Create a random vector of 0 and 1
gender <- sample(0:1, 100, replace=T)

# Create a factor variable with the label option
factor_gender <- factor(gender,
                     ordered = TRUE,
                     labels =c('Male', 'Female'))
head(factor_gender)
```

```
## [1] Male   Male   Female Female Male   Female
## Levels: Male < Female
```

Continuous variables

Continuous class variables are the default value in R. They are stored as numeric or integer. We can see it from the dataset below. `mtcars` is a built in dataset. It gathers information on different types of car. We can import it by using `mtcars` and check the class of the variable `mpg`, mile per gallon. It returns a numeric values, indicating a continuous variable.

```
dataset <- mtcars
class(dataset$mpg)
```

```
## [1] "numeric"
```

2.1.7 Object in R

2.1.8 Data frame

A **data frame** is a list of vectors which are of equal length. A matrix contains only one type of data, while a data frame accepts different data types (numeric, character, factor, etc.).

Create a data frame

You can create a data frame by passing the variable `a,b,c,d` into the `data.frame()` function. you can name the columns with `name()` and simply specify the name of the variables.

```
data.frame(df), stringsAsFactors = TRUE)
arguments:

- df: It can be a matrix to convert as a data frame or a collection of variables to join
- stringsAsFactors: Convert string to factor by default
```

you can create our first data set by combining four variables of same length.

```
# Create a, b, c, d variables
a <- c(10,20,30,40)
b <- c('book', 'pen', 'textbook', 'pencil_case')
c <- c(TRUE,FALSE,TRUE,FALSE)
d <- c(2.5, 8, 10, 7)

# Join the variables to create a data frame
df <- data.frame(a,b,c,d)
df
```

```
##    a           b     c    d
## 1 10        book  TRUE  2.5
## 2 20         pen FALSE  8.0
## 3 30    textbook  TRUE 10.0
## 4 40 pencil_case FALSE  7.0
```

you can see the column headers have the same name as the variables. you can change the column name with the function `names()`. Check the example below:

```
# Name the data frame
names(df) <- c('ID', 'items', 'store', 'price')
df
```

```
##   ID       items store price
## 1 10        book  TRUE   2.5
## 2 20         pen FALSE   8.0
## 3 30    textbook  TRUE  10.0
## 4 40 pencil_case FALSE   7.0
```

```
# Print the structure
str(df)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ ID   : num  10 20 30 40
##  $ items: Factor w/ 4 levels "book","pen","pencil_case",..: 1 2 4 3
##  $ store: logi  TRUE FALSE TRUE FALSE
##  $ price: num  2.5 8 10 7
```

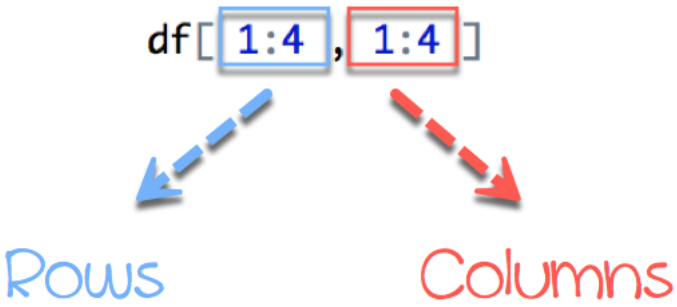
Note, by default, data frame return string variables as factor.

Slice a factor

It is possible to **slice** values of a data frame. you select the rows and columns to return into bracket precede by the name of the data frame.

A data frame is composed by rows and columns, `df[A, B]`. `A` represents the rows and `B` the columns. you can slice either by specifying which rows or columns.

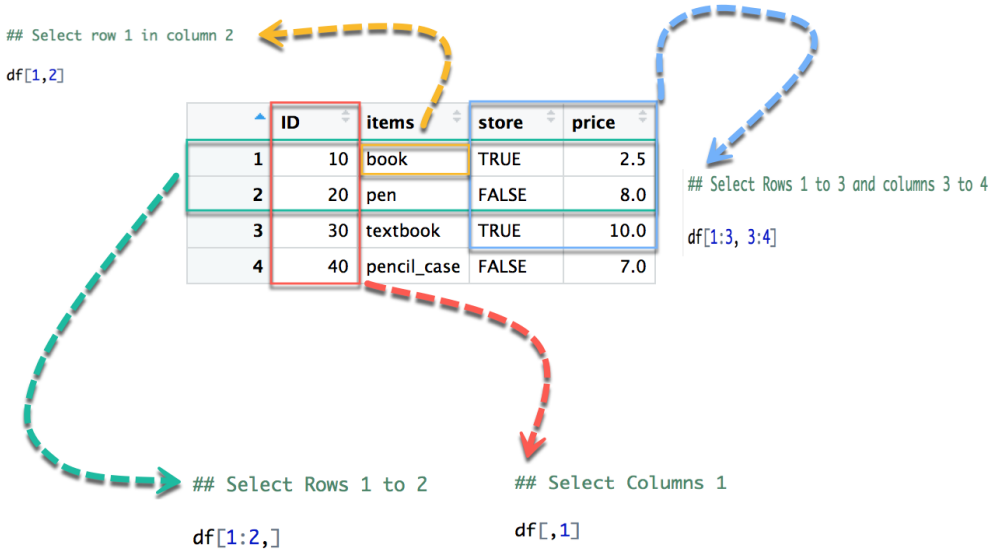
From picture 1, the left part represents the **rows** and the right part is the **columns**. Note that the symbol `:` means **to**. For instance, `1:3` intends to select values from 1 **to** 3.



In below diagram you display how to access different selection of the data frame:

- The yellow arrow selects the **row** 1 in **column** 2
- The green arrow selects the **rows** 1 to 2
- The red arrow selects the **column** 1
- The blue arrow selects the **rows** 1 to 3 and **columns** 3 to 4

Note that, if you let the left part blank, R will select **all the rows**. By analogy, if you let the right part blank, R will select **all the columns**.



you can run the code in the console:

```
## Select row 1 in column 2

df[1,2]
```

```
## [1] book
## Levels: book pen pencil_case textbook
```

```
## Select Rows 1 to 2

df[1:2,]
```

```
##   ID items store price
## 1 10  book  TRUE   2.5
## 2 20   pen FALSE   8.0
```

```
## Select Columns 1

df[,1]
```

```
## [1] 10 20 30 40
```

```
## Select Rows 1 to 3 and columns 3 to 4

df[1:3, 3:4]
```

```
##   store price
## 1  TRUE   2.5
## 2 FALSE   8.0
## 3  TRUE  10.0
```

It is also possible to select the columns with their names. For instance, the code below extracts two columns: `ID` and `store`.

```
# Slice with columns name
df[, c('ID', 'store')]
```

```
##   ID store
## 1 10  TRUE
## 2 20 FALSE
## 3 30  TRUE
## 4 40 FALSE
```

Append a Column to Data Frame

You can also append a column to a Data Frame. You need to use symbol '$ to append a new variable.

```
# Create a new vector
quantity <- c(10, 35, 40, 5)

# Add `quantity` to the `df` data frame
df$quantity <- quantity

df
```

```
##   ID       items store price quantity
## 1 10        book  TRUE   2.5       10
## 2 20         pen FALSE   8.0       35
## 3 30    textbook  TRUE  10.0       40
## 4 40 pencil_case FALSE   7.0        5
```

Note: The number of elements in the vector has to be equal to the no of elements in data frame. Executing the following statement

```
quantity <- c(10, 35, 40)
# Add `quantity` to the `df` data frame
df$quantity <- quantity
```

Select a column of a data frame

Sometimes, we need to store a column of a data frame for future use or perform operation on a column. We can use the $ sign to select the column from a data frame.

```
# Select the column ID
select_id <- df$ID
```

Subset a data frame

In the previous section, we selected an entire column without condition. It is possible to **subset** based on whether or not a certain condition was true.

You use the `subset()` function.

```
subset(x, condition)
arguments:

- x: data frame used to perform the subset
- condition: define the conditional statement
```

You want to return only the items with `quantity` above 10, you can do :

```
# Select quantity above 10
subset(df, subset = quantity > 10)
```

```
##   ID    items store price quantity
## 2 20      pen FALSE     8       35
## 3 30 textbook  TRUE    10       40
```

2.1.9 List

A **list** is a great tool to store many kinds of object in the order expected. We can include matrices, vectors data frames or lists. We can imagine a list as a bag in which we want to put many different items. When we need to use an item, we open the bag and use it. A list is similar, we can store a collection of objects and use them when we need them.

You can use 'list() function to create a list.

```
list(element_1, ...)
arguments:

- element_1: store any type of R object
- ...: pass as many object as specifying. each object needs to be separated by a comma
```

In the example below, you create three different objects, a vector, a matrix and a data frame.

```
# Vector with numerics from 1 up to 5
vect  <- 1:5

# A 2x 5 matrix
mat  <- matrix(1:9, ncol = 5)
dim(mat)
```

```
## [1] 2 5
```

```
# select the 10th row of the built-in R data set EuStockMarkets
df <- EuStockMarkets[1:10,]
```

Now, you can put the three objects into a list.

```
# Construct list with these vec, mat and df:
my_list <- list(vect, mat, df)
my_list
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8    1
##
## [[3]]
```

```
##           DAX    SMI    CAC   FTSE
##  [1,] 1628.75 1678.1 1772.8 2443.6
##  [2,] 1613.63 1688.5 1750.5 2460.2
##  [3,] 1606.51 1678.6 1718.0 2448.2
##  [4,] 1621.04 1684.1 1708.1 2470.4
##  [5,] 1618.16 1686.6 1723.1 2484.7
##  [6,] 1610.61 1671.6 1714.3 2466.8
##  [7,] 1630.75 1682.9 1734.5 2487.9
##  [8,] 1640.17 1703.6 1757.4 2508.4
##  [9,] 1635.47 1697.5 1754.0 2510.5
## [10,] 1645.89 1716.3 1754.3 2497.4
```

Select elements from list

After we built our list, we can access it quite easily. We need to use the '[[index]] to select an element in a list. The value inside the double square bracket represents the position of the item in a list we want to extract. For instance, we pass 2 inside the parenthesis, R returns the second element listed.

Let's try to select the second items of the list named `my_list`, we use `my_list[[2]]`.

```
# Print second element of the list
my_list[[2]]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8    1
```

Build-in data frame

Before to create our own data frame, we can have a look at the R data set available online. The prison dataset is a 714x5 dimension. We can get a quick look at the top of the data frame with `head()` function. By analogy, `tail()` displays the bottom of the data frame. You can specify the number of rows shown with `head (df, 5)`. We will learn more about the function 'read.csv() in future tutorial.

```
# Print the head of the data
library(dplyr)
PATH <- 'https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/prison.csv'
df <- read.csv(PATH)[1:5]
head(df, 5)
```

```
##   X state year govelec  black
## 1 1     1   80       0 0.2560
## 2 2     1   81       0 0.2557
## 3 3     1   82       1 0.2554
## 4 4     1   83       0 0.2551
## 5 5     1   84       0 0.2548
```

you can check the structure of the data frame with `str`:

```
# Structure of the data
str(df)
```

```
## 'data.frame':    714 obs. of  5 variables:
##  $ X      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ state  : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ year   : int  80 81 82 83 84 85 86 87 88 89 ...
##  $ govelec: int  0 0 1 0 0 0 1 0 0 0 ...
##  $ black  : num  0.256 0.256 0.255 0.255 0.255 ...
```

All variables are stored in the *numerical* format.

2.2 Import data with R

Data could exist in various formats. For each format R has a specific function and argument. This tutorial explains how to import data to R.

This chapter is divided as follow:

- Read CSV files
- Read Excel files
- Read SAS, SPSS, STATA files
- Read RDA files

2.2.1 Read CSV

One of the most widely data store is the `.csv` (comma-separated values) file formats. R loads an array of libraries during the start-up, including the `utils` package. This package is convenient to open csv files combined with the `reading.csv()` function. Here is the syntax for read.csv:

```
read.csv(file, header = TRUE, sep = ",")
argument:

-file: PATH where the file is stored
- header: confirm if the file has an header or not, by default, the header is set to TRUE
- sep: the symbol used to split the variable.By default, `,`.
```

We will read the data file name `mtcats`. The csv file is stored online. If your `.csv` file is stored locally, you can replace the PATH inside the code snippet. Don't forget to wrap it inside ''. The PATH needs to be a string value.

For mac user, the path for the download folder is:

- `"/Users/USERNAME/Downloads/FILENAME.csv"`

For windows user:

- `"C:\Users\USERNAME\Downloads\FILENAME.csv"` Note that, you should always specify the extension of the file name.
- `.csv`
- `.xlsx`
- `.txt`
- ...

```
PATH <- 'https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/mtcars.csv'

df <- read.csv(PATH, header =  TRUE, sep = ',')
length(df)
```

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

```
class(df$X)
```

```
## [1] "factor"
```

R, by default, returns character values as `Factor`, you can turn off this setting by adding `stringsAsFactors = FALSE`.

```
df <- read.csv(PATH, header =  TRUE, sep = ',', stringsAsFactors = FALSE)
class(df$X)
```

```
## [1] "character"
```

The class for the variable `X` is now a `character`.

2.2.2 Read Excel files

Excel files are very popular among data analysts. Spreadsheets are easy to work with and flexible. R is equipped with a library readxl to import Excel spreadsheet.

Use this code

```
require(readxl)
```

```
## Loading required package: readxl
```

to check if `readxl` is installed in your machine. If you install `r` with `r-conda-essential`, the library is already installed. You should see in the command window:

```
Loading required package: readxl.
```

If the package does not exit, you can install it with the conda

- `readxl`: Open excel spreadsheet. If you have install R with `r-essential`. It is already in the library
  - Anaconda: `conda install -c mittner r-readxl`

Use the following command to load the library to import excel files.

```
library(readxl)
```

We use the examples included in the package readxl during this tutorial. Use code

```
readxl_example()
```

```
##  [1] "clippy.xls"    "clippy.xlsx"   "datasets.xls"  "datasets.xlsx"
##  [5] "deaths.xls"    "deaths.xlsx"   "geometry.xls"  "geometry.xlsx"
##  [9] "type-me.xls"   "type-me.xlsx"
```

to see all the available spreadsheets in the libray.

To check the location of the spreadsheet named clippy.xls, simple use

```
readxl_example("geometry.xls")
```

```
## [1] "/Users/Thomas/anaconda3/lib/R/library/readxl/extdata/geometry.xls"
```

If you install R with `conda`, the spreadsheets are located in `anaconda3/lib/R/library/readxl/extdata/geometry.xls`

The function `read_excel()` is of great use when it comes to opening xls and xlsx extention.

The syntax is:

```
read_excel(PATH, sheet = NULL, range= NULL, col_names = TRUE)
arguments:

- PATH: Path where the excel is located
- sheet: Select the sheet to import. By default, all
- range: Select the range to import. By default, all non-null cells
- col_names: Select the columns to import. By default, all non-null columns
```

We can import the spreadsheets from the `readxl` library and count the number of column in the first sheet.

```
# Store the path of `datasets.xlsx`
example <- readxl_example("datasets.xlsx")
# Import the spreadsheet
```

```
df <- read_excel(example)
# Count the number of columns
length(df)
```

```
## [1] 5
```

The file `datasets.xlsx` is composed of 4 sheets. We can find out which sheets are available in the workbook by using `excel_sheets()` function:

```
example <- readxl_example("datasets.xlsx")
excel_sheets(example)
```

```
## [1] "iris"     "mtcars"   "chickwts" "quakes"
```

If a worksheet includes many sheets, it is easy to select one by using the `sheet` arguments. You can specify the name of the sheet or the index. You can verify if both function returns the same output with `identical()`.

```
quake <- read_excel(example, sheet = "quakes", col_names = TRUE)
quake_1 <-read_excel(example, sheet = 4, col_names = TRUE)
identical(quake, quake_1)
```

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

You can control what cells to read in 2 ways:

1. Use `n_max` argument to return `n` rows
2. use `range`argument combined with `cell_rows` or `cell_cols`

You can use the argument `n_max` with the number of rows to import in R. For instance, you set `n_max` equals to 5 to import the first five rows. Note, if your spreadsheet does not have header, change `TRUE` to `FALSE` in the `col_names` argument.

```
# Read the first five row
iris <- read_excel(example, n_max = 5, col_names = TRUE)
```

Sepal.Length <dbl>	Sepal.Width <dbl>	Petal.Length <dbl>	Petal.Width <dbl>	Species <chr>
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa

5 rows

If you change `col_names` to `FALSE`, R creates the headers automaticaly.

```
# Read the first five row
iris_no_header <- read_excel(example, n_max = 5, col_names = FALSE)
```

In the data frame `iris_no_header`, R created five new variables named `X__1`, `X__2`, `X__3`, `X__4` and `X__5`

X__1 <chr>	X__2 <chr>	X__3 <chr>	X__4 <chr>	X__5 <chr>
Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa

5 rows

You can also use the argument `range` to select rows and columns in the spreadsheet. In the code below, you use the excel style to select the range A1 to B5.

```
# Read rows A1 to B5
example_1 <- read_excel(example, range = "A1:B5", col_names = TRUE)
dim(example_1)
```

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

You can see that the `example_1`returns 4 rows with 2 columns. The dataset has header, that the reason the dimension is 4x2.

In the second example, you use the function `cell_rows()` which controls the range of rows to return. If you want to import the rows 1 to 5, you can set `cell_rows(1:5)`. Note that, `cell_rows(1:5)` returns the same output as `cell_rows(5:1)`.

```
# Read 5 rows of all columns
example_2 <- read_excel(example, range = cell_rows(1:5),  col_names = TRUE)
dim(example_2)
```

```
## [1] 4 5
```

The `example_2` however is a 4x5 matrix. The iris dataset has 5 columns with header. We return the first four rows with header of all columns.

In case you want to import rows which do not begin in the first row, you have to include `col_names = FALSE`. If you use `range = cell_rows(2:5)`, it becomes obvious our data frame does not have header anymore.

```
iris_row_with_header <- read_excel(example, range = cell_rows(2:3), col_names = TRUE)
iris_row_no_header <- read_excel(example, range = cell_rows(2:3), col_names = FALSE)
```

```
> read_excel(example, range = cell_rows(2:3), col_names = TRUE)
# A tibble: 1 x 5
  `5.1` `3.5` `1.4` `0.2` setosa   <---- Wrong columns name
  <dbl> <dbl> <dbl> <dbl>  <chr>
1   4.9     3   1.4   0.2 setosa
> read_excel(example, range = cell_rows(2:3), col_names = FALSE)
# A tibble: 2 x 5
  X__1  X__2  X__3  X__4   X__5   <---- Correct columns name
  <dbl> <dbl> <dbl> <dbl>  <chr>
1   5.1   3.5   1.4   0.2 setosa
2   4.9   3.0   1.4   0.2 setosa
```

We can select the columns with the letter, like in Excel.

```
# Select columns 1 and B
col <- read_excel(example, range = cell_cols("A:B"))
dim(col)
```

```
## [1] 150   2
```

Note: `range = cell_cols("A:B")`, returns output all cells with non-null value. The dataset contains 150 rows, therefore, `read_excel()` returns rows up to 150. This is verified with the 'dim() function.

`read_excel()` returns `NA` when a symbol without numerical value appears in the cell. You can count the number of missing values with the combination of two functions:

1. `sum()`
2. 'is.na

Here is the code

```
iris_na <- read_excel(example, na = "setosa")
sum(is.na(iris_na))
```

```
## [1] 50
```

We have 50 values missing, which are the rows belonging to the `setosa` species.

2.2.3 Import data from other Statistical software

You will import different files format with the `heaven`package. This package support SAS, STATA and SPSS softwares. We can use the folloing function to open different types of dataset, according to the extension of the file:

- SAS: `read_sas()`
- STATA: `read_dta()` (or `read_stata()`, which are identical)
- SPSS: `read_sav()` or `read_por()`. We need to check the extension

Only one argument is required within these function. You need to know the PATH where the file is stored and that's it, you are ready to open all the files from SAS, STATA and SPSS. These three function accepts an URL with the same extensions.

```
library(haven)
```

- `haven`: Import/export data with format differetn from csv. If you have install R with `r-essential`. It is already in the library
  - Anaconda: `conda install -c conda-forge r-haven`

Read SAS

For our example, you are going to use the `admission` dataset.

```
PATH_sas <- 'https://github.com/thomaspernet/data_csv_r/blob/master/data/data_sas.sas7bdat?raw=true'

df <- read_sas(PATH_sas)
head(df)
```

```
## # A tibble: 6 x 4
##   ADMIT   GRE   GPA  RANK
##   <dbl> <dbl> <dbl> <dbl>
## 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
```

Read STATA

Next up are STATA data files; you can use read_dta() for these. You use exactly the same dataset but store in `.dta` file.

```
PATH_stata <- 'https://github.com/thomaspernet/data_csv_r/blob/master/data/stata.dta?raw=true'

df <- read_dta(PATH_stata)
head(df)
```

```
## # A tibble: 6 x 4
##   admit   gre   gpa  rank
##   <dbl> <dbl> <dbl> <dbl>
```

```
## 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
```

Read SPSS

You use the `read_sav()`function to open a SPSS file. There is no difficulties in the task.

```
PATH_spss <- 'https://github.com/thomaspernet/data_csv_r/blob/master/data/spss.sav?raw=true'

df <- read_sav(PATH_spss)
head(df)
```

```
## # A tibble: 6 x 4
##   admit   gre   gpa  rank
##   <dbl> <dbl> <dbl> <dbl>
## 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
```

2.2.4 Best practices Data Import

When you want to import data into R, it is useful to implement following checklist. It will make it easy to import data correctly into R:

- The typical format for a spreadsheet is to use the first rows as the header (usually variables name).
- Avoid to name a dataset with blank spaces; it can lead to interpreting as a separate variable. Alternatively, prefer to use _ or '-.
- Short names are preferred
- Do not include symbol in the name: i.e: `exchange_rate_$_eur` is not correct. Prefer to name it: `exchange_rate_dollar_eur`
- Use `NA` for missing values otherwise; you need to clean the format later.

2.2.5 Summary

Following table summarizes the function to use in order to import different types of file in R. The column one states the library related to the function. The last column refers to the default argument.

Library	Objective	Function	Default Arguments
utils	Read CSV file	read.csv()	file, header =,TRUE, sep = ","
readxl	Read EXCEL file	read_excel()	path, range = NULL, col_names = TRUE
haven	Read SAS file	read_sas()	path
haven	Read STATA file	read_stata()	path
haven	Read SPSS fille	read_sav()	path

Following table shows the different ways to import a selection with 'read_excel() function.

Function	Objectives	Arguments
read_excel()	Read n number of rows	n_max = 10
	Select rows and columns like in excel	range = "A1:D10"
	Select rows with indexes	range= cell_rows(1:3)
	Select columns with letters	range = cell_cols("A:C")

2.3 Data Manipulation

2.3.1 Merge data

Very often, you have data from multiple sources. To perform an analysis, you need to **merge** two data frames together with one or more **common key variables**.

Full match

A full match returns values that have a counterpart in the destination table. The values that are not match won't be return in the new data frame. The partial match, however, return the missing values as `NA`.

You will see a simple **inner join**. The inner join keyword selects records that have matching values in both tables. To join two datasets, you can use `merge()` function. You will use three arguments :

```
merge(x, y, by.x = bx, by.y = y)
Arguments:

- x: The origin data frame
- y: The data frame to merge
- by.x: The column used for merging in x data frame. Column x to merge on
- by.y: The column used for merging in y data frame. Column y to merge on
```

Example: Create First Dataset with variables

- surname
- nationality

Create Second Dataset with variables

- surname
- movies

The common key variable is surname. You can merge both data and check if the dimensionality is 7x3.

You add `stringsAsFactors=FALSE` in the data frame because you don't want R to convert string as factor, you want the variable to be treated as character.

```
# Create origin dataframe
producers <- data.frame(
    surname = c("Spielberg", "Scorsese", "Hitchcock", "Tarantino", "Polanski"),
    nationality = c("US", "US", "UK", "US", "Poland"),    stringsAsFactors=FALSE)

# Create destination dataframe
movies <- data.frame(
    surname = c("Spielberg", "Scorsese", "Hitchcock",
             "Hitchcock", "Spielberg", "Tarantino", "Polanski"),
    title = c("Super 8",
              "Taxi Driver",
              "Psycho",
              "North by Northwest",
```

```
              "Catch Me If You Can",
              "Reservoir Dogs",
              "Chinatown"),
               stringsAsFactors=FALSE)

# Merge two datasets
m1 <- merge(producers, movies, by.x = "surname")
dim(m1)
```

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

Let's merge data frames when the common key variables have different names.

You change surname to name in the movies data frame. You use the function identical(x1, x2) to check if both dataframes are identical.

```
# Change name of `books` dataframe
colnames(movies)[colnames(movies) == 'surname'] <- 'name'

# Merge with different key value
m2 <- merge(producers, movies, by.x = "surname", by.y = "name")

# Print head of the data
head(m2)
```

```
##     surname nationality               title
## 1 Hitchcock          UK              Psycho
## 2 Hitchcock          UK  North by Northwest
## 3  Polanski      Poland           Chinatown
## 4  Scorsese          US         Taxi Driver
## 5 Spielberg          US             Super 8
## 6 Spielberg          US Catch Me If You Can
```

```
# Check if data are identical
identical(m1, m2)
```

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

Partial match

It is not surprising that two data frames do not have the same common key variables. In the **full matching**, the data frame returns **only** rows found in both x and y data frame. With **partial merging**, it is possible to keep the rows with no matching rows in the other data frame. These rows will have `NA`s in those columns that are usually filled with values from y. You can do that by setting `all.x= TRUE`.

For instance, you can add a new producer, Lucas, in the producer data frame without the movie references in movies data frame. If you set `all.x= FALSE`, R will join only the matching values in both data set. In your case, the producer Lucas will not be join to the merge because it is missing from one dataset.

Let's see the dimension of each output when you specify `all.x= TRUE` and when you don't.

```
# Create a new producer
add_producer <-  c('Lucas', 'US')

#  Append it to the `producer` dataframe
producers <- rbind(producers, add_producer)

# Use a partial merge
```

```
m3 <-merge(producers, movies, by.x = "surname", by.y = "name", all.x = TRUE)
m3
```

```
##     surname nationality               title
## 1 Hitchcock          UK              Psycho
## 2 Hitchcock          UK  North by Northwest
## 3     Lucas          US                <NA>
## 4  Polanski      Poland           Chinatown
## 5  Scorsese          US         Taxi Driver
## 6 Spielberg          US             Super 8
## 7 Spielberg          US Catch Me If You Can
## 8 Tarantino          US      Reservoir Dogs
```

```
# Use a full merge
m4 <-merge(producers, movies, by.x = "surname", by.y = "name", all.x = FALSE)
m4
```

```
##     surname nationality               title
## 1 Hitchcock          UK              Psycho
## 2 Hitchcock          UK  North by Northwest
## 3  Polanski      Poland           Chinatown
## 4  Scorsese          US         Taxi Driver
## 5 Spielberg          US             Super 8
## 6 Spielberg          US Catch Me If You Can
## 7 Tarantino          US      Reservoir Dogs
```

```
# Compare the dimension of each data frame
dim(m1)
```

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

```
dim(m2)
```

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

```
dim(m3)
```

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

```
dim(m4)
```

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

As you can see, the dimension of the new data frame 8x3 compare with 7x3 for `m1` and `m2`. R includes `NA` for the missing producer in the `producer` data frame.

2.3.2 Sort a data frame

In data analysis you can **sort** your data according to a certain variable in the dataset. In R, we can use the help of the function `order()`. In R, we can easily sort a vector of continuous variable or factor variable. Arranging the data can be of **ascending** or **descending** order. The syntax is:

```
sort(x, decreasing = FALSE, na.last = TRUE):
Argument:

- x: A vector containing continuous or factor variable
- decreasing: Control for the order of the sort method. By default, decreasing is set to  `FALSE`.
- na.last: Indicates whether the `NA` 's value should be put last or not
```

For instance, we can create a **tibble** data frame and sort one or multiple variables. A tibble data frame is a new approach to data frame. It improves the syntax of data frame and avoid frustrating data type formatting, especially for character to factor. It is also a convenient way to create a data frame by hand, which is our purpose here. To learn more about tibble, please refer to the vignette

```
library(dplyr)
data_frame <- tibble(
  c1 = rnorm(50, 5, 1.5),
  c2 = rnorm(50, 5, 1.5),
  c3 = rnorm(50, 5, 1.5),
  c4 = rnorm(50, 5, 1.5),
  c5 = rnorm(50, 5, 1.5)
)

# Sort by c1
df <-data_frame[order(data_frame$c1),]

# Sort by c3 and c4
df <-data_frame[order(data_frame$c3, data_frame$c4),]
head(df)
```

```
## # A tibble: 6 x 5
##         c1       c2       c3       c4       c5
##      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 7.149536 4.018123 2.768810 1.572147 6.043326
## 2 2.714650 2.722409 2.799125 5.855761 3.948152
## 3 7.198332 4.581330 2.863258 3.120065 3.216330
## 4 5.854579 5.576278 2.923360 4.979901 5.562087
## 5 4.146997 5.090241 3.153015 3.521260 5.597195
## 6 7.380250 3.326120 3.187876 6.744447 7.667644
```

```
# Sort by c3(descending) and c4(acending)
df <-data_frame[order(-data_frame$c3, data_frame$c4),]
head(df)
```

```
## # A tibble: 6 x 5
##         c1       c2       c3       c4       c5
##      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 5.713264 4.241064 8.461968 2.400172 4.611601
## 2 5.283188 4.410788 8.309154 7.279618 5.405082
## 3 5.001658 4.943549 8.112868 2.591730 6.111502
## 4 4.797418 7.523264 7.803936 5.765163 4.362098
## 5 6.594650 4.548536 7.196881 5.841231 3.997732
## 6 3.085112 8.130750 7.161737 4.182814 6.148398
```

3 Introduction to programming

3.1 Function in R

A **function**, in a programming environment, is a set of instructions. A programmer builds a function to avoid **repeating** the same task, or reduce **complexity**.

A function should be

- written to carry out a specified a tasks

- may or may not include arguments
- contain a body
- may or may not return one or more values.

In this chapter, we will learn how to use built-in function and how to create our own function. The roadmap is:

- R important built-in function
- Write function in R
- Environment scoping

A general approach of function is to use the argument part as **inputs**, feed the **body** part and finaly return an **output**. The Syntax of a function is the following:

```
function (arglist)  {
  #Function body
}
```

3.1.1 R important buil-in functions

There are a lot of built-in function in R. R matches your input parameters with its function arguments, either by value or by position, then executes the function body. Function arguments can have default values: if you do not specify these arguments, R will take the default value.

```
sort
```

```
## function (x, decreasing = FALSE, ...)
## {
##     if (!is.logical(decreasing) || length(decreasing) != 1L)
##         stop("'decreasing' must be a length-1 logical vector.\nDid you intend to set 'partial'?")
##     UseMethod("sort")
## }
## <bytecode: 0x7fd0f89cd750>
## <environment: namespace:base>
```

note: It is possible to see the source code of a function by running the name of the function itself in the console.

We will see three groups of function in action

- General function
- Math's function
- Statistical function

3.1.2 General functions

We are already familiar with `cbind()`, `rbind()`, `range()`, `sort()`, `order()` functions. Each of these functions has a specific task, takes arguments to return an output.

Following are important functions one must know:

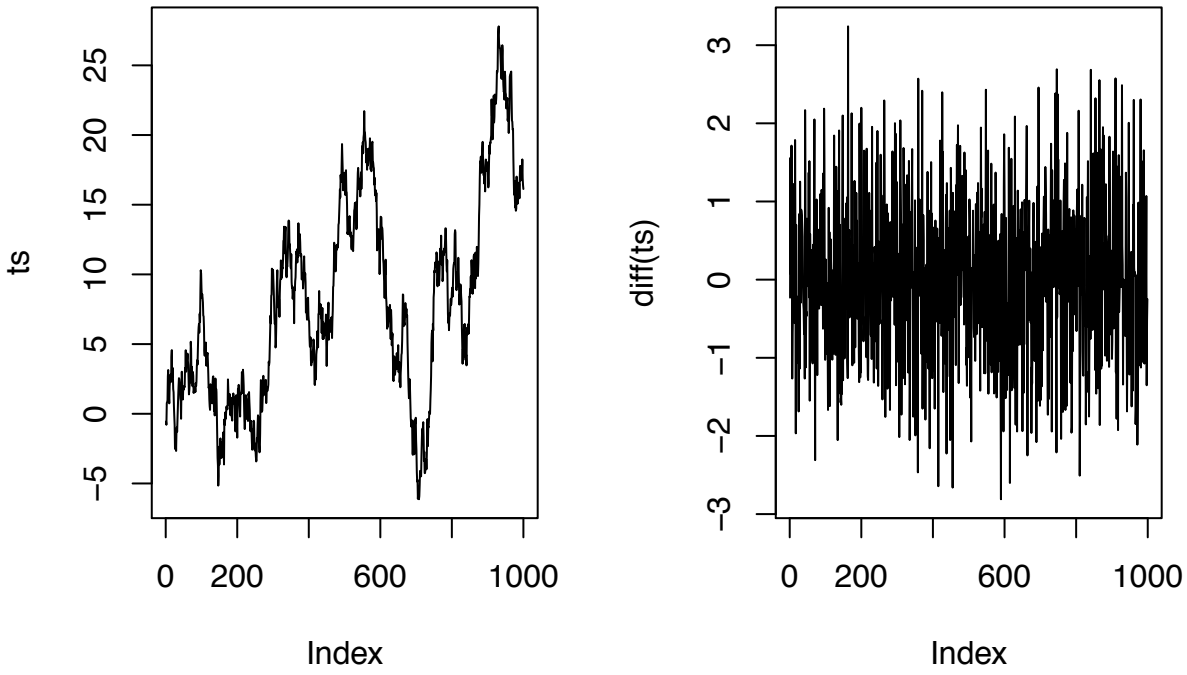
the function `diff()`

If you work on **time serie**s, you need to stationary the series by taking their lag values. A **stationary process** allows constant mean, variance and autocorrelation over time. This mainly improves the prediction of a time series. It can be easily done with the function `diff()`. You can build a random time-series data with a trend and then use the function `diff()` to stationary the series. The 'diff() function accepts one argument, a vector, and return suitable lagged and iterated difference.

the function `length()`

note: We often need to create random data, but for learning and comparison we want the numbers to be identical across machines. To ensure we all generate the same data, we use the `set.seed()` function with arbitrary values of 123. The `set.seed()` function is generated through the process of pseudorandom number generator that make every modern computers to have the same sequence of numbers. If we don't use set.seed() function, we will all have different sequence of numbers.

```
set.seed(123)
## Create the data
x = rnorm(1000)
ts <- cumsum(x)
## Stationary the serie
diff_ts <- diff(ts)
par(mfrow=c(1,2))
## Plot the series
plot(ts, type='l')
plot(diff(ts), type='l')
```



In many cases, you want to know the **length** of a vector to make computation or in `for` loop. The `length()` function counts the number of rows in vector x. The following codes import the `cars` dataset and return the number of rows.

note: `length()` returns the number of elements in a vector. If the function is passed into a matrix or a data frame, the number of columns is returned.

```
dt <- cars
## number columns
length(df)
```

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

```
## number rows
length(dt[,1])
```

```
## [1] 50
```

3.1.3 Math functions

R integrates an array of mathematical functions.

Operator	Description
abs(x)	Takes the absolute value of x
log(x,base=y)	Takes the logarithm of x with base y; if baseis not specified, returns the natural logarithm
exp(x)	Returns the exponential of x
sqrt(x)	Returns the square root of x
factorial(x)	Returns the factorial of x (x!)

```
# Create a sequence of number from45 to 55
x_vector <- seq(45,55, by = 1)
#logarithm
log(x_vector)
```

```
##  [1] 3.806662 3.828641 3.850148 3.871201 3.891820 3.912023 3.931826
##  [8] 3.951244 3.970292 3.988984 4.007333
```

```
#exponential
exp(x_vector)
```

```
##  [1] 3.493427e+19 9.496119e+19 2.581313e+20 7.016736e+20 1.907347e+21
##  [6] 5.184706e+21 1.409349e+22 3.831008e+22 1.041376e+23 2.830753e+23
## [11] 7.694785e+23
```

```
#squared root
sqrt(x_vector)
```

```
##  [1] 6.708204 6.782330 6.855655 6.928203 7.000000 7.071068 7.141428
##  [8] 7.211103 7.280110 7.348469 7.416198
```

```
#factorial
factorial(x_vector)
```

```
##  [1] 1.196222e+56 5.502622e+57 2.586232e+59 1.241392e+61 6.082819e+62
##  [6] 3.041409e+64 1.551119e+66 8.065818e+67 4.274883e+69 2.308437e+71
## [11] 1.269640e+73
```

3.1.4 Statistical functions

R standard installation contains wide range of statistical functions. In this tutorial, we will briefly look at the most important function.

Operator	Description
mean(x)	Mean of x
median(x)	Median of x
var(x)	Variance of x
sd(x)	Standard deviation of x
scale(x)	Standard scores (z-scores) of x
quantile(x)	The quartiles of x
sumary(x)	Summary of x: mean, min, max etc..

```r
speed <- dt$speed
# Mean speed of cars dataset
mean(speed)
```

```
## [1] 15.4
```

```r
# Median speed of cars dataset
median(speed)
```

```
## [1] 15
```

```r
# Variance speed of cars dataset
var(speed)
```

```
## [1] 27.95918
```

```r
# Standard deviation speed of cars dataset
sd(speed)
```

```
## [1] 5.287644
```

```r
# Standardise vector speed of cars dataset
head(scale(speed), 5)
```

```
##           [,1]
## [1,] -2.155969
## [2,] -2.155969
## [3,] -1.588609
## [4,] -1.588609
## [5,] -1.399489
```

```r
# Quantile speed of cars dataset
quantile(speed)
```

```
##   0%  25%  50%  75% 100%
##    4   12   15   19   25
```

```r
# Summary speed of cars dataset
summary(speed)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     4.0    12.0    15.0    15.4    19.0    25.0
```

Up to this point, we have learned a lot of R built-in functions.

note: Be careful with the class of the argument, i.e. numeric, Boolean or string. For instance, if we need to pass a string value, we need to enclose the string in quotation mark: "ABC"" .

It is common practice to store the values of a function for future use. For instance, we can store the residuals of a simple linear model to check the normality hypothesis.

3.1.5 Write function in R

In some occasion, you need to write your own function because you have to accomplish a particular task and no function exists. A user-defined function involves a **name**, **arguments** and a **body**.

```
function.name <- function(arguments)
{
  computations on the arguments
  some other code
```

```
}
```

You define the function similarly to variables, by **assigning** the directive **function(arguments)** to the **variable**, **function.name**, followed by the rest.

In the next session, we will go through step by step to create formulas with one or multiple arguments.

note: A good practice is to name a user-defined function different from a built-in function. It avoids confusion.

3.1.6 One argument function

In the next snippet, you define a simple square function. The function passes a value and returns the square.

```
square_function<- function(n)
{
  # compute the square of integer `n`
  n^2
}
square_function(4)
```

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

Code Explanation:

- The function is named square_function; it can be called whatever we want.
- It receives an argument `n`. We didn't specify the type of variable so that the user can pass an integer, a vector or a matrix
- The function takes the input `n` and returns the square of the input.

When you are done using the function, you can remove it with the `rm()` function.

```
rm(square_function)
square_function
```

On the console, we can see an error message :`Error: object 'square_function' not found` telling the function does not exist.

3.1.7 Environment scoping

In R, the **environment** is a **collection** of objects like functions, variables, data frame, etc.

R opens an environment each time Rstudio is prompted.

The top-level environment available is the **global environment**, called `R_GlobalEnv`. And there is the **local environment**.

You can list the content of the current environment.

```
# Only the first 10th elements are listed
ls(environment())[1:10]
```

```
##  [1] "diff_ts"         "dt"              "speed"
##  [4] "square_function" "ts"              "x"
##  [7] "x_vector"        NA                NA
## [10] NA
```

You can see all the variables and function created in the `R_GlobalEnv`.

note: the argument `n` of the function `square_function` is **not in this global environment**.

A **new** environment is created for each function. In the above example, the function `square_function()` creates a new environment inside the global environment.

To clarify the difference between **global** and **local environment**, you create two functions which return a similar output. These function takes a value `x` as an argument and add it to `y` define *outside* and *inside* the function

```
y <- 10
f <- function(x) {
  x + y
}

f(5)
```

```
## [1] 15
```

```
y
```

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

```
rm(y)
```

The function `f` returns the output 15; this is because `y` is defined in the global environment. Any variable defined in the global environment can be used locally. The variable `y` has the value of 10 during all scripts and can be accessible at any time.

Let's see what happens if the variable `y` is defined inside the function.

We dropped `y` prior to run this code using `rm(r)`.

```
f <- function(x) {
  y <- 10
  x + y
}
f(5)
y
```

The output is also 15 when you call `f(5)` but returns an error when you try to print the value `y`. The variable `y` is not in the global environment.

Finally, R uses the most recent variable definition to pass inside the body of a function. Let's consider the following example:

```
y <- 2
f <- function(x) {
  y <- 4
  x + y
}

f(5)
```

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

R ignores the `y` values defined outside the function because we explicitly created a `y` variable inside the body of the function.

3.1.8 Many arguments function

You can write a function with more than one argument. Consider the function called `time`. It is a straightforward function multiplying two variables.

```
times <- function(x,y) {
  x*y
}
times(2,4)
```

```
## [1] 8
```

3.1.9 When shall we write function?

Data scientist need to do many repetitive tasks. Most of the time, we copy and paste chunks of code repetitively. For example, normalization of a variable is highly recommended before we run a machine learning algorithm.

The formula to normalize a variable is:

$$normalize = \frac{x - x_{min}}{x_{max} - x_{min}}$$

You already know how to use the min() and max() function in R. You use the tibble library to create the data frame. Tibble is so far the most convenient function to create a data set from scratch.

```
library(tibble)
data_frame <- tibble(
  c1 = rnorm(50, 5, 1.5),
  c2 = rnorm(50, 5, 1.5),
  c3 = rnorm(50, 5, 1.5)
)
```

You will proceed in two steps to compute the function described above. In the first step, you will create a variable called `c1_norm` which is the rescaling of `c1`. In step two, you just copy and paste the code of `c1_norm` and change with `c2` and `c3`.

Detail of the function with the column c1:

- Nominator: $x - x_{min}$: `data_frame$c1 -min(data_frame$c1))`
- Denominator:$x_{max} - x_{min}$: `max(data_frame$c1)-min(data_frame$c1))`

Therefore, we can divide them to get the normalized value of column `c1`:

```
(data_frame$c1 -min(data_frame$c1))/(max(data_frame$c1)-min(data_frame$c1))
```

We can create `c1_norm`, `c2_norm` and `c3_norm`:

```
# Create c1_norm: rescaling of c1
data_frame$c1_norm <- (data_frame$c1 -min(data_frame$c1))/(max(data_frame$c1)-min(data_frame$c1))
# show the first five values
head(data_frame$c1_norm, 5)
```

```
## [1] 0.2886787 0.2804736 0.4703763 0.4491567 0.0000000
```

It works. You can copy and paste

```
data_frame$c1_norm <- (data_frame$c1 -min(data_frame$c1))/(max(data_frame$c1)-min(data_frame$c1))
```

then change `c1_norm` to `c2_norm` and `c1` to `c2`. We do the same to create `c3_norm`

```
# Column c2_norm
data_frame$c2_norm <- (data_frame$c2 - min(data_frame$c2))/(max(data_frame$c2)-min(data_frame$c2))
```

```
# Column c3_norm
data_frame$c3_norm <- (data_frame$c3 - min(data_frame$c3))/(max(data_frame$c3)-min(data_frame$c3))
```

We perfectly rescaled the variables `c1`, `c2` and `c3`.

However, this method is prone to mistake. We could copy and forget to change the column name after pasting. Therefore, a good practice is to write a function each time you need to paste same code more than twice. We can rearrange the code into a formula and call it whenever it is needed. To write our own function, we need to give:

- Name: `normalize`.
- the number of arguments: You only need one argument, which is the column we use in our computation.
- The body: This is simply the formula we want to return.

We will proceed step by step to create the function normalize.

Step 1

You create the nominator, which is $x - x_{min}$. In R, we can store the nominator in a variable like this:

```
nominator <- x-min(x)
```

Step 2

You compute the denominator: $x_{max} - x_{min}$. You can replicate the idea of step 1 and store the computation in a variable:

```
denominator <- max(x)-min(x)
```

Step 3

We perform the division between the nominator and denominator.

```
normalize <- nominator/denominator
```

Step 4

To return value to calling function you need to pass normalize inside 'return() to get the output of the function.

```
return(normalize)
```

Step 5

You are ready to use the function by wrapping everything inside the bracket.

```
normalize <- function(x){
  # step 1: create the nominator
  nominator <- x-min(x)
  # step 2: create the denominator
  denominator <- max(x)-min(x)
  # step 3: divide nominator by denominator
  normalize <- nominator/denominator
  # return the value
  return(normalize)
}
```

Let's test your function with the variable `c1`:

```
normalize(data_frame$c1)
```

```
##  [1] 0.2886787 0.2804736 0.4703763 0.4491567 0.0000000 0.6670762 0.5201213
##  [8] 0.9226957 0.6010405 0.3906637 0.9935270 1.0000000 0.2472571 0.5608726
## [15] 0.4344636 0.5084751 0.5159993 0.2392318 0.5267855 0.7987615 0.4432263
```

```
## [22] 0.4434425 0.7335992 0.6406569 0.1673950 0.5161877 0.7809786 0.7367032
## [29] 0.5517526 0.6077342 0.2513036 0.5294877 0.2963623 0.3886087 0.6475707
## [36] 0.2624606 0.5233160 0.5533098 0.4839211 0.8123155 0.2837454 0.5863482
## [43] 0.4571924 0.4252497 0.5599546 0.2841072 0.2296527 0.3818332 0.7992279
## [50] 0.4840796
```

Functions are more comprehensive way to perform a repetitive task. You can use the normalize formula over different columns, like below:

```
data_frame$c1_norm_function <- normalize(data_frame$c1)
data_frame$c2_norm_function <- normalize(data_frame$c2)
data_frame$c3_norm_function <- normalize(data_frame$c3)
```

Even though the example is simple, we can infer the power of a formula. The above code is easier to read and especially avoid to mistakes when pasting codes.

3.1.10 Formulas with condition

Sometime, we need to include conditions into a formula to allow the code to return different outputs.

In Machine Learning tasks, you need to split the dataset between a train set and a test set. The train set allows the algorithm to learn from the data. In order to test the performance of our model, you can use the test set to return the performance measure. R does not have a function to create two datasets. You can write our own function to do that. Our function takes two arguments and is called 'split_data(). The idea behind is simple; you multiply the length of dataset (i.e. number of observations) with 0.8. For instance, if you want to split the dataset 80/20, and our dataset contains 100 rows, then our function will multiply 0.8*100 = 80. 80 rows will be selected to become our training data.

You will use the `airquality` dataset to test our user-defined function. The airquality dataset has 153 rows. You can see it with the code below:

```
nrow(airquality)
```

```
## [1] 153
```

We will proceed as follow:

```
split_data <- function(df, train = TRUE)
Arguments:

-   df: Define the dataset
-   train: Specify if the function returns the train set or test set. By default, set to TRUE
```

Your function has two arguments. The arguments train is a Boolean parameter. If it is set to `TRUE`, your function creates the train dataset, otherwise, it creates the test dataset.

You can proceed like you did with the `normalise()` function. You write the code as if it was only one-time code and then wrap everything with the condition into the body to create the function.

Step 1

You need to compute the length of the dataset. This is done with the function `nrow()`. `nrow()` returns the total number of rows in the dataset. You call the variable length.

```
lenght <- nrow(airquality)
length
```

```
## function (x)  .Primitive("length")
```

Step 2

You multiply the length by 0.8. It will return the number of rows to select. It should be 153*0.8 = 122.4

```
total_row <- lenght *0.8
total_row
```

```
## [1] 122.4
```

You want to select 122 rows among the 153 rows in the airquality dataset. You create a list containing values from 1 to total_row. You store the result in the variable called split

```
split <- 1:total_row
split[1:5]
```

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

split chooses the first 122 rows from the dataset. For instance, you can see that your variable split gathers the value 1, 2, 3, 4, 5 and so on. These values will be the index when you will select the rows to return.

Step 3

We need to select the rows in the `airquality` dataset based on the values stored in the split variable. This is done like this:

```
train_df <- airquality[split, ]
head(train_df)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Step 4

You can create the test dataset by using the remaining rows, 123:153. This is done by using - in front of split.

```
test_df <- airquality[-split, ]
head(test_df)
```

```
##     Ozone Solar.R Wind Temp Month Day
## 123    85     188  6.3   94     8  31
## 124    96     167  6.9   91     9   1
## 125    78     197  5.1   92     9   2
## 126    73     183  2.8   93     9   3
## 127    91     189  4.6   93     9   4
## 128    47      95  7.4   87     9   5
```

Step 5

You can create the condition inside the body of the function. Remember, you have an argument train that is a Boolean set to `TRUE` by default to return the train set. To create the condition, you use the if syntax:

```
 if (train ==TRUE){
    train_df <- airquality[split, ]
      return(train)
  } else {
    test_df <- airquality[-split, ]
      return(test)
  }
```

This is it, you can write the function. You only need to change `airquality` to df because you want to try our function to any data frame, not only `airquality`:

```
split_data <- function(df, train = TRUE){
  lenght <- nrow(df)
  split <- 1:total_row
  if (train ==TRUE){
    train_df <- df[split, ]
      return(train_df)
  } else {
    test_df <- df[-split, ]
      return(test_df)
  }
}
```

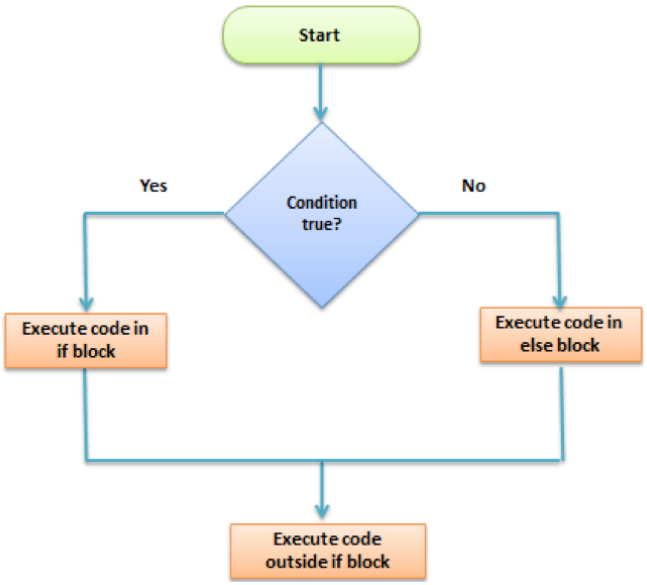
Let's try our function on the `aiquality` dataset. we should have one train set with 122 rows and a test set with 31 rows.

3.2 Conditional Statement

3.2.1 One condition statement

An `if else` statement is a great tool for the developer trying to return an output based on condition. In R, the syntax is:

```
if (condition) {
   expr1
   } else {
   expr2
   }
```



You want to examine whether a variable stored as `quantity` is above 20. If `quantity`is greater than 20, the code print `You sold a lot!` otherwise `Not enough for today`.

```
# Create quantity vector
quantity <-  25

# Set the is-else statement
if (quantity > 20) {
   print('You sold a lot!')
} else {
   print('Not enought for today')
}
```

```
## [1] "You sold a lot!"
```

note: Make sure you correctly write the indentations. Code with many conditions can become unreadable when the indentations are not in correct position.

3.2.2 Multiple conditions statement

You can customize further the control level with the `else if` statement. With `else if`, it is possible to add as many conditions as you want. The syntax is:

```
if (condition1) {
  expr1
} else if (condition2) {
  expr2
} else if (condition3) {
  expr3
} else {
  expr4
}
```

You are interested to know if you sold quantities between 20 and 30. If you do, then the statement return `Average day`, if it is above, it prints `What a great day!`, otherwise `Not enought for today`.

You can try to change the amount of quantity.

```
# Create vector quantiy
quantity <-  10

# Create multiple condition statement
if (quantity < 20) {
  print('Not enough for today')
} else if (quantity > 20  & quantity <= 30) {
  print('Average day')
} else {
  print('What a great day!')
}
```

```
## [1] "Not enough for today"
```

3.2.3 Nested if else

The `else if` statement is a clear way to understand the conditions in our code. It basically avoids to embbed many `if` inside the statement. Below, you add an easy example. VAT has different rate according to the product purchased. Imagine you have three different kind of products with different VAT applied:

Categorie	Products	VAT
A	Book, magasine, newspaper, etc..	8%
B	Vegetable, meat, beverage, etc..	10%
C	Tee-shirt, jean, pant, etc..	20%

You can write a chain to apply the correct VAT rate to the product a customer bought.

```
categorie <- 'A'
price <- 10

if (categorie =='A'){
  cat('A vat rate of 8% is applied.', 'The total price is',price *1.08)
} else if (categorie =='B'){
    cat('A vat rate of 10% is applied.', 'The total price is',price *1.10)
} else {
    cat('A vat rate of 20% is applied.', 'The total price is',price *1.20)
}
```

```
## A vat rate of 8% is applied. The total price is 10.8
```

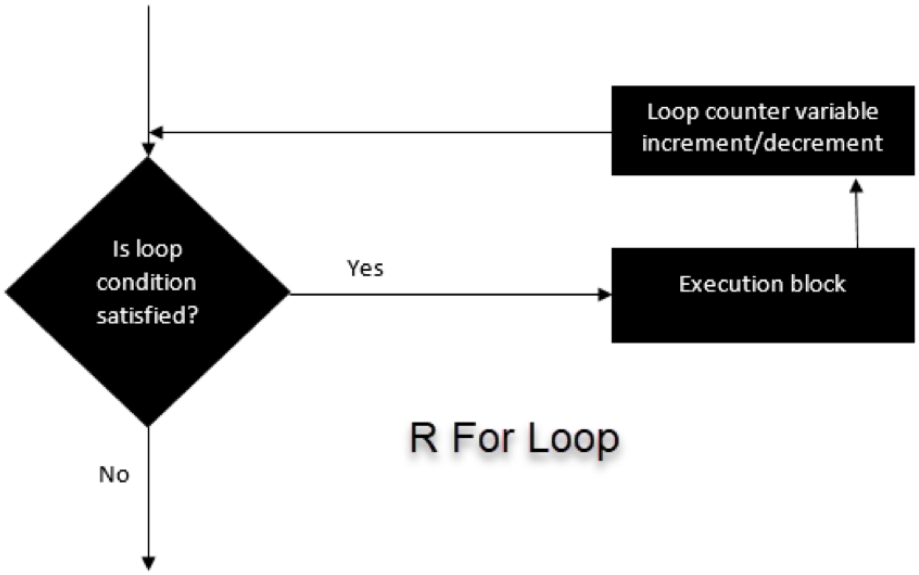
3.2.4 For loop

A `for` loop is very valuable when you need to iterate over a list of elements or a range of numbers. Loop can be used to iterate over a list, data frame, vector, matrix or any other object. The braces and square bracket are compulsory.

The basic syntax is the following:

```
For (i in vector) {
    Exp
}
```

Here, R will loop over all the `i` in `vector` and do the computation written inside the 'Exp.



3.2.5 Examples

Example 1

We iterate over all the elements of a vector and print the current value.

```
# Create fruit vector
fruit <- c('Apple', 'Orange', 'Passion fruit', 'Banana')

# Create the for statement
for ( i in fruit){
  print(i)
}
```

```
## [1] "Apple"
## [1] "Orange"
## [1] "Passion fruit"
## [1] "Banana"
```

Example 2

Creates a non-linear function by using the polynomial of x between 1 and 4 and you store it in a list.

```
# Create an empty list
list <- c()
# Create a for statement to populate the list
for (i in seq(1, 4, by=1)) {
  list[[i]] <- i*i
}
print(list)
```

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

The `for` loop is very valuable for machine learning task. After you trained a model, you need to regularise the model to avoid over-fitting. Regularization is a very tedious task because you need to find the value that minimises the loss function. To help us detect those value, you can make use of a `for` loop to iterate over a range of values and define the best candidate.

3.2.6 Loop over a list

Looping over a list is just as easy and convenient as looping over a vector.

Let's see an example

```
# Create a list with many three vectors
fruit <- list(Basket = c('Apple', 'Orange', 'Passion fruit', 'Banana'), Money = c(10, 12, 15), purchase

for (p  in fruit) {
  print(p)
}
```

```
## [1] "Apple"         "Orange"        "Passion fruit" "Banana"
## [1] 10 12 15
## [1] FALSE
```

3.2.7 Loop over a matrix

A matrix has 2-dimension, rows and columns. To iterate over a matrix, you have to define two `for` loop, namely one for the rows and another for the column.

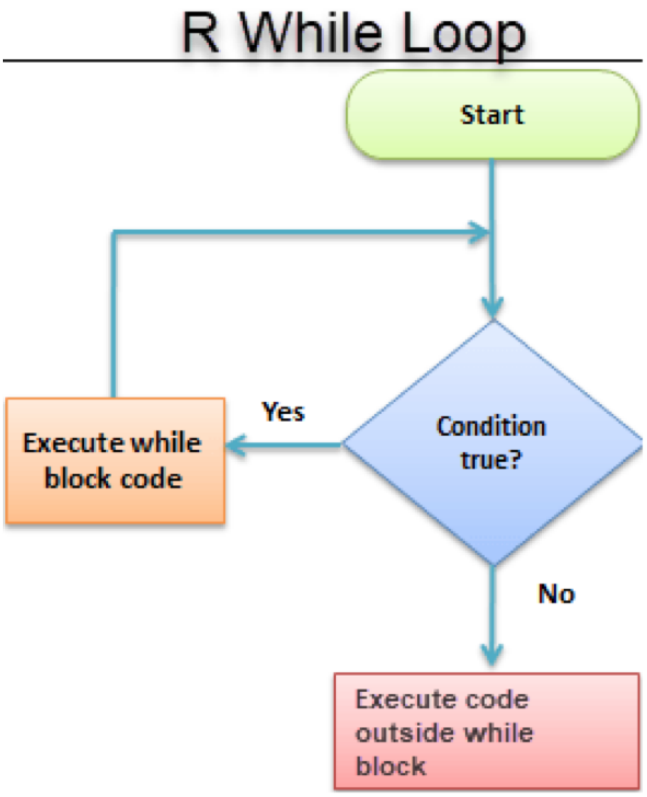
```
# Create a matrix
mat <- matrix(data = seq(10, 20, by=1), nrow = 6, ncol =2)

# define the double for loop
for (i in 1:nrow(mat)) {
  for (j in 1:ncol(mat)) {
  print(paste("Row", i, "and column", j, "have values of", mat[i,j]))
  }
}
```

```
## [1] "Row 1 and column 1 have values of 10"
## [1] "Row 1 and column 2 have values of 16"
## [1] "Row 2 and column 1 have values of 11"
## [1] "Row 2 and column 2 have values of 17"
## [1] "Row 3 and column 1 have values of 12"
## [1] "Row 3 and column 2 have values of 18"
## [1] "Row 4 and column 1 have values of 13"
## [1] "Row 4 and column 2 have values of 19"
## [1] "Row 5 and column 1 have values of 14"
## [1] "Row 5 and column 2 have values of 20"
## [1] "Row 6 and column 1 have values of 15"
## [1] "Row 6 and column 2 have values of 10"
```

3.2.8 While loop

A loop is a statement that keeps running until a condition is satisfied. The syntax for a while loop is the following:

```
while (condition) {
    Exp
    }
```



note: Remember to write a closing condition at some point otherwise the loop will go on indefinitely.

Example 1

Let's go through a very simple example to understand the concept of while loop. You will create a loop and after each run add 1 to the stored variable. You need to close the loop, therefore we explicitely tells R to stop looping when the variable reached 10.

note: If you want to see current loop value, you need to wrap the variable inside the function `print()`.

```
# Create a variable with value 1
begin <- 1
# Create the loop
while (begin <= 10){
  # See which we are
  cat('This is loop number', begin)
  # add 1 to the variable begin after each loop
  begin <- begin +1
  print(begin)
}
```

```
## This is loop number 1[1] 2
## This is loop number 2[1] 3
## This is loop number 3[1] 4
## This is loop number 4[1] 5
## This is loop number 5[1] 6
## This is loop number 6[1] 7
## This is loop number 7[1] 8
## This is loop number 8[1] 9
## This is loop number 9[1] 10
```

```
## This is loop number 10[1] 11
```

Expample 2

You bought a stock at price of 50 dollars. If the price goes below 45, we want to short it. Otherwise, we keep it in our portfolio. The price can fluctuate between -10 to +10 around 50 after each loop. You can write the code as follow:

```
set.seed(123)
# Set variable stock and price
stock <- 50
price <- 50
# Loop variable counts the number of loops
loop <- 1
# Set the while statement
while (price > 45){
  # Create a random price between 40 and 60
  price <- stock + sample(-10:10, 1)
  # Count the number of loop
  loop = loop +1
  # Print the number of loop
  print(loop)
}
```

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

```
cat('it took', loop, 'loop before we short the price. The lowest price is', price)
```

```
## it took 7 loop before we short the price. The lowest price is 40
```

3.3 Apply Family

This chapter aims at introducing the `apply()` function collection. The `apply()` function is the most basic of the collection, we will talk as well about `sapply()`, `lapply()` and `tapply()`. The apply collection can be viewed as a substitute to the loop

The `apply()` collection comes from with **r essential** if we install R with Anaconda. The `apply()` function can be feed with many functions to perfom redundant application on a collection of object (data frame, list, vector, etc.). The purpose of `apply()` is primarily to avoid explicit uses of loop constructs. They can be used for an input list, matrix or array and apply a function. Any function can be passed into `apply()`.

3.3.1 What occasions to use `apply()` collection

We use `apply()` over a matrice. This function takes 5 arguments:

```
apply(X, MARGIN, FUN)
arguments

- x: an array or matrix
- MARGIN:  take a value or range between 1 and 2 in order to define where to apply the function:
     - `MARGIN=1`: the manipulation is performed on rows
```

```
    - `MARGIN=2`: the manipulation is performed on columns
    - `MARGIN=c(1,2)` the manipulation is performed on rows and columns
- FUN: tells which function to apply
```

The simplest example is to sum a matrice over all the columns. The code `apply(m1, 2, sum)` will apply the sum function to the matrix 5x6 and return the sum of each column accessible in the dataset.

```
m1 <- matrix(rnorm(30), nrow=5, ncol=6)
a_m1 <- apply(m1, 2, sum)
a_m1
```

```
## [1]  1.1107161  0.8521523 -0.1270120 -2.0832386 -2.0494636  1.1421964
```

A best practice is to store the values before printing it to the console.

As a reminder, R has different build-in functions:

- `mean(x, na.rm = FALSE)`
- `sd(x)`
- `median(x)`
- `sum(x)`
- `min(x)`
- `max(x)`
- `scale(x)`
- `abs(x)`
- `sqrt(x)`
- `round(x, digit = n)`
- `log(x)`
- ‘exp(x)
- ...

The syntax of `lapply()` is:

```
lapply(X, FUN)
Arguments:

- X: A vector or an object
- FUN: Function applied to each element of x
```

`l` in `lapply()` stands for list. The difference between `lapply()` and `apply()` lies between the output return. The output of `lapply()` is a list. `lapply()` can be used for other objects like data frames and lists.

`lapply()` function does not need `MARGIN`.

A very easy example can be to change the string value of a matrix to lower case with `tolower` function. We construct a matrix with the name of the famous movies. The name is in upper case format.

```
movies <- c("SPYDERMAN", "BATMAN", "VERTIGO", "CHINATOWN")
movies_lower <-lapply(movies, tolower)
str(movies_lower)
```

```
## List of 4
##  $ : chr "spyderman"
##  $ : chr "batman"
##  $ : chr "vertigo"
##  $ : chr "chinatown"
```

We can use `unlist()` to convert the list into a vector.

```
movies_lower <-unlist(lapply(movies, tolower))
str(movies_lower)
```

```
##  chr [1:4] "spyderman" "batman" "vertigo" "chinatown"
```

`sapply()` function does the same jobs as `lapply()` function but returns a vector.

```
sapply(X, FUN)
Arguments:

- X: A vector or an object
- FUN: Function applied to each element of x
```

We can measure the minimum speed and stopping distances of cars from the `cars` dataset.

```
dt <- cars
# Minimum
lmn_cars <- lapply(dt, min)
smn_cars <- sapply(dt, min)
lmn_cars
```

```
## $speed
## [1] 4
##
## $dist
## [1] 2
```

```
smn_cars
```

```
## speed  dist
##     4     2
```

```
# maximum
lmxcars <- lapply(dt, max)
smxcars <- sapply(dt, max)
lmxcars
```

```
## $speed
## [1] 25
##
## $dist
## [1] 120
```

```
smxcars
```

```
## speed  dist
##    25   120
```

We can use a user built-in function into `lapply()` or `sapply()`. We create a function named `avg` to compute the average of the minimum and maximum of the vector.

```
avg <- function(x) {
  ( min(x) + max(x) ) / 2
}

fcars <- sapply(dt, avg)
fcars
```

```
## speed  dist
##  14.5  61.0
```

`sapply()` function is more efficient than `lapply()` in the output returned because `sapply()` store values direclty into a vector. In the next example, you will see this is not always the case.

We can summarize the difference between `apply()`, `sapply()` and ‘lapply() in the following table:

Function	Arguments	Objective	Input	Output
apply	apply(x, MARGIN, FUN)	Apply a function to the rows or columns or both	Data frame or matrix	vector, list, array
lapply	lapply(X, FUN)	Apply a function to all the elements of the input	List, vector or data frame	list
sapply	sappy(X FUN)	Apply a function to all the elements of the input	List, vector or data frame	vector or matrix

3.3.2 Slice vector

You can use `lapply()` or `sapply()` interchangeable to slice a data frame. We create a function, `below_average()`, that takes a vector of numerical values and returns a vector that only contains the values that are strictly above the average. You compare both results with the `identical()` function.

```
below_ave <- function(x) {
  ave <- mean(x)
  return(x[x > ave])
}

dt_s<- sapply(dt, below_ave)
dt_l<- lapply(dt, below_ave)
identical(dt_s, dt_l)
```

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

The function `tapply()` computes a measure (mean, median, min, max, etc..) or a function for each factor variable in a vector.

```
tapply(X, INDEX, FUN = NULL)
Arguments:

- X: An object, usually a vector
- INDEX: A list containing factor
- FUN: Function applied to each element of x
```

Part of the job of a data scientist or researchers is to compute summaries of variables. For instance, measure the average or more complex function with eigen values. Most of the data are grouped by ID, city, countries, and so on. Summarizing over group reveals more interesting patterns.

To understand how it works, let's use the `iris` dataset. This dataset is very famous in the world of machine learning. The purpose of this dataset is to predict the class of each of the three flower species: `Sepal`, `Versicolor`, `Virginica`. The dataset collects information for each species about their length and width.

As a prior work, you can compute the median of the length for each species. `tapply()` is a quick way to perform this computation.

```
data(iris)
tapply(iris$Sepal.Width, iris$Species, median)
```

```
##     setosa versicolor  virginica
##        3.4        2.8        3.0
```

4 Introduction to Data Preparation

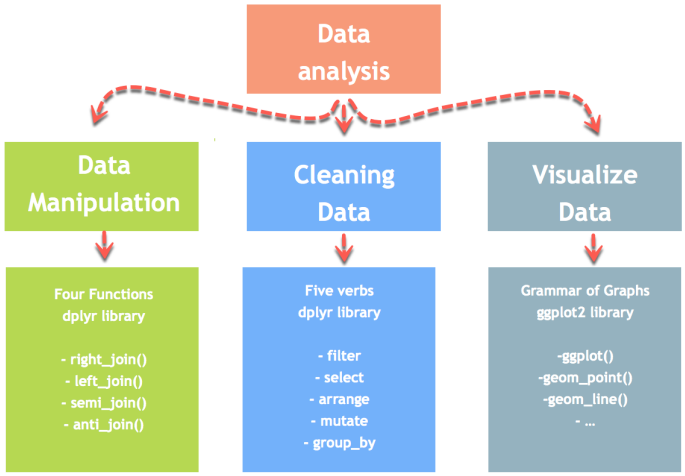
Data preparation can be divided into three parts

- **Extraction**: First, you need to collect the data from many sources and combine them.
- **Transform**: This step involves the data manipulation. Once you have consolidated all the sources of data, you can begin to clean the data.
- **Visualize**: The last move is to visualize your data to check irregularity.

One of the most significant challenges faced by data scientist is the data manipulation. Data is never available in the desired format. The data scientist needs to spend at least half of his time, cleaning and manipulating the data. That is one of the most critical assignments in the job. If the data manipulation process is not complete, precise and rigorous, the model will not perform correctly.

R has a library called `dplyr` to help in data transformation. The `dplyr` library is fundamentally created around four functions to manipulate the data and five verbs to clean the data. After that, you can use the ggplot library to analyze and visualize the data.

In this chapter, you will learn how to use the `dplyr` library to manipulate a data frame.



You need to install `dplyr` library:

- `caret`: Data manipulation library. If you have install R with `r-essential`. It is already in the library
  - Anaconda: `conda install -c r r-dplyr`

4.1 Data Wrangling

4.1.1 Merge with `dplyr()`

`dplyr` provides a nice and convenient way to combine datasets. You may have many sources of input data, and at some point, you need to combine them. A join with `dplyr` adds variables to the right of the original dataset. The beauty is `dplyr` is that it handles four types of joins similar to SQL

- `Left_join()`
- `right_join()`
- `inner_join()`
- `full_join()`

You will study all the joins types via an easy example.

First of all, you build two datasets. Table 1 contains two variables, `ID` and `y`, whereas table 2 gathers `ID` and `z`. In each situation, you need to have a **key-pair** variables. In your case, `ID` is your **key** variable. The function will look for identical values in both tables and binds the returning values to the right of table 1.

Table 1

ID	y
A	5
B	5
C	8
D	0
F	9

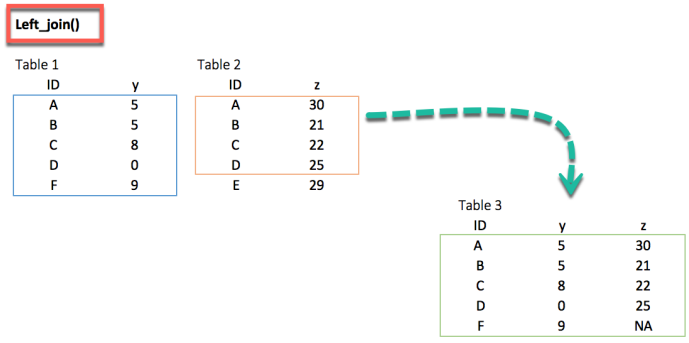
Table 2

ID	z
A	30
B	21
C	22
D	25
E	29

```
library(dplyr)
df_primary <- tribble(
  ~ID, ~y,
  "A", 5,
  "B", 5,
  "C", 8,
  "D", 0,
  "F", 9
)

df_secondary <- tribble(
  ~ID, ~y,
  "A", 30,
  "B", 21,
  "C", 22,
  "D", 25,
  "E", 29
)
```

Function `left_join()`

The most common way to merge two datasets is to use the `left_join()` function. You can see from the picture below that the key-pair matches perfectly the rows `A`, `B`, `C` and `D` from both datasets. However, `E` and `F` are left over. How do you treat these two observations?. With the `left_join()`, you will keep all the variables in the original table and don't consider the variables that do not have a key-paired in the destination table. In your example, the variable `E` does not exist in table 1. Therefore, the row will be dropped. The variable `F` comes from the origin table; it will be kept after the `left_join()` and return `NA` in the column `z`. The figure below reproduces what will happen with a `left_join()`.

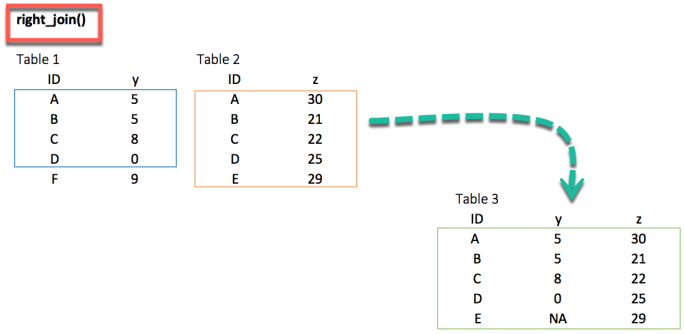


```
left_join(df_primary, df_secondary, by = 'ID')
```

```
## # A tibble: 5 x 3
##      ID   y.x   y.y
##   <chr> <dbl> <dbl>
## 1     A     5    30
## 2     B     5    21
## 3     C     8    22
## 4     D     0    25
## 5     F     9    NA
```

Function `right_join()`

The `right_join()` function works exactly like `left_join()`. The only difference is the row dropped. The value `E`, available in the destination, exists in the new table and takes the value `NA` for the column `y`.



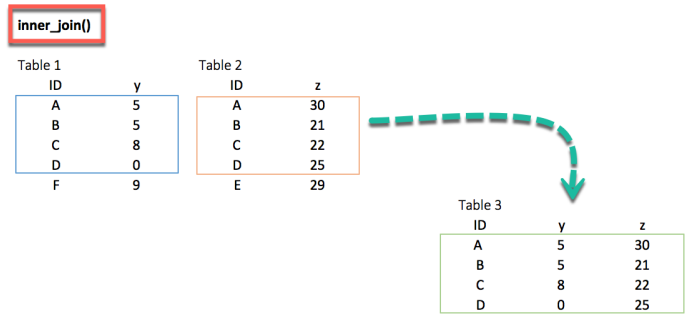

```
right_join(df_primary, df_secondary, by = 'ID')
```

```
## # A tibble: 5 x 3
##      ID   y.x   y.y
##   <chr> <dbl> <dbl>
## 1     A     5    30
## 2     B     5    21
## 3     C     8    22
## 4     D     0    25
## 5     E    NA    29
```

Function `inner_join()`

When you are 100% sure that the two datasets won't match, you can consider to return **only** rows existing in **both** dataset. This is possible when you need a clean dataset or when you don't want to impute missing values with the mean or median.

The `inner_join()`comes to help. This function excludes the unmatched rows.

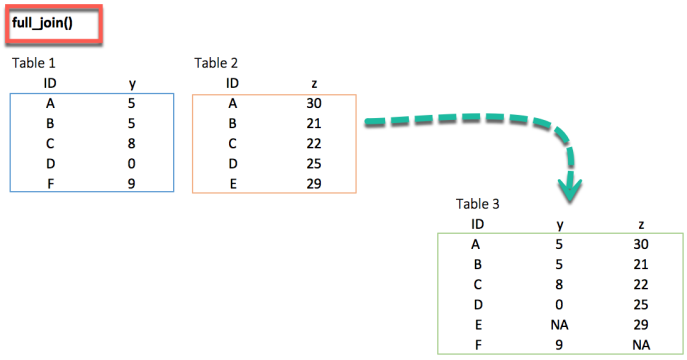


```
inner_join(df_primary, df_secondary, by = 'ID')
```

```
## # A tibble: 4 x 3
##      ID   y.x   y.y
##   <chr> <dbl> <dbl>
## 1     A     5    30
## 2     B     5    21
## 3     C     8    22
## 4     D     0    25
```

Function `full_join()`

Finally, the `full_join()` function keeps all observations and replace missing values with `NA`.

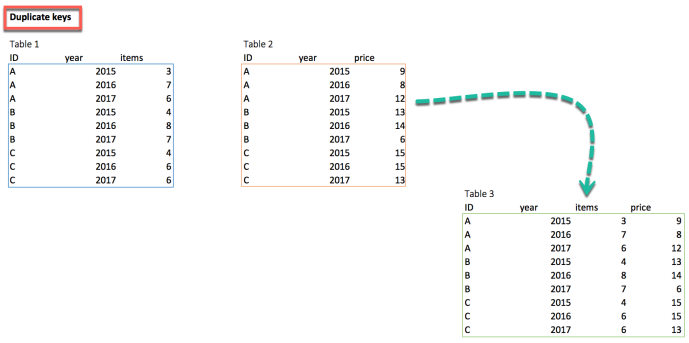



```
full_join(df_primary, df_secondary, by = 'ID')
```

```
## # A tibble: 6 x 3
##      ID   y.x   y.y
##   <chr> <dbl> <dbl>
## 1     A     5    30
## 2     B     5    21
## 3     C     8    22
## 4     D     0    25
## 5     F     9    NA
## 6     E    NA    29
```

Multiple keys pairs

Last but not least, you can have multiple keys in your dataset. Consider the following dataset where you have years or a list of products bought by the customer.



If you try to merge both table, R throws an error. To remedy the situation, you can pass two key-pairs variables. That is, `ID` and `year`appear in both datasets. You can use the following code to merge table1 and table 2.

```
df_primary <- tribble(
  ~ID, ~year, ~items,
  "A", 2015,3,
  "A", 2016,7 ,
  "A", 2017, 6,
  "B", 2015,4,
  "B", 2016, 8,
  "B", 2017,7,
  "C", 2015, 4,
  "C", 2016, 6,
  "C", 2017, 6
)

df_secondary <- tribble(
  ~ID, ~year, ~prices,
  "A", 2015,9,
  "A", 2016,8 ,
  "A", 2017,12,
  "B", 2015,13,
  "B", 2016, 14,
  "B", 2017,6,
  "C", 2015, 15,
  "C", 2016, 15,
  "C", 2017, 13
)

left_join(df_primary, df_secondary, by = c('ID', 'year'))
```

```
## # A tibble: 9 x 4
##      ID  year items prices
##   <chr> <dbl> <dbl>  <dbl>
## 1     A  2015     3      9
## 2     A  2016     7      8
## 3     A  2017     6     12
## 4     B  2015     4     13
## 5     B  2016     8     14
## 6     B  2017     7      6
## 7     C  2015     4     15
## 8     C  2016     6     15
## 9     C  2017     6     13
```

4.1.2 Data transformation

Following are four important functions to tidy the data:

- `gather()`: Transform the data from wide to long
- `spread()`: Transform the data from long to wide
- `separate()`: Split one variables into two
- `unit()`: Unit two variables into one

You use the `tidyr` library. This library belongs to the collection of library to manipulate, clean and visualize the data.

- `tidyr`: Manipulate data frame. If you have install R with `r-essential`. It is already in the library

– Anaconda: `conda install -c r r-tidyr`

Instead, you can use the console:

```
install.packages("tidyr")
```

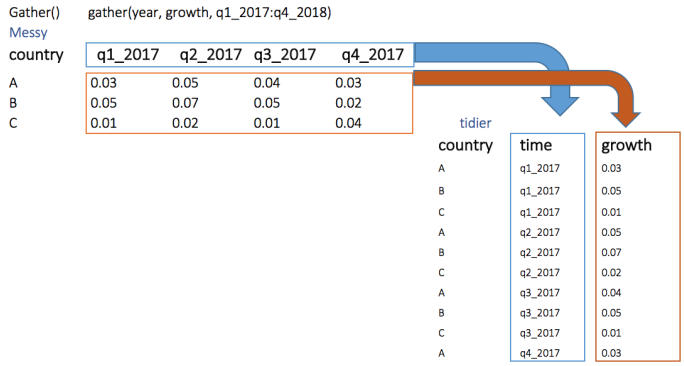
to install `tidyr`

Function `gather()`

The objectives of the `gather()` function is to transform the data from wide to long.

```
gather(data, key, value, na.rm = FALSE)
arguments:

- data: The data frame used to reshape the dataset
- key: Name of the new column created
- value: Select the columns used to fill the key column
- na.rm: Remove missing values. FALSE by default
```

Below, you can visualize the concept of reshaping wide to long. You want to create a single column names `growth`, filled by the rows of the `quarter` variables.




```
library(tidyr)
# Create messy dataset
messy <- data.frame(
  country = c("A", "B", "C"),
  q1_2017 = c(0.03, 0.05, 0.01),
  q2_2017 = c(0.05, 0.07, 0.02),
  q3_2017 = c(0.04, 0.05, 0.01),
  q4_2017 = c(0.03, 0.02, 0.04)
)
messy
```

```
##   country q1_2017 q2_2017 q3_2017 q4_2017
## 1       A    0.03    0.05    0.04    0.03
## 2       B    0.05    0.07    0.05    0.02
## 3       C    0.01    0.02    0.01    0.04
```

```
# Reshape the data
tidier <- messy %>%
          gather(quarter, growth, q1_2017:q4_2017)
tidier
```

```
##    country quarter growth
## 1        A q1_2017   0.03
## 2        B q1_2017   0.05
```

```
## 3        C q1_2017   0.01
## 4        A q2_2017   0.05
## 5        B q2_2017   0.07
## 6        C q2_2017   0.02
## 7        A q3_2017   0.04
## 8        B q3_2017   0.05
## 9        C q3_2017   0.01
## 10       A q4_2017   0.03
## 11       B q4_2017   0.02
## 12       C q4_2017   0.04
```

In the `gather()` function, you create two new variable `quarter` and `growth` because your original dataset has one group variable: i.e. `country` and the key-value pairs.

Function `spread()`

the `spread()` function does the opposite of gather.

```
spread(data, key, value)
arguments:

- data: The data frame used to reshape the dataset
- key: Column to reshape long to wide
- value: Rows used to fill the new column
```

You can reshape the `tidier` dataset back to `messy` with 'spread()

```
# Reshape the data
messy_1 <- tidier %>%
          spread(quarter, growth)
messy_1
```

```
##   country q1_2017 q2_2017 q3_2017 q4_2017
## 1       A    0.03    0.05    0.04    0.03
## 2       B    0.05    0.07    0.05    0.02
## 3       C    0.01    0.02    0.01    0.04
```

Function `separate()`

The `separate()` function splits a column into two according to a separator. This function is helpful in some situation where the variable is a date. Our analysis can require focusing on month and year and you want to separate the column into two new variables.

```
separate(data, col, into, sep= "", remove = TRUE)
arguments:

- data: The data frame used to reshape the dataset
- col: The column to split
- into: The name of the new variables
- sep: Indicates the symbol used that unit the variable name, i.e:  "-", "_", "&"
remove: Remove the old column. By default sets to TRUE.
```

You can split the quarter from the year in the `tidier` dataset by applying the `separate()` function.

```
separate_tidier <- tidier %>%
                   separate(quarter, c("Qrt", "year"), sep ="_")
head(separate_tidier)
```

```
##   country Qrt year growth
## 1       A  q1 2017   0.03
```

```
## 2       B  q1 2017   0.05
## 3       C  q1 2017   0.01
## 4       A  q2 2017   0.05
## 5       B  q2 2017   0.07
## 6       C  q2 2017   0.02
```

Function `unite()`

The `unite()` function concanates two columns into one.

```
unit(data, col, conc ,sep= "", remove = TRUE)
arguments:

- data: The data frame used to reshape the dataset
- col: Name of the new colunm
- conc: Name of the columns to concanate
- sep: Indicates the symbol used that unit the variable name, i.e: "-", "_", "&"
- remove: Remove the old columns. By default sets to TRUE
```

In the above example, you separated quarter from year. What if you want to merge them. You use the following code:

```
unit_tidier <- separate_tidier %>%
                    unite(Quarter, Qrt, year, sep ="_")
head(unit_tidier)
```

```
##   country Quarter growth
## 1       A q1_2017   0.03
## 2       B q1_2017   0.05
## 3       C q1_2017   0.01
## 4       A q2_2017   0.05
## 5       B q2_2017   0.07
## 6       C q2_2017   0.02
```

4.1.3 Summary

The table below summarises the four functions used in `dplyr` to merge two datasets.

Library	Objectives	Function	Arguments	Multiple keys
Dplyr	Merge two datasets. Keep all observations from the origin table	left_join()	data, origin, destination, by = "ID"	origin, destination, by = c("ID", "ID2")
Dplyr	Merge two datasets. Keep all observations from the destination table	right_join()	data, origin, destination, by = "ID"	origin, destination, by = c("ID", "ID2")
Dplyr	Merge two datasets. Excludes all unmatched rows	inner_join()	data, origin, destination, by = "ID"	origin, destination, by = c("ID", "ID2")
Dplyr	Merge two datasets. Keeps all observations	full_join()	data, origin, destination, by = "ID"	origin, destination, by = c("ID", "ID2")

The last table summarises how to transform a dataset with the `gather()`, `spread()`, `separate()` and `unit()` functions.

Library	Objectives	Function	Arguments
tidyr	Transform the data from wide to long	gather()	(data, key, value, na.rm = FALSE)
tidyr	Transform the data from long to wide	spread()	(data, key, value)
tidyr	Split one variables into two	separate()	(data, col, into, sep= “”, remove = TRUE)
tidyr	Unit two variables into one	unit()	(data, col, conc ,sep= “”, remove = TRUE)

4.2 Data selection

In this chapter, you will learn how to manipulate the data. The library called `dplyr` contains valuable verbs to navigate inside the dataset.

Through this chapter, you will use the `Travel times` dataset. The dataset collects information on the trip leads by a driver between his home and his workplace. There are fourteen variables in the dataset, including:

- `DayOfWeek`: Identify the day of the week the driver uses his car
- `Distance`: The total distance of the journey
- `MaxSpeed`: The maximum speed of the journey
- `TotalTime`: The length in minutes of the journey

The dataset has around 200 observations in the dataset, and the rides occurred between Monday to Friday.

First of all, you need to:

- load the dataset
- check the structure of the data.

One handy feature with `dplyr` is the `glimpse()` function. This is an improvement over `str()`. We can use `glimpse()` to see the structure of the dataset and decide what manipulation is required.

```
library(dplyr)
PATH <- "https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/travel_times.csv"
df <- read.csv(PATH)
glimpse(df)
```

```
## Observations: 205
## Variables: 14
## $ X              <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
## $ Date           <fctr> 1/6/2012, 1/6/2012, 1/4/2012, 1/4/2012, 1/3/20...
## $ StartTime      <fctr> 16:37, 08:20, 16:17, 07:53, 18:57, 07:57, 17:3...
## $ DayOfWeek      <fctr> Friday, Friday, Wednesday, Wednesday, Tuesday,...
## $ GoingTo        <fctr> Home, GSK, Home, GSK, Home, GSK, Home, GSK, GS...
## $ Distance       <dbl> 51.29, 51.63, 51.27, 49.17, 51.15, 51.80, 51.37...
## $ MaxSpeed       <dbl> 127.4, 130.3, 127.4, 132.3, 136.2, 135.8, 123.2...
## $ AvgSpeed       <dbl> 78.3, 81.8, 82.0, 74.2, 83.4, 84.5, 82.9, 77.5,...
## $ AvgMovingSpeed <dbl> 84.8, 88.9, 85.8, 82.9, 88.1, 88.8, 87.3, 85.9,...
## $ FuelEconomy    <fctr> , , , , , , -, -, 8.89, 8.89, 8.89, 8.89, 8.89...
## $ TotalTime      <dbl> 39.3, 37.9, 37.5, 39.8, 36.8, 36.8, 37.2, 37.9,...
## $ MovingTime     <dbl> 36.3, 34.9, 35.9, 35.6, 34.8, 35.0, 35.3, 34.3,...
## $ Take407All     <fctr> No, No, No, No, No, No, No, No, No, No, No, No...
## $ Comments       <fctr> , , , , , , , , , , , , , , , Put snow tires o...
```

This is obvious that the variable `Comments` needs further diagnostic. The first observations of the `Comments` variable are only missing values.

```
sum(df$Comments =="")
```

```
## [1] 181
```

Code Explanation

- `sum(df$Comments =="")`: Sum the observations equalts to `""` in the column `comments` from `df`

4.2.1 Filter observations from statement

Function `select()`

We will begin with the `select()` verb. We don't necessarily need all the variables, and a good practice is to select only the variables you find relevant.

We have 181 missing observations, almost 90 percent of the dataset. If you decide to exclude them, you won't be able to carry on the analysis.

The other possibility is to drop the variable `Comment` with the `select()` verb.

We can select variables in different ways with `select()`. Note that, the first argument is the dataset.

```
- `select(df, A, B ,C)`: Select the variables A, B and C from df dataset.
- `select(df, A:C)`: Select all variables from A to C from df dataset.
- `select(df, -C)`: Exclude C from the dataset from df dataset.
```

You can use the third way to exclude the `Comments` variable.

```
step_1_df <- select(df, -Comments)
dim(df)
```

```
## [1] 205  14
```

```
dim(step_1_df)
```

```
## [1] 205  13
```

The original dataset has 14 features while the `step_1_df` has 13.

Function `Filter()`

The `filter()` verb helps to keep the observations following a criteria. The `filter()` works exactly like `select()`, you pass the data frame first and then a condition separated by a comma:

```
filter(df, condition)
arguments:

- df: dataset used to filter the data
- condition:  Condition used to filter the data
```

One criteria

First of all, you can count the number of observations within each level of a factor variable.

```
table(step_1_df$GoingTo)
```

```
##
##  GSK Home
##  105  100
```

Code Explanation

- `table()`: Count the number of observations by level. Note, only factor level variable are accepted
- `table(step_1_df$GoingTo)`: Count the number of of trips toward the final destination.

The function `table()` indicates 105 rides are going to `GSK` and 100 to `Home`.

We can filter the data to return one dataset with 105 observations and another one with 100 observations.

```
# Select observations if GoingTo == Home
select_home <- filter(df, GoingTo =="Home")
dim(select_home)
```

```
## [1] 100  14
```

```
# Select observations if GoingTo == Work
select_work <- filter(df, GoingTo =="GSK")
dim(select_work)
```

```
## [1] 105  14
```

Multiple criterions

We can filter a dataset with more than one criteria. For instance, you can extract the observations where the destination is `Home` and occured on a `Wednesday`.

```
select_home_wed <- filter(df, GoingTo =="Home" & DayOfWeek == "Wednesday")
dim(select_home_wed)
```

```
## [1] 23 14
```

23 observations matched this criterion.

4.2.2 Pipeline

The creation of a dataset requires a lot of operations, such as:

- importing
- merging
- selecting
- filtering
- and so on

The `dplyr` library comes with a practical operator, `%>%`, called the **pipeline**. The pipeline feature makes the manipulation clean, fast and less prompt to error.

This operator is a code which performs steps without saving intermediate steps to the hard drive. If you are back to our example from above, you can select the variables of interest and filter them. We have three steps:

- Step 1: Import data: Import the gps data
- Step 2: Select data: Select `GoingTo` and `DayOfWeek`
- Step 3: Filter data: Return only `Home` and `Wednesday`

We can use the hard way to do it:

```
# Step 1
step_1 <- read.csv(PATH)
# Step 2
step_2 <- select(step_1, GoingTo, DayOfWeek)
# Step 3
step_3 <- filter(step_2, GoingTo=="Home", DayOfWeek=="Wednesday")
head(step_3)
```

```
##   GoingTo DayOfWeek
## 1    Home Wednesday
## 2    Home Wednesday
## 3    Home Wednesday
## 4    Home Wednesday
## 5    Home Wednesday
## 6    Home Wednesday
```

That is not a convenient way to perform many operations, especially in a situation with lots of steps. The environment ends up with a lot of objects stored.

Let's use the pipeline operator `%>%` instead. We only need to define the data frame used at the beginning and all the process will flow from it.

Basic syntax of pipeline

```
New_df <- df %>%
                step 1 %>%
                step 2 %>%
                ...
arguments

- New_df: Name of the new data frame
- df: Data frame used to compute the step
- step: Instruction for each step
- Note: The last instruction does not need the pipe operator `%`, you don't have instructions to pipe ar

Note: Create a new variable is optional. If not included, the output will be displayed in the console.
```

You can create your first pipe following the steps enumerated above.

```
# Create the data frame filter_home_wed. It will be the object return at the end of the pipeline
filter_home_wed <-
  # Step 1
  read.csv(PATH) %>%
  # Step 2
                   select(GoingTo, DayOfWeek) %>%
  # Step 3
                   filter(GoingTo=="Home", DayOfWeek=="Wednesday")

identical(step_3, filter_home_wed)
```

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

We are ready to create a stunning dataset with the pipeline operator.

Function `arrange()`

In the previous tutorial, you learn how to sort the values with the function `sort()`. The library `dplyr` has its sorting function. It works like a charm with the pipeline. The `arrange()` verb can reorder one or many rows, either ascending (default) or descending.

```
- `arrange(A)`: Ascending sort of variable A
- `arrange(A, B)`: Ascending sort of variable A and B
- `arrange(desc(A), B)`: Descending sort of variable A and ascending sort of B
```

We can sort the distance by destination.

```
# Sort by destination and distance
step_2_df <- step_1_df %>%
```

```
            arrange(GoingTo, Distance)
head(step_2_df)
```

```
##     X       Date StartTime DayOfWeek GoingTo Distance MaxSpeed AvgSpeed
## 1 193  7/25/2011     08:06    Monday     GSK    48.32    121.2     63.4
## 2 196  7/21/2011     07:59  Thursday     GSK    48.35    129.3     81.5
## 3 198  7/20/2011     08:24 Wednesday     GSK    48.50    125.8     75.7
## 4 189  7/27/2011     08:15 Wednesday     GSK    48.82    124.5     70.4
## 5  95 10/11/2011     08:25   Tuesday     GSK    48.94    130.8     85.7
## 6 171  8/10/2011     08:13 Wednesday     GSK    48.98    124.8     72.8
##   AvgMovingSpeed FuelEconomy TotalTime MovingTime Take407All
## 1           78.4        8.45      45.7       37.0         No
## 2           89.0        8.28      35.6       32.6        Yes
## 3           87.3        7.89      38.5       33.3        Yes
## 4           77.8        8.45      41.6       37.6         No
## 5           93.2        7.81      34.3       31.5        Yes
## 6           78.8        8.54      40.4       37.3         No
```

4.2.3 Summary

In the table below, you summarize all the operations you learnt during the tutorial.

Verb	Objective	Code	Explanation
glimpse	check the structure of a df	glimpse(df)	Identical to str()
select()	Select/exclude the variables	select(df, A, B ,C)	Select the variables A, B and C
		select(df, A:C)	Select all variables from A to C
		select(df, -C)	Exclude C
filter()	Filter the df based a one or many conditions	filter(df, condition1)	One condition
		filter(df, condition1	ondition2)
arrange()	Sort the dataset with one or many variables	arrange(A)	Ascending sort of variable A
		arrange(A, B)	Ascending sort of variable A and B
		arrange(desc(A), B)	Descending sort of variable A and ascending sort of B
%>%	Create a pipeline between each step	step 1 %>% step 2 %>% step 3	

4.2.4 Aggregate the dataset

Summary of a variable is important to have an idea about the data. Although, summarizing a variable by group gives better information on the distribution of the data.

In this tutorial, you will learn how summarize a dataset by group with the `dplyr` library.

For this tutorial, you will use the `batting` dataset. The original dataset contains 102816 observations and 22 variables. You will only use 20 percent of this dataset and use the following variables:

- `playerID`: Player ID code. Factor
- `yearID`: Year. Factor
- `teamID`: Team. factor
- `lgID`: League. Factor: AA AL FL NL PL UA
- `AB`: At bats. Numeric

- `G`: Games: number of games by a player. Numeric
- `R`: Runs. Numeric
- `HR`: Homeruns. Numeric
- `SH`: Sacrifice hits. Numeric

Before you perform summary, you will do the following steps to prepare the data:

- Step 1: Import the data
- Step 2: Select the relevant variables
- Step 3: Sort the data

```
library(dplyr)
# Step 1
data <-  read.csv("https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/batting_lahman
# Step 2
  select(c(playerID, yearID, AB, teamID, lgID, G, R, HR, SH)) %>%
# Step 3
  arrange(playerID, teamID, yearID)
```

A good practice when you import a dataset is to use the `glimpse()` function to have an idea about the structure of the dataset.

```
# Structure of the data
glimpse(data)
```

```
## Observations: 20,563
## Variables: 9
## $ playerID <fctr> aardsda01, aardsda01, aaronha01, aaronha01, aaronha0...
## $ yearID   <int> 2009, 2010, 1973, 1957, 1962, 1975, 1986, 1979, 1980,...
## $ AB       <int> 0, 0, 392, 615, 592, 465, 0, 0, 0, 0, 0, 0, 45, 610, ...
## $ teamID   <fctr> SEA, SEA, ATL, ML1, ML1, ML4, BAL, CAL, CAL, CAL, LA...
## $ lgID     <fctr> AL, AL, NL, NL, NL, AL, AL, AL, AL, AL, NL, AL, NA, ...
## $ G        <int> 73, 53, 120, 151, 156, 137, 66, 37, 40, 24, 32, 18, 1...
## $ R        <int> 0, 0, 84, 118, 127, 45, 0, 0, 0, 0, 0, 0, 3, 70, 20, ...
## $ HR       <int> 0, 0, 40, 44, 45, 12, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0...
## $ SH       <int> 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, NA, 5, 8, 0, NA, ...
```

Function `summarise()`

The syntax of `summarise()` is basic and consistent with the other verbs included in the `dplyr` library.

```
summarise(df, variable_name = condition)
arguments:

- `df`: Dataset used to construct the summary statistics
- `variable_name = condition`: Formula to create the new variable
```

Look at the code below:

```
summarise(data, mean_run = mean(R))
```

```
##   mean_run
## 1 19.20114
```

Code Explanation

- `summarise(data, mean_run = mean(R))`: Creates a variable named `mean_run` which is the average of the column `run` from the dataset `data`.

You can add as many variables as you want. You return the average games played and the average sacrifice hits.

```
summarise(data, mean_games = mean(G),
                mean_SH = mean(SH, na.rm = TRUE))
```

```
##   mean_games  mean_SH
## 1   51.98361 2.340085
```

Code Explanation

- `mean_SH = mean(SH, na.rm = TRUE)`: Summarize a second variable. You set `na.rm = TRUE` because the column `SH` contains missing observations.

4.2.5 Group_by vs no group_by

The function `summerise()` without `group_by()` does not make any sense. It is more instructive to construct a summary statistic by group. The library `dplyr` applies a function automatically to the group you passed inside the verb `group_by`.

Note that, `group_by` works perfectly with all the other verbs (i.e. `mutate()`, `filter()`, `arrange()`, ...).

It is convenient to use the pipeline operator when you have more than one step. You can compute the average homerun by baseball league.

```
data %>%
  group_by(lgID) %>%
  summarise(mean_run = mean(HR))
```

```
## # A tibble: 7 x 2
##     lgID  mean_run
##   <fctr>     <dbl>
## 1     AA 0.9166667
## 2     AL 3.1270988
## 3     FL 1.3131313
## 4     NL 2.8595953
## 5     PL 2.5789474
## 6     UA 0.6216216
## 7   <NA> 0.2867133
```

Code Explanation

- `data`: Dataset used to construct the summary statistics
- `group_by(lgID)`: Compute the summary by grouping the variable 'lgID
- `summarise(mean_run = mean(HR))`: Compute the average homerun

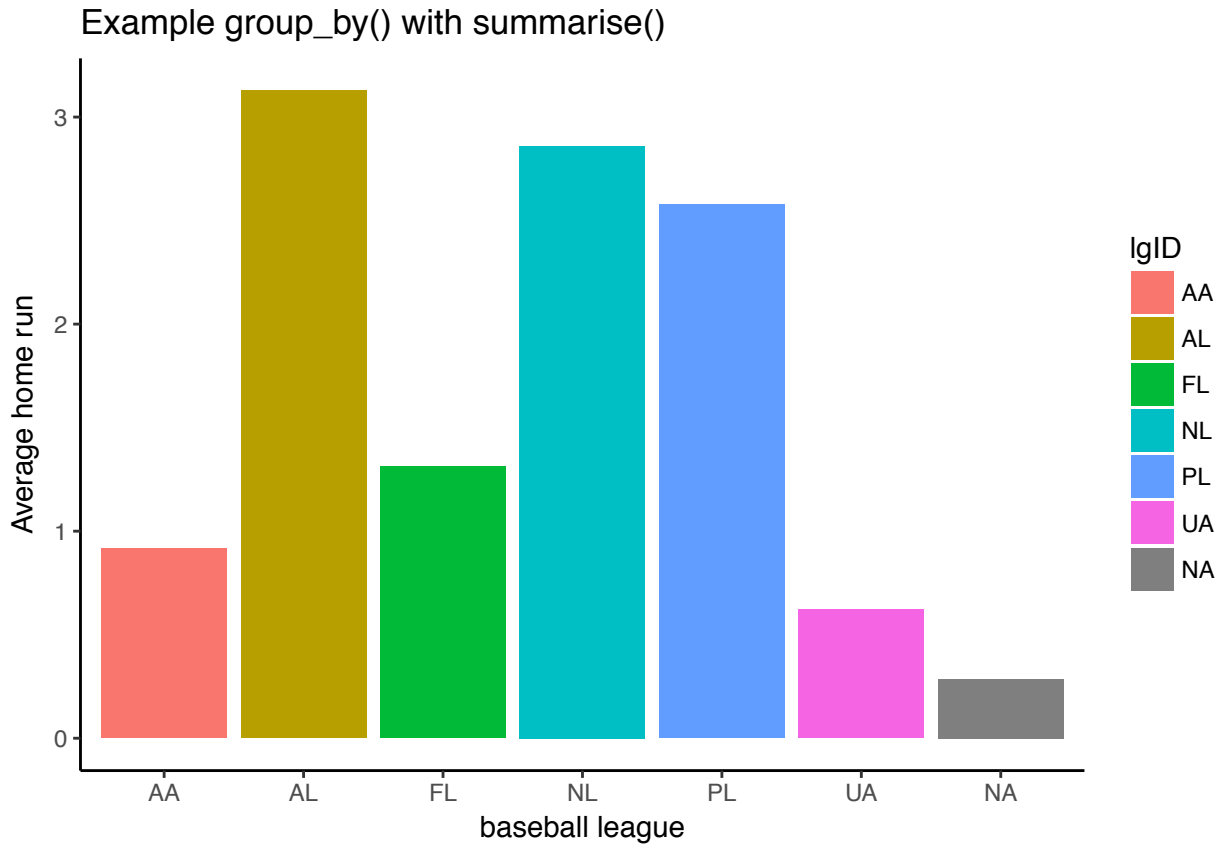
The pipe operator works with `ggplot()` as well. You can easily show the summary statistic with a graph. All the steps are pushed inside the pipeline until the grap is plot. It seems more visual to see the average homerun by league with a bar char. The code below demonstrates the power of combining `group_by()`, `summarise()` and `ggplot()` together.

You will do the following step:

- Step 1: Select data frame
- Step 2: Group data
- Step 3: Summarize the data
- Step 4: Plot the summary statistics

```
library(ggplot2)

# Step 1
data %>%
```

```r
# Step 2
  group_by(lgID)%>%
# Step 3
  summarise(mean_home_run = mean(HR)) %>%
# Step 4
  ggplot(aes(x = lgID, y = mean_home_run, fill = lgID))+
  geom_bar(stat="identity") +
  theme_classic()+
  labs(
    x ="baseball league",
    y = "Average home run",
    title = paste(
        "Example group_by() with summarise()"
    )
  )
```



4.2.6 Functions compatible

The verb `summarise()` is compatible with almost all the functions in R. Here is a short list of useful functions you can use together with `summarise()`:

Objective	Function	Description
Basic	mean()	Average of vector x
	median()	Median of vector x
	sum()	Sum of vector x

Objective	Function	Description
variation	sd()	standard deviation of vector x
	IQR()	Interquartile of vector x
Range	min()	Minimum of vector x
	max()	Maximum of vector x
	quantile()	Quantile of vector x
Position	first()	Use with group_by() First observation of the group
	last()	Use with group_by(). Last observation of the group
	nth()	Use with group_by(). nth observation of the group
Count	n()	Use with group_by(). Count the number of rows
	n_distinct()	Use with group_by(). Count the number of distinct observations

In the previous example, you didn't store the summary statistic in a data frame.

You can proceed in two steps to generate a date frame from a summary:

- Step 1: Store the data frame for further use
- Step 2: Use the dataset to create a line plot

Step 1

You compute the average number of games played by year.

```
## Mean
ex1 <- data %>%
    group_by(yearID) %>%
    summarise(mean_game_year = mean(G))
head(ex1)
```

```
## # A tibble: 6 x 2
##   yearID mean_game_year
##    <int>          <dbl>
## 1   1871       23.42308
## 2   1872       18.37931
## 3   1873       25.61538
## 4   1874       39.05263
## 5   1875       28.39535
## 6   1876       35.90625
```

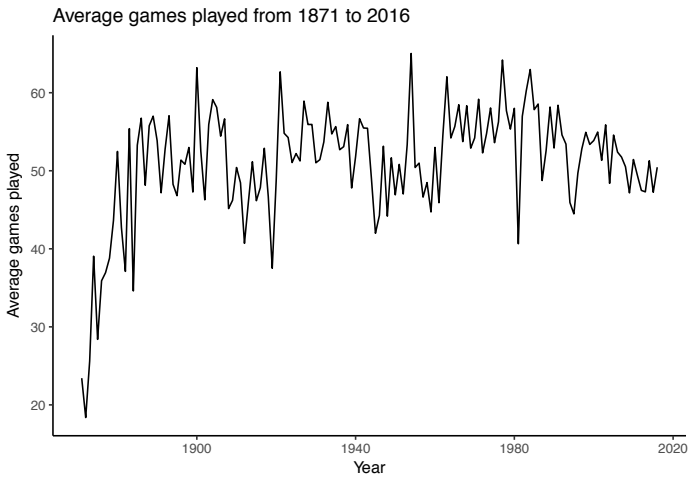
Code Explanation

- The summary statistic of `batting` dataset is stored in the data frame `ex1`.

Step 2

you show the summary statistic with a line plot and see the trend.

```
# Plot the graph
ggplot(ex1, aes(x = yearID, y = mean_game_year)) +
  geom_line() +
  theme_classic() +
    labs(
    x ="Year",
    y = "Average games played",
    title = paste(
        "Average games played from 1871 to 2016"
    )
  )
```



4.2.7 Subsetting

The function `summarise()` is compatible with subsetting.

```
## Subsetting + Median
data %>%
    group_by(lgID) %>%
    summarise(median_at_bat_league = median(AB),
              # Compute the median without the zero
              median_at_bat_league_no_zero = median(AB[AB > 0]))
```

```
## # A tibble: 7 x 3
##     lgID median_at_bat_league median_at_bat_league_no_zero
##   <fctr>                <dbl>                         <dbl>
## 1     AA                  130                           131
## 2     AL                   38                            85
## 3     FL                   88                            97
## 4     NL                   56                            67
## 5     PL                  238                           238
## 6     UA                   35                            35
## 7   <NA>                  101                           101
```

Code Explanation

- `median_at_bat_league_no_zero = median(AB[AB > 0])`: The variable `AB` contains lots of 0. You can compare the median of the **at bat** variable with and without 0.

Sum

Another useful function to aggregate the variable is `sum()`.

You can check which leagues have the more homeruns.

```
## Sum
data %>%
    group_by(lgID) %>%
    summarise(sum_homerun_league = sum(HR))
```

```
## # A tibble: 7 x 2
##     lgID sum_homerun_league
##   <fctr>              <int>
## 1     AA                341
```

```
## 2     AL              29426
## 3     FL                130
## 4     NL              29817
## 5     PL                 98
## 6     UA                 46
## 7   <NA>                 41
```

Standard deviation

Spread in the data is computed with the standard deviation or `sd()` in R.

```
# Spread
data %>%
    group_by(teamID) %>%
    summarise(sd_at_bat_league = sd(HR))
```

```
## # A tibble: 148 x 2
##    teamID sd_at_bat_league
##    <fctr>            <dbl>
##  1    ALT               NA
##  2    ANA        8.7816395
##  3    ARI        6.0765503
##  4    ATL        8.5363863
##  5    BAL        7.7350173
##  6    BFN        1.3645163
##  7    BFP        0.4472136
##  8    BL1        0.6992059
##  9    BL2        1.7106757
## 10    BL3        1.0000000
## # ... with 138 more rows
```

There are lots of inequality in the quantity of homerun done by each team.

Minimum and maximum

You can access the minimum and the maximum of a vector with the function `min()` and `max()`.

The code below returns the lowest and highest number of games in a season played by a player.

```
# Min and max
data %>%
    group_by(playerID) %>%
    summarise(min_G = min(G),
              max_G = max(G))
```

```
## # A tibble: 10,395 x 3
##     playerID min_G max_G
##       <fctr> <dbl> <dbl>
##  1 aardsda01    53    73
##  2 aaronha01   120   156
##  3  aasedo01    24    66
##  4  abadfe01    18    18
##  5 abadijo01    11    11
##  6 abbated01     3   153
##  7 abbeybe01    11    11
##  8 abbeych01    80   132
##  9 abbotgl01     5    23
## 10 abbotji01    13    29
## # ... with 10,385 more rows
```

Count

Count observations by group is always a good idea. With R, you can aggregate the the number of occurence with `n()`.

For instance, the code below computes the number of years played by each player.

```
# count observations
data %>%
    group_by(playerID) %>%
    summarise(number_year = n()) %>%
    arrange(desc(number_year))
```

```
## # A tibble: 10,395 x 2
##     playerID number_year
##       <fctr>       <int>
##  1 pennohe01          11
##  2 joosted01          10
##  3 mcguide01          10
##  4  rosepe01          10
##  5 davisha01           9
##  6 johnssi01           9
##  7  kaatji01           9
##  8 keelewi01           9
##  9 marshmi01           9
## 10 quirkja01           9
## # ... with 10,385 more rows
```

First and last

You can select the first, last or nth position of a group.

For instance, you can find the first and last year of each player.

```
# first and last
data %>%
      group_by(playerID) %>%
      summarise(first_appearance = first(yearID),
                last_appearance = last(yearID))
```

```
## # A tibble: 10,395 x 3
##     playerID first_appearance last_appearance
##       <fctr>            <int>           <int>
##  1 aardsda01             2009            2010
##  2 aaronha01             1973            1975
##  3  aasedo01             1986            1990
##  4  abadfe01             2016            2016
##  5 abadijo01             1875            1875
##  6 abbated01             1905            1897
##  7 abbeybe01             1894            1894
##  8 abbeych01             1895            1897
##  9 abbotgl01             1973            1979
## 10 abbotji01             1992            1996
## # ... with 10,385 more rows
```

nth observation

The fonction `nth()` is complementary to `first()` and `last()`. You can access the nth observation within a group with the index to return.

For instance, you can filter only the second year that a team played.

```
# nth
data %>%
      group_by(teamID) %>%
      summarise(second_game = nth(yearID,2)) %>%
      arrange(second_game)
```

```
## # A tibble: 148 x 2
##    teamID second_game
##    <fctr>       <int>
##  1    BS1        1871
##  2    CH1        1871
##  3    FW1        1871
##  4    NY2        1871
##  5    RC1        1871
##  6    BR1        1872
##  7    BR2        1872
##  8    CL1        1872
##  9    MID        1872
## 10    TRO        1872
## # ... with 138 more rows
```

Distinct number of observations

The function `n()` returns the number of observations in a current group. A closed function to `n()` is `n_distinct()`, which count the number of unique values.

In the next example, you add up the total of players a team recruited during the all periods.

```
# distinct values
data %>%
    group_by(teamID) %>%
    summarise(number_player = n_distinct(playerID)) %>%
    arrange(desc(number_player))
```

```
## # A tibble: 148 x 2
##    teamID number_player
##    <fctr>         <int>
##  1    CHN           751
##  2    SLN           729
##  3    PHI           699
##  4    PIT           683
##  5    CIN           679
##  6    BOS           647
##  7    CLE           646
##  8    CHA           636
##  9    DET           623
## 10    NYA           612
## # ... with 138 more rows
```

Code Explanation

- `group_by(teamID)`: Group by year and team
- `summarise(number_player = n_distinct(playerID))`: Count the distinct number of players by team
- `arrange(desc(number_player))`: Sort the data by the number of player

4.2.8 Multiple groups

A summary statistic can be realized among multiple groups.

```
# Multiple groups
data %>%
  group_by(yearID, teamID) %>%
  summarise(mean_games = mean(G)) %>%
  arrange(desc(teamID, yearID))
```

```
## # A tibble: 2,829 x 3
## # Groups:   yearID [146]
##    yearID teamID mean_games
##     <int> <fctr>      <dbl>
##  1   1884    WSU   20.41667
##  2   1891    WS9   46.33333
##  3   1886    WS8   22.00000
##  4   1887    WS8   51.00000
##  5   1888    WS8   27.00000
##  6   1889    WS8   52.42857
##  7   1884    WS7    8.00000
##  8   1875    WS6   14.80000
##  9   1873    WS5   16.62500
## 10   1872    WS4    4.20000
## # ... with 2,819 more rows
```

Code Explanation

- `group_by(yearID, teamID)`: Group by year **and** team
- `summarise(mean_games = mean(G))`: Summarize the number of game player
- `arrange(desc(teamID, yearID))`: Sort the data by team and year

4.2.9 Filter

Before you intend to do an operation, you can filter the dataset. The dataset starts in 1871, and the analysis does not need the years prior to 1980.

```
# Filter
data %>%
    filter(yearID > 1980) %>%
    group_by(yearID) %>%
    summarise(mean_game_year = mean(G))
```

```
## # A tibble: 36 x 2
##    yearID mean_game_year
##     <int>          <dbl>
##  1   1981       40.64583
##  2   1982       56.97790
##  3   1983       60.25128
##  4   1984       62.97436
##  5   1985       57.82828
##  6   1986       58.55340
##  7   1987       48.74752
##  8   1988       52.57282
##  9   1989       58.16425
## 10   1990       52.91556
```

```
## # ... with 26 more rows
```

Code Explanation

- `filter(yearID > 1980)`: Filter the data to show only the relevant years (i.e. after 1980)
- `group_by(yearID)`: Group by year
- `summarise(mean_game_year = mean(G))`: Summarize the data

4.2.10 Ungroup

Last but not least, you need to remove the grouping before you want to change the level of the computation.

```
# Ungroup the data
data %>%
     filter(HR >0)%>%
     group_by(playerID) %>%
     summarise(average_HR_game = sum(HR)/sum(G)) %>%
     ungroup() %>%
     summarise(total_average_homerun = mean(average_HR_game))
```

```
## # A tibble: 1 x 1
##   total_average_homerun
##                   <dbl>
## 1            0.06882226
```

Code Explanation

- `filter(HR >0)` : Exclude zero homerun
- `group_by(playerID)`: group by player
- `summarise(average_HR_game = sum(HR)/sum(G))`: Compute average homerun by player
- `ungroup()`: remove the grouping
- `summarise(total_average_homerun = mean(average_HR_game))`: Summarize the data

4.2.11 Summary

When you want to return a summary by group, you can use:

```
# group by X1, X2, X3
group(df, X1, X2, X3)
```

you need to ungroup the data with:

```
ungroup(df)
```

The table below summarizes the function you learnt with `summarise()`

method	function	code
mean	mean	summarise(df,mean_x1 = mean(x1))
median	median	summarise(df,median_x1 = median(x1))
sum	sum	summarise(df,sum_x1 = sum(x1))
standard deviation	sd	summarise(df,sd_x1 = sd(x1))
interquartile	IQR	summarise(df,interquartile_x1 = IQR(x1))
minimum	min	summarise(df,minimum_x1 = min(x1))
maximum	max	summarise(df,maximum_x1 = max(x1))
quantile	quantile	summarise(df,quantile_x1 = quantile(x1))
first observation	first	summarise(df,first_x1 = first(x1))
last observation	last	summarise(df,last_x1 = last(x1))

method	function	code
nth observation	nth	summarise(df,nth_x1 = nth(x1, 2))
number of occurrence	n	summarise(df,n_x1 = n(x1))
number of distinct occurrence	n_distinct	summarise(df,n_distinct _x1 = n_distinct(x1))

4.2.12 Data cleaning

Missing values in data science arise when an observation is missing in a column of a data frame or contains a character value instead of numeric value. Missing values must be dropped or replaced in order to draw correct conclusion from the data.

In this chapter, you will learn how to deal with missing values with the `dplyr` library. `dplyr` library is part of an ecosystem to realize a data analysis.

Function `mutate()`

The fourth verb in the `dplyr` library is helpful to create new variables or change the values of existing variables.

You will proceed in two parts. You will learn how to:

- exclude missing values from a data frame
- impute missing values with the mean and median

The verb `mutate()` is very easy to use. You can create a new variable following this syntax:

```
mutate(df, name_variable_1 = condition, ...)
arguments

- df: Data frame used to create a new variable
- name_variable_1: Name and the formula to create the new variable
- ...: No limit constraint. Possibility to create more than one variable inside `mutate()
```

4.2.13 Exclude missing values

The `na.omit()` method from the `dplyr` library is a simple way to exclude missing observation. Dropping all the `NA` from the data is easy but it does not mean it is the most elegant solution. During analysis, it is wise to use variety of methods to deal with missing values

To tackle the problem of missing observations, you will use the titanic dataset. In this dataset, you have access to the information of the passengers on board during the tragedy. This dataset has many `NA` that need to be taken care of.

You will upload the csv file from the internet and then check which columns have 'NA. To return the columns with missing data, you can use the following code:

Let's upload the data and verify the missing values.

```
PATH <- "https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/titanic_csv.csv"
df_titanic <- read.csv(PATH, sep = ",")
# Return the column names containing missing observations
list_na <- colnames(df_titanic)[ apply(df_titanic, 2, anyNA) ]
list_na
```

```
## [1] "age"  "fare"
```

Here,

```
colnames(df_titanic)[ apply(df_titanic, 2, anyNA) ]
```

Gives the name of columns that do not have data.

The columns `age` and `fare` have missing values.

You can drop them with the `na.omit()`.

```
library(dplyr)
# Exclude the missing observations
df_titanic_drop <- df_titanic %>%
                   na.omit()
dim(df_titanic_drop)
```

```
## [1] 1045   13
```

The dataset contains 1045 rows compared to 1309 with the original dataset.

4.2.14 Impute missing values

You could also impute (populate) missing values with the median or the mean. A good practice is to create two separate variables for the mean and the median. Once created, you can replace the missing values with the newly formed variables.

You will use the apply method to compute the mean of the column with `NA`. Let's see an example

Step 1

Earlier in the tutorial, you stored the columns name with the missing values in the list called list_na. You will use this list

Step 2

Now you need to compute of the mean with the argument 'na.rm = TRUE. This argument is compulsory because the columns have missing values, and this tells R to ignore them.

```
# Create mean
average_missing <- apply(df_titanic[,colnames(df_titanic) %in% list_na],
      2,
      mean,
      na.rm =  TRUE)
average_missing
```

```
##      age     fare
## 29.88113 33.29548
```

Code Explanation:

You pass 4 arguments in the apply method.

- df: `df_titanic[,colnames(df_titanic) %in% list_na]`. This code will return the columns name from the list_na object (i.e. "age" and "fare")
- '2: Compute the function on the columns
- `mean`: Compute the mean
- `na.rm = TRUE`: Ignore the missing values

You successfully created the mean of the columns containing missing observations. These two values will be used to replace the missing observations.

Step 3

Replace the `NA` Values

The verb `mutate()` from the `dplyr` library is useful in creating a new variable. You don't necessarily want to change the original column so you can create a new variable without the `NA`. Mutate is easy to use, you just choose a variable name and define how to create this variable.

Here is the complete code

```
# Create a new variable with the mean and median
df_titanic_replace <- df_titanic %>%
            mutate(replace_mean_age  = ifelse(is.na(age), average_missing[1], age),
                   replace_mean_fare = ifelse(is.na(fare), average_missing[2], fare))

sum(is.na(df_titanic_replace$age))
```

```
## [1] 263
```

```
sum(is.na(df_titanic_replace$replace_mean_age))
```

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

Code Explanation:

You create two variables, replace_mean_age and replace_mean_fare as follow:

- `replace_mean_age  = ifelse(is.na(age), average_missing[1], age)`
- 'replace_mean_fare = ifelse(is.na(fare), average_missing[2], fare)

If the column `age` has missing values, then replace with the first element of `average_missing` (mean of age), else keep the original values. Same logic for `fare`.

```
sum(is.na(df_titanic_replace$age))
```

```
## [1] 263
```

```
## [1] 263
sum(is.na(df_titanic_replace$replace_mean_age))
```

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

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

The original column age has 263 missing values while the newly created variable have replaced them with the mean of the variable `age`.

Step 4

You can replace the missing observation with the median as well.

```
median_missing <- apply(df_titanic[,colnames(df_titanic) %in% list_na],
      2,
      median,
      na.rm =  TRUE)

df_titanic_replace <- df_titanic %>%
            mutate(replace_median_age  = ifelse(is.na(age), median_missing[1], age),
                   replace_median_fare = ifelse(is.na(fare), median_missing[2], fare))
head(df_titanic_replace)
```

```
##   X pclass survived                                            name    sex
## 1 1      1        1                   Allen, Miss. Elisabeth Walton female
## 2 2      1        1                  Allison, Master. Hudson Trevor   male
## 3 3      1        0                    Allison, Miss. Helen Loraine female
## 4 4      1        0            Allison, Mr. Hudson Joshua Creighton   male
```

```
## 5 5      1        0 Allison, Mrs. Hudson J C (Bessie Waldo Daniels) female
## 6 6      1        1                              Anderson, Mr. Harry   male
##       age sibsp parch ticket     fare   cabin embarked
## 1 29.0000     0     0  24160 211.3375      B5        S
## 2  0.9167     1     2 113781 151.5500 C22 C26        S
## 3  2.0000     1     2 113781 151.5500 C22 C26        S
## 4 30.0000     1     2 113781 151.5500 C22 C26        S
## 5 25.0000     1     2 113781 151.5500 C22 C26        S
## 6 48.0000     0     0  19952  26.5500     E12        S
##                         home.dest replace_median_age replace_median_fare
## 1                     St Louis, MO            29.0000            211.3375
## 2 Montreal, PQ / Chesterville, ON             0.9167            151.5500
## 3 Montreal, PQ / Chesterville, ON             2.0000            151.5500
## 4 Montreal, PQ / Chesterville, ON            30.0000            151.5500
## 5 Montreal, PQ / Chesterville, ON            25.0000            151.5500
## 6                     New York, NY            48.0000             26.5500
```

Step 5

A big data set could have lots of missing values and the above method could be cumbersome. You can execute all the above steps above in one line of code using 'sapply() method. Though you would not know the vales of mean and median.

`sapply` does not create a data frame, so you can wrap the `sapply()` function within `data.frame()` to create a data frame object.

```
# Quick code to replace missing values with the mean
df_titanic_impute_mean <- data.frame(
  sapply(
    df_titanic,
    function(x) ifelse(is.na(x),
                       mean(x, na.rm = TRUE),
                       x)))
```

4.2.15 Summary

You have three methods to deal with missing values:

- Exclude all of the missing observations
- Impute with the mean
- Impute with the median

The table below summarizes how to remove all the missing observations

Library	Objective	Code
base	List missing observations	colnames(df)[apply(df, 2, anyNA)]
dplyr	Remove all missing values	na.omit(df)

The table below summarises the two ways to impute missing values.

Method	Objective	Details	Advantages	Disavantages
Step by step	Impute missing	Check columns with missing, compute mean/median, store the value, replace with mutate()	Know the value of means/median	More step. Can be slow with big dataset

Method	Objective	Details	Advantages	Disavantages
Quick way	Impute missing	Use sapply() and data.frame() to automatically search and replace missing values with mean/median	Short code and fast	Don't know the imputation values

Example of the codes with the mean:

- Step by step

```
# Step 1: Check columns with missing
list_na <- colnames(df)[ apply(df, 2, anyNA) ]
# Step 2: Compute mean/median
average_missing <- apply(df[,colnames(df) %in% list_na],
      2,
      mean,
      na.rm =  TRUE)
# Step 3: Replace with mutate()
df_mean_replace <- df %>%
            mutate(replace_mean_X1  = ifelse(is.na(X1), average_missing[1], X1))
```

- Quick way

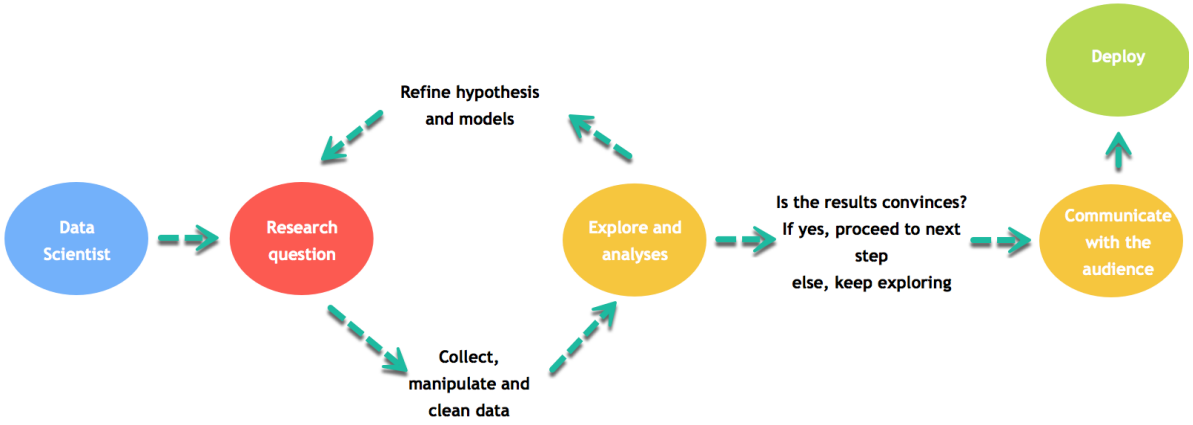
```
data.frame(sapply(df,function(x) ifelse(is.na(x),mean(x, na.rm = TRUE),x)))
```

4.3 Data Visualization

Graphs are the third part of the process of data analysis. The first part is about **data extraction**, the second part deals with **cleaning and manipulating the data**. At last, the data scientist may need to **communicate his results graphically**.

The job of the data scientist can be reviewed in the following picture

- The first task of a data scientist is to define a research question. This research question depends on the objectives and goals of the project.
- After that, one of the most prominent tasks is the feature engineering. The data scientist needs to collect, manipulate and clean the data
- When this step is completed, he can start to explore the dataset. Sometimes, it is necessary to refine and change the original hypothesis due to a new discovery.



When the **explanatory** analysis is achieved, the data scientist has to consider the capacity of the reader to **understand the underlying concepts and models**. His results should be presented in a format that all

stakeholders can understand. One of the best methods to **communicate** the results is through a **graph**. Graphs are incredible tools to simplify complex analysis.

This part of the tutorial focuses on how to make graphs/charts with R.

In this tutorial, you are going to use `ggplot2` package. This package is built upon the consistent underlying of the book *Grammar of graphics* written by Wilkinson, 2005. `ggplot2` is very flexible, incorporates many themes and plot specification at a high level of abstraction. With `ggplot2`, you can't plot 3-dimensional graphics and create interactive graphics.

In `ggplot2`, a graph is composed of the following arguments:

- data
- aesthetic mapping
- geometric object
- statistical transformations
- scales
- coordinate system
- position adjustments
- faceting

You will learn how to control those arguments in the tutorial.

The basic syntax of `ggplot2` is:

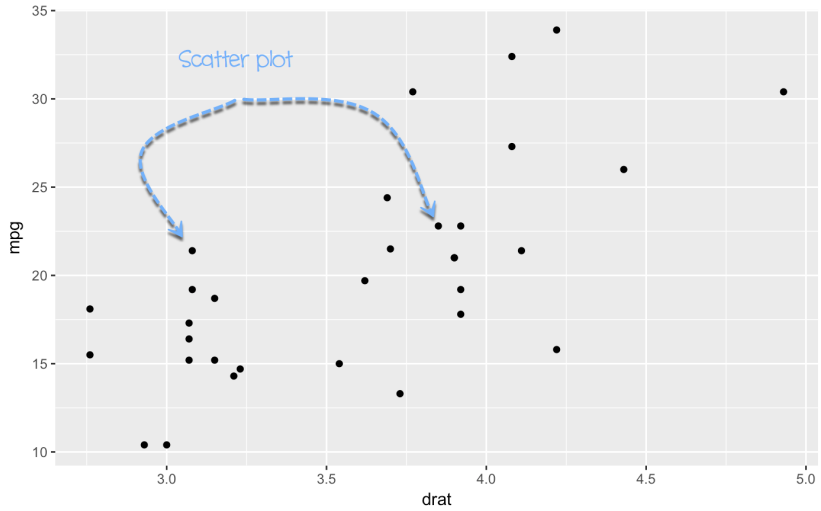
```
ggplot(data, mapping = aes()) +
geometric object

arguments:
data:  Dataset used to plot the graph
mapping: Control the x and y-axis
geometric object: The type of plot you want to show. The most common objects are:

- Point: `geom_point()`
- Bar: `geom_bar()`
- Line: `geom_line()`
- Histogram: `geom_histogram()`
```

4.3.1 Scatterplot

Let's see how ggplot works with the `mtcars` dataset. You start by plotting a scatterplot of the `mpg` variable and `drat` variable.

Basic scatter plot

```
library(ggplot2)
ggplot(mtcars, aes(x= drat, y = mpg))+
  geom_point()
```



Code Explanation

- You first pass the dataset mtcars to ggplot.

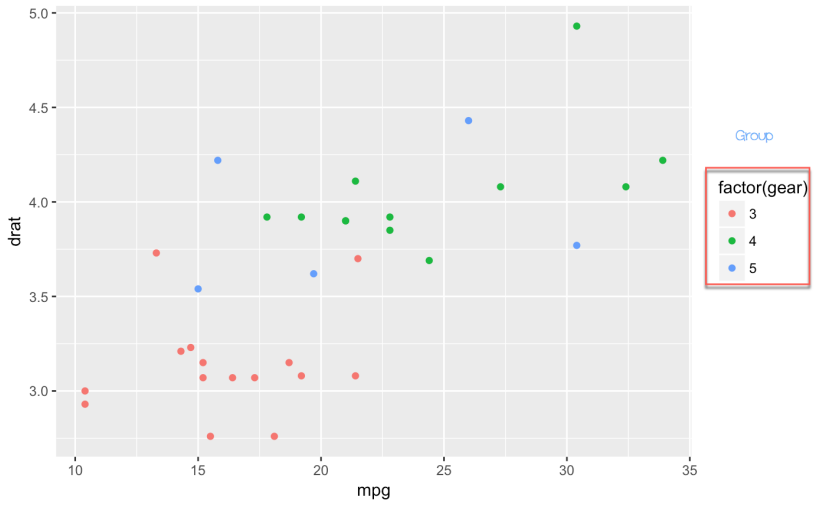
- Inside the `aes()` argument, you add the x-axis and y-axis.
- The `+` sign means you want R to keep reading the code. It makes the code more readable by breaking it.

- Use 'geom_point() for the geometric object.

Scatter plot with groups

Sometimes, it can be interesting to distinguish the values by a group of data (i.e. factor level data).

```
ggplot(mtcars, aes(x= mpg, y = drat))+
  geom_point(aes(color = factor(gear)))
```



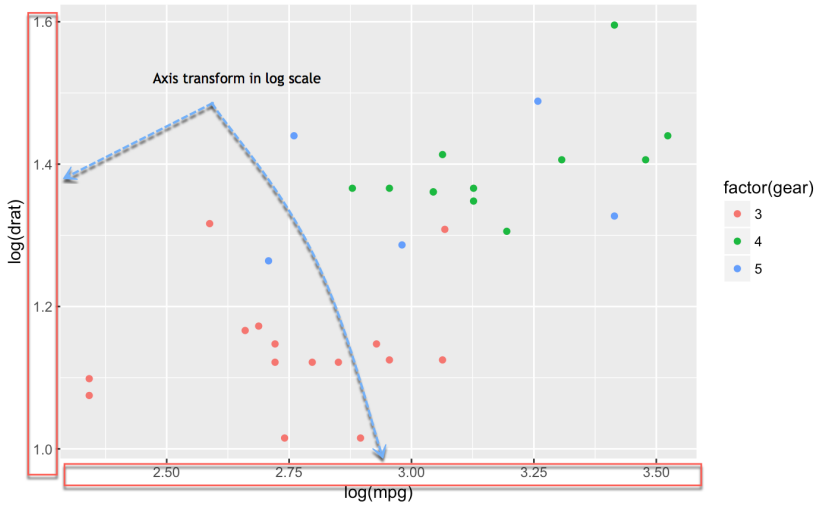
Code Explanation

- The `aes()` inside the `geom_point()` controls the color of the group. The group should be a factor variable. Thus, you convert the variable 'gear in a factor.
- Altogether, you have the code `aes(color = factor(gear))` that changes the color of the dots.

Change axis

Rescale the data is a big part of the data scientist job. In rare occasion data comes in a nice bell shape. One solution to make your data less sensitive to outliers is to rescale them.

```
ggplot(mtcars, aes(x= log(mpg), y = log(drat))) +
  geom_point(aes(color = factor(gear)))
```



Code Explanation

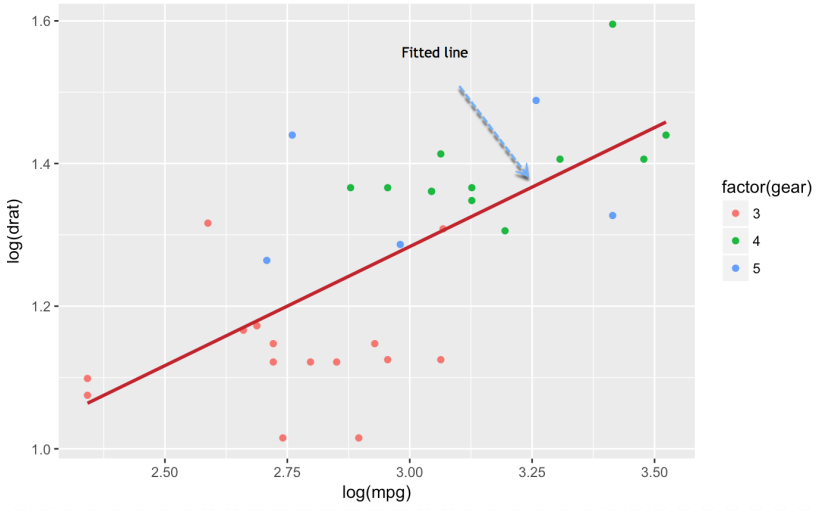
- You transform the x and y variables in `log()` directly inside the `aes()` mapping.

Note that any other transformation can be applied such as standardization or normalization.

Scatter plot with fitted values

You can add another level of information to the graph. You can plot the fitted value of a linear regression.

```
library(ggplot2)
my_graph <- ggplot(mtcars, aes(x= log(mpg), y = log(drat)))+
  geom_point(aes(color = factor(gear)))+
  stat_smooth(method = "lm",
              col = "#C42126",
              se = FALSE,
              size = 1)
```



Code Explanation

- `graph`: You store your graph into the variable `graph`. It is helpful for further use or avoid too complex line of codes
- The argument `stat_smooth()` controls for the smoothing method
- `method = "lm"`: Linear regression
- `col = "#C42126"`: Code for the red color of the line
- `se = FALSE`: Don't display the standard error
- `size = 1`: the size of the line is 1

Note that other smoothing methods are available

- `glm`
- `gam`
- `loess`: default value
- `rim`

Add information to the graph

So far, you haven't added information in the graphs. Graphs need to be informative. The reader should see the story behind the data analysis just by looking at the graph without referring additional documentation. Hence, graphs need good labels. You can add labels with `labs()`function.

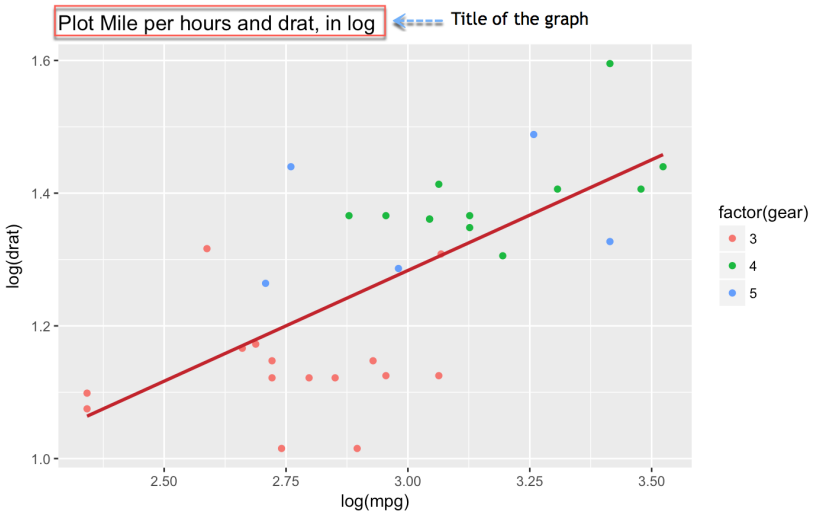
The basic syntax for `lab()` is :

```
lab(title = "Hello Guru99")
argument:

- title: Control the title

It is possible to change or add title with:
  - subtitle: Add subtitle below title
  - caption: Add caption below the graph
  - x: rename x-axis
  - y: rename y-axis

Example:
lab(title = "Hello Guru99", subtitle = "My first plot")
```

Add a title

One mandatory information to add is obviously a title.

```
my_graph +
    labs(
        title = "Plot Mile per hours and drat, in log"
         )
```



Code Explanation

- `my_graph`: You use the graph you stored. It avoids rewriting all the codes each time you add new information to the graph.
- You wrap the title inside the `lab()`.

Add a title with a dynamic name

A dynamic title is helpful to add more precise information in the title.

You can use the `paste()` function to print static text and dynamic text. The basic syntax of `paste()` is:

```
paste("This is a text", A)
arguments

- " ": Text inside the quotation marks are the static text
- A: Display the variable stored in A
- Note you can add as much static text and variable as you want. You need to separate them with a comma
```

Example:

```
A <- 2010
paste("The first year is", A)
```

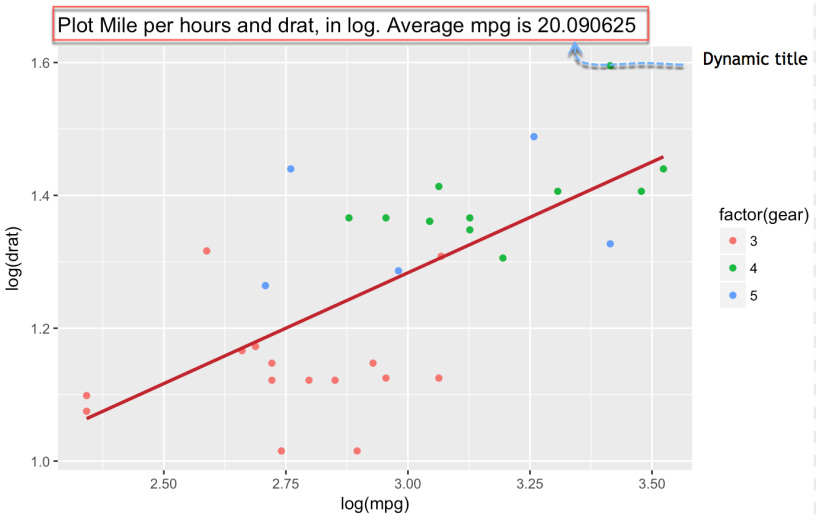
```
## [1] "The first year is 2010"
```

```
B <- 2018
paste("The first year is", A, "and the last year is", B)
```

```
## [1] "The first year is 2010 and the last year is 2018"
```

You can add a dynamic name to your graph, namely the average of `mpg`.

```
mean_mpg <- mean(mtcars$mpg)
my_graph +
  labs(
        title = paste("Plot Mile per hours and drat, in log. Average mpg is", mean_mpg)
        )
```



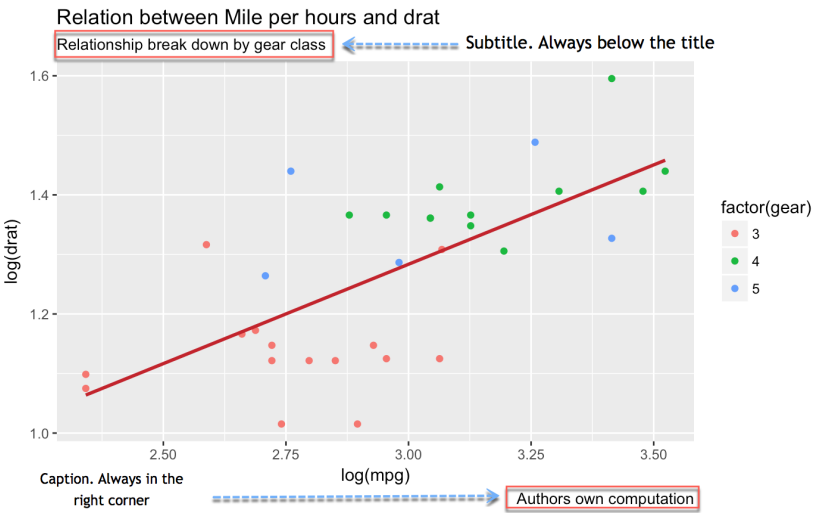
Code Explanation

- You create the average of `mpg` with `mean(mtcars$mpg)` stored in `mean_mpg` variable
- You use the `paste()` with `mean_mpg` to create a dynamic title returning the mean value of `mpg`

Add a subtitle

Two additional details can make your graph more explicit. You are talking about the subtitle and the caption. The subtitle goes right below the title. The caption can inform about who did the computation and the source of the data.

```
my_graph +
  labs(
    title =
      "Relation between Mile per hours and drat",
    subtitle =
      "Relationship break down by gear class",
    caption = "Authors own computation"
  )
```



Code Explanation

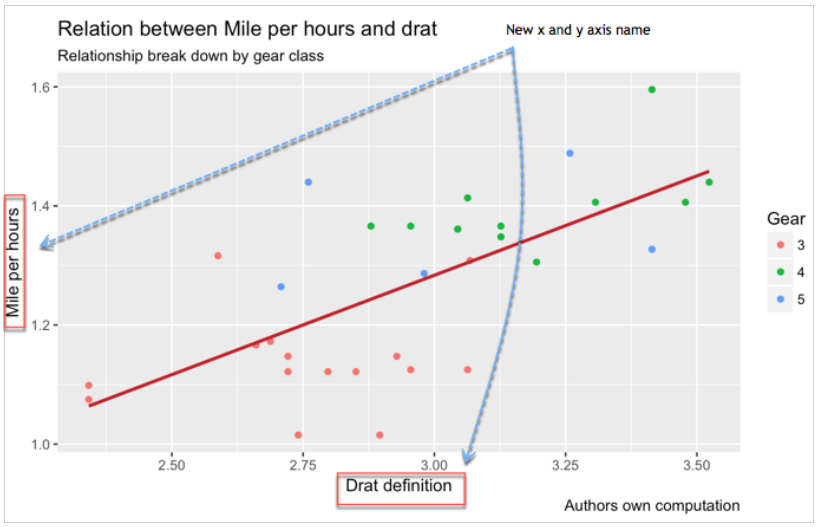
- Inside the `lab()`, you added:

  - `title = "Relation between Mile per hours and drat"`: Add title
  - `subtitle = "Relationship break down by gear class"`: Add subtitle
  - `caption = "Authors own computation`: Add caption
  - You separate each new information with a comma, `,`
- Note that you break the lines of code. It is not compulsory, and it only helps to read the code more easily

Rename x-axis and y-axis

Variables itself in the dataset might not always be explicit or by convention use the `_` when there are multiple words (i.e. `GDP_CAP`). You don't want such name to appear in your graph. It is important to change the name or add more details, like the units.

```
my_graph +
  labs(
    x = "Drat definition",
    y = "Mile per hours",
    color= "Gear",
    title =
      "Relation between Mile per hours and drat",
    subtitle =
      "Relationship break down by gear class",
    caption = "Authors own computation"
  )
```



Code Explanation

- Inside the `lab()`, you added:
  - `x = "Drat definition"`: Change the name of x-axis
  - 'y = "Mile per hours": Change the name of y-axis

Control the scales

You can control the scale of the axis.

The function `seq()` is convenient when you need to create a sequence of number. The basic syntax is:

```
seq(begin, last, by = x)
arguments:

- begin: First number of the sequence
```

- last: Last number of the sequence
- by= x: The step. For instance, if x is 2, the code adds 2 to `begin-1` until it reaches `last`

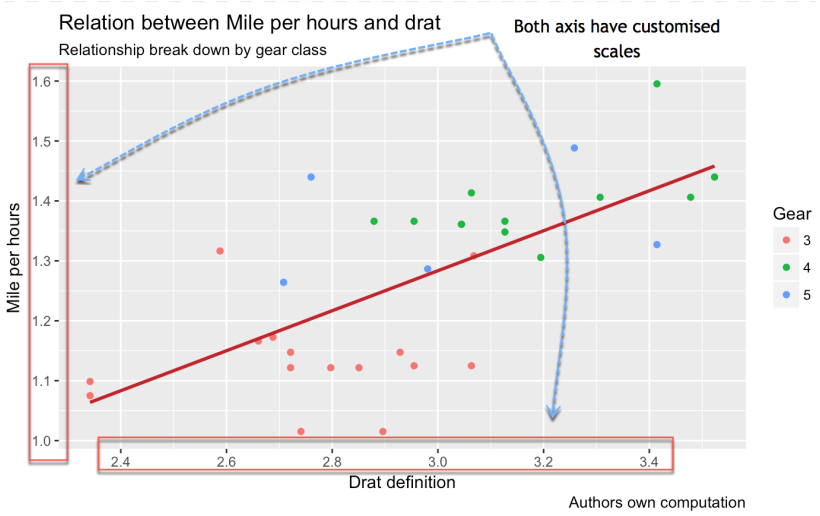
For instance, if you want to create a range from 0 to 12 with a step of 3, you will have four numbers, 0 4 8 12

```
seq(0, 12,4)
```

```
## [1]  0  4  8 12
```

You can control the scale of the x-axis and y-axis as below

```
my_graph +
  scale_x_continuous(breaks = seq(1, 3.6, by = 0.2))+
  scale_y_continuous(breaks = seq(1, 1.6, by = 0.1))+
  labs(
    x = "Drat definition",
    y = "Mile per hours",
    color= "Gear",
    title = "Relation between Mile per hours and drat",
    subtitle = "Relationship break down by gear class",
    caption = "Authors own computation"
  )
```



Code Explanation

- The function `scale_y_continuous()` controls the **y-axis**

- The function `scale_x_continuous()` controls the **x-axis**.
- The parameter `breaks` controls the split of the axis. You can manually add the sequence of number or use the `seq()`function:
  - `seq(1, 3.6, by = 0.2)`: Create six numbers from 2.4 to 3.4 with a step of 3
  - `seq(1, 1.6, by = 0.1)`: Create seven numbers from 1 to 1.6 with a step of 1

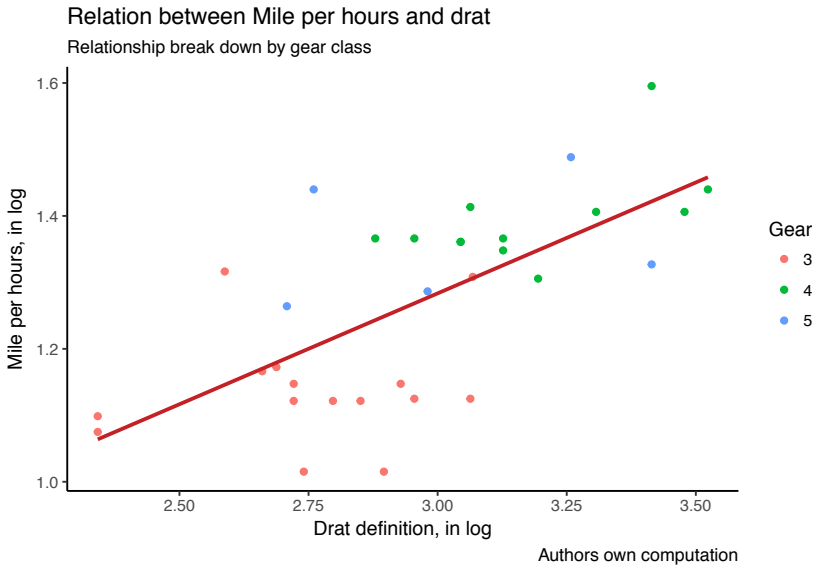
Theme

Finally, R allows us to customize out plot with different themes. The library `ggplot2` includes eights themes:

- `theme_bw()`
- `theme_light()`
- `theme_classis()`
- `theme_linedraw()`
- `theme_dark()`

- theme_minimal()
- theme_gray()
- theme_void()

```
my_graph+
  theme_classic()+
  labs(
    x = "Drat definition, in log",
    y = "Mile per hours, in log",
    color= "Gear",
    title = "Relation between Mile per hours and drat",
    subtitle = "Relationship break down by gear class",
    caption = "Authors own computation"
  )
```



4.3.2 Save Plots

After all these steps, it is time to save and share your graph. You add `ggsave('NAME OF THE FILE)` right after you plot the graph and it will be stored on the hard drive.

The graph is saved in the working directory. To check the working directory, you can run this code:

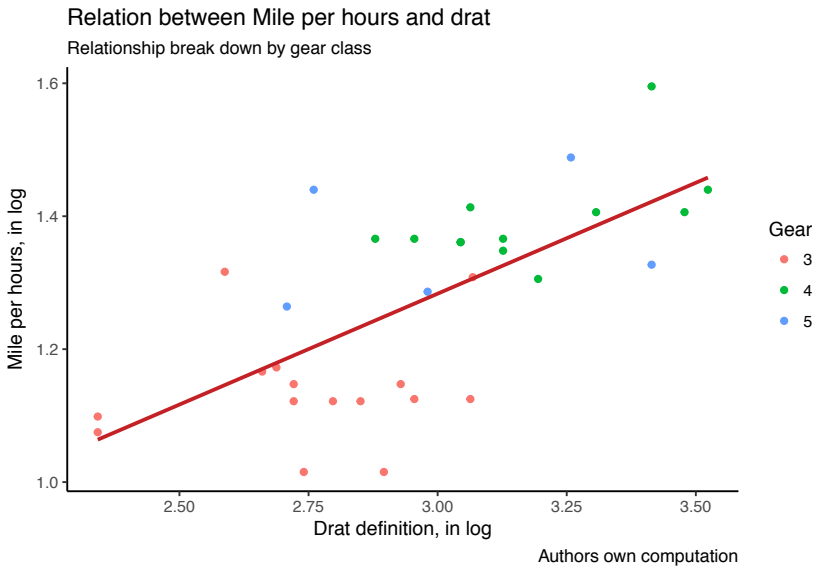
```
directory <- getwd()
directory
```

```
## [1] "/Users/Thomas/Dropbox/Learning/book_R"
```

Let's plot your fantastic graph, saves it and checks the location

```
my_graph +
  theme_classic()+
  labs(
    x = "Drat definition, in log",
    y = "Mile per hours, in log",
    color= "Gear",
    title = "Relation between Mile per hours and drat",
    subtitle = "Relationship break down by gear class",
```

```
    caption = "Authors own computation"
  )
```




```
ggsave("my_fantastic_plot.png")
```

```
## Saving 6.5 x 4.5 in image
```

note: For pedagogical purpose only, you created a function called `open_folder()` to open the directory folder for you. You just need to run the code below and see where the picture is stored. You should see a file names `my_fantastic_plot.png`.

```
# Run this code to create the function
open_folder <- function(dir){
  if (.Platform['OS.type'] == "windows"){
    shell.exec(dir)
  } else {
    system(paste(Sys.getenv("R_BROWSER"), dir))
  }
}

# Call the function to open the folder
open_folder(directory)
```

4.3.3 Summary

You can summarize the arguments to create a scatter plot in the table below:

Objective	Code
Basic scatter plot	ggplot(df, aes(x = x1, y = y)) + geom_point()
Scatter plot with color group	ggplot(df, aes(x = x1, y = y)) + geom_point(aes(color = factor(x1)) + stat_smooth(method = "lm")
Add fitted values	ggplot(df, aes(x = x1, y = y)) + geom_point(aes(color = factor(x1))
Add title	ggplot(df, aes(x = x1, y = y)) + geom_point() + labs(title = paste("Hello Guru99"))
Add subtitle	ggplot(df, aes(x = x1, y = y)) + geom_point() + labs(subtitle = paste("Hello Guru99"))

Objective	Code
Rename x	ggplot(df, aes(x = x1, y = y)) + geom_point() + labs(x = "X1")
Rename y	ggplot(df, aes(x = x1, y = y)) + geom_point() + labs(y = "y1")
Control the scale	ggplot(df, aes(x = x1, y = y)) + geom_point() + scale_y_continuous(breaks = seq(10, 35, by = 10)) + scale_x_continuous(breaks = seq(2, 5, by = 1)
Create logs	ggplot(df, aes(x =log(x1), y = log(y))) + geom_point()
Theme	ggplot(df, aes(x = x1, y = y)) + geom_point() + theme_classic()
Save	ggsave("my_fantastic_plot.png")

Complete code:

```
ggplot(mtcars, aes(x= log10(x1), y = log10(x2)))+
  geom_point(aes(color = factor(x3)))+
  geom_smooth(se = FALSE)+
  theme_classic()+
  labs(
    x = "X1, in log",
    y = "y, in log",
    color= "Gear",
    title = "Hello world",
    subtitle = "My fantastic plot",
    caption = "Made by Guru99"
  )
ggsave("my_fantastic_plot.png")
```

4.3.4 Bar chart

A bar chart is a great way to display categorical variables in the x-axis. This type of graph denotes two aspects in the y-axis.

1. The first one counts the number of occurrence between groups.
2. The second one shows a summary statistic (min, max, average, and so on) of a variable in the y-axis.

You will use the `mtcars` dataset with has the following variables:

- `cyl`: Number of the cylinder in the car. Numeric variable
- `am`: Type of transmission. 0 for automatic and 1 for manual. Numeric variable
- 'mpg: Miles per gallon. Numeric variable

To create graph in R, you can use the library `ggplot` which creates ready-for-publication graphs. The basic syntax of this library is:

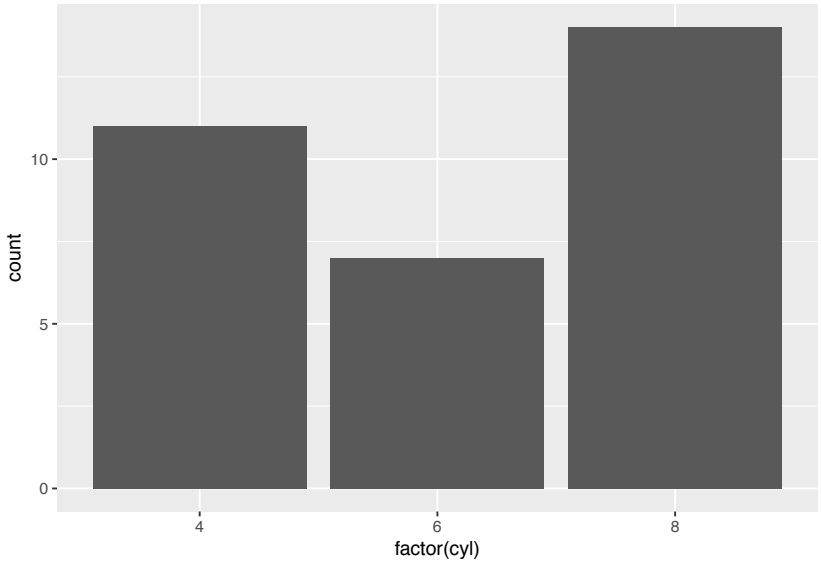
```
ggplot(data, mapping = aes()) +
geometric object

arguments:
data:  dataset used to plot the graph
mapping: Control the x and y axis
geometric object: The type of plot you want to show. The most common object are:

- Point: `geom_point()`
- Bar: `geom_bar()`
- Line: `geom_line()`
- Histogram: `geom_histogram()`
```

In this tutorial, you are interested with the geometric object `geom_bar()` that create the **bar chart**.

Bar chart: count

Our first graph shows the frequency of cylinder with `geom_bar()`. The code below is the most basic syntax.

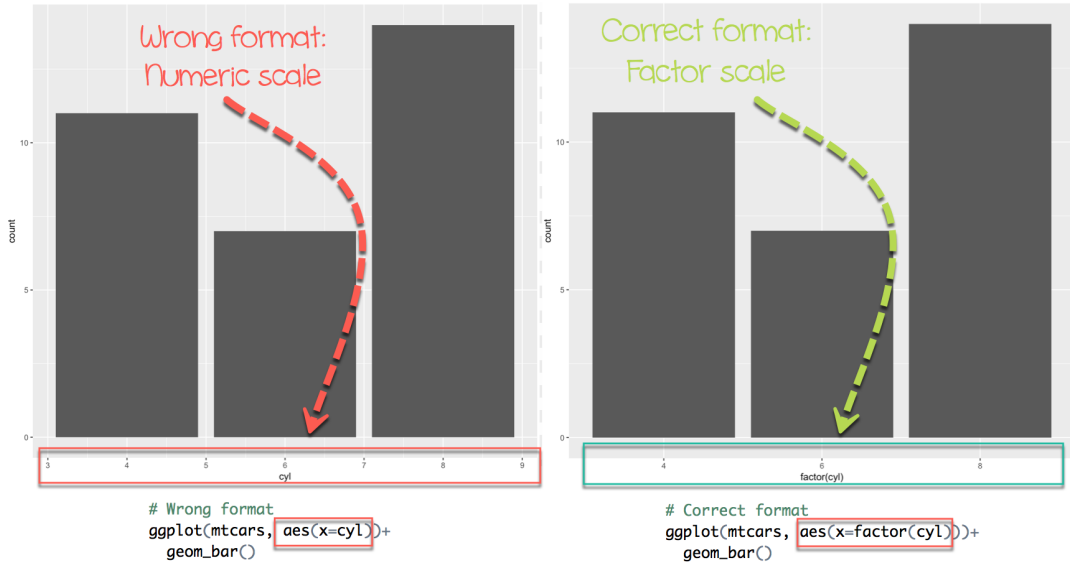
```
library(ggplot2)
# Most basic bar chart
ggplot(mtcars, aes(x=factor(cyl)))+
  geom_bar()
```



Code Explanation

- You pass the dataset `mtcars` to ggplot.
- Inside the `aes()` argument, you add the x-axis as a factor variable.
- The `+` sign means you want R to keep reading the code. It is makes the code more readable by breaking it.
- Use `geom_bar()` for the geometric object.

note: make sure you convert the variables into a factor otherwise R treats the variables as numeric. See the example below.



Customize the graph

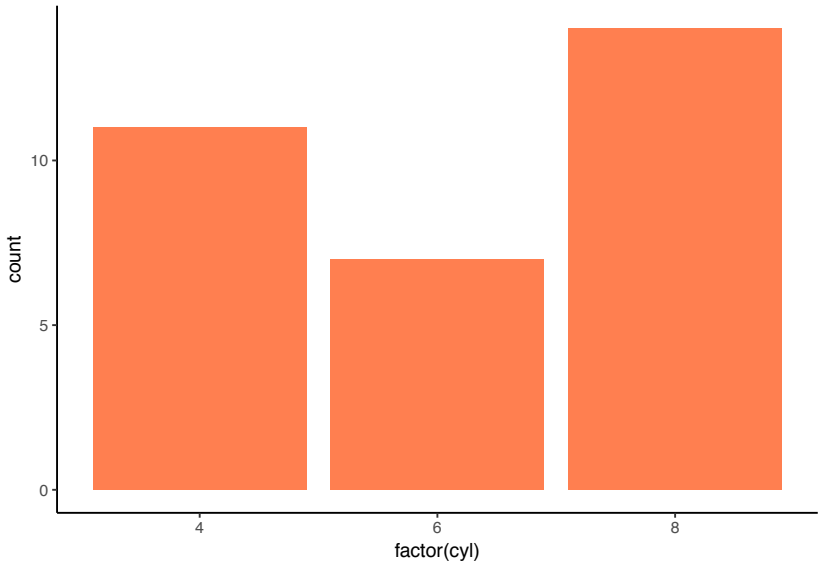
Four arguments can be passed to customize the graph:

```
- `stat`: Control the type of formatting. By default, `bin` to plot a count in the y-axis. For continuou
- `alpha`: Control density of the color
- `fill`: Change the color of the bar
- `size`: Control the size the bar
```

Change the color of the bars

You can change the color of the bars. Note that the colors of the bars are all similar.

```
# Change the color of the bars
ggplot(mtcars, aes(x=factor(cyl)))+
  geom_bar(fill = "coral") +
  theme_classic()
```



Code Exlanation

- The colors of the bars are controlled by the `aes()` mapping inside the geometric object (i.e. not in the `ggplot()`). You can change the color with the `fill` argument. Here, you choose the `coral` color.

you can use this code:

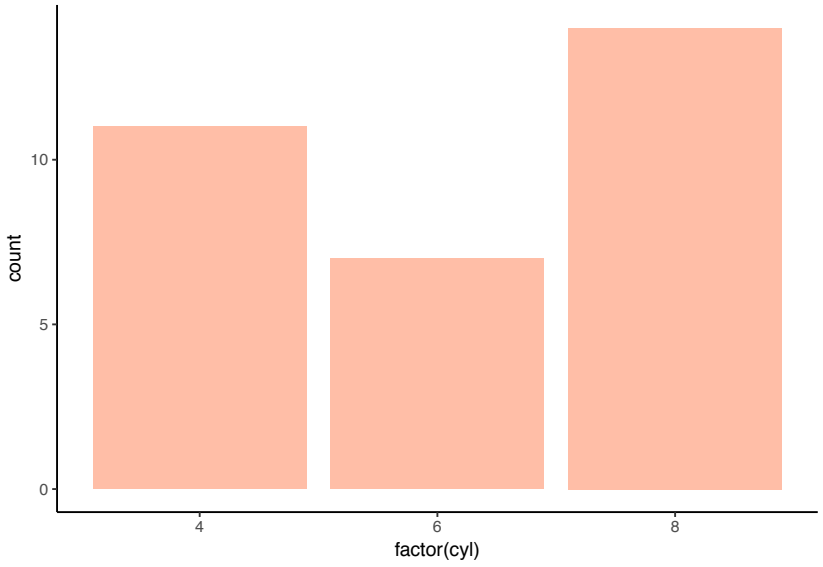
```
grDevices::colors()
```

to see all the colors available in R. There are around 650 colors.

Change the intensity

You can increase or decrease the intensity of the bars' color

```
# Change intensity
ggplot(mtcars, aes(factor(cyl)))+
  geom_bar(fill = "coral",
           alpha = 0.5) +
  theme_classic()
```



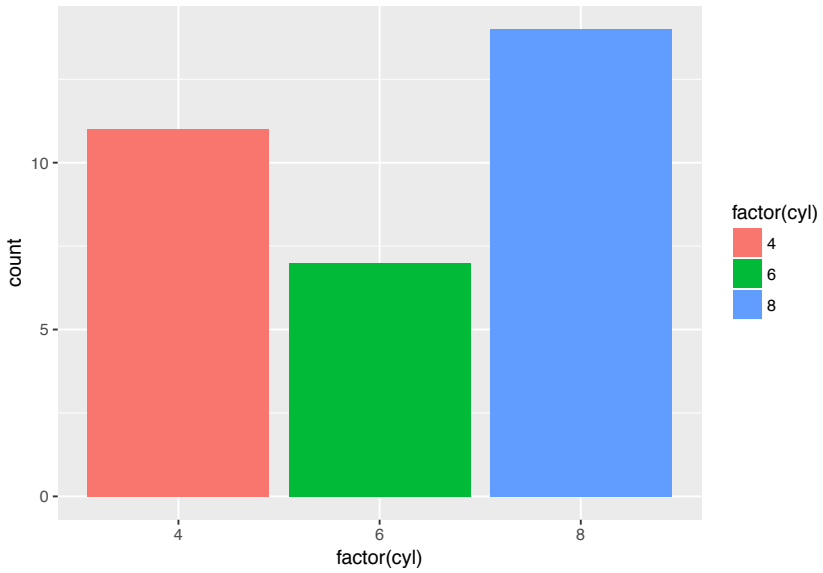
Code Explanation

- To increase/decrease the intensity of the bar, you can change the value of the `alpha`. A large `alpha` increases the intensity and low `alpha` reduces the intensity. `alpha` ranges from 0 to 1. If 1, then the color is the same as the palette. If 0, color is white. You choose `alpha = 0.1`.

Color by groups

You can change the colors of the bars, meaning one different color for each group. For instance, `cyl` variable has three levels, then you can plot the bar chart with three colors.

```
# Color by group
ggplot(mtcars, aes(factor(cyl), fill = factor(cyl)))+
  geom_bar()
```



Code Explanation

- The argument `fill` inside the `aes()` allows changing the color of the bar. You change the color by setting `fill = x-axis variable`. In your example, the x-axis variable is `cyl`; `fill = factor(cyl)`

Add a group in the bars

You can further split the y-axis based on another factor level. For instance, you can count the number of automatic and manual transmission based on the cylinder type.

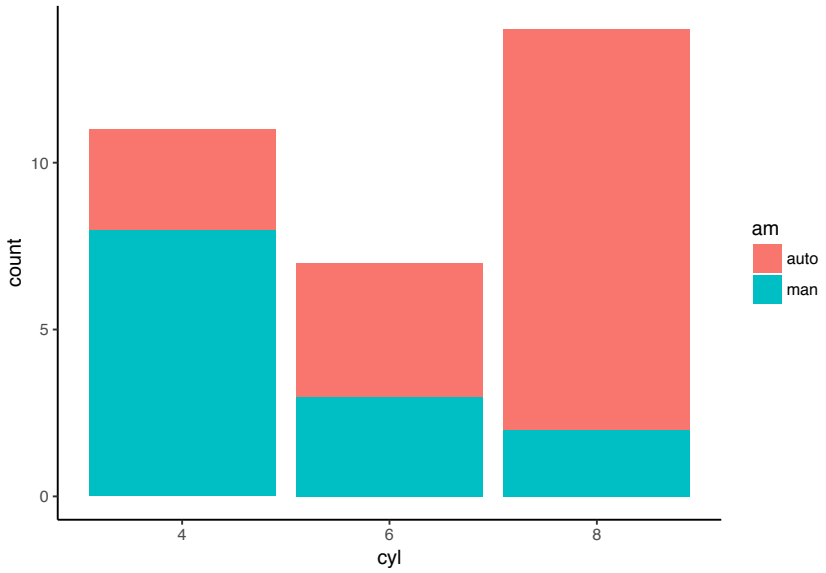
You will proceed as follow:

- Step 1: Create the data frame with `mtcars` dataset
- Step 2: Label the `am` variable with `auto` for automatic transmission and `man` for manual transmission. Convert `am` and `cyl` as a factor so that you don't need to use `factor()` in the `ggplot()` function.
- Step 3: Plot the bar chart to count the number of transmission by cylinder

```
library(dplyr)
# Step 1
data <- mtcars %>%
# Step 2
        mutate(am = factor(am, labels= c("auto", "man")),
                               cyl = factor(cyl))
```

You have the dataset ready, you can plot the graph;

```
# Step 3
ggplot(data, aes(x= cyl, fill = am))+
  geom_bar()+
  theme_classic()
```



Code Explanation

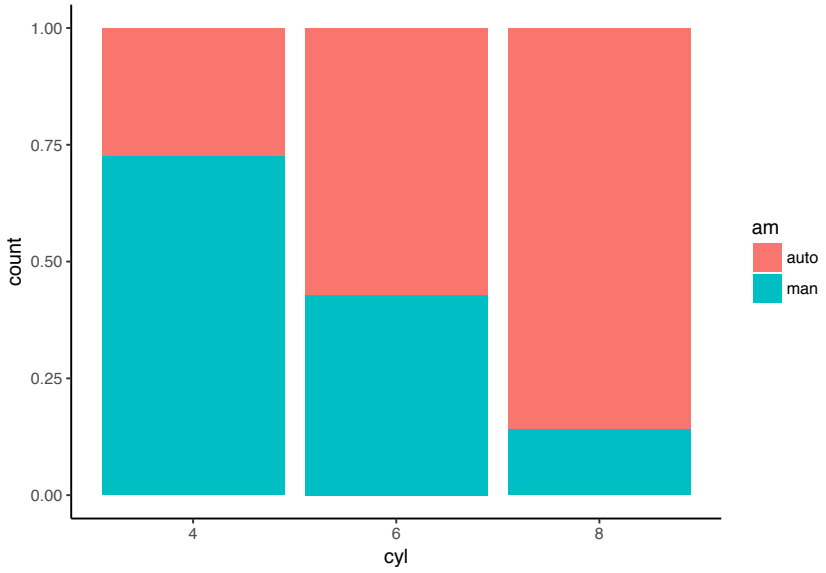
- The `ggplot()` contains the dataset `data` and the `aes()`.
- In the `aes()` you include the variable x-axis and which variable is required to fill the bar (i.e. `am`)
- `geom_bar()`: Create the bar chart

The mapping will fill the bar with two colors, one for each level. It is effortless to change the group by choosing other factor variables in the dataset.

Bar chart in percentage

You can visualize the bar in percentage instead of the raw count.

```
# Bar chart in percentage
ggplot(data, aes(x= cyl,fill = am))+
  geom_bar(position = "fill")+
  theme_classic()
```



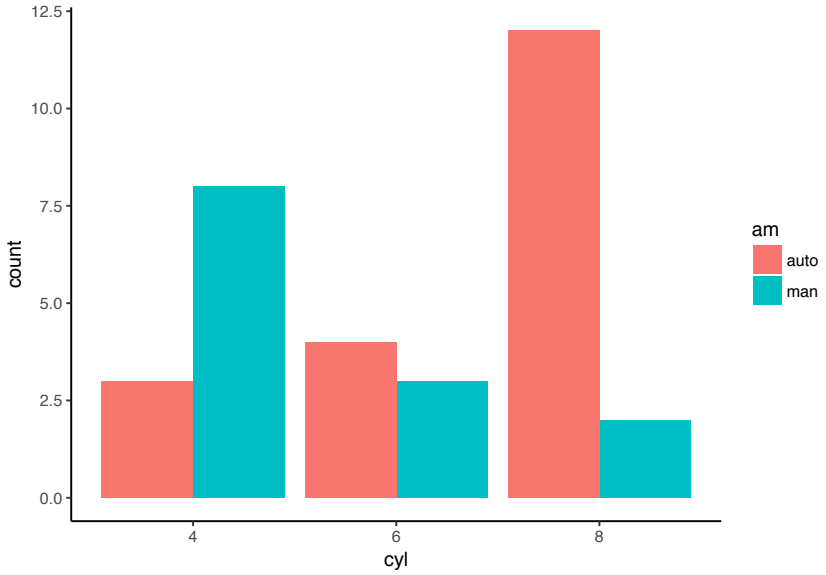
Code Explanation

- Use `position = "fill"` in the `geom_bar()` argument to create a graphic with percentage in the y-axis.

Side by side bars

It is easy to plot the bar chart with the group variable side by side..

```
# Bar chart side by side
ggplot(data, aes(x= cyl, fill = am))+
  geom_bar(position=position_dodge())+
  theme_classic()
```



Code Explanation

- `position=position_dodge()`: Explicitly tells how to arrange the bars

4.3.5 Histogram

In the second part of the bar chart tutorial, you can represent the group of variables with values in the y-axis.

Your objective is to create a graph with the average mile per gallon for each type of cylinder. To draw an informative graph, you will follow these steps:

- Step 1: Create a new variable with the average mile per gallon by cylinder
- Step 2: Create a basic histogram
- Step 3: Change the orientation
- Step 4: Change the color
- Step 5: Change the size
- Step 6: Add labels to the graph

Step 1: Create a new variable

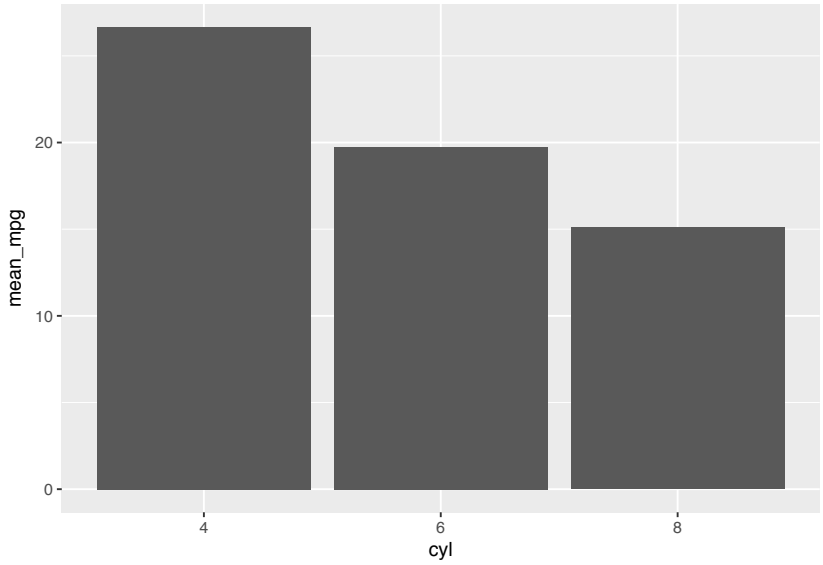
You create a data frame named `data_histogram` which simply returns the average miles per gallon by the number of cylinders in the car. You call this new variable `mean_mpg`, and you round the mean with two decimals.

```
# Step 1
data_histogram <- mtcars %>%
        mutate(cyl = factor(cyl)) %>%
        group_by(cyl)%>%
        summarise(mean_mpg = round(mean(mpg), 2))
```

Step 2: Create a basic histogram

You can plot the histogram. It is not ready to communicate to be delivered to client but gives us an intuition about the trend.

```
ggplot(data_histogram, aes(x= cyl, y = mean_mpg))+
  geom_bar(stat="identity")
```



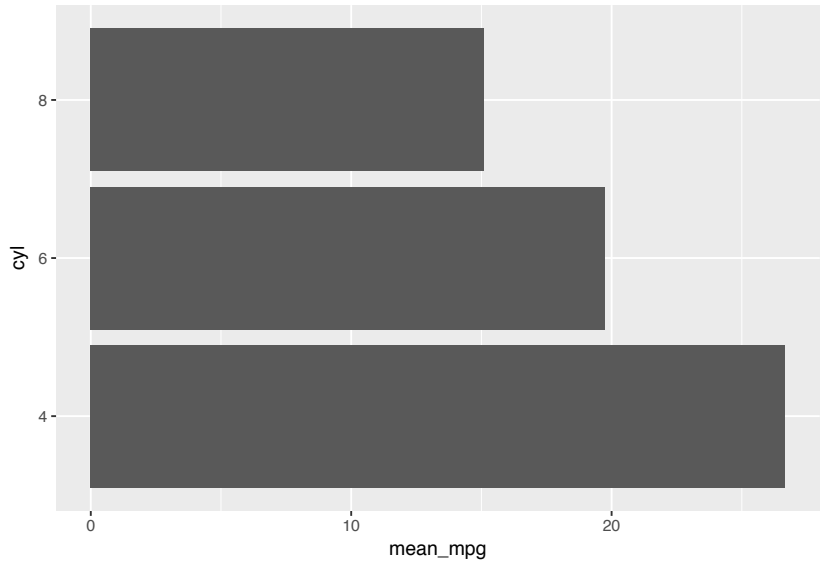
Code Explanation

- The `aes()` has now two variables. The `cyl` variable refers to the x-axis, and the `mean_mpg` is the y-axis.
- You need to pass the argument `stat="identity"` to refer the variable in the y-axis as a numerical value. `geom_bar` uses `stat="bin"` as default value.

Step 3: Change the orientation

You change the orientation of the graph from vertical to horizontal.

```
ggplot(data_histogram, aes(x= cyl, y = mean_mpg))+
  geom_bar(stat="identity") +
  coord_flip()
```



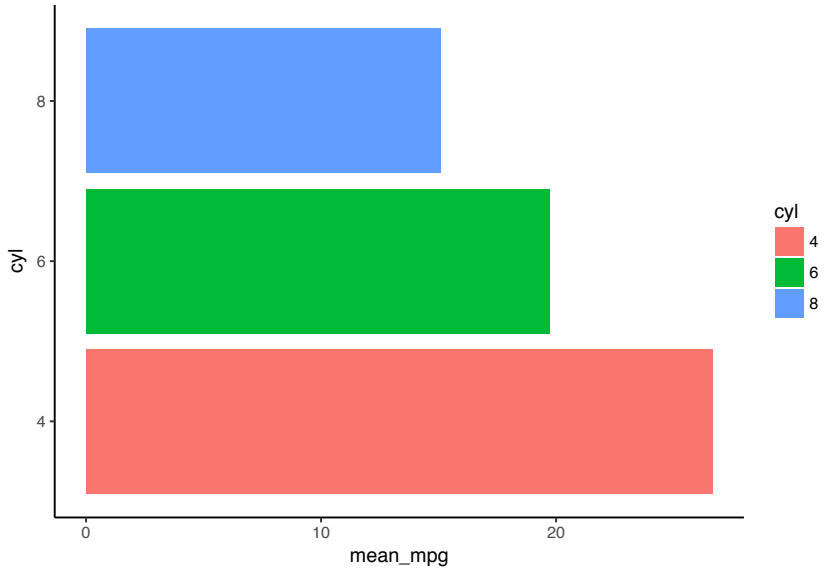
Code Explanation

- You can control the orientation of the graph with `coord_flip()`.

Step 4: Change the color

You can differentiate the colors of the bars according to the factor level of the x-axis variable.

```
ggplot(data_histogram, aes(x= cyl, y = mean_mpg, fill = cyl))+
         geom_bar(stat="identity")+
         coord_flip()+
  theme_classic()
```



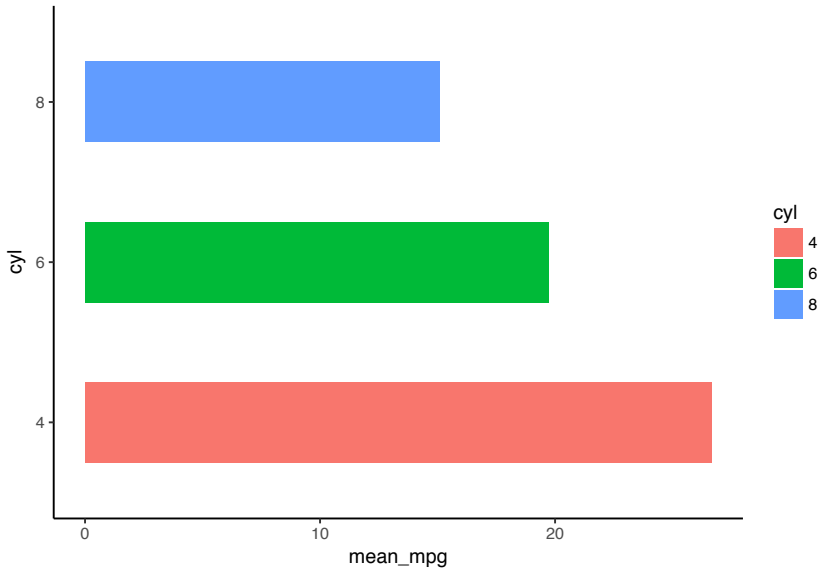
Code Explanation

- You can plot the graph by groups with the `fill= cyl` mapping. R takes care automatically of the colors based on the levels of `cyl` variable.

Step 5: Change the size

To make the graph looks prettier, you reduce the width of the bar.

```
graph <- ggplot(data_histogram, aes(x= cyl, y = mean_mpg, fill = cyl)) +
  geom_bar(stat="identity",
           width=0.5) +
  coord_flip()+
  theme_classic()

graph
```



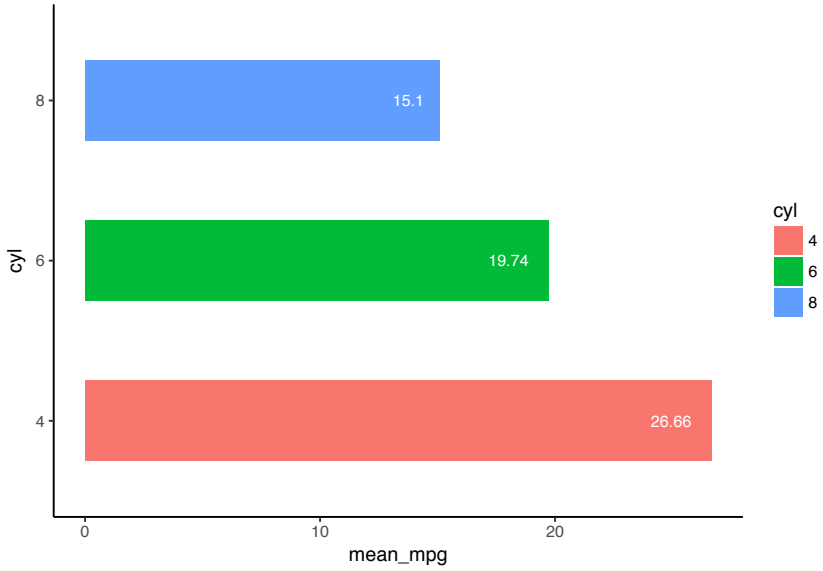
Code Explanation

- The `width` argument inside the `geom_bar()` controls the size of the bar. Larger value increases the width.
- Note, you store the graph in the variable `graph`. you do so because the next step will not change the code of the variable `graph`. It improves the readability of the code.

Step 6: Add labels to the graph

The last step consists to add the value of the variable `mean_mpg` in the label.

```
graph +
  geom_text(aes(label=mean_mpg),
            hjust=1.5,
            color="white",
            size=3)+
  theme_classic()
```



Code Explanation

- The function `geom_text()` is useful to control the aesthetic of the text.
  - `label=`: Add a label inside the bars
  - `mean_mpg`: Use the variable `mean_mpg` for the label
- `hjust` controls the location of the label. Values closed to 1 displays the label at the top of the bar, and higher values bring the label to the bottom. If the orientation of the graph is vertical, change `hjust` to `vjust`.
- `color="white"`: Change the color of the text. Here you use the white color.
- `size=3`: Set the size of the text.

4.3.6 Summary

A bar chart is useful when the x-axis is a categorical variable. The y-axis can be either a count or a summary statistic. The table below summarizes how to control bar chart with `ggplot2`:

Objective	code
Count	ggplot(df, eas(x= factor(x1)) + geom_bar()
Count with different color of fill	ggplot(df, eas(x= factor(x1), fill = factor(x1))) + geom_bar()
Count with groups, stacked	ggplot(df, eas(x= factor(x1), fill = factor(x2))) + geom_bar(position=position_dodge())
Count with groups, side by side	ggplot(df, eas(x= factor(x1), fill = factor(x2))) + geom_bar()
Count with groups, stacked in %	ggplot(df, eas(x= factor(x1), fill = factor(x2))) + geom_bar(position=position_dodge())
Values	ggplot(df, eas(x= factor(x1)+ y = x2) + geom_bar(stat="identity")

4.3.7 Box plot

You can use the geometric object `geom_boxplot()` from `ggplot2` library to draw a box plot. Box plot helps to **visualise the distribution of the data by quartile and detect the presence of outliers.**

You will use the `airquality` dataset to introduce box plot with `ggplot`. This dataset measures the airquality of New York from May to September 1973. The dataset contains 154 observations. You will use the following variables:

- `Ozone`: Numerical variable
- `Wind`: Numerical variable
- `Month`: May to September. Numerical variable

Before you start to create your first box plot, you need to manipulate the data as follow:

- Step 1: Import the data
- Step 2: Drop unnecessary variables
- Step 3: Convert `Month` in factor level
- Step 4: Create a new categorical variable dividing the month with three level: `begin`, `middle` and `end`.
- Step 5: Remove missing observations

All these steps are done with `dplyr` and the pipeline operator `%>%`.

```
library(dplyr)
library(ggplot2)
# Step 1
data_air <- airquality %>%
# Step 2
```

```
            select(-c(Solar.R, Temp)) %>%
# Step 3
            mutate(Month = factor(Month, order = TRUE, labels = c("May", "June", "July", "August", "Sep
#Step 4
                   day_cat = factor(ifelse(Day < 10, "Begin",
                                    ifelse(Day <20, "Middle", "End"))))
```

A good practice is to check the structure of the data with the function `glimpse()`.

```
glimpse(data_air)
```

```
## Observations: 153
## Variables: 5
## $ Ozone   <int> 41, 36, 12, 18, NA, 28, 23, 19, 8, NA, 7, 16, 11, 14, ...
## $ Wind    <dbl> 7.4, 8.0, 12.6, 11.5, 14.3, 14.9, 8.6, 13.8, 20.1, 8.6...
## $ Month   <ord> May, May, May, May, May, May, May, May, May, May, May,...
## $ Day     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,...
## $ day_cat <fctr> Begin, Begin, Begin, Begin, Begin, Begin, Begin, Begi...
```

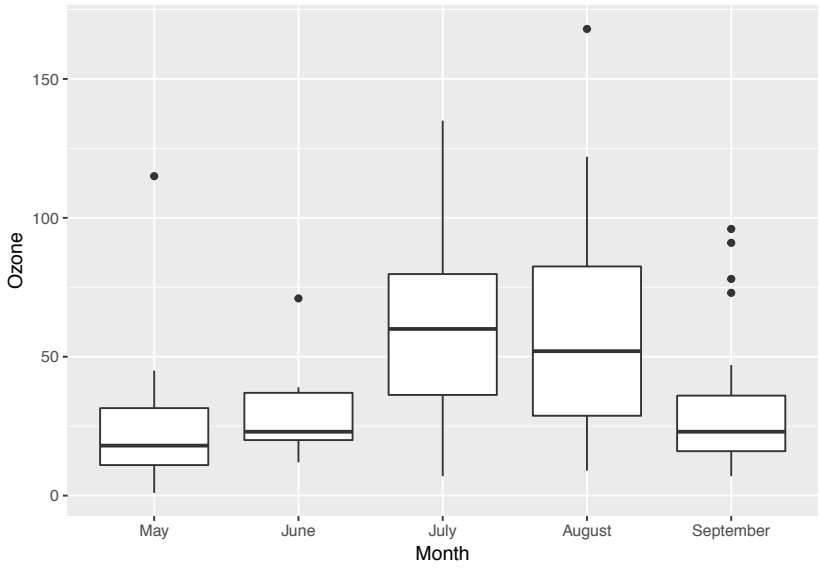
There are `NA`'s in the dataset. Removing them is wise.

```
# Step 5
data_air_nona <- data_air %>% na.omit()
```

4.3.8 Basic box plot

Let's plot the basic box plot with the distribution of ozone by month.

```
# Store the graph
box_plot <- ggplot(data_air_nona, aes(x=Month, y=Ozone))
# Add the geometric object box plot
box_plot +
  geom_boxplot()
```



Code Explanation

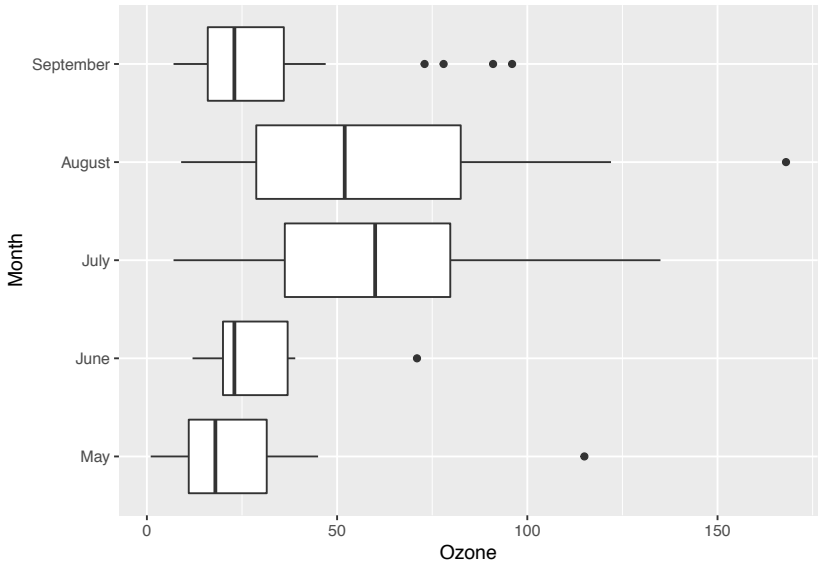
- First step: Store the graph for further use

  - `box_plot`: You store your graph into the variable `box_plot` It is helpful for further use or avoid too complex line of codes
- Second step: Add your geometric object box plot
  - You pass the dataset `data_air_nona` to ggplot.
  - Inside the `aes()` argument, you add the x-axis and y-axis.
  - The `+` sign means you want R to keep reading the code. It makes the code more readable by breaking it.
  - Use `geom_boxplot()` to create a box plot.

Change side of the graph

You can flip the side of the graph.

```
box_plot +
  geom_boxplot()+
  coord_flip()
```



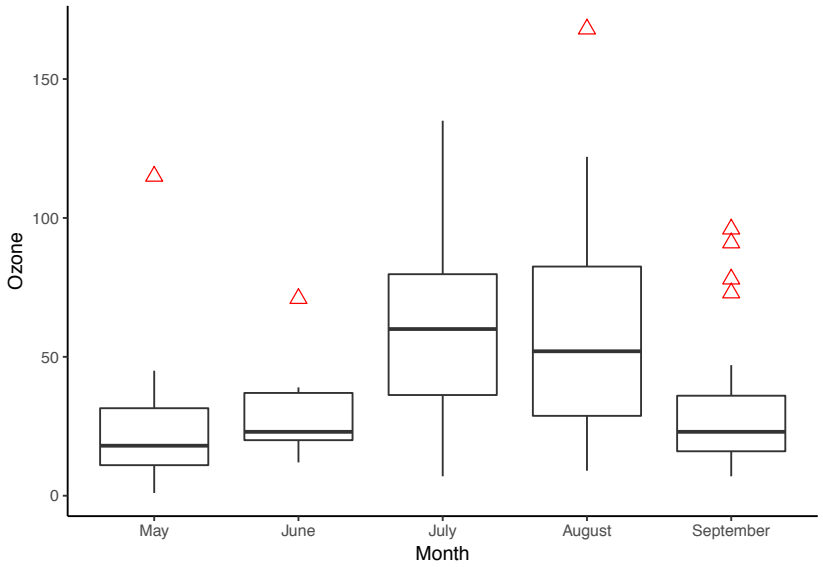
Code Explanation

- `box_plot`: You use the graph you stored. It avoids rewriting all the codes each time you add new information to the graph
- `geom_boxplot()`: Create the box plot
- coord_flip()‘: Flip the side of the graph

Change color of outlier

You can change the color, shape and size of the outliers.

```
box_plot +
  geom_boxplot(outlier.colour="red",
               outlier.shape=2,
               outlier.size=3) +
  theme_classic()
```



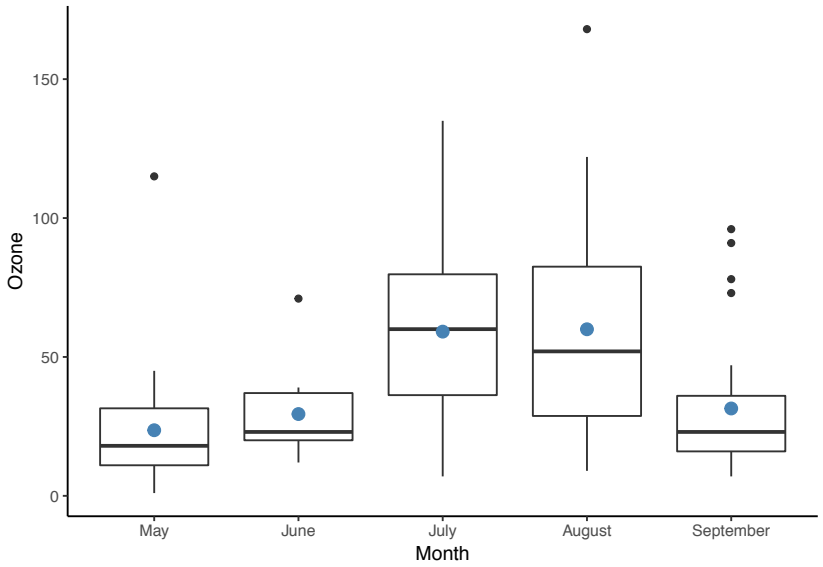
Code Explanation

- `outlier.colour="red"`: Control the color of the outliers
- `outlier.shape=2`: Change the shape of the outlier. 2 refers to triangle
- `outlier.size=3`: Change the size of the triangle. The size is proportional to the number.

Add a summary statistic

You can add a summary statistic to the box plot.

```
box_plot+
  geom_boxplot() +
   stat_summary(fun.y=mean,
                geom = "point",
                size=3,
                color ="steelblue") +
  theme_classic()
```



Code Explanation

- `stat_summary()` allows adding a summary to the box plot

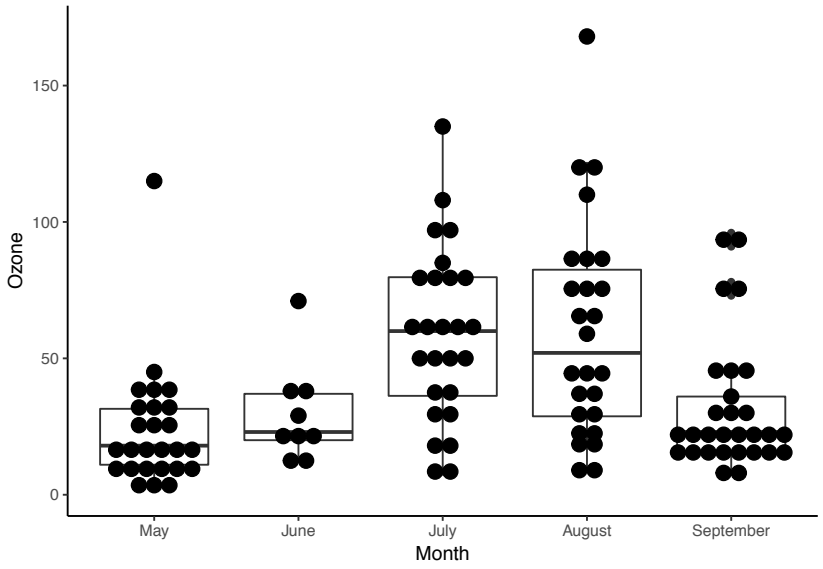
- The argument `fun.y` controls the statistics returned. You will use `mean`.

- Note: Other statistics are available such as `min` and `max`.More than one statistics can be exhibited in the same graph.
- `geom = "point"`: Plot the average with a point
- `size=3`: Size of the point
- `color ="steelblue"`: Color of the points

4.3.9 box plot with dots

In the next plot, you add the dot plot layers. Each dot represents an observation.

```
box_plot+
  geom_boxplot()  +
  geom_dotplot(binaxis='y',
               dotsize=1,
               stackdir='center')+
  theme_classic()
```



Code Explanation

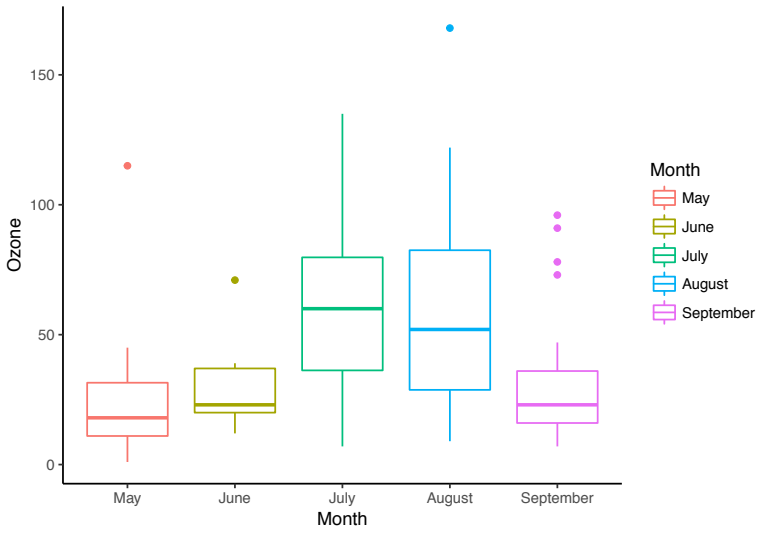
- `geom_dotplot()` allows adding dot to the bin width
- `binaxis='y'`: Change the position of the dots along the y-axis. By default, x-axis
- `dotsize=1`: Size of the dots
- `stackdir='center'`: Way to stack the dots: Four values:
  - "up" (default),
  - "down"
  - "center"
  - "centerwhole"

4.3.10 Control aesthetic of the box plot

Change the color of the box

You can change the colors of the group.

```
ggplot(data_air_nona, aes(x = Month, y = Ozone, color= Month)) +
  geom_boxplot()  +
  theme_classic()
```



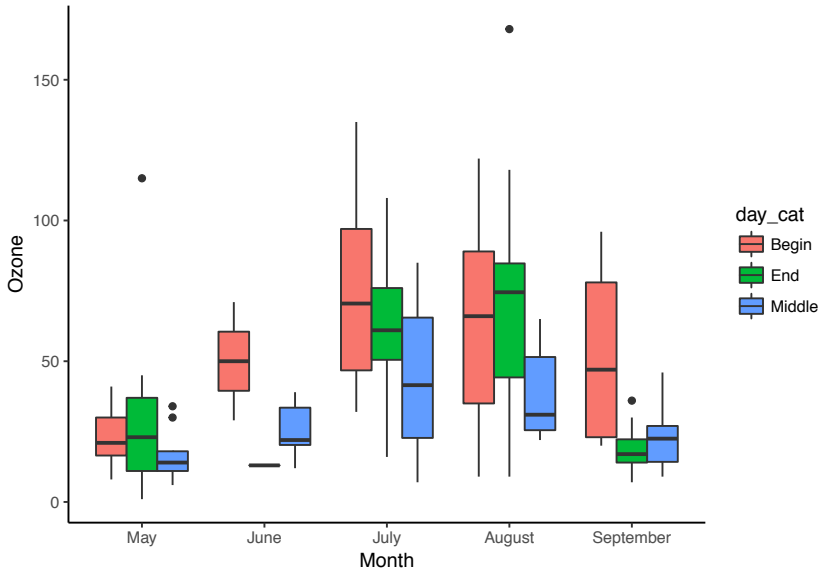
Code Explanation

- The colors of the groups are controlled in the `aes()` mapping. You can use `color= Month` to change the color of the box according to the months

Box plot with multiple groups

It is also possible to add multiple groups. You can visualize the difference in the air quality according to the day of the measure.

```
ggplot(data_air_nona, aes(Month, Ozone)) +
  geom_boxplot(aes(fill= day_cat))+
  theme_classic()
```



Code Explanation

- The `aes()` mapping of the geometric object controls the groups to display (this variable has to be a

factor).

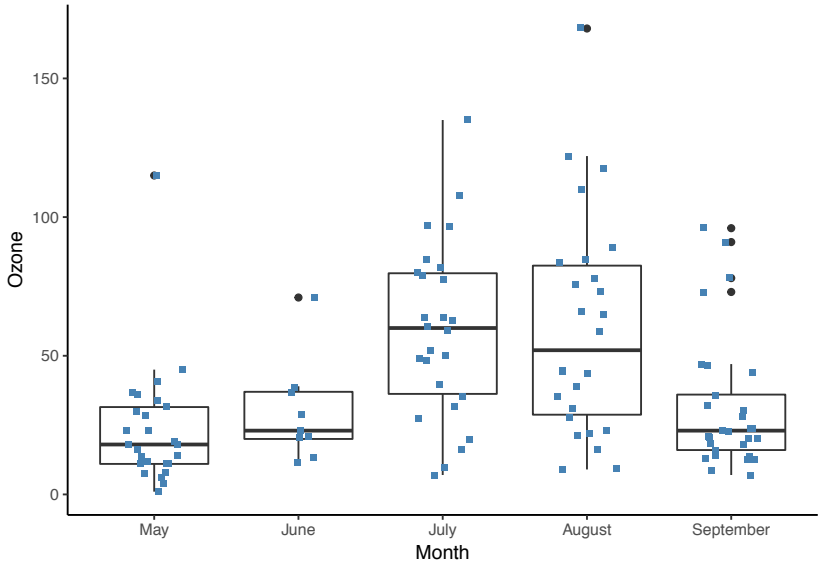
- `aes(fill= day_cat)` allows creating three boxes for each month in the x-axis

box plot with jittered dots

Another way to show the dot is with jittered points. This is a convenient way to visualize points with a categorical variable.

This method avoids the overlapping of the discrete data.

```
box_plot +
  geom_boxplot()  +
  geom_jitter(shape=15,
              color = "steelblue",
              position=position_jitter(width = 0.21))+
  theme_classic()
```

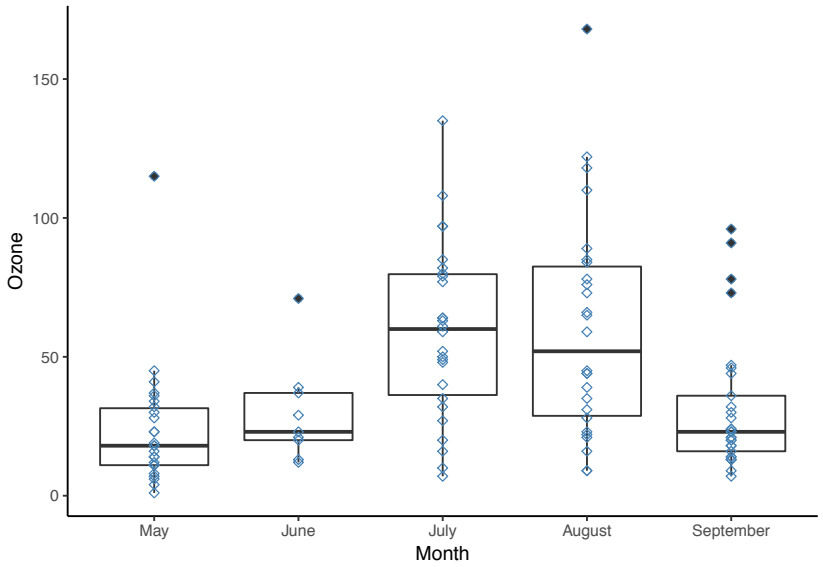


Code Explanation

- `geom_jitter()` adds a little decay to each point.
- `shape=15` changes the shape of the points. 15 represents the squares
- `color = "steelblue"`: Change the color of the point
- `position=position_jitter(width = 0.21)`: Way to place the overlapping points. `position_jitter(width = 0.21)` means you move the points by 20 percent from the x-axis. By default, 40 percent.

You can see the difference between the first graph with the jitter method and the second with the point method.

```
box_plot +
  geom_boxplot()  +
  geom_point(shape=5,
              color = "steelblue")+
  theme_classic()
```



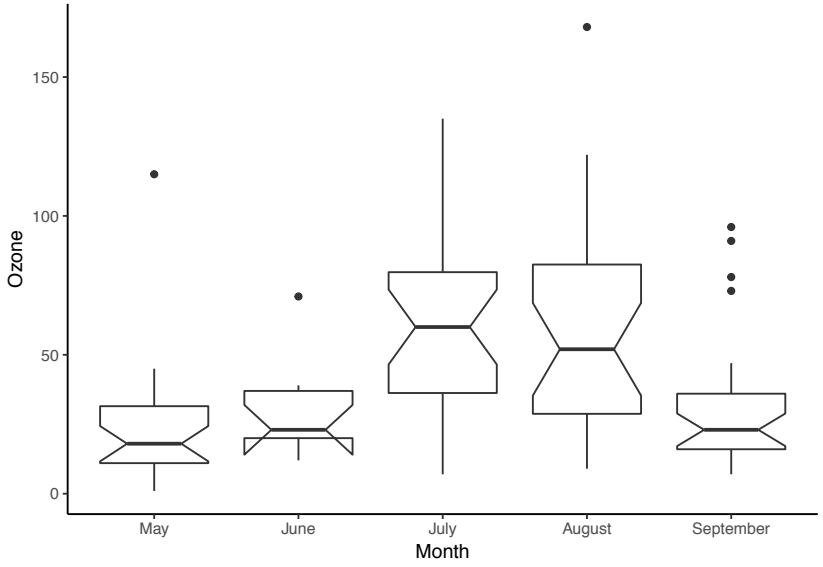
Notched box plot

You can use an interesting feature of `geom_boxplot()`, a notched box plot. The notch plot narrows the box around the median. The main purpose of a notched box plot is to compare the significance of the median between groups. There are strong evidence two groups have different medians when the notches do not overlap. A notch is computed as follow:

$$median \pm 1.57 * \frac{IQR}{\sqrt{n}}$$

with IQR is the interquartile and n number of observations.

```
box_plot +
  geom_boxplot(notch=TRUE) +
  theme_classic()
```



Code Explanation

- `geom_boxplot(notch=TRUE)`: Create a notched box plot

4.3.11 Summary

You can summarize the different types of box plot in the table below:

Objective	Code
Basic box plot	ggplot(df, aes(x = x1, y =y)) + geom_boxplot()
flip the side	ggplot(df, aes(x = x1, y =y)) + geom_boxplot() + coord_flip()
Notched box plot	ggplot(df, aes(x = x1, y =y)) + geom_boxplot(notch=TRUE)
Box plot with jittered dots	ggplot(df, aes(x = x1, y =y)) + geom_boxplot() + geom_jitter(position = position_jitter(0.21))

4.4 Export data

In this chapter, we will learn how to export data from R environment to different formats.

To export data to the hard drive, you need a path and an extension. First of all, the path is the location of the data will be stored. In this tutorial, you will see how to store data on:

- The hard drive
- Google Drive
- Dropbox

Secondly, R allows the users to export the data into different types of files. We cover the essential file's extension:

- csv
- xlsx
- RDS
- SAS
- SPSS
- STATA

Overall, it is not difficult to export data from R.

4.4.1 Hard drive

To begin with, you can save the data directly into the working directory. The following code prints the path of your working directory:

```
directory <- getwd()
directory
```

```
## [1] "/Users/Thomas/Dropbox/Learning/book_R"
```

By default, file will be saved in this path. You can, of course, set a different path. For instance, you can change the path to the download folder. We will see in the function description how to change the path.

For Mac OS:

```
/Users/USERNAME/Downloads/
```

For Windows:

```
C:\Users\USERNAME\Downloads\
```

First of all, let's import the `mtcars` dataset and get the mean of `mpg` and `disp` grouped by `gear`.

```
library(dplyr)
df <- mtcars %>%
  select(mpg, disp, gear) %>%
  group_by(gear) %>%
  summarise(mean_mpg = mean(mpg), mean_disp = mean(disp))
df
```

```
## # A tibble: 3 x 3
##    gear mean_mpg mean_disp
##   <dbl>    <dbl>     <dbl>
## 1     3 16.10667  326.3000
## 2     4 24.53333  123.0167
## 3     5 21.38000  202.4800
```

The table contains three rows and three columns. You can create a CSV file with the function `write.csv()`.

4.4.2 Export CSV

The basic syntax is:

```
write.csv(df, path, sep = "\t")
arguments

- df: Dataset to save. Need to be the same name of the data frame in the environment.
- path: A string. Set the destination path. Path+ filename + extention i.e. "/Users/USERNAME/Downloads/
- Note: A decimal is used to the separator.
```

Example:

```
r   write.csv(df, "table_car.csv")
```

Code Explanation

- `write.csv(df, "table_car.csv")`: Create a CSV file in the hard drive:
- `df`: name of the data frame in the environment
- `"table_car.csv"`: Name the file table_car and store it as csv

note: You can use the function `write.csv2()` to separate the rows with a semicolon.

```
write.csv2(df, "table_car.csv")
```

note: For pedagogical purpose only, we created a function called `open_folder()` to open the directory folder for you. You just need to run the code below and see where the csv file is stored. You should see a file names `table_car.csv`.

```
# Run this code to create the function
open_folder <- function(dir){
  if (.Platform['OS.type'] == "windows"){
    shell.exec(dir)
  } else {
    system(paste(Sys.getenv("R_BROWSER"), dir))
  }
}

# Call the function to open the folder
open_folder(directory)
```

4.4.3 Export excel file

Export data to Excel is trivial for Windows users and trickier for Mac OS user. Both users will use the library `xlsx` to create an Excel file. The slight difference comes from the installation of the library. Indeed, the library `xlsx` uses Java to create the file, Java is not installed by default on the mac OS machine.

Windows users

If you are a Windows users, you can install the library directly with conda:

```
conda install -c r r-xlsx
```

Once the library installed, you can use the function `write.xlsx()`. A new Excel workbook is created in the working directory

```
library(xlsx)
write.xlsx(df, "table_car.xlsx")
```

Mac users

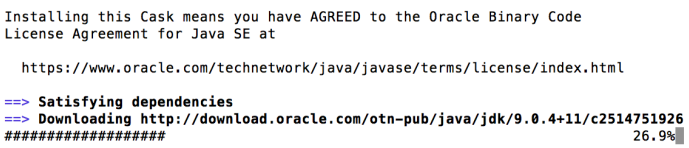
If you are a Mac OS user, you need to follow these steps:

- Step 1: Install latest version of Java
- Step 2: Install library `rJava`
- Step 3: Install library `xlsx`

Step 1

The easiest way to install Java might be with **Homebrew**. If you have Homebrew already installed on your machine, you can copy and paste the following code to the terminal:

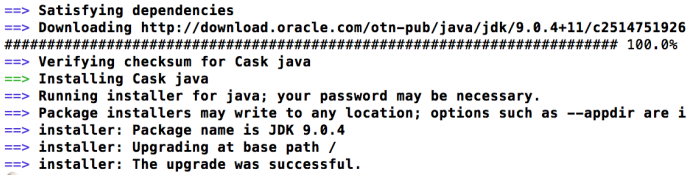
```
brew cask install java
```



If you don't have Homebrew already on your machine, you need to install it. The next two lines of code install Homebrew and Java

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew cask install java
```

You have now the latest version of Java on your machine.



You can go back to Rstudio and check which version of Java is installed.

```
system("java -version")
```

At the time of the tutorial, the latest version of Java is 9.0.4.

Step 2

You need to install `rjava` in R. We recommended you to install R and Rstudio with Anaconda. Anaconda managed the dependencies between libraries. In this sense, Anaconda will do all the job to install correctly and efficiently `rjava`.

First of all, you need to update conda

```
conda - conda update
```

and then install `rjava`

- `rjava`: Add Java to R.
- Anaconda: `conda install -c r r-rjava`

You should be able to open `rjava` in Rstudio

```
library(rJava)
```

```
## Loading required package: methods
```

Step 3

Finally, it is time to install `xlsx`.

- `xlsx`: Export file to excel.
- Anaconda: `conda install -c r r-xlsx`

Just as the windows users, you can save data with the function `write.xlsx()`

```
library(xlsx)
```

```
## Loading required package: xlsxjars
```

```
write.xlsx(df, "table_car.xlsx")
```

4.4.4 Export to different software

Exporting data to different software is as simple as importing them. The library `haven` provides a convenient way to export data to

- spss
- sas
- stata

First of all, import the library. If you don't have `haven`, you can go here to install it.

```
library(haven)
```

SPSS file

Below is the code to export the data to SPSS software:

```
r   write_sav(df, "table_car.sav")
```

Export SAS file

Just as simple as spss, you can export to sas

```
write_sas(df, "table_car.sas7bdat")
```

Export STATA file

Finally, `haven` library allows to write `.dta` file.

```
write_dta(df, "table_car.dta")
```

R

If you want to save a data frame or any other R object, you can use the `save()` function.

```
save(df, file = 'table_car.RData')
```

4.4.5 Interact with the cloud services

Last but not least, R is equipped with amazing libraries to interact with the cloud computing. The last part of this tutorial deals with export/import files from:

- Google Drive
- Dropbox

note: This part of the tutorial assumes you have an account with Google and Dropbox. If not, you can easily create one for

- Gmail:
- Dropbox:

4.4.6 Google Drive

You need to install the library `googledrive` to access the function allowing to interact with Google Drive.

The library is not yet available at Anaconda. You want to use Anaconda to manage all your libraries. To install libraries out of the conda libraries, you need to check the library path (i.e. where R goes to find libraries):

```
lib <- .libPaths()
lib
```

```
## [1] "/Users/Thomas/anaconda3/lib/R/library"
```

You should see `anaconda3/lib/R/library`. That is where you want to install your library. Copy this path.

For non-conda user, installing a library is easy, you can use the function `install.packages('NAME OF PACKAGE)` with the name of the package inside the parenthesis. Don't forget the `' '`. Note that, R is supposed to install the package in the 'libPaths() automatically. It is worth to see it in action.

```
install.packages("googledrive", lib)
```

and you open the library.

```
library(googledrive)
```

Upload to Google Drive

To upload a file to Google drive, you need to use the function `drive_upload()`.

Each time you restart Rstudio, you will be prompted to allow access `tidyverse` to Google Drive.

The basic syntax of `drive_upload()` is

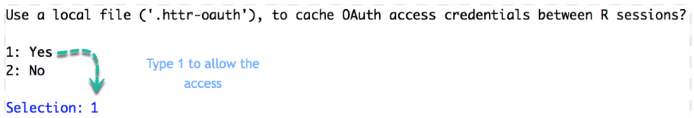
```
drive_upload(file, path = NULL, name = NULL)
arguments:

  - file: Full name of the file to upload (i.e. including the extension)
- path: Location of the file
- name: You can rename it as you wish. By default, it is the local name.
```

After you launch the code, you need to confirm several questions

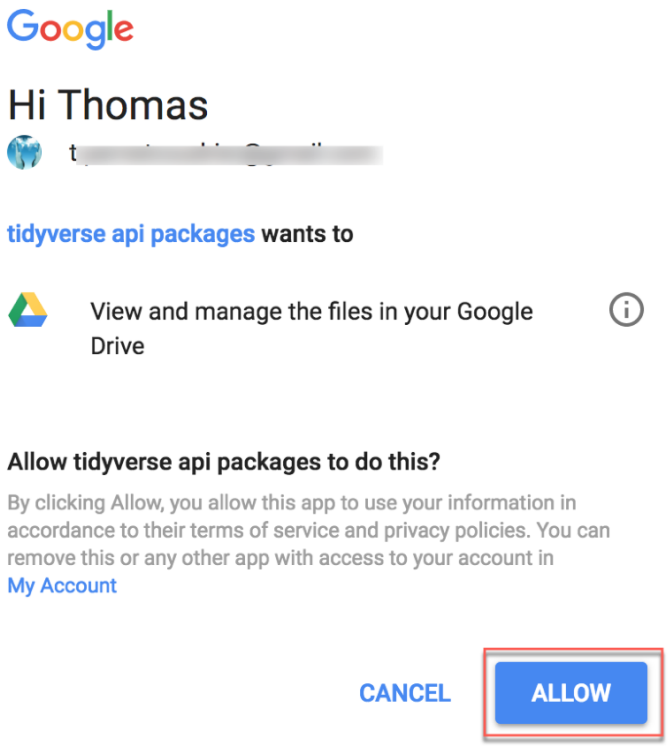
```
drive_upload("table_car.csv", name = "table_car")
```

```
## Local file:
##   * table_car.csv
## uploaded into Drive file:
##   * table_car: 1Yxn6SB3Kb27B89McNhp3Uz4jYdzbLBPK
## with MIME type:
##   * text/csv
```

You type 1 in the console to confirm the access



Then, you are redirected to Google API to allow the access. Click Allow.



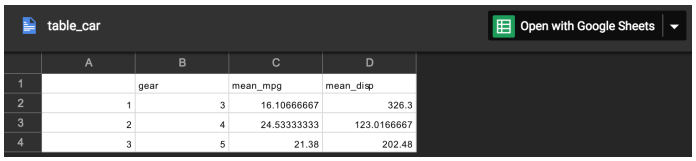
When the authentication is completed, you can quit your browser.

In the Rstudio's console, you can see the summary of the step done. Google successfully uploaded the file located locally on the Drive. Google assigned an ID to each file in the drive.

You can see this file in Google Spreadsheet.

```
drive_browse("table_car")
```

You will be redirected to Google Spreadsheet



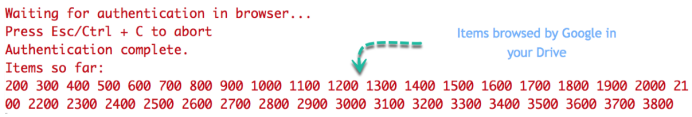
Import from Google Drive

Upload a file from Google Drive with the ID is convenient. If you know the file name, you can get its ID as follow:

note: Depending on your internet connection and the size of your Drive, it takes times.

```
x <- drive_get("table_car")
as_id(x)
```

```
## [1] "1Yxn6SB3Kb27B89McNhp3Uz4jYdzbLBPK"
## attr(,"class")
## [1] "drive_id"
```



You stored the ID in the variable `x`. The function `drive_download()` allows downloading a file from Google Drive.

The basic syntax is:

```
drive_download(file, path = NULL, overwrite = FALSE)
arguments:

- file:  Name or id of the file to download
- path: Location to download the file. By default, it is downloaded to the working directory, and the na
- overwrite = FALSE: If the file already exists, don't overwrite it. If set to TRUE, the old file is era
```

You can finally download the file:

```
download_google <- drive_download(as_id(x), overwrite = TRUE)
```

```
## File downloaded:
##   * table_car
## Saved locally as:
##   * table_car
```

Code Explanation

- `drive_download()`: Function to download a file from Google Drive
- `as_id(x)`: Use the ID to browse the file in Google Drive
- `overwrite = TRUE`: If file exists, overwrite it, else execution halted To see the name of the file locally, you can use:

```
r   google_file <- download_google$local_path   google_file
```

```
## [1] "table_car"
```

It is trivial to open the file. The file is stored in your working directory. Remember, you need to add the extenstion of the file to open it in R. You can create the full name with the function `paste()` (i.e. `table_car.csv`)

```
path <- paste(google_file, ".csv", sep ="")
google_table_car <- read.csv(path)
google_table_car
```

```
##   X gear mean_mpg mean_disp
## 1 1    3 16.10667  326.3000
## 2 2    4 24.53333  123.0167
## 3 3    5 21.38000  202.4800
```

Finally, you can remove the file from your Google drive.

```
## clean-up
drive_find("table_car") %>% drive_rm()
```

4.4.7 Export to Dropbox

R interacts with Dropbox via the `rdrop2` library. The library is not available at Anaconda as well. You can install it via the console

```
install.packages('rdrop2')
```

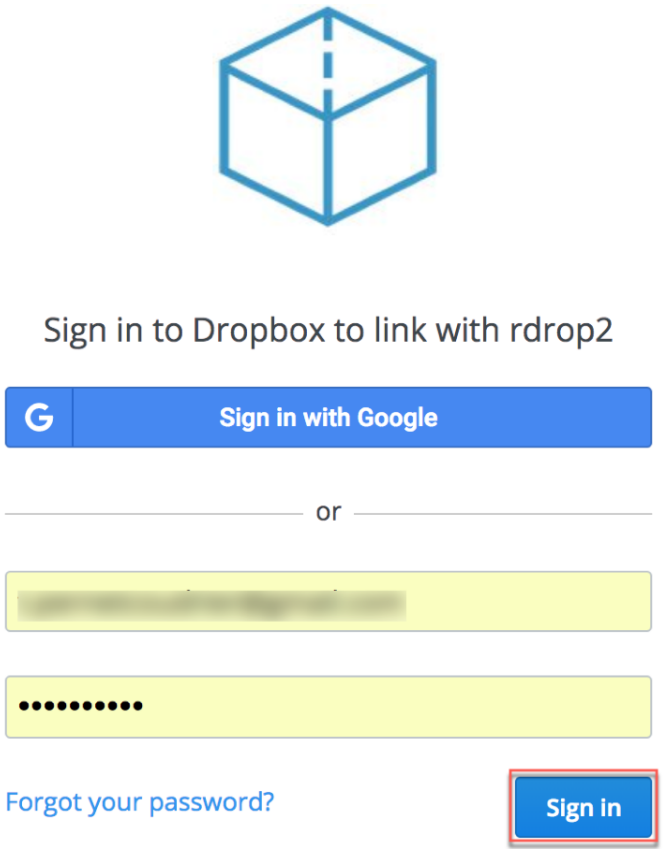
```
library(rdrop2)
```

You need to provide a temporary access to Dropbox with your credential. After the identification is done, R can create, remove upload and download to your Dropbox.

First of all, you need to give the access to your account. The credentials are cached during all session.

```
drop_auth()
```

You will be redirected to Dropbox to confirm the authentification. You need to sign in to link Dropbox with `rdrop2`



You can create a folder with the function `drop_create()`.

- `drop_create('my_first_drop')`: Create a folder in the first branch of Dropbox
- `drop_create('First_branch/my_first_drop')`: Create a folder inside the existing `First_branch` folder.

```
drop_create('my_first_drop')
```

To upload the .csv file into your Dropbox, use the function `drop_upload()`.

Basic syntax:

```
drop_upload(file, path = NULL, mode = "overwrite")
arguments:

  - file: local path
- path: Path on Dropbox
- mode = "overwrite":  By default, overwrite an existing file. If set to `add`, the upload is not compl
```

```
drop_upload('table_car.csv', path = "my_first_drop")
```

You can read the csv file from Dropbox with the function `drop_read_csv()`

```
dropbox_table_car <- drop_read_csv("my_first_drop/table_car.csv")
dropbox_table_car
```

```
##   X gear mean_mpg mean_disp
## 1 1    3 16.10667  326.3000
## 2 2    4 24.53333  123.0167
## 3 3    5 21.38000  202.4800
```

When you are done using the file and want to delete it. You need to write the path of the file in the function `drop_delete()`

```
drop_delete('my_first_drop/table_car.csv')
```

It is also possible to delete a folder

```
drop_delete('my_first_drop')
```

4.4.8 Summary

We can summarize all the functions in the table below

Library	Objective	Function
base	Export csv	write.csv()
xlsx	Export excel	write.xlsx()
haven	Export spss	write_sav()
haven	Export sas	write_sas()
haven	Export stata	write_dta()
base	Export R	save()
googledrive	Upload Google Drive	drive_upload()
googledrive	Open in Google Drive	drive_browse()
googledrive	Retrieve file ID	drive_get(as_id())
googledrive	Dowload from Google Drive	download_google()
googledrive	Remove file from Google Drive	drive_rm()
rdrop2	Authentification	drop_auth()
rdrop2	Create a folder	drop_create()
rdrop2	Upload to Dropbox	drop_upload()
rdrop2	Read csv from Dropbox	drop_read_csv
rdrop2	Delete file from Dropbox	drop_delete()

5 Data analysis

5.1 Statistical inference

Stastitical inference is the art of generating conclusions about the distribution of the data. A data scientist is often exposed to question that can only be answered scientifically. Therefore, statistical inference is a strategy to test whether a hypothesis is true, i.e. validated by the data.

A common strategy to assess hypothesis is to conduct a **t-test**. A **t-test** can tell whether two groups have the same mean. A *t-test* is also called a **Student Test**. A *t-test* can be estimated for:

1. A single vector (i.e. one-sample *t-test*)
2. Two vectors from the same sample group (i.e. paired *t-test*).

You assume that both vectors are randomly sampled, independent and come from a normally distributed population with unknown but equal variances.

In this tutorial, you will learn:

- One sample *t-test*
- Paired *t-test*

5.1.1 What is the *t-test*?

The basic idea behind a **t-test** is to use statistic to evaluate two contrary hypothesis:

- H0: NULL hypothesis: The average is the same as the sample used
- H1: True hypothesis: The average is different from the sample used

The **t-test** is commonly used with small sample sizes. To perform a **t-test**, you need to assume normality of the data.

The basic syntax for `t.test()` is:

```
t.test(x, y = NULL,
       mu = 0, var.equal = FALSE)
arguments:
- x : A vector to compute the one-sample *t-test*
- y: A second vector to compute the two sample *t-test*
- mu: Mean of the population
- var.equal: Specify if the variance of the two vectors are equal. By default, set to `FALSE`
```

5.1.2 One-sample *t-test*

The *t-test*, or student's test, compares the mean of a vector against a theoretical mean, μ. The formula used to compute the *t-test* is:

$$t = \frac{m - \mu}{\frac{s}{\sqrt{n}}}$$

Here

- m refers to the mean
- μ to the theoretical mean
- s is the standard deviation
- n the number of observations.

To evaluate the statistical significance of the *t-test*, you need to compute the **p-value**. The *p-value* ranges from 0 to 1, and is interpreted as follow:

- A *p-value* lower than 0.05 means you are strongly confident to reject the null hypothesis, thus H1 is accepted.
- A *p-value* higher than 0.05 indicates that you don't have enough evidences to reject the null hypothesis.

You can construst the *p-value* by looking at the corresponding absolute value of the *t-test* in the Student distribution with a degrees of freedom equals to $df = n - 1$.

For instance, if you have 5 observations, you need to compare our *t-value* with the *t-value* in the Student distribution with 4 degrees of freedom and at 95 percent confidence interval. To reject the null hypothesesis, the *t-value* should be higher than 2.77.

Cf table below:

	90%	95%	97.5%	99%	99.5%	99.95%	1-Tail Confidence Level
	80%	90%	95%	98%	99%	99.9%	2-Tail Confidence Level
	0.100	0.050	0.025	0.010	0.005	0.0005	1-Tail Alpha
df	0.20	0.10	0.05	0.02	0.01	0.001	2-Tail Alpha
1	3.0777	6.3138	12.7062	31.8205	63.6567	636.6192	
2	1.8856	2.9200	4.3027	6.9646	9.9248	31.5991	
3	1.6377	2.3534	3.1824	4.5407	5.8409	12.9240	
4	1.5332	2.1318	2.7764	3.7469	4.6041	8.6103	
5	1.4759	2.0150	2.5706	3.3649	4.0321	6.8688	
6	1.4398	1.9432	2.4469	3.1427	3.7074	5.9588	

The t value for 4 degrees of freedom is 2.77 for 95% confidence interval

Example:

Suppose you are a company producing cookies. Each cookie is supposed to contain 10 grams of sugar. The cookies are produced by a machine that adds the sugar in a bowl before mixing everything together. You believe the machine does not add 10 grams of sugar for each cookie. If your assumption is true, the machine needs to be fixed. You stored the level of sugar of thirty cookies.

Note: You can create a randomized vector with the function `rnorm()`. This function generates normally distributed values. The basic syntax is:

```
rnorm(n, mean, sd)
arguments

- n: Number of observations to generate
- mean: The mean of the distribution. Optional
- sd: The standard deviation of the distribution. Optional
```

You can create a distribution with 30 observations with a mean of 9.99 and a standard deviation of 0.04.

```
set.seed(123)
sugar_cookie <- rnorm(30, mean =9.99, sd = 0.04)
head(sugar_cookie)
```

```
## [1]  9.967581  9.980793 10.052348  9.992820  9.995172 10.058603
```

You can use a one-sample *t-test* to check whether the level of sugar is different than the recipe. You can draw a hypothesis test:

- H0: The average level of sugar is equal to 10
- H1: The average level of sugar is different than 10

You use a significance level of 0.05.

```
# H0 : mu = 10
t.test(sugar_cookie, mu = 10)
```

Here is the output

Degree of freedom = n-1

```
        One Sample t-test

data:  sugar_cookie
t = -1.6588, df = 29, p-value = 0.1079
alternative hypothesis: true mean is not equal to 10
95 percent confidence interval:
  9.973463 10.002769
sample estimates:
mean of x
 9.988116
```

P value: below 0.05, we can reject the Null hypothesis

The true mean is between this interval with a probability of 95%

Mean of x

The *p-value* of the one sample *t-test* is 0.1079 and above 0.05. You can be confident at 95% that the amount of sugar added by the machine is between 9.973 and 10.002 grams. You cannot reject the null (H0) hypothesis. There is not enough evidence that amount of sugar added by the machine does not follow the recipe.

5.1.3 Paired *t-test*

The paired *t-test*, or dependant sample *t-test*, is used when the mean of the treated group is computed twice. The basic application of the paired *t-test* is:

- A/B testing: Compare two variants
- Case control studies: Before/after treatment

Example:

A beverage company is interested in knowing the performance of a discount program on the sales. The company decided to follow the daily sales of one of its shops where the program is being promoted. At the end of the program, the company wants to know if there is a statistical difference between the average sales of the shop before and after the program.

- The company tracked the sales everyday before the program started. This is our first vector.
- The program is promoted for one week and the sales are recorded every day. This is our second vector.
- You will perform the *t-test* to judge the effectiveness of the program. This is called a paired t test because the values of both vectors come from the same distribution (i.e. the same shop).

The hypothesis testing is:

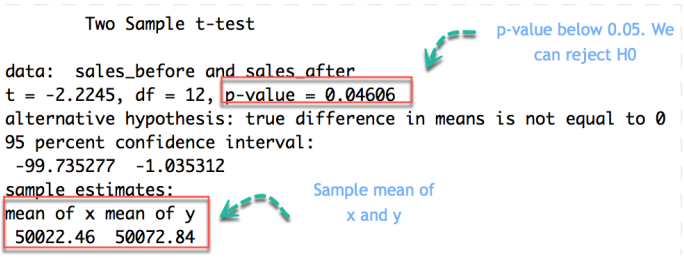
- H0: No difference in mean
- H1: The two means are different

Remember, one assumption in the *t-test* is an unknown but equal variance. In reality, the data barely have equal mean and it leads to incorrect results for the *t-test*.

One solution to relax the equal variance assumption is to use the *Welch's test*. R assumes the two variances are not equal by default. In our dataset, both vectors have the same variance, you can set `var.equal= TRUE`.

You create two random vectors from a Gaussian distribution with a higher mean for the sales after the program.

```
set.seed(123)
# sales before the program
sales_before <- rnorm(7, mean =50000, sd = 50)
# sales after the program. This has higher mean
sales_after <- rnorm(7, mean =50075, sd =50)
# draw the distribution
t.test(sales_before,sales_after, var.equal= TRUE)
```



You obtained a *p-value* of 0.04606, lower than the threshold of 0.05. You conclude the averages of the two groups are significantly different. The program improves the sales of shops.

5.1.4 Summary

The *t-test* belongs to the family of inferential statistics. It is commonly employed to find out if there is a statistical difference between the means of two groups.

You can summarize the *t-test* is the table below:

test	Hypothesis to test	*p-value*	code	optional argument
one-sample *t-test*	Mean of a vector is different from the theoretical mean	0.05	t.test(x, mu = mean)	
paired sample *t-test*	Mean A is different from mean B for the same group	0.06	t.test(A,B, mu = mean)	var.equal= TRUE

If you assume the variances are equal, you need to change the parameter `var.equal= TRUE`.

5.2 ANOVA

Analysis of Variance (ANOVA) helps you test differences between two or more group means. ANOVA test is elaborated around the different sources of variation (variation between and within group) in a typical variable. A primarily ANOVA test provides evidences of the existence of the mean equality between group. This statistical method is an extension of the *t-test*. It is used in a situation where the factor variable has more than one group.

5.2.1 One-way ANOVA

There are many situations where you need to compare the mean between multiple groups. For instance, the marketing department wants to know if three teams have the same sales performance.

- Team: 3 level factor: A, B, and C
- Sale: A measure of performance

The ANOVA test can tell if the three groups have similar performances.

To clarify if the data comes from the same population, you can perform a **one-way analysis of variance** (one-way ANOVA hereafter). This test, like any other statistical tests, gives evidences whether the H0 hypothesis can be accepted or rejected

Hypothesis in one-way ANOVA test:

- H0: The means between groups are identical
- H1: At least, the mean of one group is different

In other words, the H0 hypothesis implies that there is not enough evidence to prove the mean of the group (factor) are different from another.

This test is similar to the **t-test**, although ANOVA test is recommended in situation there are more than 2 groups. Except that, the **t-test** and *ANOVA* provides similar results.

We assume that each factor is randomly sampled, independent and comes from a normally distributed population with unknown but equal variances.

Interpret ANOVA test

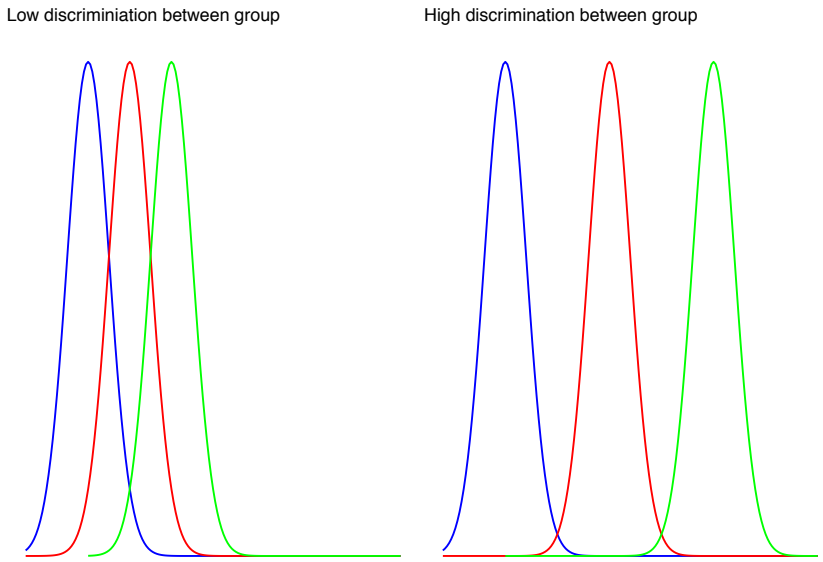
The F-statistic is used to test if the data are from significantly different populations, i.e. different sample means.

To compute the F-statistic, you need to divide the **between-group variability** over the **within-group variability**.

The **between-group** variability reflects the differences between the groups inside all of the population. Look at the two graphs below to understand the concept of between-group variance.

The left graph shows very little variation between the three group, and it is very likely that the three means tends to the **overall** mean (i.e. mean for the three groups).

The right graph plots three distributions far apart and none of them overlap. There is a high chance the difference between the total mean and the groups mean will be large.

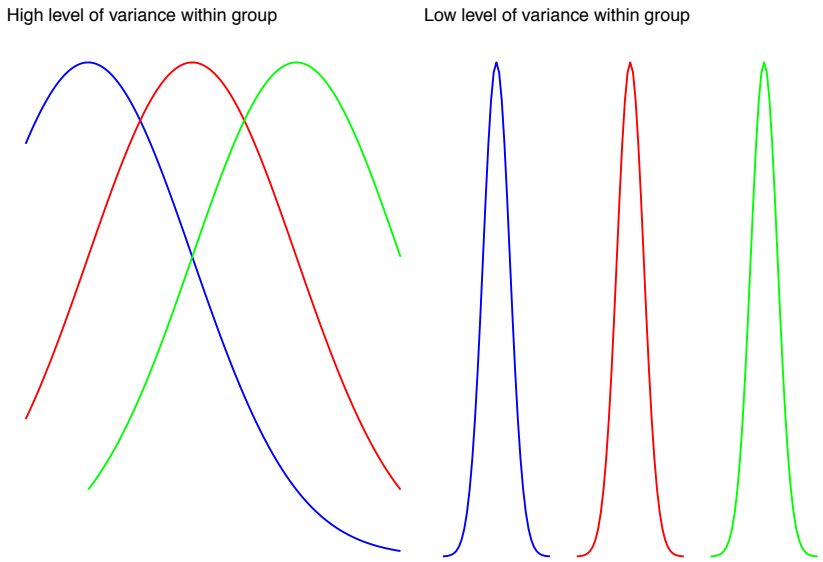


The **within-group** variability considers the difference inside the groups. The variation comes from the individual observations; some points might be totally different than the group means. The **within-group** variability picks up this effect and refer to the sample error.

To understand visually the concept of within-group variability, you can look at the graph below.

The left part plots the distribution of three different groups. You increased the spread for each sample and it is clear the individual variance is large. The F-test will decrease, meaning you tend to accept the null hypothesis.

The right part shows exactly the same samples (identical mean) but with lower variability. It leads to an increase of the F-test and tends in favor of the alternative hypothesis.



You can use both measure to construct the F-statistics. This is very intuitive to understand the F-statistic. If the numerator increases, it means the between group variability is high and it is likely the groups in the

sample are drawn from completely different distributions.

In other words, a low F-statistic indicates little or no significant difference between the group's average.

You will use the `poison` dataset to implement the one-way ANOVA test. The dataset contains 48 rows and 3 variables:

- `Time`: Survival time of the animal
- `poison`: Type of poison used: factor level: 1,2 and 3
- `treat`: Type of treatment used: factor level: 1,2 and 3

Before you start to compute the ANOVA test, you need to prepare the data as follow:

- Step 1: Import the data
- Step 2: Remove unnecessary variable
- Step 3: Convert the variable `poison` as ordered level

```
library(dplyr)
PATH <-  "https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/poisons.csv"
# Step 1
df <- read.csv(PATH) %>%
# Step 2
          select(-X) %>%
# Step 3
          mutate(poison = factor(poison, ordered = TRUE))
glimpse(df)
```

```
## Observations: 48
## Variables: 3
## $ time   <dbl> 0.31, 0.45, 0.46, 0.43, 0.36, 0.29, 0.40, 0.23, 0.22, 0...
## $ poison <ord> 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 1, 1, 1, 1, 2, 2, 2...
## $ treat  <fctr> A, A, A, A, A, A, A, A, A, A, A, A, B, B, B, B, B, B, ...
```

Our objective is to test the following assumption:

- H0: There is no difference in survival time average between group
- H1: The survival time average is different for at least one group.

In other words, you want to know if there is a statistical difference between the mean of the survival time according to the type of poison given to the Guinea pig.

You will proceed as follow:

- Step 1: Check the format of the variable `poison`
- Step 2: Print the summary statistic: count, mean and standard deviation
- Step 3: Plot a box plot
- Step 4: Compute the one-way ANOVA test
- Step 5: Run a pairwise t test

Step 1: Format

You can check the level of the `poison with the following code. You should see three character values because you convert them in factor with the`mutate‘ verb.

```
levels(df$poison)
```

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

Step 2: Summary statistics

You compute the mean and standard deviation. They are the first moments to define the population parameters.

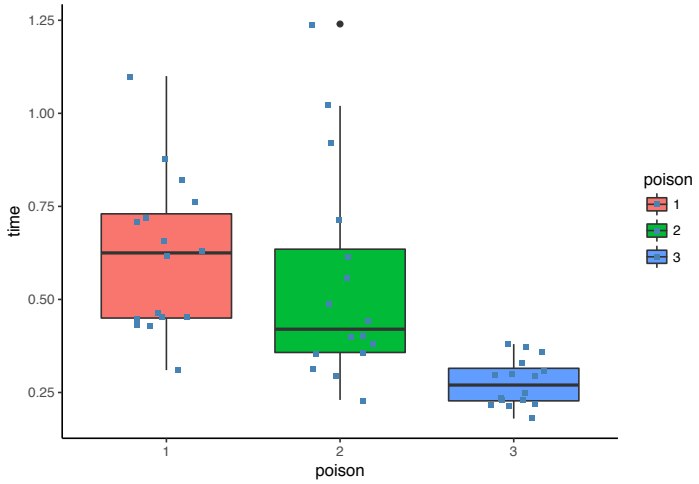
```
df %>%
  group_by(poison) %>%
  summarise(
    count_poison = n(),
    mean_time = mean(time, na.rm = TRUE),
    sd_time = sd(time, na.rm = TRUE)
  )
```

```
## # A tibble: 3 x 4
##   poison count_poison mean_time    sd_time
##    <ord>        <int>     <dbl>      <dbl>
## 1      1           16  0.617500 0.20942779
## 2      2           16  0.544375 0.28936641
## 3      3           16  0.276250 0.06227627
```

Step 3: Box plot

In step three, you can graphically check if there is a difference between the distribution. Note that you include the jittered dot.

```
ggplot(df, aes(x= poison, y =time, fill = poison))+
  geom_boxplot()+
    geom_jitter(shape=15,
                color = "steelblue",
                position=position_jitter(0.21))+
  theme_classic()
```



Step 4: One way ANOVA

You can run the one-way ANOVA test with the command `aov`. The basic syntax for an ANOVA test is:

```
aov(formula, data)
Arguments:

- formula: The equation you want to estimate
- data: The dataset used
```

The syntax of the formula is:

```
y ~ X1+ X2+...+Xn # X1 +  X2 +... refers to the independant variables
y ~ . # use all the remaining variables as independant variables
```

You can answer our question: Is there any difference in the survival time between the Guinea pig, knowing the type of poison given.

Note that, it is advised to store the model and use the function `summary()` to get a better print of the results.

```
anova_one_way <- aov(time ~ poison, data = df)
summary(anova_one_way)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## poison       2  1.033  0.5165   11.79 7.66e-05 ***
## Residuals   45  1.972  0.0438
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Code Explanation

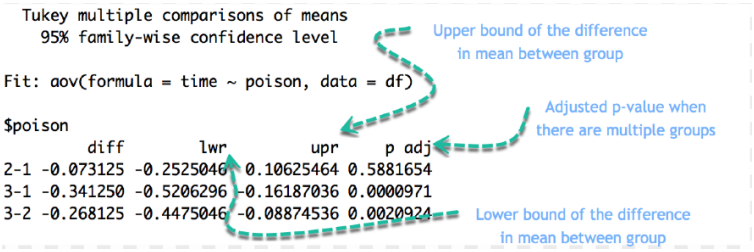
- `aov(time ~ poison, data = df)`: Run the ANOVA test with the following formula $time = poison$
- `summary(anova_one_way)`: Print the summary of the test

The *p-value* is lower than the usual threshold of 0.05. You are confident to say there are a statistical difference between the groups, indicated by the *.

5.2.2 Pairwise comparison

The one-way ANOVA test does not inform which group has a different mean. Instead, you can perform a Tukey test with the function `TukeyHSD()`.

```
TukeyHSD(anova_one_way)
```



5.2.3 Two-way ANOVA

A two-way ANOVA test adds another group variable to the formula. This is basically identical to the one-way ANOVA test, the formula changes slightly to:

$$y = x1 + x2$$

with y is a quantitative variable and $x1$ and $x2$ are categorical variables.

5.2.4 Hypothesis in two way ANOVA test

- H0: The means are equal for both variables (i.e. factor variable)
- H1: The means are different for both variables

You add `treat` variable to our model. This variable indicates the treatment given to the Guinea pig. You are interested to see if there is a statistical dependence between the poison and treatment given to the Guinea pig.

We adjust our code by adding `treat` with the other independent variable.

```
anova_two_way <- aov(time ~ poison+ treat, data = df)
summary(anova_two_way)
```

```
##             Df Sum Sq Mean Sq F value  Pr(>F)
## poison       2 1.0330  0.5165   20.64 5.7e-07 ***
## treat        3 0.9212  0.3071   12.27 6.7e-06 ***
## Residuals   42 1.0509  0.0250
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

You can conclude that both `poison`and `treat` are statistically different from 0. You can reject the NULL hypothesis and confirm that changing the treatment or the poison impact the time of survival.

5.2.5 Summary

We can summarize the test in the table below:

Test	code	hypothesis	*p-value*
One way ANOVA	aov(y ~ X, data = df)	H1: Average is different for at least one group	0.05
Pairwise	TukeyHSD(ANOVA summary)		0.05
Two way ANOVA	aov(y ~ X1 + X2, data = df)	H1: Average is different for both group	0.05

5.3 Correlation

A bivariate relationship describes a relationship -or correlation- between two variables, x and y. In this tutorial, we discuss the concept of **correlation** and show how it can be used to measure the relationship between any two variables.

There are two primary methods to compute the correlation between two variables.

- Pearson: Parametric correlation
- Spearman: Non-parametric correlation

5.3.1 Pearson

The Pearson correlation method is usually used as a primary check for the relationship between two variables.

The **coefficient of correlation**, r, is a measure of the strength of the **linear** relationship between two variables x and y. It is computed as follow:

$r = \frac{Cov(x,y)}{\sigma_x \sigma_y}$

with

- $\sigma_x = \sum (x - \bar{x})^2$, i.e. standard deviation of x
- $\sigma_y = \sum (y - \bar{y})^2$, i.e. standard deviation of y

The correlation ranges between -1 and 1.

- A value of r near or equal to 0 implies little or no linear relationship between y and x.
- In contrast, the closer r comes to 1 or -1, the stronger the linear relationship.

We can compute the *t-test* as follow and check the distribution table with a degree of freedom equals to $n-2$:

$$t = \frac{r}{\sqrt{1-r^2}}\sqrt{n-2}$$

5.3.2 Spearman rank correlation

A rank correlation sorts the observations by rank and computes the level of similarity between the rank. A rank correlation has the advantage of being robust to outliers and is not linked to the distribution of the data. Note that, a rank correlation is suitable for the ordinal variable.

Spearman's rank correlation, ρ, is always between -1 and 1 with a value close to the extremity indicates strong relationship. It is computed as follow:

$$\rho = \frac{Cov(rg_x, rg_y)}{\sigma_{rg_x}\sigma_{rg_y}}$$

with $Cov(rg_x, rg_y)$ stated the covariances between rank rg_x and rg_y. The denominator calculates the standard deviations.

In R, we can use the `cor()` function. It takes three arguments, y, x and the `method`.

```
cor(x, y, method)
arguments:

- x: First vector
- y: Second vector
- method: The formula used to compute the correlation. Three string values:

    - "pearson"
    - "kendall"
    - "spearman"

An optional argument can be added if the vectors contain missing value:
- use = "complete.obs"
```

We will use the `BudgetUK` dataset. This dataset reports the budget allocation of British households between 1980 and 1982. There are 1519 observations with ten features, among them:

- `wfood`: share food share spend
- `wfuel`: share fuel spend
- `wcloth`: budget share for clothing spend
- `walc`: share alcohol spend
- `wtrans`: share transport spend
- `wother`: share of other goods spend
- `totexp`: total household spend in pound
- `income`: total net household income
- `age`: age of household
- `children`: number of children

```
library(dplyr)
PATH <-  "https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/british_household.csv"
data <- read.csv(PATH) %>%
        filter(income <500) %>%
        mutate(log_income = log(income),
               log_totexp = log(totexp),
```

```
                children_fac = factor(children, order= TRUE, labels = c("No", "Yes"))) %>%
        select(-c(X, X.1, children, totexp, income))

glimpse(data)
```

```
## Observations: 1,516
## Variables: 10
## $ wfood        <dbl> 0.4272, 0.3739, 0.1941, 0.4438, 0.3331, 0.3752, 0...
## $ wfuel        <dbl> 0.1342, 0.1686, 0.4056, 0.1258, 0.0824, 0.0481, 0...
## $ wcloth       <dbl> 0.0000, 0.0091, 0.0012, 0.0539, 0.0399, 0.1170, 0...
## $ walc         <dbl> 0.0106, 0.0825, 0.0513, 0.0397, 0.1571, 0.0210, 0...
## $ wtrans       <dbl> 0.1458, 0.1215, 0.2063, 0.0652, 0.2403, 0.0955, 0...
## $ wother       <dbl> 0.2822, 0.2444, 0.1415, 0.2716, 0.1473, 0.3431, 0...
## $ age          <int> 25, 39, 47, 33, 31, 24, 46, 25, 30, 41, 48, 24, 2...
## $ log_income   <dbl> 4.867534, 5.010635, 5.438079, 4.605170, 4.605170,...
## $ log_totexp   <dbl> 3.912023, 4.499810, 5.192957, 4.382027, 4.499810,...
## $ children_fac <ord> Yes, Yes, Yes, Yes, No, No, No, No, No, No, Yes, ...
```

Code Explanation

- We first import the data and have a look with the `glimpse()` function from the `dplyr` library.
- Three points are above 500K, so we decided to exclude them.
- It is a common practice to convert a monetary variable in log. It helps to reduce the impact of outliers and decreases the skewness in the dataset.

We can compute the correlation coefficient between `income` and `wfood` variables with the "pearson" and "spearman" methods.

```
cor(data$log_income, data$wfood, method = "pearson")
```

```
## [1] -0.2466986
```

```
cor(data$log_income, data$wfood, method = "spearman")
```

```
## [1] -0.2501252
```

5.3.3 Correlation matrix

The bivariate correlation is a good start but we can get a broader picture with multivariate analysis. A correlation with many variables is pictured inside a **correlation matrix**. A correlation matrix is a matrix $n * n$ that represents the pair correlation of all the variables.

The `cor()` function returns a correlation matrix. The only difference with the bivariate correlation is we don't need to specify which variables. By default R computes the correlation between all the variables.

Note that, a correlation cannot be computed for factor variable. We need to make sure we drop categorical feature before we pass the data frame inside `cor()`.

A correlation matrix is symmetrical which means the values above the diagonal have the same values as the one below. It is more visual to show half of the matrix.

We exclude `children_fac` because it is a factor level variable. `cor` does not perform correlation on a categorical variable.

```
# the last column of data is a factor level. We don't include it in the code
mat_1 <-as.dist(round(cor(data[,1:9]),2))
mat_1
```

```
##            wfood wfuel wcloth  walc wtrans wother   age log_income
## wfuel       0.11
## wcloth     -0.33 -0.25
## walc       -0.12 -0.13  -0.09
## wtrans     -0.34 -0.16  -0.19 -0.22
## wother     -0.35 -0.14  -0.22 -0.12  -0.29
## age         0.02 -0.05   0.04 -0.14   0.03   0.02
## log_income -0.25 -0.12   0.10  0.04   0.06   0.13  0.23
## log_totexp -0.50 -0.36   0.34  0.12   0.15   0.15  0.21       0.49
```

Code Explanation

- `cor(data)`: Display the correlation matrix
- `round(data, 2)`: Round the correlation matrix with two decimals
- `as.dist()`: Shows the second half only

5.3.4 Significance level

The significance level is useful in some situations when we use the `pearson` or `spearman` method. The function `rcorr()` from the library `Hmisc` computes for us the `*p-value*`.

- `hmisc`: Add significance to correlation matrix. If you have install R with `r-essential`. It is already in the library
  - Anaconda: `conda install -c r r-hmisc`

The `rcorr()` requires a data frame to be stored as a matrix. We can convert our data into a matrix before to compute the correlation matrix with the `*p-value*`.

```
library("Hmisc")
data_rcorr <- as.matrix(data[,1:9])

mat_2 <- rcorr(data_rcorr)
# mat_2 <- rcorr(as.matrix(data)) returns the same output
```

The list object `mat_2` contains three elements:

- r: Output of the correlation matrix
- n: Number of observation
- P: *p-value*

We are interested by the third element, the *p-value*. It is common to show the correlation matrix with the *p-value* instead of the coefficient of correlation.

```
p_value <- round(mat_2[["P"]], 3)
p_value
```

```
##            wfood wfuel wcloth  walc wtrans wother   age log_income
## wfood         NA 0.000  0.000 0.000  0.000  0.000 0.365      0.000
## wfuel      0.000    NA  0.000 0.000  0.000  0.000 0.076      0.000
## wcloth     0.000 0.000     NA 0.001  0.000  0.000 0.160      0.000
## walc       0.000 0.000  0.001    NA  0.000  0.000 0.000      0.105
## wtrans     0.000 0.000  0.000 0.000     NA  0.000 0.259      0.020
## wother     0.000 0.000  0.000 0.000  0.000     NA 0.355      0.000
## age        0.365 0.076  0.160 0.000  0.259  0.355    NA      0.000
## log_income 0.000 0.000  0.000 0.105  0.020  0.000 0.000         NA
## log_totexp 0.000 0.000  0.000 0.000  0.000  0.000 0.000      0.000
##            log_totexp
## wfood               0
```

```
## wfuel               0
## wcloth              0
## walc                0
## wtrans              0
## wother              0
## age                 0
## log_income          0
## log_totexp         NA
```

Code Explanation

- `mat_2[["P"]]`: The *p-value*s are stored in the element called P
- `round(mat_2[["P"]], 3)`: Round the elements with three digits

5.3.5 Visualize correlation matrix

A heat map is another way to show a correlation matrix. The GGally library is an extension of ggplot2. Currently, it is not available in the conda library. We can install directly in the console.

```
install.packages("GGally")
```

The library includes different functions to show the summary statistics such as the correlation and distribution of all the variables in a matrix.

The `ggcorr()` function has lots of arguments. We will introduce only the arguments we will use in the tutorial:

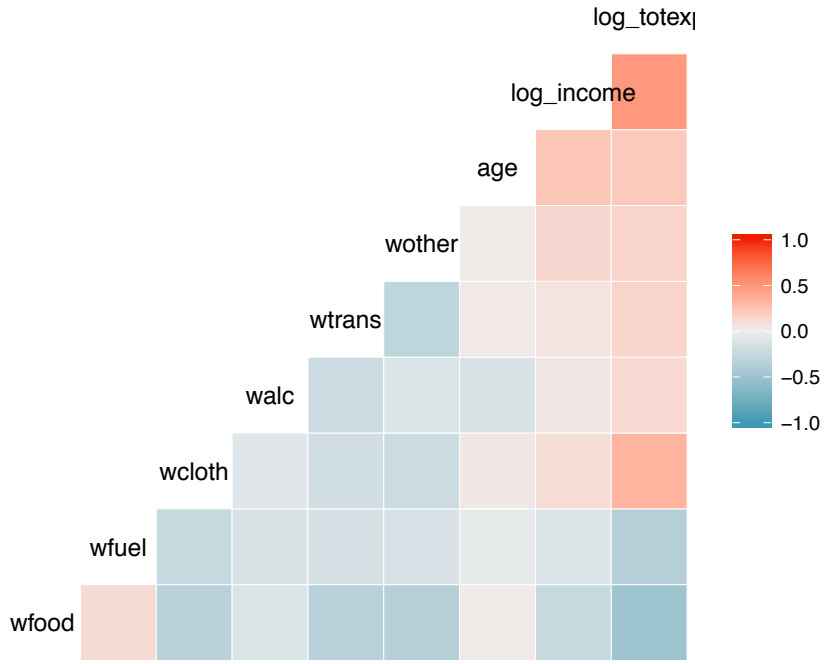
The function ggcorr

```
ggcorr(df, method = c("pairwise", "pearson"),
  nbreaks = NULL, digits = 2, low = "#3B9AB2",
  mid = "#EEEEEE", high = "#F21A00",
  geom = "tile", label = FALSE,
  label_alpha = FALSE)
arguments:

- df: Dataset used
- method: Formula to compute the correlation. By default, pairwise and Pearson are computed
- nbreaks: Return a categorical range for the coloration of the coefficients. By default, no break and
- digits: Round the correlation coefficient. By default, set to 2
- low: Control the lower level of the coloration
- mid: Control the middle level of the coloration
- high: Control the high level of the coloration
- geom: Control the shape of the geometric argument. By default, "tile"
- label: Boolean value. Display or not the label. By default, set to `FALSE`
```

5.3.6 Heat map

The most basic plot of the package is a heat map. The legend of the graph shows a gradient color from - 1 to 1, with hot color indicating strong positive correlation and cold color, a negative correlation.

```
library(GGally)
ggcorr(data)
```



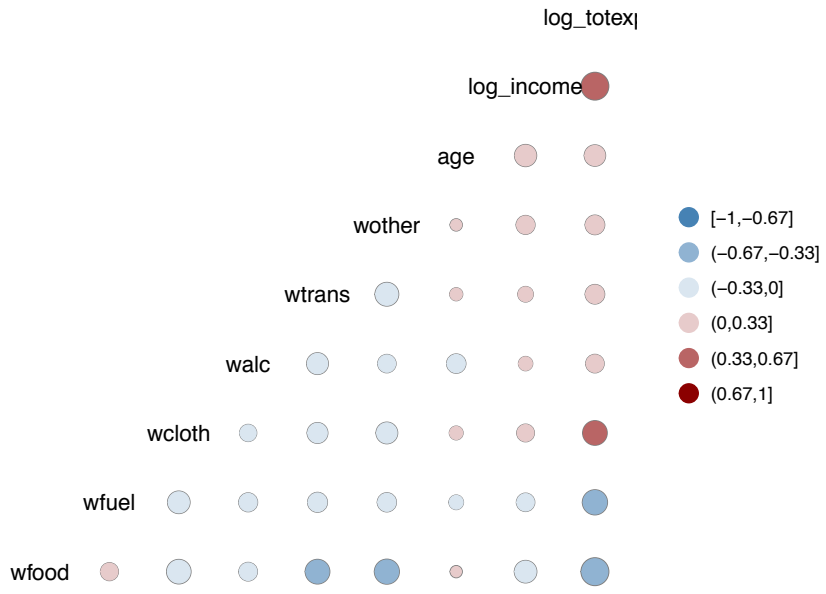
Code Explanation

- `ggcorr(data)`: Only one argument is needed, which is the data frame name. Factor level variables are not included in the plot.

Add control to the heat map

We can add more controls to the graph.

```
ggcorr(data,
       nbreaks = 6,
       low = "steelblue",
       mid = "white",
       high = "darkred",
       geom = "circle")
```



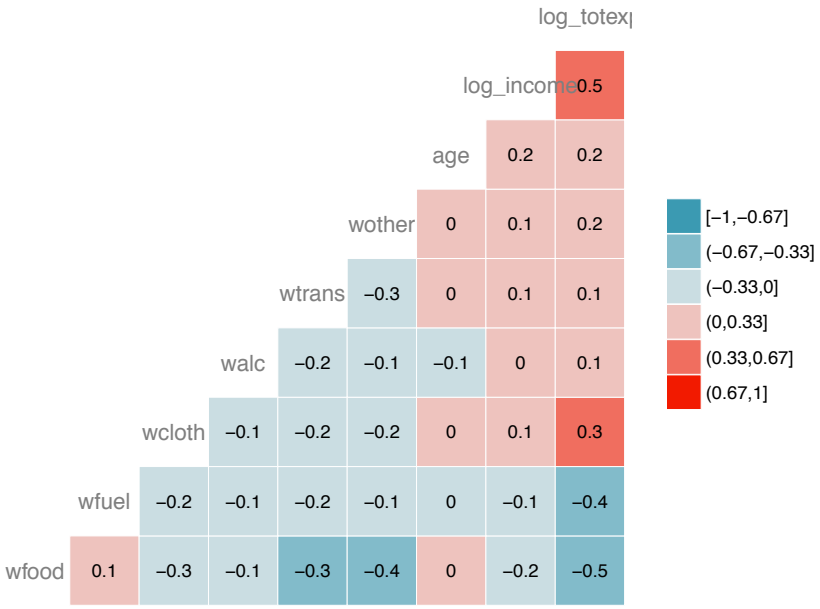
Code Explanation

- `nbreaks=6`: break the legend with 6 ranks.
- `low = "steelblue"`: Use lighter colors for negative correlation
- `mid = "white"`: Use white colors for middle ranges correlation
- `high = "darkred"`: Use dark colors for positive correlation
- `geom = "circle"`: Use circle as the shape of the windows in the heat map. The size of the circle is proportional to the absolute value of the correlation.

Add label to the heat map

`GGally` allows us to add a label inside the windows.

```
ggcorr(data,
       nbreaks = 6,
       label = TRUE,
       label_size = 3,
       color = "grey50")
```



Code Explanation

- `label = TRUE`: Add the values of the coefficients of correlation inside the heat map.
- `color = "grey50"`: Choose the color, i.e. grey
- `label_size = 3`: Set the size of the label equals to 3

The function ggpairs

Finally, we introduce another function from the `GGaly` library. `ggpair` produces a graph in a matrix format. We can display three kinds of computation within one graph. The matrix is a $n * n$ dimension, with n equals the number of observations. The upper/lower part displays $(n-1) * 2$ windows and n in the diagonal. We can control what information we want to show in each part of the matrix. The formula for `ggpair` is:

```
ggpair(df, columns = 1:ncol(df),title = NULL,
       upper = list(continuous = "cor"),
       lower = list(continuous = "smooth"),
  mapping = NULL)
arguments:

- df: Dataset used
```

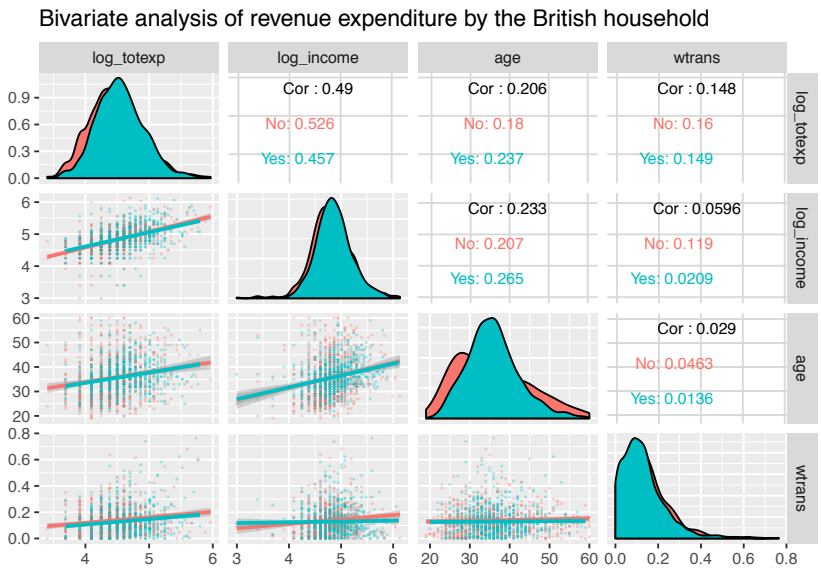
```
- columns: Select the columns to draw the plot
- title:  Include a title
- upper: Control the boxes above the diagonal of the plot. Need to supply the type of computations or g
- Lower: Control the boxes below the diagonal.
- mapping: Indicates the aesthetic of the graph. For instance, we can compute the graph for different g
```

Bivariate analysis with ggpair with grouping

The next graph plots three information:

- The correlation matrix between `log_totexp`, `log_income`, `age` and `wtrans` variable grouped by whether the household has a kid or not.
- Plot the distribution of each variable by group
- Display the scatter plot with the trend by group

```
library(ggplot2)
ggpairs(data, columns = c("log_totexp", "log_income", "age", "wtrans"), title =  "Bivariate analysis of
        upper = list(continuous = wrap("cor",
                       size = 3)),
        lower = list(continuous = wrap("smooth",
                        alpha = 0.3,
                        size=0.1)),
    mapping = aes(color = children_fac)
)
```



Code Explanation

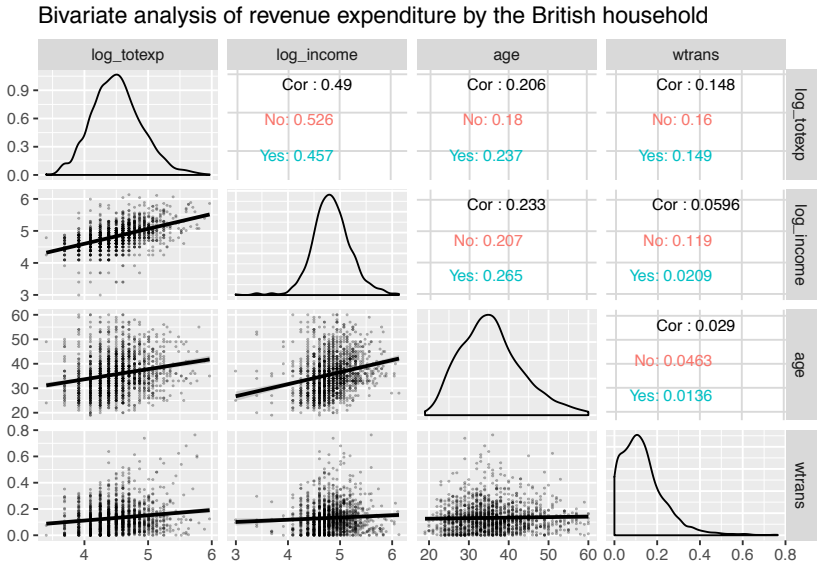
- `columns = c("log_totexp", "log_income", "age", "wtrans")`: Choose the variables to show in the graph
- `title =  "Bivariate analysis of revenue expenditure by the British household"`: Add a title
- upper = list(): Control the upper part of the graph. I.e. Above the diagonal
- `continuous = wrap("cor", size = 3))`: Compute the coefficient of correlation. We wrap the argument `continuous` inside the `wrap()` function to control for the aesthetic of the graph (i.e. `size = 3`) -lower = list(): Control the lower part of the graph. I.e. Below the diagonal.
- `continuous = wrap("smooth",alpha = 0.3,size=0.1)`: Add a scatter plot with a linear trend. We wrap the argument `continuous` inside the `wrap()` function to control for the aesthetic of the graph (i.e. `size=0.1`, `alpha=0.3`)

- `mapping = aes(color = children_fac)`: We want each part of the graph to be stacked by the variable `children_fac`, which is a categorical variable taking the value of 1 if the household does not have kids and 2 otherwise

Bivariate analysis with ggpair with partial grouping

The graph below is a little bit different. We change the position of the mapping inside the `upper` argument.

```
ggpairs(data, columns = c("log_totexp", "log_income", "age", "wtrans"), title =  "Bivariate analysis of
        upper = list(continuous = wrap("cor",
                     size = 3),
                     mapping = aes(color = children_fac)),
        lower = list(
                      continuous = wrap("smooth",
                      alpha = 0.3,
                      size=0.1))
)
```



Code Explanation

- Exact same code as previous example except for:
- `mapping = aes(color = children_fac)`: Move the list in `upper = list()`. We only want the computation stacked by group in the upper part of the graph.

5.3.7 Summary

We can summarize the function in the table below:

library	Objective	method	code
Base	bivariate correlation	Pearson	cor(df$x$1, dfx2, method = "pearson")
Base	bivariate correlation	Spearman	cor(df$x$1, dfx2, method = "spearman")
Base	Multivariate correlation	pearson	cor(df, method = "pearson")
Base	Multivariate correlation	Spearman	cor(df, method = "spearman")
Hmisc	P value		rcorr(as.matrix(data[,1:9]))[["P"]]
Ggally	heat map		ggcorr(df)
	Multivariate plots		cf code below

code for multivariate plots:

```
ggpairs(df, columns = c("x1", "x2", "x3", "x4"), title =  "Bivariate analysis of df data frame",
        upper = list(continuous = wrap("cor",
                     size = 3),
                     mapping = aes(color = x1)),
        lower = list(
                      continuous = wrap("smooth",
                      alpha = 0.3,
                      size=0.1))
)

with x1 as factor level
```

5.4 Machine learning

Machine learning is becoming widespread among data scientist and is deployed in hundreds of products you use daily. One of the first ML application was **spam filter**.

Following are other application of Machine Learning

- Identification of unwanted spam messages in email
- Segmentation of customer behavior for targeted advertising
- Reduction of fraudulent credit card transactions
- Optimization of energy use in home and office building
- Facial recognition

Before you start to implement machine learning algorithm, let's study the difference between **supervised** learning and **unsupervised** learning. This tutorial introduces some fundamental concepts in data science.

In this chapter, you narrow-down your analysis to supervised/unsupervised learning.

5.4.1 Supervised learning

In **supervised learning**, the training data you feed to the algorithm includes a *label*.

Classification task is probably the most used supervised learning technique. One of the first classification task researchers tackled was the spam filter. The objective of the learning is to predict whether an email is classified as spam or ham. The machine, after the training step, can detect the class of email.

Regressions are commonly used in the machine learning field to predict continuous value. Regression task can predict the value of a **dependent variable** based on a set of **independent variables** (also called predictors or regressors). For instance, linear regressions can predict a stock price, weather forecast, sales and so on.

Here is the list of some fundamental supervised learning algorithms.

- K-Nearest Neighbors
- Linear regression*
- Logistic regression*
- Support Vector Machine (SVM)
- Decision trees and Random Forest*
- Neural Networks

5.4.2 Unsupervised learning

In **unsupervised learning**, the training data is unlabeled. The system tries to learn without a reference. Below is a list of unsupervised learning algorithms.

- Clustering
- K-mean*
- Hierarchical Cluster Analysis*
- Expectation Maximization
- Visualization and dimensionality reduction
- Principal Component Analysis
- Kernel PCA
- Locally-Linear Embedding

5.4.3 Simple Linear regression

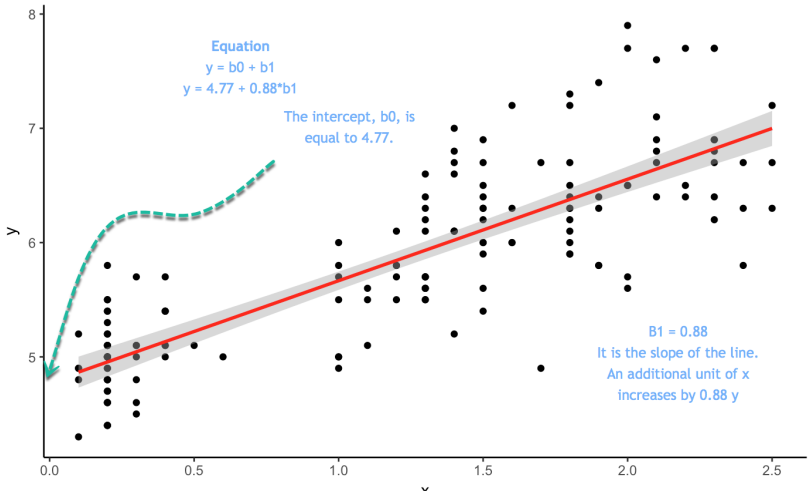
Linear regression answers a simple question: Can you measure an exact relationship between one target variables and a set of predictors?

The simplest of probabilistic models is the *straight line model*:

$y = \beta_0 + \beta_1 x + \varepsilon$ where

- y = Dependent variable
- x = Independent variable
- ε = random error component
- β_0 = intercept
- β_1 = Coefficient of x

Consider the following plot:



The equation is $y = \beta_0 + \beta_1 + \varepsilon$. β_0 is the intercept. If x equals to 0, y will be equal to the intercept, 4.77. β_1 is the slope of the line. It tells in which proportion y varies when x varies.

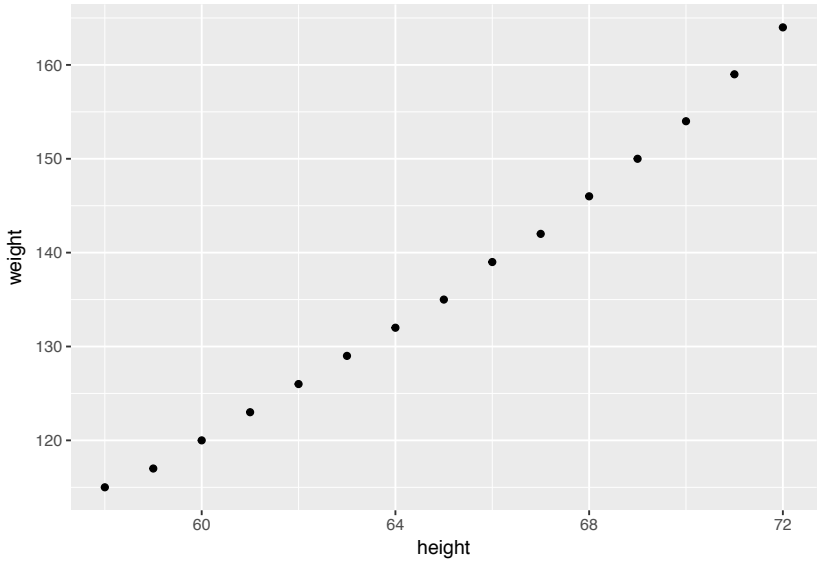
To estimate the optimal values of β_0 and β, you use a method called **Ordinary Least Squares (OLS)**. This method tries to find the parameters that minimize the sum of the squared errors, that is the vertical distance between the predicted y values and the actual y values. The difference is known as the **error term**.

Before you estimate the model, you can determine whether a linear relationship between y and x is plausible by plotting a scatterplot.

Scatterplot

We will use a very simple dataset to explain the concept of simple linear regression. We will import the *Average Heights and Weights for American Women*. The dataset contains 15 observations. You want to measure whether `Heights` are positively correlated with `Weights`.

```
library(ggplot2)
path <- 'https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/women.csv'
df <- read.csv(path)
ggplot(df, aes(x=height, y =  weight))+
  geom_point()
```



The scatterplot suggests a general tendency for y to increase as x increases. In the next step, you will measure by how much *weight* increases for each additional *height*.

5.4.4 Least Squares Estimates

In a simple OLS regression, the computation of α_0 and β_0 is straightforward. We don't mean to show the derivation in this tutorial. We will only write the formula.

We want to estimate: $y = \beta_0 + \beta_1 x + \varepsilon$

The goal of the OLS regression is to minimize the following equation:

$\sum(y_i - \hat{y}_i)^2 = \sum e_i^2$

where y_i is the actual values and $\hat{y}_i$ is the predicted values.

The solution for β is $\beta_0 = \bar{y} - \beta_1 \bar{x}$

Note that $\bar{x}$ means the average value of x

The solution for β is $\beta = \frac{Cov(x,y)}{Var(x)}$

In R, you can use the `cov()`and `var()`function to estimate β and you can use the `mean()` function to estimate α

```
beta <- cov(df$height, df$weight)/var(df$height)
beta
```

```
## [1] 3.45
```

```
alpha <- mean(df$weight) - beta*mean(df$height)
alpha
```

```
## [1] -87.51667
```

The beta coefficient implies that for each additional height, the weight increases by 3.45.

Estimating simple linear equation manually is not ideal. R provides a suitable function to estimate the parameters. You will see this function shortly. Before that, we will introduce how to compute by hand a simple linear regression model. In your journey of data scientist, you will barely or never estimate a simple linear model. In most situation, regression tasks are performed on a lot of estimators.

5.4.5 Multiple Linear regression

More practical applications of regression analysis employ models that are more complex than the simple straight-line model. The probabilistic model that includes more than one independent variable is called **multiple regression models**. The general form of this model is:

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_k x_k + \varepsilon$

In matrix notation, you can rewrite the model:

- $Y = \beta X + \varepsilon$

The dependent variable y is now a function of k independent variables. The value of the coefficient β_i determines the contribution of the independent variable x_i and β_0.

We briefly introduce the assumption you make about the random error ε of the OLS:

- Mean equal to 0
- Variance equal to σ^2
- Normal distribution
- Random errors are independent (in a probabilistic sense)

You need to solve for β, the vector of regression coefficients that minimise the sum of the squared errors between the predicted and actual Y values.

The closed-form solution is:

$\beta = (X^T X)^{-1} X^T Y$

with:

- T indicates the **transpose** of the matrix X
- $(X^T X)^{-1}$ indicates the **invertible matrix**

We use the `mtcars` dataset. You are already familiar with the dataset. Your goals are to predict the mile per gallon over a set of features.

5.4.6 Continuous variables

For now, you will only use the continuous variables and put aside categorical features. The variable `am` is a binary variable taking the value of 1 if the transmission is manual and 0 for automatic cars and V/S is also a binary variable.

```
library(dplyr)
df <- mtcars %>%
      select(-c(am, vs, cyl, gear, carb))
glimpse(df)
```

```
## Observations: 32
## Variables: 6
## $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19....
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 1...
## $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, ...
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.9...
## $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3...
## $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 2...
```

We can use the `lm()` function to compute the parameters. The basic syntax of this function is:

```
lm(formula, data, subset)
Arguments:
   formula: The equation to estimate
   data: The dataset used
   subset: Estimate the model on a subset of the dataset
```

The syntax of the formula is:

```
y ~ X1+ X2+...+Xn # With intercept
y ~ . # use all the remaining variables as independant variables
y ~ X1+ X2+...+Xn -1 # Without intercept
```

R handles interaction terms easily for two differents notations:

```
- x:z: compute the interaction of x and z
- x*z: compute the effect of x, z and the interactions x:z. i.e x+z+x:z
```

Your objective is to estimate the mile per gallon based on a set of variables. The equation to estimate is:

$$mpg = \beta_0 + \beta_1 disp + \beta_2 hp + \beta_3 drat + \beta_4 wt + \varepsilon$$

You estimate our first linear regression and store the result in the `fit` object.

```
model <- mpg ~.
fit <- lm(model, df)
fit
```

```
##
## Call:
## lm(formula = model, data = df)
##
## Coefficients:
## (Intercept)         disp           hp         drat           wt
##    16.53357      0.00872     -0.02060      2.01577     -4.38546
##        qsec
##     0.64015
```

The output does not provide enough information about the quality of the fit. You can access more details such as the significance of the coefficients, the degree of freedom and the shape of the residuals with the `summary()` function.

```
summary(fit) ## return the *p-value* and coefficient
```

```
##
## Call:
## lm(formula = model, data = df)
##
## Residuals:
```

```
##     Min      1Q  Median      3Q     Max
## -3.5404 -1.6701 -0.4264  1.1320  5.4996
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 16.53357   10.96423   1.508  0.14362
## disp         0.00872    0.01119   0.779  0.44281
## hp          -0.02060    0.01528  -1.348  0.18936
## drat         2.01578    1.30946   1.539  0.13579
## wt          -4.38546    1.24343  -3.527  0.00158 **
## qsec         0.64015    0.45934   1.394  0.17523
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.558 on 26 degrees of freedom
## Multiple R-squared:  0.8489, Adjusted R-squared:  0.8199
## F-statistic: 29.22 on 5 and 26 DF,  p-value: 6.892e-10
```

Read the table

- The above table proves that there is a strong negative relationship between wt and mileage and positive relationship with drat.
- Only the variable wt has a statistical impact on mpg. Remember, to test a hypothesis in statistic, you use:
  - H0: No statistical impact
  - H1: The predictor has a meaningful impact on y
  - If the *p-value* is lower than 0.05, it indicates the variable is statistically significant
- *Adjusted R-squared*: Variance explained by the model. In your model, the model explained 82 percent of the variance of y. R squared is always between 0 and 1. The higher the better

You can run the ANOVA test to estimate the effect of each feature on the variances with the `anova()` function.

```
anova(fit)
```

```
## Analysis of Variance Table
##
## Response: mpg
##           Df Sum Sq Mean Sq  F value   Pr(>F)
## disp       1 808.89  808.89 123.6185 2.23e-11 ***
## hp         1  33.67   33.67   5.1449 0.031854 *
## drat       1  30.15   30.15   4.6073 0.041340 *
## wt         1  70.51   70.51  10.7754 0.002933 **
## qsec       1  12.71   12.71   1.9422 0.175233
## Residuals 26 170.13    6.54
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

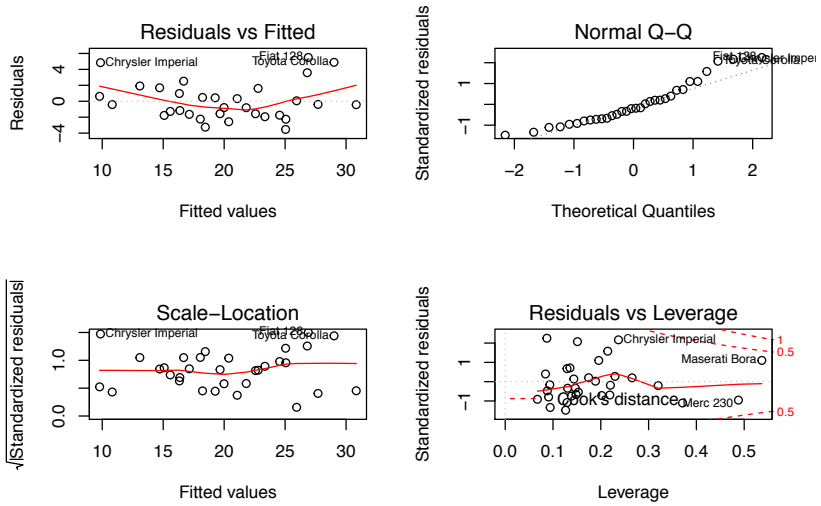
A more conventional way to estimate the model performance is to display the residual against different measures.

You can use the `plot()` function to show four graphs:

```
- Residuals vs Fitted values
- Normal Q-Q plot: Theoretical Quantile vs Standardized residuals
- Scale-Location: Fitted values  vs Square roots of the standardised residuals
- Residuals vs Leverage: Leverage vs Standardized residuals
```

You add the code `par(mfrow=c(2,2))` before `plot(fit)`. If you don't add this line of code, R prompts you to hit the enter command to display the next graph.

```
par(mfrow=c(2,2))
plot(fit)
```



Code Explanation

- `(mfrow=c(2,2))`: return a window with the four graphs side by side.
- The first 2 adds the the number of rows
- The second 2 adds the number of columns.
- If you write `(mfrow=c(3,2))`: you will create a 3 rows 2 columns window

The `lm()` formula returns a list containing a lot of useful information. You can access them with the `fit` object you have created, followed by the '$ sign and the information you want to extract.

- coefficients : `fit$coefficients`
- residuals: `fit$residuals`
- fitted value: `fit$fitted.values`

5.4.7 Factors regression

In the last model estimation, you regress `mpg` on continuous variables only. It is straightforward to add factor variables to the model. You add the variable `am` to your model. It is important to be sure the variable is a factor level and not continuous.

```
df <- mtcars %>%
      mutate(cyl = factor(cyl),
             vs = factor(vs),
             am = factor(am),
             gear = factor(gear),
             carb = factor(carb))
summary(lm(model, df))
```

```
##
## Call:
## lm(formula = model, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
```

```
## -3.5087 -1.3584 -0.0948  0.7745  4.6251
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 23.87913   20.06582   1.190   0.2525
## cyl6        -2.64870    3.04089  -0.871   0.3975
## cyl8        -0.33616    7.15954  -0.047   0.9632
## disp         0.03555    0.03190   1.114   0.2827
## hp          -0.07051    0.03943  -1.788   0.0939 .
## drat         1.18283    2.48348   0.476   0.6407
## wt          -4.52978    2.53875  -1.784   0.0946 .
## qsec         0.36784    0.93540   0.393   0.6997
## vs1          1.93085    2.87126   0.672   0.5115
## am1          1.21212    3.21355   0.377   0.7113
## gear4        1.11435    3.79952   0.293   0.7733
## gear5        2.52840    3.73636   0.677   0.5089
## carb2       -0.97935    2.31797  -0.423   0.6787
## carb3        2.99964    4.29355   0.699   0.4955
## carb4        1.09142    4.44962   0.245   0.8096
## carb6        4.47757    6.38406   0.701   0.4938
## carb8        7.25041    8.36057   0.867   0.3995
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.833 on 15 degrees of freedom
## Multiple R-squared:  0.8931, Adjusted R-squared:  0.779
## F-statistic:  7.83 on 16 and 15 DF,  p-value: 0.000124
```

R uses the first factor level as a base group. You need to compare the coefficients of the other group against the base group.

5.4.8 Features selection

The last part of this tutorial deals with the **stepwise regression** algorithm. The purpose of this algorithm is to add and remove potential candidates in the models and keep those who have a significant impact on the dependent variable. This algorithm is meaningful when the dataset contains a large list of predictors. You don't need to manually add and remove the independent variables. The stepwise regression is built to select the best candidates to fit the model.

Let's see in action how it works. You use the `mtcars` dataset with the continuous variables only for pedagogical illustration. Before you begin analysis, its good to establish variations between the data with a correlation matrix. The `GGally` library is an extension of `ggplot2`.

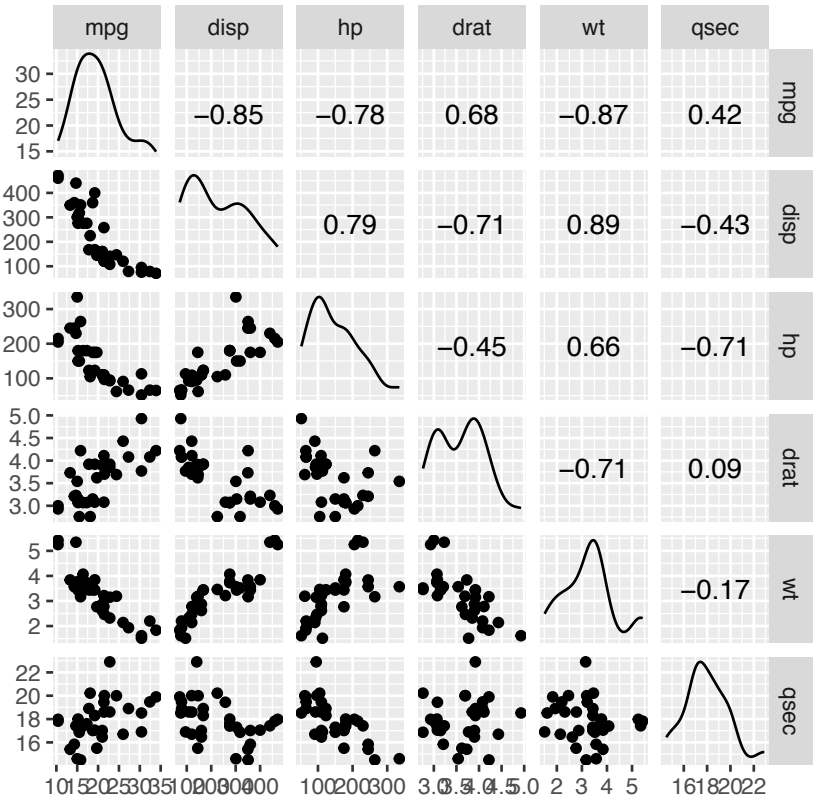
The library includes different functions to show summary statistics such as correlation and distribution of all the variables in a matrix. You will use the `ggscatmat` function, but you can refer to the vignette for more information about the `GGally` library.

The basic syntax for `ggscatmat()` is:

```
ggscatmat(df, columns = 1:ncol(df), corMethod = "pearson")
arguments:

- df:  A matrix of continuous variables
- columns: Pick up the columns to use in the function. By default, all columns are used
- corMethod: Define the function to compute the correlation between variable. By default, the algorithm
```

You display the correlation for all your variables and decides which one will be the best candidates for the first step of the stepwise regression. There are some strong correlations between your variables and the dependent variable, `mpg`.

```
library(GGally)
df <- mtcars %>%
      select(-c(am, vs, cyl, gear, carb))
ggscatmat(df,columns = 1:ncol(df))
```



5.4.9 Stepwise regression

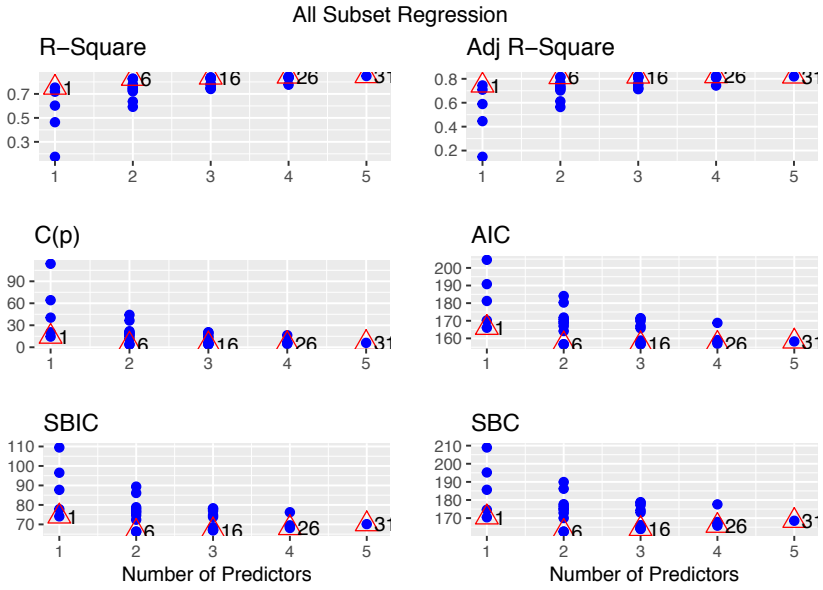
Variables selection is an important part to fit a model. The stepwise regression will perform the searching process automatically. To have a clue on how many possible choices there are in the dataset, you compute 2^k with k is the number of predictors. The amount of possibilities grows bigger with the number of independent variables. That's why you need to have an automatic search.

You need to install the `olsrr` package from CRAN. The package is not available yet in Anaconda. Hence, you install it directly from the command line:

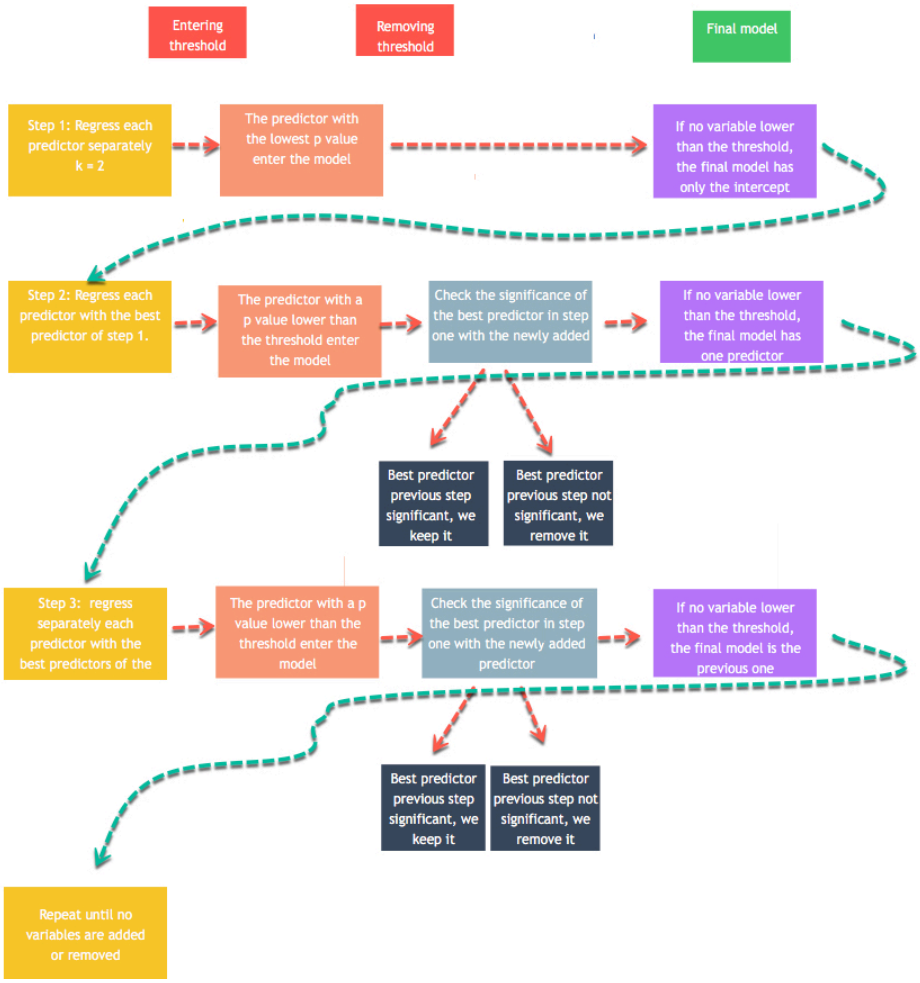
```
install.packages("olsrr")
```

We can plot all the subsets of possibilities with the fit criteria (i.e. R-square, Adjusted R-square, Bayesian criteria)

```
library(olsrr)
model <- mpg ~.
fit <- lm(model, df)
test <- ols_all_subset(fit)
plot(test)
```



Linear regression models use the ***t-test*** to estimate the statistical impact of an independent variable on the dependent variable. Researchers set the maximum threshold at 10 percent, with lower values indicates a stronger statistical link. The strategy of the stepwise regression is constructed around this test to add and remove potential candidates. The algorithm works as follow:



- Step 1: Regress each predictor on `y` separately. Namely, regress `x_1` on `y`, `x_2` on `y` to `x_n`. Store the ***p-value*** and keep the regressor with a *p-value* lower than a defined threshold (0.1 by default). The predictors with a significance lower than the threshold will be added to the final model. If no variable has a *p-value* lower than the entering threshold, then the algorithm stops, and you have your final model with a constant only.
- Step 2: Use the predictor with the lowest *p-value* and adds separately one variable. You regress a constant, the best predictor of step one and a third variable. You add to the stepwise model, the new predictors with a value lower than the entering threshold. If no variable has a *p-value* lower than 0.1, then the algorithm stops, and you have your final model with one predictor only. You regress the stepwise model to check the significance of the step 1 best predictors. If it is higher than the removing threshold, you keep it in the stepwise model. Otherwise, you exclude it.
- Step 3: You replicate step 2 on the new best stepwise model. The algorithm adds predictors to the stepwise model based on the entering values and excludes predictor from the stepwise model if it does not satisfy the excluding threshold.
- The algorithm keeps on going until no variable can be added or excluded.

You can perform the algorithm with the function `ols_stepwise()` from the `olsrr` package.

```
ols_stepwise(fit, pent = 0.1, prem = 0.3, details = FALSE)
arguments:

- fit:  Model to fit. Need to use `lm()`before to run `ols_stepwise()`
- pent: Threshold of the *p-value* used to enter a variable into the stepwise model. By defaut, 0.1
- prem: Threshold of the *p-value* used to exclude a variable into the stepwise model. By defaut, 0.3
- details: Print the details of each step
```

Before that, we show you the steps of the algorithm. Below is a table with the dependent and independente variables:

Dependent variable	Independent variables
mpg	disp
	hp
	drat
	wt
	qsec

Start

To begin with, the algorithm starts by running the model on each independent variables independently . The table shows the *p-value* for each model.

```
## [[1]]
##  (Intercept)         disp
## 3.576586e-21 9.380327e-10
##
## [[2]]
##  (Intercept)           hp
## 6.642736e-18 1.787835e-07
##
## [[3]]
##  (Intercept)         drat
## 0.1796390847 0.0000177624
##
## [[4]]
##  (Intercept)           wt
```

```
## 8.241799e-19 1.293959e-10
##
## [[5]]
## (Intercept)        qsec
##  0.61385436  0.01708199
```

To enter the model, the algorithm keeps the variable with the lowest *p-value*. From the above output, it is `wt`

Step 1

In the first step, the algorithm runs `mpg` on `wt` and the other variables separately

```
## [[1]]
##  (Intercept)           wt         disp
## 4.910746e-16 7.430725e-03 6.361981e-02
##
## [[2]]
##  (Intercept)           wt           hp
## 2.565459e-20 1.119647e-06 1.451229e-03
##
## [[3]]
##  (Intercept)           wt         drat
## 2.737824e-04 1.589075e-06 3.308544e-01
##
## [[4]]
##  (Intercept)           wt         qsec
## 7.650466e-04 2.518948e-11 1.499883e-03
```

Each variable is a potential candidate to enter the final model. However, the algorithm keeps only the variable with the lower *p-value*. It turns out `hp` has a slighlty lower *p-value* than `qsec`. Therefore, `hp` enters the final model

Step 2*

The algorithm repeats the first step but this time with two independent variables in the final model.

```
## [[1]]
##  (Intercept)           wt           hp         disp
## 1.161936e-16 1.330991e-03 1.097103e-02 9.285070e-01
##
## [[2]]
##  (Intercept)           wt           hp         drat
## 5.133678e-05 3.642961e-04 1.178415e-03 1.987554e-01
##
## [[3]]
##  (Intercept)           wt           hp         qsec
## 2.784556e-03 3.217222e-06 2.441762e-01 2.546284e-01
```

None of the variables that entered the final model has a *p-value* sufficiently low. The algorithm stops here, we have the final model:

```
##
## Call:
## lm(formula = mpg ~ wt + hp, data = df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -3.941 -1.600 -0.182  1.050  5.854
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 37.22727    1.59879  23.285  < 2e-16 ***
## wt          -3.87783    0.63273  -6.129 1.12e-06 ***
## hp          -0.03177    0.00903  -3.519  0.00145 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.593 on 29 degrees of freedom
## Multiple R-squared:  0.8268, Adjusted R-squared:  0.8148
## F-statistic: 69.21 on 2 and 29 DF,  p-value: 9.109e-12
```

You can use the function `ols_stepwise()` to compare the results.

```
stp_s <- ols_stepwise(fit, details = TRUE)
```

```
## We are selecting variables based on p value...

## 1 variable(s) added....

## Variable Selection Procedure
##  Dependent Variable: mpg
##
##  Stepwise Selection: Step 1
##
##  Variable wt Entered
##
##                         Model Summary
## ----------------------------------------------------------------
## R                       0.868       RMSE                 3.046
## R-Squared               0.753       Coef. Var           15.161
## Adj. R-Squared          0.745       MSE                  9.277
## Pred R-Squared          0.709       MAE                  2.341
## ----------------------------------------------------------------
##  RMSE: Root Mean Square Error
##  MSE: Mean Square Error
##  MAE: Mean Absolute Error
##
##                                ANOVA
## --------------------------------------------------------------------
##                Sum of
##               Squares        DF    Mean Square      F         Sig.
## --------------------------------------------------------------------
## Regression     847.725         1        847.725    91.375    0.0000
## Residual       278.322        30          9.277
## Total         1126.047        31
## --------------------------------------------------------------------
##
##                                  Parameter Estimates
## ----------------------------------------------------------------------------------------
##       model      Beta    Std. Error    Std. Beta      t        Sig      lower     upper
## ----------------------------------------------------------------------------------------
## (Intercept)    37.285         1.878                 19.858    0.000    33.450    41.120
##          wt    -5.344         0.559       -0.868    -9.559    0.000    -6.486    -4.203
## ----------------------------------------------------------------------------------------

## 1 variable(s) added...
```

```
## Stepwise Selection: Step 2
##
##  Variable hp Entered
##
##                         Model Summary
## ----------------------------------------------------------------
## R                       0.909       RMSE                 2.593
## R-Squared               0.827       Coef. Var           12.909
## Adj. R-Squared          0.815       MSE                  6.726
## Pred R-Squared          0.781       MAE                  1.901
## ----------------------------------------------------------------
##  RMSE: Root Mean Square Error
##  MSE: Mean Square Error
##  MAE: Mean Absolute Error
##
##                                ANOVA
## --------------------------------------------------------------------
##                 Sum of
##                Squares        DF    Mean Square      F         Sig.
## --------------------------------------------------------------------
## Regression     930.999         2        465.500    69.211    0.0000
## Residual       195.048        29          6.726
## Total         1126.047        31
## --------------------------------------------------------------------
##
##                                  Parameter Estimates
## ----------------------------------------------------------------------------------------------
##       model      Beta    Std. Error    Std. Beta      t        Sig      lower     upper
## ----------------------------------------------------------------------------------------------
## (Intercept)    37.227         1.599                 23.285    0.000    33.957    40.497
##          wt    -3.878         0.633       -0.630    -6.129    0.000    -5.172    -2.584
##          hp    -0.032         0.009       -0.361    -3.519    0.001    -0.050    -0.013
## ----------------------------------------------------------------------------------------------

## No more variables to be added or removed.
```

```
stp_s
```

```
## Stepwise Selection Method
##
## Candidate Terms:
##
## 1 . disp
## 2 . hp
## 3 . drat
## 4 . wt
## 5 . qsec
##
## ---------------------------------------------------------------------------------------
##                               Stepwise Selection Summary
## ---------------------------------------------------------------------------------------
##                      Added/                   Adj.
## Step    Variable    Removed     R-Square    R-Square     C(p)        AIC         RMSE
## ---------------------------------------------------------------------------------------
##    1       wt       addition       0.753       0.745    14.5350    166.0294    3.0459
```

```
##    2       hp       addition       0.827       0.815     3.8080    156.6523    2.5934
## ---------------------------------------------------------------------------------------
```

The algorithm founds a solution after 2 steps, and return the same output as we had before.

At the end, you can say the models is explained by two variables and an intercept. Mile per gallon is negatively correlated with Gross horsepower and Weight

5.4.10 Summary

Ordinary least squared regression can be summarized in the table below:

Library	Objective	Function	Arguments
base	Compute a linear regression	lm()	formula, data
base	Summarize model	summarise()	fit
base	Exctract coefficients	lm()$coefficient	
base	Exctract residuals	lm()$residuals	
base	Exctract fitted value	lm()$fitted.values	
olsrr	Run stepwise regression	ols_stepwise()	fit, pent = 0.1, prem = 0.3, details = FALSE

note: Remember to transform caterorical variable in factor before to fit the model.

5.4.11 Decision trees

Decision trees are versatile Machine Learning algorithm that can perform both classification and regression tasks. They are very powerful algorithms, capable of fitting complex datasets. Besides, decision trees are fundamental components of random forests, which are among the most potent Machine Learning algorithms available today.

In this session, you will do learn:

- Training and Visualizing a decision trees for a classification task
- Making prediction
- Regularization Hyper-parameters

5.4.12 Training and Visualizing a decision trees

To build your first decision trees, we will proceed as follow:

- Step 1: Import the data
- Step 2: Clean the dataset
- Step 3: Create train/test set
- Step 4: Build the model
- Step 5: Make prediction
- Step 6: Measure performance
- Step 7: Tune the hyper-parameters

Step 1: Import the data

You load the titanic dataset. If you are curious about the fate of the titanic, you can watch this video on Youtube. The purpose of this dataset is to predict which people are more likely to survive after the collision

with the iceberg. The dataset contains 13 variables and 1309 observations. The dataset is ordered by the variable `X`.

```
set.seed(678)
path <- 'https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/titanic_csv.csv'
titanic <- read.csv(path)
head(titanic)
```

```
##   X pclass survived                                            name    sex
## 1 1      1        1                   Allen, Miss. Elisabeth Walton female
## 2 2      1        1                  Allison, Master. Hudson Trevor   male
## 3 3      1        0                    Allison, Miss. Helen Loraine female
## 4 4      1        0            Allison, Mr. Hudson Joshua Creighton   male
## 5 5      1        0 Allison, Mrs. Hudson J C (Bessie Waldo Daniels) female
## 6 6      1        1                              Anderson, Mr. Harry   male
##       age sibsp parch ticket     fare   cabin embarked
## 1 29.0000     0     0  24160 211.3375      B5        S
## 2  0.9167     1     2 113781 151.5500 C22 C26        S
## 3  2.0000     1     2 113781 151.5500 C22 C26        S
## 4 30.0000     1     2 113781 151.5500 C22 C26        S
## 5 25.0000     1     2 113781 151.5500 C22 C26        S
## 6 48.0000     0     0  19952  26.5500     E12        S
##                         home.dest
## 1                     St Louis, MO
## 2 Montreal, PQ / Chesterville, ON
## 3 Montreal, PQ / Chesterville, ON
## 4 Montreal, PQ / Chesterville, ON
## 5 Montreal, PQ / Chesterville, ON
## 6                     New York, NY
```

```
tail(titanic)
```

```
##         X pclass survived                      name    sex  age sibsp
## 1304 1304      3        0     Yousseff, Mr. Gerious   male   NA     0
## 1305 1305      3        0      Zabour, Miss. Hileni female 14.5     1
## 1306 1306      3        0     Zabour, Miss. Thamine female   NA     1
## 1307 1307      3        0 Zakarian, Mr. Mapriededer   male 26.5     0
## 1308 1308      3        0       Zakarian, Mr. Ortin   male 27.0     0
## 1309 1309      3        0        Zimmerman, Mr. Leo   male 29.0     0
##      parch ticket    fare cabin embarked home.dest
## 1304     0   2627 14.4583              C
## 1305     0   2665 14.4542              C
## 1306     0   2665 14.4542              C
## 1307     0   2656  7.2250              C
## 1308     0   2670  7.2250              C
## 1309     0 315082  7.8750              S
```

From the head and tail output, you can notice the data is not shuffled. This is a big issue! When you will split your data between a train set and test set, you will select **only** the passenger from class 1 and 2 (No passenger from class 3 are in the top 80 percent of the observations), which means the algorithm will never see the features of passenger of class 3. This mistake will lead to poor prediction.

To overcome this issue, you can use the function `sample()`.

```
shuffle_index <- sample(1:nrow(titanic))
head(shuffle_index)
```

```
## [1]  288  874 1078  633  887  992
```

Code Explanation

- `sample(1:nrow(titanic))`: Generate a random list of index from 1 to 1309 (i.e. the maximum number of rows).

You will use this index to shuffle the titanic dataset.

```
titanic <- titanic[shuffle_index, ]
head(titanic)
```

```
##         X pclass survived
## 288   288      1        0
## 874   874      3        0
## 1078 1078      3        1
## 633   633      3        0
## 887   887      3        1
## 992   992      3        1
##                                                      name    sex age
## 288                                    Sutton, Mr. Frederick   male  61
## 874                   Humblen, Mr. Adolf Mathias Nicolai Olsen   male  42
## 1078                                 O'Driscoll, Miss. Bridget female  NA
## 633  Andersson, Mrs. Anders Johan (Alfrida Konstantia Brogren) female  39
## 887                                     Jermyn, Miss. Annie female  NA
## 992                                        Mamee, Mr. Hanna   male  NA
##      sibsp parch ticket    fare cabin embarked           home.dest
## 288      0     0  36963 32.3208   D50        S     Haddenfield, NJ
## 874      0     0 348121  7.6500 F G63        S
## 1078     0     0  14311  7.7500              Q
## 633      1     5 347082 31.2750              S Sweden Winnipeg, MN
## 887      0     0  14313  7.7500              Q
## 992      0     0   2677  7.2292              C
```

Step 2: Clean the data

The structure of the data shows some variables have `NA`'s. Data clean up to be done as follows:

- Drop variables `home.dest`,`cabin`, `name`, `X` and `ticket`
- Create factor variables for `pclass` and `survived`
- Drop the `NA`

```
library(dplyr)
# Drop variables
clean_titanic <- titanic %>%
                 select(-c(home.dest, cabin, name, X, ticket)) %>%
# Convert to factor level
                 mutate(pclass = factor(pclass, levels = c(1,2,3), labels= c('Upper', 'Middle', 'Lower'
                        survived = factor(survived, levels = c(0,1), labels = c('Died', 'Survived'))) %
  na.omit()

glimpse(clean_titanic)
```

```
## Observations: 1,045
## Variables: 8
## $ pclass   <fctr> Upper, Lower, Lower, Upper, Middle, Upper, Middle, U...
## $ survived <fctr> Died, Died, Died, Survived, Died, Survived, Survived...
## $ sex      <fctr> male, male, female, female, male, male, female, male...
```

```
## $ age      <dbl> 61.0, 42.0, 39.0, 49.0, 29.0, 37.0, 20.0, 54.0, 2.0, ...
## $ sibsp    <int> 0, 0, 1, 0, 0, 1, 0, 0, 4, 0, 0, 1, 1, 0, 0, 0, 1, 1,...
## $ parch    <int> 0, 0, 5, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 2, 0, 4, 0,...
## $ fare     <dbl> 32.3208, 7.6500, 31.2750, 25.9292, 10.5000, 52.5542, ...
## $ embarked <fctr> S, S, S, S, S, S, S, S, S, C, S, S, S, Q, C, S, S, C...
```

Before you train your model, you need to add two steps:

- Create a train and test set: You train the model on the train set and test the prediction on the test set (i.e. unseen data)
- Install `rpart.plot` from the console

Step 3: Create train/test set

The common practice is to split the data 80/20, 80 percent of the data serves to train the model, and 20 percent to make predictions. You need to create two separate data frames. You don't want to touch the test set until you finish to build your model. You can create a function name `create_train_test()` that takes three arguments.

```
create_train_test(df, size = 0.8, train = TRUE)
arguments:

- df: Dataset used to train the model.
- size: Size of the split. By default, 0.8. Numerical value
- train: If set to `TRUE`, the function creates the train set, otherwise the test set. Default value se
```

You need to add a Boolean parameter because R does not allow to return two data frames simultaneously.

```
create_train_test <- function(data, size=0.8, train = TRUE){
  n_row = nrow(data)
  total_row = size*n_row
  train_sample <- 1:total_row
  if (train ==TRUE){
    return(data[train_sample, ])
  } else {
    return(data[-train_sample, ])
  }
}
```

Code Explanation

- `function(data, size=0.8, train = TRUE)`: Add the arguments in the function
- `n_row = nrow(data)`: Count number of rows in the dataset
- `total_row = size*n_row`: Return the nth row to construct the train set
- `train_sample <- 1:total_row`: Select the first row to the nth rows
- `if (train ==TRUE){ } else { }`: If condition sets to true, return the train set, else the test set.

You can test your function and check the dimension.

```
data_train <-create_train_test(clean_titanic, 0.8, train = TRUE)
data_test <-create_train_test(clean_titanic, 0.8, train = FALSE)

dim(data_train)
```

```
## [1] 836   8
```

```
dim(data_test)
```

```
## [1] 209   8
```

The train dataset has 1046 rows while the test dataset has 262 rows.

You use the function `prop.table()` combined with `table()` to verify if the randomization process is correct.

```
prop.table(table(data_train$survived))
```

```
##
##      Died  Survived
## 0.5944976 0.4055024
```

```
prop.table(table(data_test$survived))
```

```
##
##      Died  Survived
## 0.5789474 0.4210526
```

In both dataset, the amount of survivors is the same, about 40 percent.

Install `rpart.plot`

`rpart.plot` is not available from `conda` libraries. You can install it from the console.

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

Step 4: Build the model

You are ready to build the model. The syntax for `Rpart()` function is:

```
rpart(formula, data=, method='')
arguments:

- formula: The function to predict
- data: Specifies the data frame
- method:
    - "class" for a classification tree
    - "anova" for a regression tree
```

You use the `class` method because you predict a class.

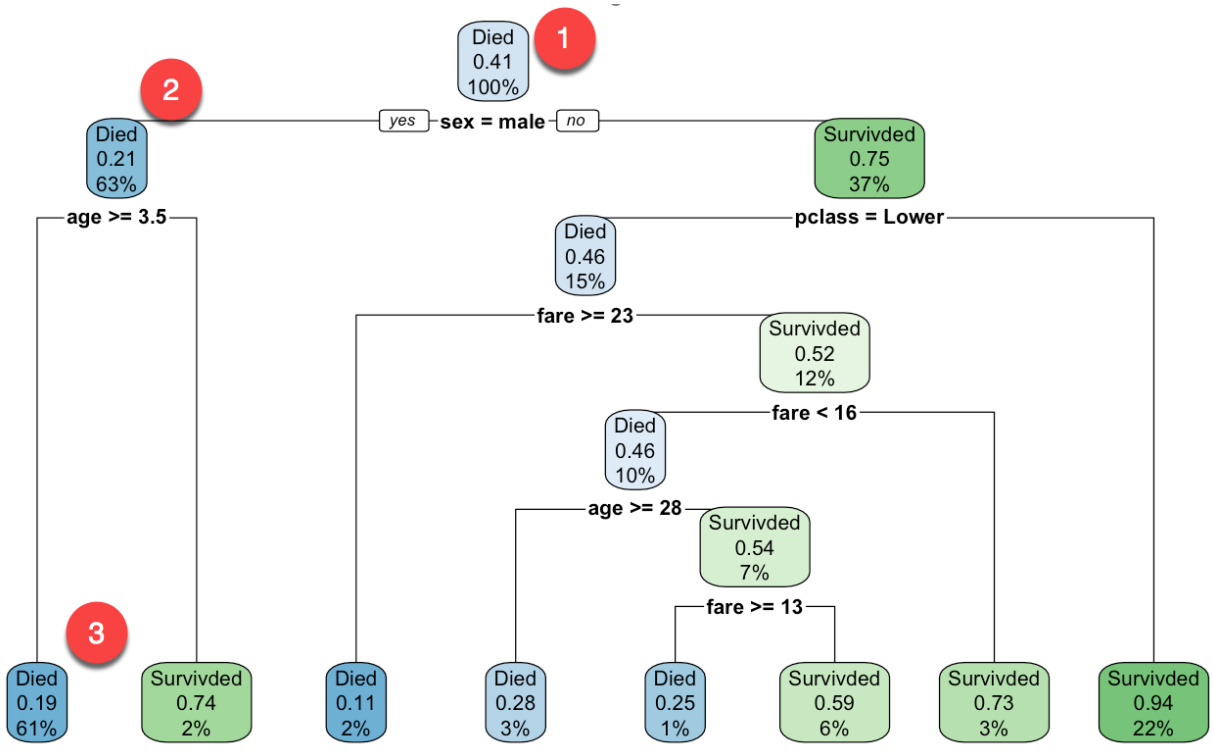
```
library(rpart)
library(rpart.plot)
fit <- rpart(survived ~., data = data_train, method = 'class')
```

Code Explanation

- `rpart()`: Function to fit the model. The arguments are:
  - `survived ~.`: Formula of the Decision Trees
  - `data = data_train`: Dataset
  - `method = 'class'`: Fit a binary model

You can plot the model

```
rpart.plot(fit, extra= 101)
```



Code Explanation

- `rpart.plot(fit, extra= 104)`: Plot the tree. The `extra` features are set to `106` to display the probability of the 2nd class (useful for binary responses). You can refer to the vignette for more information about the other choices.

You start at the *root node* (depth 0 over 3, the top of the graph):

1. At the top, it is the overall probability of survival. It shows the proportion of passenger that survided to the crash. 41 percent of passenger survived.
2. This node asks whether the gender of the passenger is male. If yes, then you go down to the root's left child node (depth 2). 63 percent are males with a survival probability of 21 percent.
3. In the second node, you ask if the male passenger is above 3.5 years old. If yes, then the chance of survival is 19 percent.
4. You keep on going like that to understand what features impact the likelihood of survival.

note: one of the many qualities of Decision Trees is that they require very little data preparation. In particular, they don't require feature scaling or centering.

By default, `rpart()` function uses the **Gini** impurity measure to split the note. The higher the Gini coefficient, the more different instances within the node.

Step 5: Make prediction

You can predict your test dataset. To make a prediction, you can use the `predict()` function. The basic syntax of predict for a decision trees is:

```
predict(fitted_model, df, type = 'class')
arguments:

- fitted_model: This is the object stored after a model estimation.
- df: Data frame used to make the prediction
- type: Type of prediction
    - 'class': for classification
```

```
    - 'prob': to compute the probability of each class
    - 'vector': Predict the mean response at the node level
```

You want to predict which passengers are more likely to survive after the collision from the test set. It means, you will know among those 209 passengers, which one will survive or not.

```
predict_unseen <- predict(fit, data_test, type = 'class')
```

Code Explanation

- `predict(fit, data_test, type = 'class')`: Predict the class (0/1) of the test set

You actually know the passenger that didn't make it and those who did.

```
table_mat <- table(data_test$survived, predict_unseen)
table_mat
```

```
##           predict_unseen
##            Died Survived
##   Died      106       15
##   Survived   30       58
```

Code Explanation

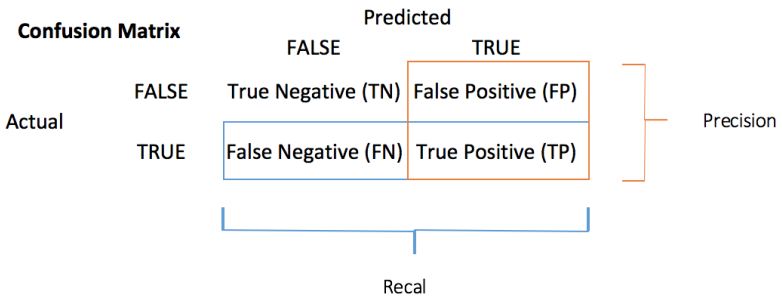
- `table(data_test$survived, predict_unseen)`: Create a $2 * 2$ table to count how many passengers are classified as survivors and passed away compare to the correct classification

The model correctly predicted 106 dead passengers but classified 15 survivors as dead. By analogy, the model misclassified 30 passengers as survivors while they turned out to be dead.

Step 6: Measure performance

You can compute an accuracy measure for classification task with the **confusion matrix**:

The **confusion matrix** is a better choice to evaluate the classification performance. The general idea is to count the number of times `True` instances are classified are `False`.



Each row in a confusion matrix represents an *actual target*, while each column represents a *predicted target*. The first row of this matrix considers dead passengers (the False class): 106 were correctly classified as dead (**True negative**), while the remaining one was wrongly classified as survivor (**False positive**). The second row considers the survivors, the positive class were 58 (**True positive**), while the **True negative** was 30.

You can compute the **accuracy test** from the confusion matrix:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

This is the proportion of true positive and true negative over the sum of the matrix. With R, you can code as follow:

```
accuracy_Test <- sum(diag(table_mat))/sum(table_mat)
```

Code Explanation

- `sum(diag(table_mat))`: Sum of the diagonal
- `sum(table_mat)`: Sum of the matrix.

You can print the accuracy of the test set:

```
print(paste('Accuracy for test', accuracy_Test))
```

```
## [1] "Accuracy for test 0.784688995215311"
```

You have a score of 78 percent for the test set You can replicate the same exercise with the train dataset.

Step 7: Tune the hyper-parameters

Decision tree has various parameters that control aspects of the fit. In `rpart` library, you can control the parameters using the `rpart.control()` function. In the following code, you introduce the parameters you will tune. You can refer to the vignette for other parameters.

```
rpart.control(minsplit = 20, minbucket = round(minsplit/3), maxdepth = 30)
Arguments:

- minsplit: Set the minimum number of observations in the node before the algorithm perform a split
- minbucket:  Set the minimum number of observations in the final note i.e. the leaf
- maxdepth: Set the maximum depth of any node of the final tree. The root node is treated a depth 0
```

We will proceed as follow:

- Construct function to return accuracy
- Tune the maximum depth
- Tune the minimum number of sample a node must have before it can split
- Tune the minimum number of sample a leaf node must have

You can write a function to display the accuracy. You simply wrap the code you used before:

1. predict: `predict_unseen <- predict(fit, data_test, type = 'class')`
2. Produce table: `table_mat <- table(data_test$survived, predict_unseen)`
3. Compute accuracy: `accuracy_Test <- sum(diag(table_mat))/sum(table_mat)`

```
accuracy_tune <- function(fit){
  predict_unseen <- predict(fit, data_test, type = 'class')
  table_mat <- table(data_test$survived, predict_unseen)
  accuracy_Test <- sum(diag(table_mat))/sum(table_mat)
  accuracy_Test
}
```

You can try to tune the parameters and see if you can improve the model over the default value. As a reminder, you need to get an accuracy higher than 0.78

You run the algorithm with the following parameters:

```
minsplit = 4
minbucket= round(5/3)
maxdepth = 3
cp=0
```

```
control <- rpart.control(minsplit = 4,
                         minbucket= round(5/3),
                         maxdepth = 3,
                         cp=0)
```

```
tune_fit <- rpart(survived ~., data = data_train, method = 'class', control = control)
accuracy_tune(tune_fit)
```

```
## [1] 0.7990431
```

You get a higher performance than the previous model. Congratulation!

5.4.13 Summary

We can summarize the functions to train a decision trees algorithm.

Library	Objective	function	class	parameters	details
rpart	Train classification trees	rpart()	class	formula, df, method	
rpart	Train regression tree	rpart()	anova	formula, df, method	
rpart	Plot the trees	rpart.plot()		fitted model	
base	predict	predict()	class	fitted model, type	
base	predict	predict()	prob	fitted model, type	
base	predict	predict()	vector	fitted model, type	
rpart	Control parameters	rpart.control()		minsplit	Set the minimum number of observations in the node before the algorithm perform a split
				minbucket	Set the minimum number of observations in the final note i.e. the leaf
				maxdepth	Set the maximum depth of any node of the final tree. The root node is treated a depth 0
rpart	Train model with control parameter	rpart()		formula, df, method, control	

note: Train the model on a training data and test the performance on an unseen dataset, i.e. test set.

5.4.14 Random Forests

Random forests are based on a simple idea: *'the wisdom of the crowd'*. Aggregate of the results of multiple predictors gives a better prediction than the best individual predictor. A group of predictors is called an **ensemble**. Thus, this technique is called **Ensemble Learning**.

In earlier tutorial, you learned how to use **Decision trees** to make a binary prediction. To improve our technique, we can train a group of **Decision Tree classifiers**, each on a different random subset of the train set. To make a prediction, we just obtain the predictions of all individuals trees, then predict the class that gets the most votes. This technique is called **Random Forest**.

We will proceed as follow to train the Random Forest:

- Step 1: Import the data
- Step 2: Train the model
- Step 3: Construct accuracy function
- Step 4: Visualize the model

5.4.15 Step 1: Import the data

To make sure you have the same dataset as in the tutorial for decision trees, the train test and test set are stored on the internet. You can import them without make any change.

```
library(dplyr)
data_train <- read.csv("https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/titanic_t
  select(-1)
data_test <- read.csv("https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/titanic_te
  select(-1)
```

5.4.16 Step 2: Train the model

One way to evaluate the performance of a model is to train it on a number of different smaller datasets and evaluate them over the other smaller testing set. This is called the **F-fold cross-validation** feature. R has a function to randomly split k number of datasets of almost the same size. For example, if $k = 9$, the model is evaluated over the nine folder and tested on the remaining test set. This process is repeated until all the subsets have been evaluated. This technique is widely used for model selection, especially when the model has parameters to tune.

Now that we have a way to evaluate our model, we need to figure out how to choose the parameters that generalized best the data.

Random forest chooses a random subset of features and builds many Decision Trees. The model averages out all the predictions of the Decisions trees.

Random forest has some parameters that can be changed to improved the generalization of the prediction. You will use the function `RandomForest()` to train the model.

Syntax for Random Forest is:

```
RandomForest(formula, ntree=n, mtry=FALSE, maxnodes = NULL)
Arguments:

- Formula: Formula of the fitted model
- ntree: number of trees in the forest
- mtry: Number of candidates draw to feed the algorithm. By default, it is the square of the number of
- maxnodes: Set the maximum amount of terminal nodes in the forest
-   importance=TRUE: Whether independent variables importance in the random forest be assessed
```

note: Random forest can be trained on more parameters. You can refer to the vignette to see the different parameters.

Tuning a model is very tedious work. There are lot of combination possible between the parameters. You don't necessarily have the time to try all of them. A good alternative is to let the machine find the best combination for you. There are two methods available:

- Random Search
- Grid Search

We will define both methods but during the tutorial, we will train the model using grid search

5.4.17 Grid Search

The grid search method is simple, the model will be evaluated over all the combination you pass in the function, using cross-validation.

For instance, you want to try the model with 10, 20, 30 number of trees and each tree will be tested over a number of `mtry` equals to 1, 2, 3, 4, 5. Then the machine will test 15 different models:

```
##    .mtry ntrees
## 1      1     10
## 2      2     10
## 3      3     10
## 4      4     10
## 5      5     10
## 6      1     20
## 7      2     20
## 8      3     20
## 9      4     20
## 10     5     20
## 11     1     30
## 12     2     30
## 13     3     30
## 14     4     30
## 15     5     30
```

The algorithm will evaluate:

```
RandomForest(formula, ntree=10, mtry=1)
RandomForest(formula, ntree=10, mtry=2)
RandomForest(formula, ntree=10, mtry=3)
RandomForest(formula, ntree=20, mtry=2)
and so on
```

Each time, the random forest experiments with a cross-validation. One shortcoming of the grid search is the number of experimentations. It can become very easily explosive when the number of combination is high. To overcome this issue, you can use the random search

5.4.18 Random Search definition

The big difference between random search and grid search is, random search will not evaluate all the combination of hyperparameter in the searching space. Instead, it will randomly choose combination at every iteration. The advantage is it lower the computational cost.

5.4.19 Set the control parameter

You will proceed as follow to construct and evaluate the model:

- Evaluate the model with the default setting
- Find the best number of `mtry`
- Find the best number of `maxnodes`
- Find the best number of `ntrees`
- Evaluate the model on the test dataset

Before you begin with the parameters exploration, you need to install two libraries.

- `caret`: R machine learning library. If you have install R with `r-essential`. It is already in the library

  - Anaconda: `conda install -c r r-caret`
- `e1071`: R machine learning library.
  - Anaconda: `conda install -c r r-e1071`

You can import them along with `RandomForest`

```
library(randomForest)
library(caret)
library(e1071)
```

Default setting

K-fold cross validation is controlled by the `trainControl()` function

```
trainControl(method = "cv", number = n, search ="grid")
arguments

- method = "cv": The method used to resample the dataset.
- number = n: Number of folders to create
- search = "grid": Use the search grid method. For randomized method, use "grid"

Note: You can refer to the vignette to see the other arguments of the function.
```

You can try to run the model with the default parameters and see the accuracy score.

note: You will use the same controls during all the tutorial.

```
# Define the control
trControl <- trainControl(method="cv",
                          number=10,
                          search="grid")
```

You will use `caret` library to evaluate your model. The library has one function called `train()` to evaluate almost all machine learning algorithm. Say differently, you can use this function to train other algorithms.

The basic syntax is:

```
train(formula, df, method = "rf", metric= "Accuracy", trControl = trainControl(), tuneGrid = NULL)
argument

- `formula`: Define the formula of the algorithm
- `method`: Define which model to train. Note, at the end of the tutorial, there is a list of all the m
- `metric` = "Accuracy": Define how to select the optimal model
- `trControl = trainControl()`: Define the control parameters
- `tuneGrid = NULL`: Return a data frame with all the possible combination
```

Let's try the build the model with the default values.

```
set.seed(1234)
# Run the model
rf_default <- train(survived ~.,
                    data=data_train,
                    method="rf",
                    metric="Accuracy",
                    trControl=trControl)
# Print the results
print(rf_default)
```

```
## Random Forest
##
```

```
## 836 samples
##   7 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 753, 752, 753, 752, 752, 752, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.7919248  0.5536486
##    6    0.7811245  0.5391611
##   10    0.7572002  0.4939620
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 2.
```

Code Explanation

- `trainControl(method="cv", number=10, search="grid")`: Evaluate the model with a grid search of 10 folder
- `train(...)`: Train a random forest model. Best model is chosen with the accuracy measure.

The algorithm uses 500 trees and tested three different values of `mtry`: 2, 6, 10.

The final value used for the model was `mtry = 2` with an accuracy of 0.78. Let's try to get a higher score.

Step 2: Search best `mtry`

You can test the model with values of `mtry` from 1 to 10

```
set.seed(1234)
tuneGrid <- expand.grid(.mtry=c(1:10))
rf_mtry <- train(survived ~.,
                    data=data_train,
                    method="rf",
                    metric="Accuracy",
                    tuneGrid=tuneGrid,
                    trControl=trControl,
                    importance = TRUE,
                    nodesize = 14,
                    ntree = 300)
print(rf_mtry)
```

```
## Random Forest
##
## 836 samples
##   7 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 753, 752, 753, 752, 752, 752, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    1    0.7572576  0.4647368
##    2    0.7979346  0.5662364
```

```
##    3    0.8075158  0.5884815
##    4    0.8110729  0.5970664
##    5    0.8074727  0.5900030
##    6    0.8099111  0.5949342
##    7    0.8050918  0.5866415
##    8    0.8050918  0.5855399
##    9    0.8050631  0.5855035
##   10    0.7978916  0.5707336
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 4.
```

Code Explanation

- `tuneGrid <- expand.grid(.mtry=c(3:10))`: Construct a vector with value from 3:10

The final value used for the model was 'mtry = 4.

The best value of `mtry` is stored in:

`rf_mtry$bestTune$mtry`

You can store it and use it when you need to tune the other parameters.

```
max(rf_mtry$results$Accuracy)
```

```
## [1] 0.8110729
```

```
best_mtry <- rf_mtry$bestTune$mtry
best_mtry
```

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

Step 3: Search the best `maxnodes`

You need to create a loop to evaluate the different values of `maxnodes`. In the following code, you will:

- Create a list
- Create a variable with the best value of the parameter `mtry`; Compulsory
- Create the loop
- Store the current value of `maxnode`
- Summarize the results

```
store_maxnode <- list()
tuneGrid <- expand.grid(.mtry=best_mtry)
for (maxnodes in c(5:15)) {
  set.seed(1234)
  rf_maxnode <- train(survived ~.,
                      data=data_train,
                      method="rf",
                      metric="Accuracy",
                      tuneGrid=tuneGrid,
                      trControl=trControl,
                      importance = TRUE,
                      nodesize = 14,
                      maxnodes=maxnodes,
                      ntree = 300)
  current_iteration <- toString(maxnodes)
  store_maxnode[[current_iteration]] <- rf_maxnode
}
```

```
results_mtry <- resamples(store_maxnode)
summary(results_mtry)
```

```
##
## Call:
## summary.resamples(object = results_mtry)
##
## Models: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
## Number of resamples: 10
##
## Accuracy
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 5  0.6785714 0.7529762 0.7903758 0.7799771 0.8168388 0.8433735    0
## 6  0.6904762 0.7648810 0.7784710 0.7811962 0.8125000 0.8313253    0
## 7  0.6904762 0.7619048 0.7738095 0.7788009 0.8102410 0.8333333    0
## 8  0.6904762 0.7627295 0.7844234 0.7847820 0.8184524 0.8433735    0
## 9  0.7261905 0.7747418 0.8083764 0.7955250 0.8258749 0.8333333    0
## 10 0.6904762 0.7837780 0.7904475 0.7895869 0.8214286 0.8433735    0
## 11 0.7023810 0.7791523 0.8024240 0.7943775 0.8184524 0.8433735    0
## 12 0.7380952 0.7910929 0.8144005 0.8051205 0.8288511 0.8452381    0
## 13 0.7142857 0.8005952 0.8192771 0.8075158 0.8403614 0.8452381    0
## 14 0.7380952 0.7941050 0.8203528 0.8098967 0.8403614 0.8452381    0
## 15 0.7142857 0.8000215 0.8203528 0.8075301 0.8378873 0.8554217    0
##
## Kappa
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 5  0.3297872 0.4640436 0.5459706 0.5270773 0.6068751 0.6717371    0
## 6  0.3576471 0.4981484 0.5248805 0.5366310 0.6031287 0.6480921    0
## 7  0.3576471 0.4927448 0.5192771 0.5297159 0.5996437 0.6508314    0
## 8  0.3576471 0.4848320 0.5408159 0.5427127 0.6200253 0.6717371    0
## 9  0.4236277 0.5074421 0.5859472 0.5601687 0.6228626 0.6480921    0
## 10 0.3576471 0.5255698 0.5527057 0.5497490 0.6204819 0.6717371    0
## 11 0.3794326 0.5235007 0.5783191 0.5600467 0.6126720 0.6717371    0
## 12 0.4460432 0.5480930 0.5999072 0.5808134 0.6296780 0.6717371    0
## 13 0.4014252 0.5725752 0.6087279 0.5875305 0.6576219 0.6678832    0
## 14 0.4460432 0.5585005 0.6117973 0.5911995 0.6590982 0.6717371    0
## 15 0.4014252 0.5689401 0.6117973 0.5867010 0.6507194 0.6955990    0
```

Code explanation:

- `store_maxnode <- list()`: The results of the model will be stored in this list
- `expand.grid(.mtry=best_mtry)`: Use the best value of `mtry`
- `for (maxnodes in c(15:25)) { ... }`: Compute the model with values of `maxnodes` starting from 15 to 25.
- `maxnodes=maxnodes`: For each iteration, `maxnodes` is equal to the current value of `maxnodes`. i.e 15, 16, 17, ...
- `key <- toString(maxnodes)`: Store as a string variable the value of `maxnode`.
- `store_maxnode[[key]] <- rf_maxnode`: Save the result of the model in the list.
- `resamples(store_maxnode)`: Arrange the results of the model
- `summary(results_mtry)`: Print the summary of all the combination.

The last value of `maxnode` has the highest accuracy. You can try with higher values to see if you can get a higher score.

```
store_maxnode <- list()
tuneGrid <- expand.grid(.mtry=best_mtry)
for (maxnodes in c(20:30)) {
  set.seed(1234)
  rf_maxnode <- train(survived ~.,
                      data=data_train,
                      method="rf",
                      metric="Accuracy",
                      tuneGrid=tuneGrid,
                      trControl=trControl,
                      importance = TRUE,
                      nodesize = 14,
                      maxnodes=maxnodes,
                      ntree = 300)
  key <- toString(maxnodes)
  store_maxnode[[key]] <- rf_maxnode
}
results_node <- resamples(store_maxnode)
summary(results_node)
```

```
##
## Call:
## summary.resamples(object = results_node)
##
## Models: 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
## Number of resamples: 10
##
## Accuracy
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 20 0.7142857 0.7821644 0.8144005 0.8075301 0.8447719 0.8571429    0
## 21 0.7142857 0.8000215 0.8144005 0.8075014 0.8403614 0.8571429    0
## 22 0.7023810 0.7941050 0.8263769 0.8099254 0.8328313 0.8690476    0
## 23 0.7023810 0.7941050 0.8263769 0.8111302 0.8447719 0.8571429    0
## 24 0.7142857 0.7946429 0.8313253 0.8135112 0.8417599 0.8690476    0
## 25 0.7142857 0.7916667 0.8313253 0.8099398 0.8408635 0.8690476    0
## 26 0.7142857 0.7941050 0.8203528 0.8123207 0.8528758 0.8571429    0
## 27 0.7023810 0.8060456 0.8313253 0.8135112 0.8333333 0.8690476    0
## 28 0.7261905 0.7941050 0.8203528 0.8111015 0.8328313 0.8690476    0
## 29 0.7142857 0.7910929 0.8313253 0.8087063 0.8333333 0.8571429    0
## 30 0.6785714 0.7910929 0.8263769 0.8063253 0.8403614 0.8690476    0
##
## Kappa
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 20 0.3956835 0.5316120 0.5961830 0.5854366 0.6661120 0.6955990    0
## 21 0.3956835 0.5699332 0.5960343 0.5853247 0.6590982 0.6919315    0
## 22 0.3735084 0.5560661 0.6221836 0.5914492 0.6422128 0.7189781    0
## 23 0.3735084 0.5594228 0.6228827 0.5939786 0.6657372 0.6955990    0
## 24 0.3956835 0.5600352 0.6337821 0.5992188 0.6604703 0.7189781    0
## 25 0.3956835 0.5530760 0.6354875 0.5912239 0.6554912 0.7189781    0
## 26 0.3956835 0.5589331 0.6136074 0.5969142 0.6822128 0.6955990    0
## 27 0.3735084 0.5852459 0.6368425 0.5998148 0.6426088 0.7189781    0
## 28 0.4290780 0.5589331 0.6154905 0.5946859 0.6356141 0.7189781    0
## 29 0.4070588 0.5534173 0.6337821 0.5901173 0.6423101 0.6919315    0
## 30 0.3297872 0.5534173 0.6202632 0.5843432 0.6590982 0.7189781    0
```

The highest accuracy score is obtained with a value of `maxnode` equals to 22.

Step 4: Search the best `ntrees`

Now that you have the best value of `mtry` and `maxnode`, you can tune the number of trees. The method is exactly the same as `maxnode`.

```
store_maxtrees <- list()
for (ntree in c(250, 300, 350, 400, 450, 500, 550, 600, 800, 1000, 2000)){
    set.seed(5678)
    rf_maxtrees <- train(survived ~.,
                         data=data_train,
                         method="rf",
                         metric="Accuracy",
                         tuneGrid=tuneGrid,
                         trControl=trControl,
                         importance = TRUE,
                         nodesize = 14,
                         maxnodes=24,
                         ntree = ntree)
    key <- toString(ntree)
    store_maxtrees[[key]] <- rf_maxtrees
}
results_tree <- resamples(store_maxtrees)
summary(results_tree)
```

```
##
## Call:
## summary.resamples(object = results_tree)
##
## Models: 250, 300, 350, 400, 450, 500, 550, 600, 800, 1000, 2000
## Number of resamples: 10
##
## Accuracy
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.7380952 0.7976190 0.8083764 0.8087010 0.8292683 0.8674699    0
## 300  0.7500000 0.7886905 0.8024240 0.8027199 0.8203397 0.8452381    0
## 350  0.7500000 0.7886905 0.8024240 0.8027056 0.8277623 0.8452381    0
## 400  0.7500000 0.7886905 0.8083764 0.8051009 0.8292683 0.8452381    0
## 450  0.7500000 0.7886905 0.8024240 0.8039104 0.8292683 0.8452381    0
## 500  0.7619048 0.7886905 0.8024240 0.8062914 0.8292683 0.8571429    0
## 550  0.7619048 0.7886905 0.8083764 0.8099062 0.8323171 0.8571429    0
## 600  0.7619048 0.7886905 0.8083764 0.8099205 0.8323171 0.8674699    0
## 800  0.7619048 0.7976190 0.8083764 0.8110820 0.8292683 0.8674699    0
## 1000 0.7619048 0.7976190 0.8121510 0.8086723 0.8303571 0.8452381    0
## 2000 0.7619048 0.7886905 0.8121510 0.8086723 0.8333333 0.8452381    0
##
## Kappa
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.4061697 0.5667400 0.5836013 0.5856103 0.6335363 0.7196807    0
## 300  0.4302326 0.5449376 0.5780349 0.5723307 0.6130767 0.6710843    0
## 350  0.4302326 0.5449376 0.5780349 0.5723185 0.6291592 0.6710843    0
## 400  0.4302326 0.5482030 0.5836013 0.5774782 0.6335363 0.6710843    0
## 450  0.4302326 0.5449376 0.5780349 0.5750587 0.6335363 0.6710843    0
## 500  0.4601542 0.5449376 0.5780349 0.5804340 0.6335363 0.6949153    0
## 550  0.4601542 0.5482030 0.5857118 0.5884507 0.6396872 0.6949153    0
```

```
## 600  0.4601542 0.5482030 0.5857118 0.5884374 0.6396872 0.7196807    0
## 800  0.4601542 0.5667400 0.5836013 0.5910088 0.6335363 0.7196807    0
## 1000 0.4601542 0.5667400 0.5961590 0.5857446 0.6343666 0.6678832    0
## 2000 0.4601542 0.5482030 0.5961590 0.5862151 0.6440678 0.6656337    0
```

You have your final model. You can train the random forest with the following parameters:

- `ntree =800`: 800 trees will be trained
- `mtry=4`: 4 features is chosen for each iteration
- `maxnodes = 24`: Maximum 24 nodes in the terminal nodes (leaves)

```
fit_rf <- train(survived ~.,
                data_train,
                method="rf",
                metric="Accuracy",
                tuneGrid=tuneGrid,
                trControl=trControl,
                importance = TRUE,
                nodesize = 14,
                ntree =800,
                maxnodes=24)
```

Step 5: Evaluate the model

The library `caret` has a function to make prediction.

```
predict(model, newdata= df)
argument

- `model`: Define the model evaluated before.
- `newdata`: Define the dataset to make prediction
```

```
prediction <-predict(fit_rf, data_test)
```

You can use the prediction to compute the confusion matrix and see the accuracy score

```
confusionMatrix(prediction, data_test$survived)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  110  32
##        Yes  11  56
##
##                Accuracy : 0.7943
##                  95% CI : (0.733, 0.8469)
##     No Information Rate : 0.5789
##     P-Value [Acc > NIR] : 3.959e-11
##
##                   Kappa : 0.5638
##  Mcnemar's Test P-Value : 0.002289
##
##             Sensitivity : 0.9091
##             Specificity : 0.6364
##          Pos Pred Value : 0.7746
##          Neg Pred Value : 0.8358
##              Prevalence : 0.5789
```

```
##          Detection Rate : 0.5263
##    Detection Prevalence : 0.6794
##       Balanced Accuracy : 0.7727
##
##        'Positive' Class : No
##
```

You have an accuracy of 0.7943 percent, which is higher than the default value.

Lastly, you can look at the feature importance with the function `varImp()`. It seems that the most important features are the `sex` and `age`. That is not surprising because the important features are likely to appear closer to the root of the tree, while les important features will often appear closed to the leaves.

```
varImp(fit_rf)
```

```
## rf variable importance
##
##              Importance
## sexmale         100.000
## age              28.014
## pclassMiddle     27.016
## fare             21.557
## pclassUpper      16.324
## sibsp            11.246
## parch             5.522
## embarkedC         4.908
## embarkedQ         1.420
## embarkedS         0.000
```

5.4.20 Summary

We can summarize how to train and evaluate a random forest with the table below:

Library	Objective	function	parameter
randomForest	Create a Random forest	RandomForest()	formula, ntree=n, mtry=FALSE, maxnodes = NULL
caret	Create K folder cross validation	trainControl()	method = "cv", number = n, search ="grid"
caret	Train a Random Forest	train()	formula, df, method = "rf", metric= "Accuracy", trControl = trainControl(), tuneGrid = NULL
caret	Predict out of sample	predict	model, newdata= df
caret	Confusion Matrix and Statistics	confusionMatrix()	model, y test
caret	variable importance	cvarImp()	model

5.4.21 Appendix

List of model used in `caret`. Only the first tenth are returned.

```
names(getModelInfo())[1:10]
```

```
##  [1] "ada"         "AdaBag"      "AdaBoost.M1" "adaboost"    "amdai"
##  [6] "ANFIS"       "avNNet"      "awnb"        "awtan"       "bag"
```

5.5 Classification

5.5.1 Logistic regression

This tutorial introduces how to perform a logistic regression. Logistic regression is used to predict a class, i.e. a probability.

Logistic regression has a convenient mathematical property and performs well to predict a binary outcome.

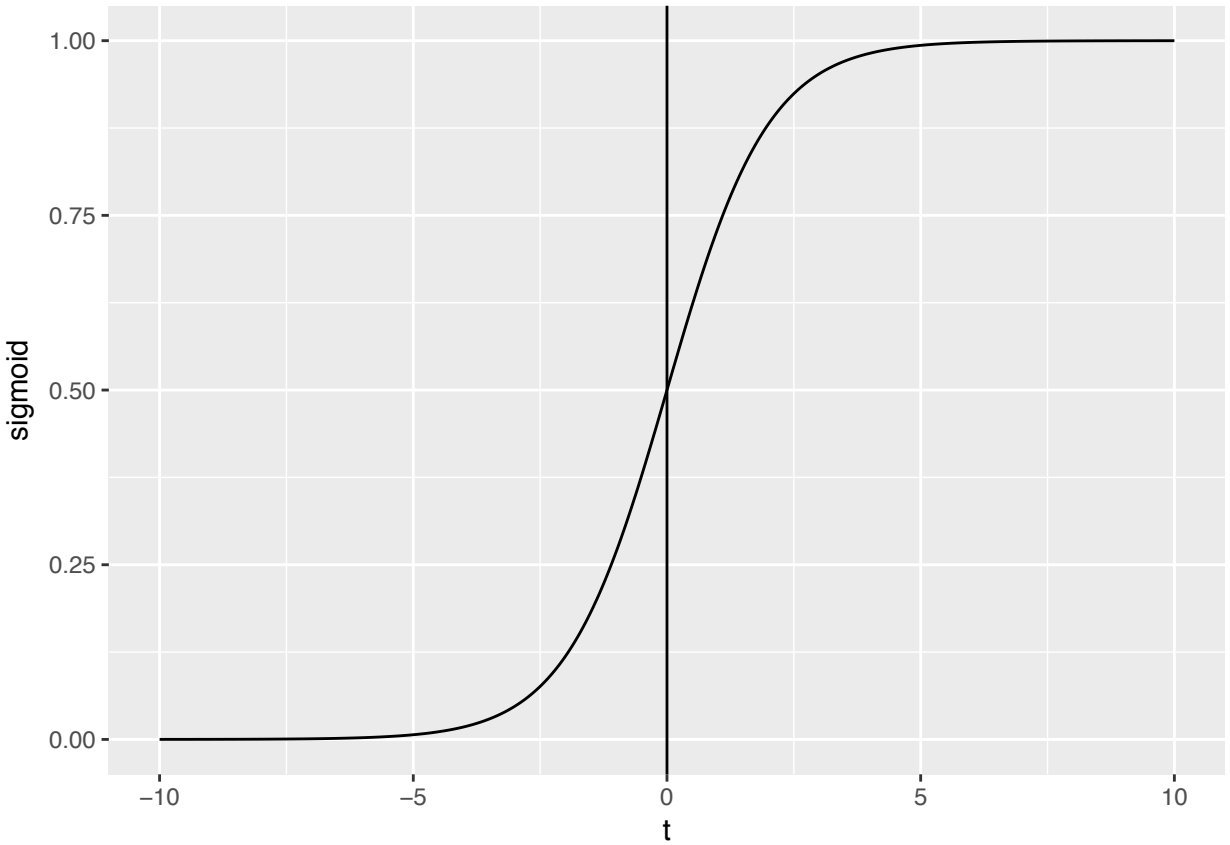
Imagine, you want to predict whether a loan is denied/accepted based on many attributes. The logistic regression is of the form 0/1. $Y = 0$ if a loan is rejected, $Y = 1$ if accepted.

A logistic regression model differs from linear regression model in two ways.

- First of all, the logistic regression accepts only dichotomous (binary) input as a dependent variable (i.e. a vector of 0 and 1).
- Secondly, the outcome is measured by the following probabilistic link function called **sigmoid** due to its S-shaped.:

$\sigma(t) = \frac{1}{1+exp(-t)}$

The output of the function is always between 0 and 1. Check image below



The sigmoid function returns values from 0 to 1. For the classification task, we need a discrete output of 0 or 1.

To convert a continuous flow into discrete value, we can set a decision bound at 0.5. All values above this threshold are classified as 1

$$\hat{y} = \begin{cases} 0 & if \quad \hat{p} < .5 \\ 1 & if \quad \hat{p} \geqslant .5 \end{cases}$$

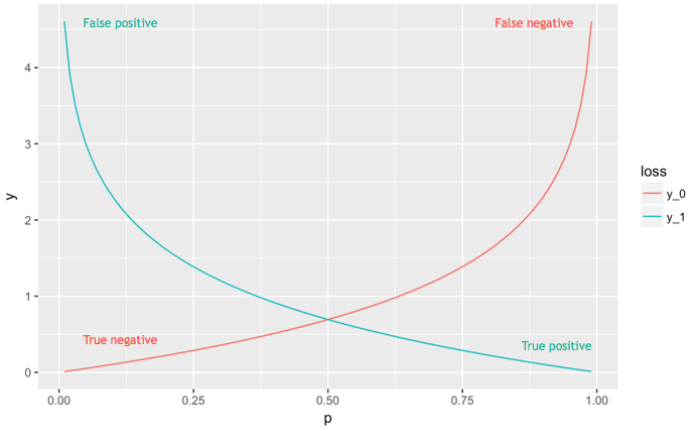
5.5.2 Train the model

To train a linear regression model, we used the **Mean Squared Error (MSE)** cost function. The MSE is easy to compute but does not fit non-linearity. The S shape of the sigmoid function prevents us from finding a single local minimum, thus a single solution.

Instead, the logistic regression takes advantage of the **Log loss** function to fit two separate functions, one for positive instances ($y = 1$) and one for negative instances ($y = 0$).

$$c(\theta) = \begin{cases} -log(p) & if \quad p < .5 \\ -log(1-p) & if \quad p > .0 \end{cases}$$

We can plot the functions and see the benefit of using the log in the cost function.



The cost function **penalizes** incorrect predictions more than correct predictions. Indeed, values close to 0 for $-log(p)$, grows vertically (green line) and values close to 1 for $-log(1-p)$ tends to infinity (red line). It means, the cost tends to be high for wrong predictions (false positive and false negative) and low for correct predictions (true positive and true negative). If the model predicts a p value close to 1, then the loss function $-log(p)$ tends to 0 for positive instances and be substantially large for the function $-log(1-p)$. The cost function of the training set averages out all the training instances. You can see the advantage of using the log, both functions increase or decrease monotonically. It makes it possible to compute the gradient and minimize the cost function.

$$J(\theta) = -\frac{1}{m}\sum[y^{(i)}log(p^{(i)}) + (1-y^{(i)})log(1-p^{(i)})]$$

5.5.3 Example Logistic Regression

Let's use the **adult** data set to illustrate Logistic regression. Adult is a famous dataset for classification task. The objective is to predict whether the annual income in dollar of an individual will exceed 50.000. The dataset contains 46,033 observations and 10 features:

- `age`: age of the individual. Numeric
- `education`: Educational level of the individual. Factor i.e. 11th, HS-grad, ...
- `marital.status`: Marital status of the individual. Factor i.e. Never-married, Married-civ-spouse, ...
- `gender`: Gender of the individual. Factor i.e. Male or Female
- `income`: Target variable. Income above or below 50K. Factor i.e. >50K, <=50K

amongst others

```
library(dplyr)
data_adult <- read.csv("https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/data_adul
glimpse(data_adult)
```

```
## Observations: 46,033
## Variables: 10
## $ X              <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,...
## $ age            <int> 25, 38, 28, 44, 34, 63, 24, 55, 65, 36, 26, 48...
## $ workclass      <fctr> Private, Private, Local-gov, Private, Private...
## $ education      <fctr> 11th, HS-grad, Assoc-acdm, Some-college, 10th...
## $ educational.num <int> 7, 9, 12, 10, 6, 15, 10, 4, 9, 13, 9, 9, 14, 1...
## $ marital.status  <fctr> Never-married, Married-civ-spouse, Married-ci...
## $ race           <fctr> Black, White, White, Black, White, White, Whi...
## $ gender         <fctr> Male, Male, Male, Male, Male, Male, Female, M...
## $ hours.per.week  <int> 40, 50, 40, 40, 30, 32, 40, 10, 40, 40, 39, 48...
## $ income         <fctr> <=50K, <=50K, >50K, >50K, <=50K, >50K, <=50K,...
```

We will proceed as follow:

- Step 1: Check continuous variables
- Step 2: Check factor variables
- Step 3: Feature engineering
- Step 4: Summary statistic
- Step 5: Train/test set
- Step 6: Build the model
- Step 7: Assess the performance of the model
- step 8: Improve the model

Your task is to predict which individual will have a revenue higher than 50K.

In this tutorial, each step will be detailed to perform an analysis on a real dataset.

Step 1: Check continuous variables

In the first step, you can see the distribution of the continuous variables.

```
continuous <- select_if(data_adult, is.numeric)
summary(continuous)
```

```
##        X              age        educational.num hours.per.week
##  Min.   :    1   Min.   :17.00   Min.   : 1.00   Min.   : 1.00
##  1st Qu.:11509   1st Qu.:28.00   1st Qu.: 9.00   1st Qu.:40.00
##  Median :23017   Median :37.00   Median :10.00   Median :40.00
##  Mean   :23017   Mean   :38.56   Mean   :10.13   Mean   :40.95
##  3rd Qu.:34525   3rd Qu.:47.00   3rd Qu.:13.00   3rd Qu.:45.00
##  Max.   :46033   Max.   :90.00   Max.   :16.00   Max.   :99.00
```

Code Explanation

- `continuous <- select_if(data_adult, is.numeric)`: Use the function `select_if()` from the `dplyr` library to select only the numerical columns
- `summary(continuous)`: Print the summary statistic

From the above table, you can see that the data have totally different scales and `hours.per.weeks` has large outliers (.i.e. look at the last quartile and maximum value).

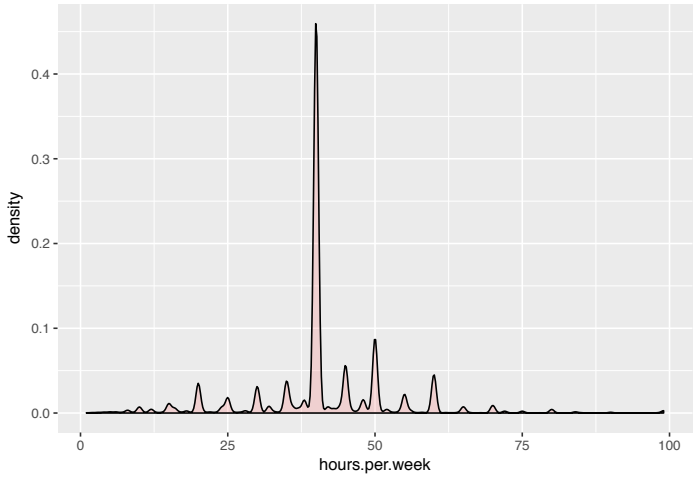
You can deal with it following two steps:

- Step 1: Plot the distribution of `hours.per.week`
- Step 2: Standardize the continuous variables

1. Plot the distribution

Let's look closer at the distribution of `hours.per.week`

```
# Histogram overlaid with kernel density curve
ggplot(continuous, aes(x=hours.per.week)) +
    geom_density(alpha=.2, fill="#FF6666")
```



The variable has lots of outliers and not well-defined distribution. You can partially tackle this problem by deleting the top 0.01 percent of the hours per week.

Basic syntax of quantile:

```
quantile(variable, percentile)
arguments:

variable:  Select the variable in the data frame to compute the percentile
percentile:  Can be a single value between 0 and 1 or multiple value. If multiple, use this format:  `c
   - `A`,`B`,`C` and `...` are all integer from 0 to 1.
```

We compute the top 2 percent percentile

```
top_one_percent <- quantile(data_adult$hours.per.week, .99)
top_one_percent
```

```
## 99%
##  80
```

Code Explanation

- `quantile(data_adult$hours.per.week, .99)`: Compute the value of the 99 percent of the working time

98 percent of the population works under 80 hours per week.

You can drop the observations above this threshold. You use the `filter` from the `dplyr` library.

```
data_adult_drop <- data_adult %>%
                  filter(hours.per.week <top_one_percent)
dim(data_adult_drop)
```

```
## [1] 45537    10
```

- 2) Standardize the continuous variables

You can standardize each column to improve the performance because your data do not have the same scale. You can use the function `mutate_if` from the `dplyr` library. The basic syntax is:

```
mutate_if(df, condition, funs(function))
arguments:

-`df`: Data frame used to compute the function
- `condition`: Statement used. Do not use parenthesis
- funs(function):  Return the function to apply. Do not use parenthesis for the function
```

You can standardize the numeric columns as follow:

```
data_adult_rescale <- data_adult_drop %>%
  mutate_if(is.numeric, funs(as.numeric(scale(.))))
head(data_adult_rescale)
```

```
##           X         age        workclass    education educational.num
## 1 -1.732680 -1.02325949          Private         11th     -1.22106443
## 2 -1.732605 -0.03969284          Private      HS-grad     -0.43998868
## 3 -1.732530 -0.79628257        Local-gov   Assoc-acdm      0.73162494
## 4 -1.732455  0.41426100          Private Some-college     -0.04945081
## 5 -1.732379 -0.34232873          Private         10th     -1.61160231
## 6 -1.732304  1.85178149 Self-emp-not-inc  Prof-school      1.90323857
##       marital.status  race gender hours.per.week income
## 1      Never-married Black   Male    -0.03995944  <=50K
## 2 Married-civ-spouse White   Male     0.86863037  <=50K
## 3 Married-civ-spouse White   Male    -0.03995944   >50K
## 4 Married-civ-spouse Black   Male    -0.03995944   >50K
## 5      Never-married White   Male    -0.94854924  <=50K
## 6 Married-civ-spouse White   Male    -0.76683128   >50K
```

Code Explanation

- `mutate_if(is.numeric, funs(scale))`: The condition is only numeric column and the function is `scale`

Step 2: Check factor variables

This step has two objectives:

- Check the level in each categorical column
- Define new levels

We will divide this step in three part:

1. Select the categorical columns
2. Store the bar chart of each column in a list
3. Print the graphs

We can select the factor columns with the code below:

```
# Select categorical column
factor <- data.frame(select_if(data_adult_rescale, is.factor))
ncol(factor)
```

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

Code Explanation

- `data.frame(select_if(data_adult, is.factor))`: We store the factor columns in `factor` in a data frame type. The library `ggplot2` requires a data frame object.

The dataset contains 6 categorical variables

The second step is more skilled. You want to plot a bar chart for each column in the data frame `factor`. It is more convenient to automatize the process, especially in situation there are lots of columns.

```
library(ggplot2)
# Create graph for each column
graph <- lapply(names(factor),
                function(x)
                  ggplot(factor, aes(get(x))) +
                         geom_bar()+
                         theme(axis.text.x = element_text(angle = 90)))
```

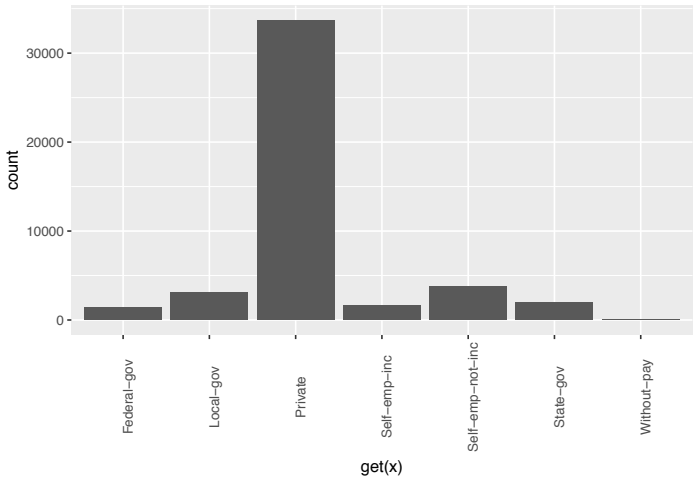
Code Explanation

- `lapply()`: Use the function `lapply()` to pass a function in all the columns of the dataset. You store the output in a list
- `function(x)`: The function will be processed for each x. Here `x` is the columns
- `ggplot(factor, aes(get(x))) + geom_bar()+ theme(axis.text.x = element_text(angle = 90))`: Create a bar char chart for each `x` element. Note, to return `x` as a column, you need to include it inside the `get()`

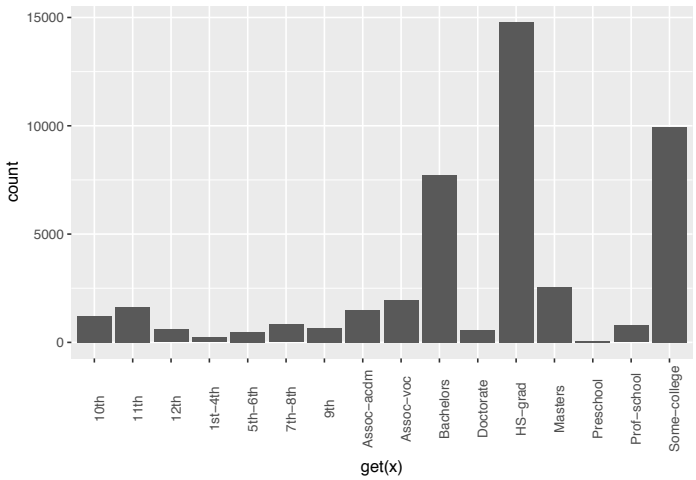
The last step is relatively easy. You want to print the 6 graphs.

```
# Print the graph
graph
```

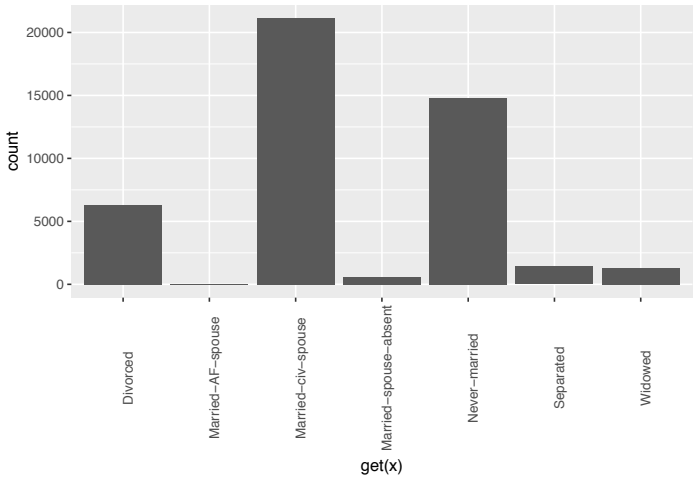
```
## [[1]]
```



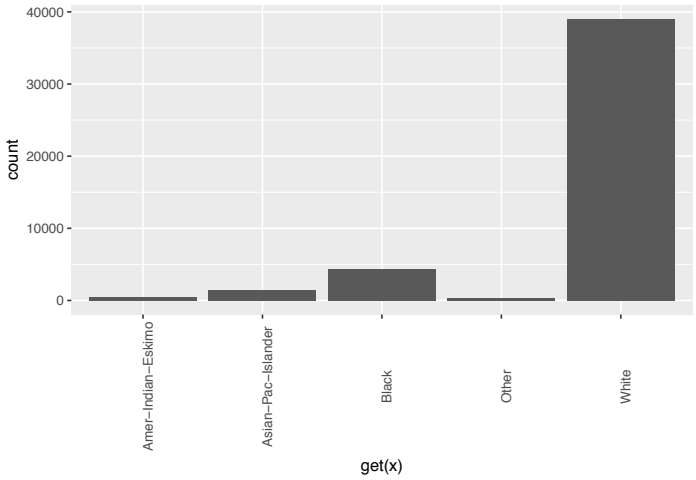

```
##
## [[2]]
```


15000
10000
5000
0
count
10th
11th
12th
1st–4th
5th–6th
7th–8th
9th
Assoc–acdm
Assoc–voc
Bachelors
Doctorate
HS–grad
Masters
Preschool
Prof–school
Some–college
get(x)

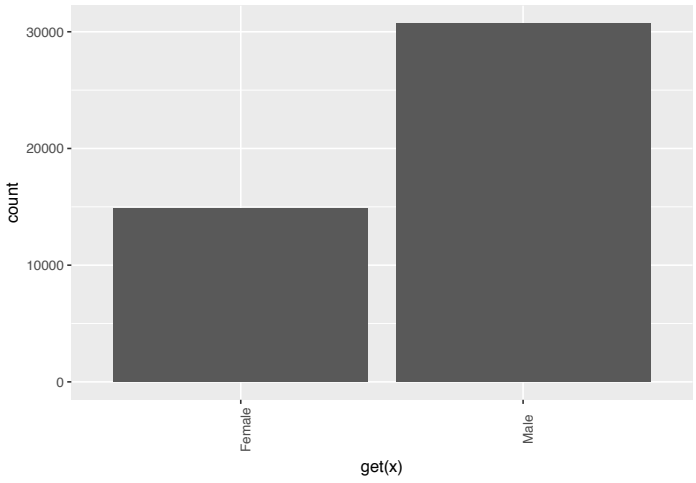
```
##
## [[3]]
```


20000
15000
10000
5000
0
count
Divorced
Married–AF–spouse
Married–civ–spouse
Married–spouse–absent
Never–married
Separated
Widowed
get(x)

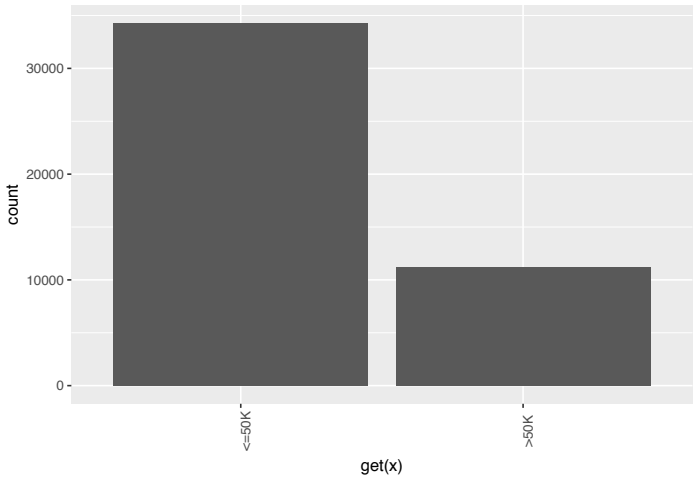
```
##
## [[4]]
```


40000
30000
20000
10000
0
count
Amer–Indian–Eskimo
Asian–Pac–Islander
Black
Other
White
get(x)

```
##
## [[5]]
```




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



Step 3: Feature Engineering

From the graph above, you can see that the variable `education` has 16 levels. This is substantial, and some levels have a relatively low number of observations. If you want to improve the amount of information you can get from this variable, you can recast it into higher level. Namely, you create larger groups with similar level of education. For instance, low level of education will be converted in `dropout`. Higher levels of education will be changed to `master`.

Here is the detail:

Old level	New level
Preschool	dropout
10th	dropout
11th	dropout
12th	dropout
1st-4th	dropout
5th-6th	dropout
7th-8th	dropout
9th	dropout
HS-Grad	HighGrad
Some-college	Community

Old level	New level
Assoc-acdm	Community
Assoc-voc	Community
Bachelors	Bachelors
Masters	Masters
Prof-school	Masters
Doctorate	PhD

```
recast_data <- data_adult_rescale %>%
                select(-X) %>%
                mutate(education = factor(ifelse(education == "Preschool" | education == "10th" | educa
       ifelse(education == "Bachelors", "Bachelors",
              ifelse(education == "Masters"| education == "Prof-school", "Master", "PhD")))))))
```

Code Explanation

- We use the verb `mutate` from `dplyr` library. We change the values of `education` with the statement `ifelse`

In the table below, you create a summary statistic to see, on average, how many years of education (z-value) it takes to reach the Bachelor, Master or PhD.

```
recast_data %>%
            group_by(education)%>%
            summarise(average_educ_year = mean(educational.num),
                      count = n()) %>%
            arrange(average_educ_year)
```

```
## # A tibble: 6 x 3
##   education average_educ_year count
##      <fctr>             <dbl> <int>
## 1   dropout       -1.76147258  5712
## 2  HighGrad       -0.43998868 14803
## 3 Community        0.09561361 13407
## 4 Bachelors        1.12216282  7720
## 5    Master        1.60337381  3338
## 6       PhD        2.29377644   557
```

It is also possible to create lower levels for the marital status. In the following code you change the level as follow:

Old level	New level
Never-married	Not-married
Married-spouse-absent	Not-married
Married-AF-spouse	Maried
Married-civ-spouse	Maried
Separated	Separated
Divorced	Separated
Widows	Widow

```
# Change level marry
recast_data <- recast_data %>%
                mutate(marital.status = factor(ifelse(marital.status == "Never-married" | marital.statu
```

You can check the number of individuals within each group.

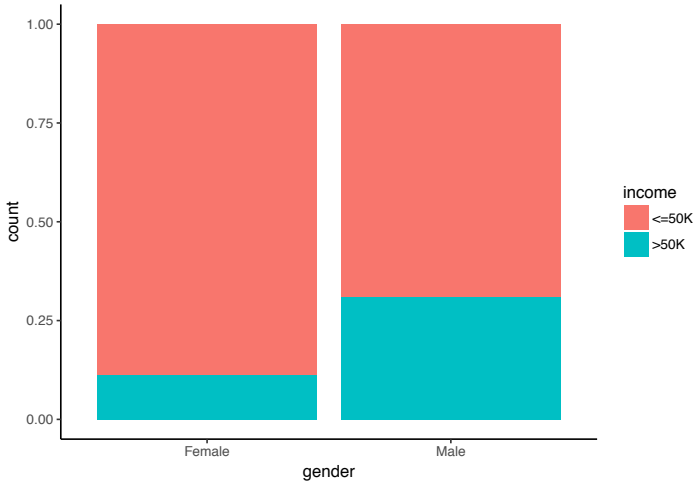
```
table(recast_data$marital.status)
```

```
##
##      Married Not_married   Separated       Widow
##        21165       15359        7727        1286
```

Step 4: Summary statistic

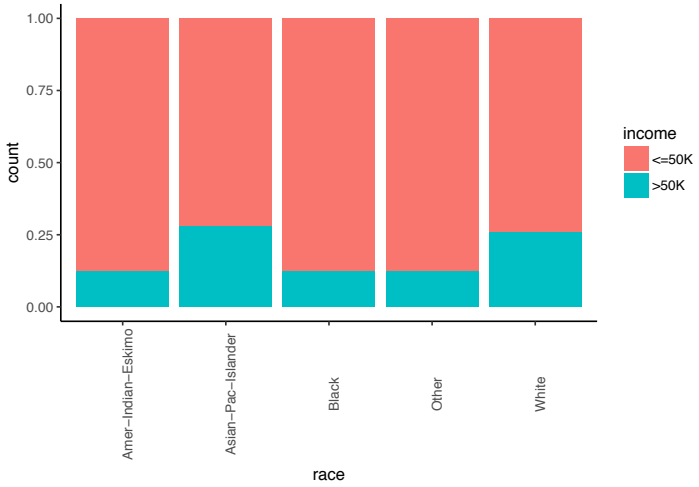
It is time to check some statistics about our target variables. In the graph below, you count the percentage of individuals earning more than 50k given their gender.

```
# Plot gender income
ggplot(recast_data, aes(x =  gender, fill = income))+
  geom_bar(position = "fill") +
  theme_classic()
```



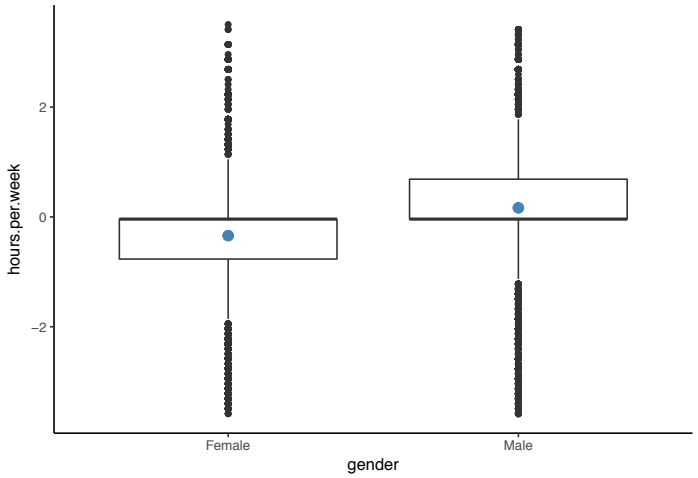
Next, check if the origin of the individual affects their earning.

```
# Plot origin income
ggplot(recast_data, aes(x =  race, fill = income))+
  geom_bar(position = "fill") +
  theme_classic()+
  theme(axis.text.x = element_text(angle = 90))
```



The number of hours work by gender.

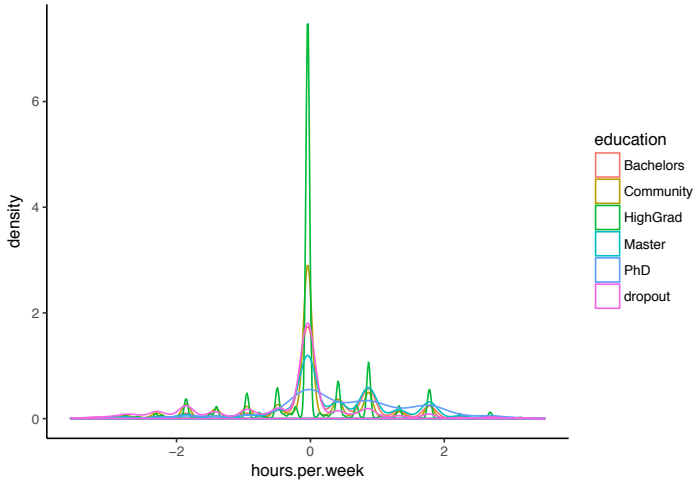
```
# box plot gender working time
ggplot(recast_data, aes( x= gender, y = hours.per.week)) +
  geom_boxplot()+
  stat_summary(fun.y=mean,
               geom = "point",
               size=3,
               color ="steelblue") +
  theme_classic()
```



The box plot confirms that the distribution of working time fits different groups. In the box plot, both genders do not have homogeneous observations.

You can check the density of the weekly working time by type of education. The distributions have many distinct picks. It can probably be explained by the type of contract in the US.

```
# Plot distribution working time by education
ggplot(recast_data, aes( x= hours.per.week)) +
  geom_density(aes(color = education), alpha =0.5)+
  theme_classic()
```



Code Explanation

- `ggplot(recast_data, aes( x= hours.per.week))`: A density plot only requires one variable
- `geom_density(aes(color = education), alpha =0.5)`: The geometric object to control the density

To confirm your thoughts, you can perform a one-way ANOVA test:

```
anova <- aov(hours.per.week ~ education, recast_data)
summary(anova)
```

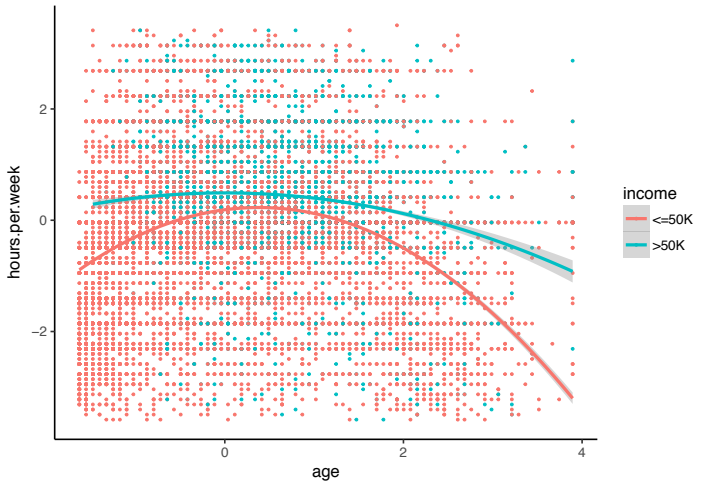
```
##                Df Sum Sq Mean Sq F value Pr(>F)
## education       5   1552  310.31   321.2 <2e-16 ***
## Residuals   45531  43984    0.97
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA test confirms the difference in average between groups.

5.5.4 Non-linearity

Before you run the model, you can see if the number of hours worked is related to age.

```
library(ggplot2)
ggplot(recast_data, aes(x = age, y = hours.per.week))+
  geom_point(aes(color= income),
             size =0.5) +
    stat_smooth(method='lm',
                formula = y~poly(x,2),
                se = TRUE,
                aes(color= income)) +
  theme_classic()
```



Code Explanation

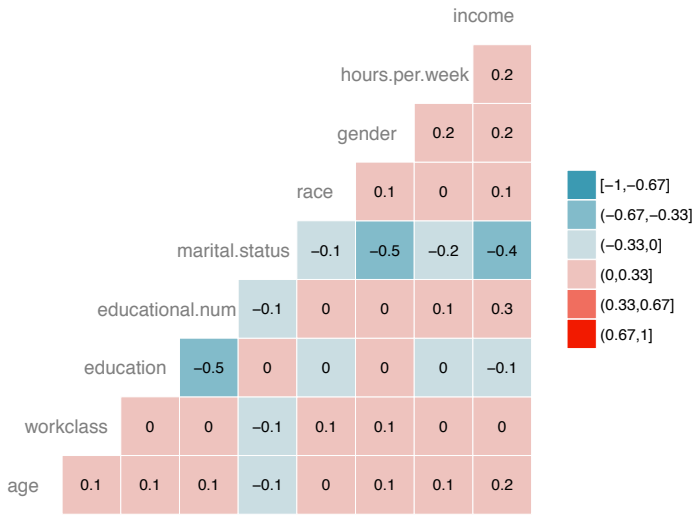
- `ggplot(recast_data, aes(x = age, y = hours.per.week))`: Set the aesthetic of the graph
- `geom_point(aes(color= income), size =0.5)`: Construct the dot plot
- `stat_smooth()`: Add the trend line with the following arguments:
  - `method='lm'`: Plot the fitted value if the linear regression
  - `formula = y~poly(x,2)`: Fit a polynomial regression
  - `se = TRUE`: Add the standard error
  - `aes(color= income)`: Break the model by `income`

In a nutshell, you can test interaction terms in the model to pick up the non-linearity effect between the weekly working time and other features. It is important to detect under which condition the working time differs.

5.5.5 Correlation

The next check is to visualize the correlation between the variables. You convert the factor level type to numeric so that you can plot a heat map containing the coefficient of correlation computed with the Spearman method.

```
library(GGally)
# Convert data to numeric
corr <-  data.frame(lapply(recast_data,as.integer))
# Plot the graph
ggcorr(corr,
       method = c("pairwise", "spearman"),
       nbreaks = 6,
       hjust = 0.8,
       label = TRUE,
       label_size = 3,
       color = "grey50")
```



Code Explanation

- `data.frame(lapply(recast_data,as.integer))`: Convert data to numeric
- `ggcorr()` plot the heat map with the following arguments:
  - `method`: Method to compute the correlation
  - `nbreaks` = 6: Number of break
  - `hjust` = 0.8: Control position of the variable name in the plot
  - `label` = TRUE: Add labels in the center of the windows
  - `label_size` = 3: Size labels
  - `color` = "grey50"): Color of the label

Step 5: Train/test set

Any supervised machine learning task require to split the data between a train set and a test set. You can use the function you created in the other supervised learning tutorials to create a train/test set.

```
set.seed(1234)
create_train_test <- function(data, size=0.8, train = TRUE){
  n_row = nrow(data)
  total_row = size*n_row
  train_sample <- 1:total_row
  if (train ==TRUE){
```

```
    return(data[train_sample, ])
  } else {
    return(data[-train_sample, ])
  }
}
data_train <-create_train_test(recast_data, 0.8, train = TRUE)
data_test <-create_train_test(recast_data, 0.8, train = FALSE)

dim(data_train)
```

```
## [1] 36429     9
```

```
dim(data_test)
```

```
## [1] 9108    9
```

Step 6: Build the model

To see how the algorithm performs, you use the `glm()` package. The **Generalized Linear Model** is a collection of models. The basic syntax is:

```
glm(formula, data=data, family=linkfunction()
Argument:

- formula:  Equation used to fit the model
- data: dataset used
- Family:
    - binomial: (link = "logit")
    - gaussian: (link = "identity")
    - Gamma:    (link = "inverse")
    - inverse.gaussian: (link = "1/mu^2")
    - poisson:  (link = "log")
    - quasi:    (link = "identity", variance = "constant")
    - quasibinomial:    (link = "logit")
    - quasipoisson: (link = "log")
```

You are ready to estimate the logistic model to split the income level between a set of features.

```
formula <- income ~ .
logit <- glm(formula, data = data_train, family = 'binomial')
summary(logit)
```

```
##
## Call:
## glm(formula = formula, family = "binomial", data = data_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.6456  -0.5858  -0.2609  -0.0651   3.1982
##
## Coefficients:
##                            Estimate Std. Error z value Pr(>|z|)
## (Intercept)                 0.07882    0.21726   0.363  0.71675
## age                         0.41119    0.01857  22.146  < 2e-16 ***
## workclassLocal-gov         -0.64018    0.09396  -6.813 9.54e-12 ***
## workclassPrivate           -0.53542    0.07886  -6.789 1.13e-11 ***
## workclassSelf-emp-inc      -0.07733    0.10350  -0.747  0.45499
```

```
## workclassSelf-emp-not-inc -1.09052    0.09140 -11.931  < 2e-16 ***
## workclassState-gov        -0.80562    0.10617  -7.588 3.25e-14 ***
## workclassWithout-pay      -1.09765    0.86787  -1.265  0.20596
## educationCommunity        -0.44436    0.08267  -5.375 7.66e-08 ***
## educationHighGrad         -0.67613    0.11827  -5.717 1.08e-08 ***
## educationMaster            0.35651    0.06780   5.258 1.46e-07 ***
## educationPhD               0.46995    0.15772   2.980  0.00289 **
## educationdropout          -1.04974    0.21280  -4.933 8.10e-07 ***
## educational.num            0.56908    0.07063   8.057 7.84e-16 ***
## marital.statusNot_married -2.50346    0.05113 -48.966  < 2e-16 ***
## marital.statusSeparated   -2.16177    0.05425 -39.846  < 2e-16 ***
## marital.statusWidow       -2.22707    0.12522 -17.785  < 2e-16 ***
## raceAsian-Pac-Islander     0.08359    0.20344   0.411  0.68117
## raceBlack                  0.07188    0.19330   0.372  0.71001
## raceOther                  0.01370    0.27695   0.049  0.96054
## raceWhite                  0.34830    0.18441   1.889  0.05894 .
## genderMale                 0.08596    0.04289   2.004  0.04506 *
## hours.per.week             0.41942    0.01748  23.998  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 40601  on 36428  degrees of freedom
## Residual deviance: 27041  on 36406  degrees of freedom
## AIC: 27087
##
## Number of Fisher Scoring iterations: 6
```

Code Explanation

- `formula <- income ~ .`: Create the model to fit
- `logit <- glm(formula, data = data_train, family = 'binomial')`: Fit a logistic model (`family = 'binomial'`) with the `data_train` data.
- `summary(logit)`: Print the summary of the model

The summary of our model reveals interesting information. The performance of a logistic regression is evaluated with specific key metrics.

- **AIC** (Akaike Information Criteria): This is the equivalent of *R2* in logistic regression. It measures the fit when a penalty is applied to the number of parameters. Smaller *AIC* values indicate the model is closer to the truth.
- **Null deviance**: Fits the model only with the intercept. The degree of freedom is n-1. We can interpret it as a Chi-square value (fitted value different from the actual value hypothesis testing).
- **Residual Deviance**: Model with all the variables. It is also interpreted as a Chi-square hypothesis testing.
- **Number of Fisher Scoring iterations**: Number of iterations before converging.

The output of the `glm()` function is stored in a list. The code below shows all the items available in the `logit` variable we constructed to evaluate the logistic regression.

```
# The list is very long, print only the first three elements
lapply(logit, class)[1:3]
```

```
## $coefficients
## [1] "numeric"
##
```

```
## $residuals
## [1] "numeric"
##
## $fitted.values
## [1] "numeric"
```

Each value can be extracted with the `$` sign follow by the name of the metrics. For instance, you stored the model as `logit`. To extract the *AIC* criteria, you use:

```
logit$aic
```

```
## [1] 27086.65
```

Step 7: Assess the performance of the model

5.5.6 Confusion matrix

The **confusion matrix** is a better choice to evaluate the classification performance compared with the different metrics you saw before. The general idea is to count the number of times `True` instances are classified are `False`.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets.

```
predict <- predict(logit,data_test, type = 'response')
# confusion matrix
table_mat <- table(data_test$income, predict > 0.5)
table_mat
```

```
##
##         FALSE TRUE
##   <=50K  6310  495
##   >50K   1074 1229
```

Code Explanation

- `predict(logit,data_test, type = 'response')`: Compute the prediction on the test set. Set `type = 'response'` to compute the response probability.
- `table(data_test$income, predict > 0.5)`: Compute the confusion matrix. `predict > 0.5` means it returns 1 if the predicted probabilities are above 0.5, else 0.

Each row in a confusion matrix represents an *actual target*, while each column represents a *predicted target*. The first row of this matrix considers the income lower than 50k (the False class): 6241 were correctly classified as individuals with income lower than 50k (**True negative**), while the remaining one was wrongly classified as above 50k (**False positive**). The second row considers the income above 50k, the positive class were 1229 (**True positive**), while the **True negative** was 1074

You can calculate the model **accuracy** by summing the *true positive* + *true negative* over the total observation

```
accuracy_Test <- sum(diag(table_mat))/sum(table_mat)
accuracy_Test
```

```
## [1] 0.8277339
```

Code Explanation

- `sum(diag(table_mat))`: Sum of the diagonal
- 'sum(table_mat): Sum of the matrix.

The model appears to suffer from one problem, it overestimates the number of false negatives. This is called the **accuracy test paradox**. We stated that the accuracy is the ratio of correct predictions to the total number of cases. We can have relatively high accuracy but a useless model. It happens when there is a dominant class. If you look back at the confusion matrix, you can see most of the cases are classified as true negative. Imagine now, the model classified all the classes as negative (i.e. lower than 50k). You would have an accuracy of 75 percent (6718/6718+2257). Your model performs better but struggles to distinguish the true positive with the true negative.

In such situation, it is preferable to have a more concise metric. We can look at:

- $Precision = TP/(TP + FP)$
- $Recall = TP/(TP + FN)$

5.5.7 Precision vs Recall

Precision looks at the accuracy of the positive prediction. **Recall** is the ratio of positive instances that are correctly detected by the classifier;

You can construct two functions to compute these two metrics

1) Construct precision

```
precision <-  function(matrix){

    # True positive
    tp <- matrix[2,2]
    # false positive
    fp <- matrix[1,2]
    return(tp/(tp+fp))
}
```

Code Explanation

- `mat[1,1]`: Return the first cell of the first column of the data frame, i.e. the true positive
- `mat[1,2]`; Return the first cell of the second column of the data frame,, i.e. the false positive

```
recall <-  function(matrix){

    # true positive
    tp <- matrix[2,2]
    # false positive
    fn <- matrix[2,1]
     return(tp/(tp+fn))
}
```

Code Explanation

- `mat[1,1]`: Return the first cell of the first column of the data frame, i.e. the true positive
- `mat[2,1]`; Return the second cell of the first column of the data frame, i.e. the false negative

You can test your functions

```
prec <- precision(table_mat)
prec
```

```
## [1] 0.712877
```

```
rec <- recall(table_mat)
rec
```

```
## [1] 0.5336518
```

When the model says it is an individual above 50k, it is correct in only 54 percent of the case, and can claim individuals above 50k in 72 percent of the case.

You can create the F_1 score based on the precision and recall. The F_1 is a harmonic mean of these two metrics, meaning it gives more weight to the lower values.

$F_1 = 2 * \frac{precision * recall}{precision + recall}$

```
f1 <- 2*((prec*rec)/(prec+rec))
f1
```

```
## [1] 0.6103799
```

Precision vs Recall tradeoff

It is impossible to have both a high precision and high recall.

If we increase the precision, the correct individual will be better predicted, but we would miss lots of them (lower recall). In some situation, we prefer higher precision than recall. There is a concave relationship between precision and recall.

- Imagine, you need to predict if a patient has a disease. You want to be as precise as possible.
- If you need to detect potential fraudulent people in the street through facial recognition, it would be better to catch many people labelled as fraudulent even though the precision is low. The police will be able to release the non-fraudulent individual.

5.5.8 The ROC curve

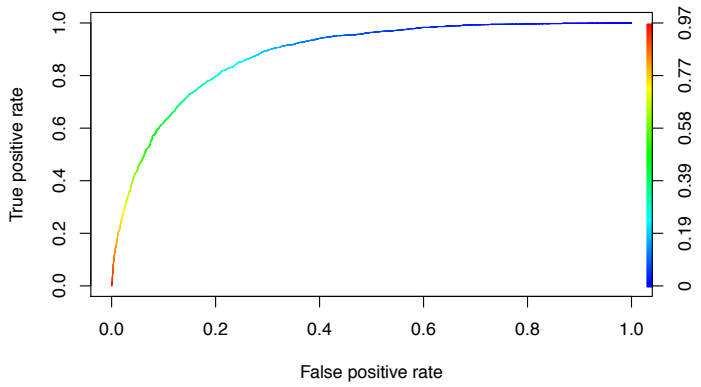
The **Receiver Operating Characteristic** curve is another common tool used with binary classification. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve shows the *true positive rate (i.e. recall)* against the *false positive rate*. The *false positive rate* is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the *true negative rate*. The *true negative rate* is also called **specificity**. Hence the ROC curve plots **sensitivity** (recall) versus $1 - specificity$

Before you need to install one library to plot the ROC.

- `RORC`: Plot graph for classification postestimation. I
  - Anaconda: `conda install -c r r-rocr --yes`

We can plot the ROC with the `prediction()` and `performance()` functions.

```
library(ROCR)
ROCRpred <- prediction(predict, data_test$income)
ROCRperf <- performance(ROCRpred, 'tpr','fpr')
plot(ROCRperf, colorize = TRUE, text.adj = c(-0.2,1.7))
```



Code Explanation

- `prediction(predict, data_test$income)`: The ROCR library needs to create a prediction object to transform the input data
- `performance(ROCRpred, 'tpr','fpr')`: Return the two combinations to produce in the graph. Here, `tpr` and `fpr` are constructed. Tot plot precision and recall together, use `prec`, 'rec.

Step 8: Improve the model

You can try to add non-linearity to the model with the interaction between `age` and `hours.per.week`, `gender` and `hours.per.week`.

You need to use the F_1 score test to compare both model

```
formula_2 <- income ~ age:hours.per.week + gender:hours.per.week + .
logit_2 <- glm(formula_2, data = data_train, family = 'binomial')
predict_2 <- predict(logit_2, data_test,  type = 'response')
table_mat_2 <- table(data_test$income, predict_2 > 0.5)

precision_2 <- precision(table_mat_2)
recall_2 <- recall(table_mat_2)
f1_2 <- 2*((precision_2*recall_2)/(precision_2+recall_2))
f1_2
```

```
## [1] 0.6109181
```

The score is slightly higher than the previous one. You can keep working on the data a try to beat the score.

5.5.9 Summary

To perform an analysis, you can follow these steps:

- Step 1: Check continuous variables
- Step 2: Check factor variables
- Step 3: Feature engineering
- Step 4: Summary statistic
- Step 5: Train/test set
- Step 5: Build the model
- Step 6: Assess the performance of the model
- step 7: Improve the model

We summarize the function to train a logisitic regression in the table below:

Package	Objective	function	argument
-	Create train/test dataset	create_train_set()	data, size, train

Package	Objective	function	argument
glm	Train a Generalized Linear Model	glm()	formula, data, family*
glm	Summarize the model	summary()	fitted model
base	Make prediction	predict()	fitted model, dataset, type = 'response'
base	Create a confusion matrix	table()	y, predict()
base	Create accuracy score	sum(diag(table())/sum(table()	
ROCR	Create ROC Step 1 Create prediction	prediction()	predict(), y
ROCR	Create ROC Step 2 Create performance	performance()	prediction(),'tpr', 'fpr'
ROCR	Create ROC Step 3 Plot graph	plot()	performance()

The other **GLM** type of models are:

```
- binomial: (link = "logit")
- gaussian: (link = "identity")
- Gamma:    (link = "inverse")
- inverse.gaussian: (link = "1/mu^2")
- poisson:  (link = "log")
- quasi:    (link = "identity", variance = "constant")
- quasibinomial:    (link = "logit")
- quasipoisson: (link = "log")
```

5.6 Cluster analysis

Cluster analysis is part of the **unsupervised learning**. A cluster is a group of data that share similar features. We can say, clustering analysis is more about discovery than prediction. The machine searches for similarity in the data. For instance, you can use cluster analysis for the following application:

- Customer segmentation: Looks for similarity between groups of customers
- Stock Market clustering: Group stock based on performances
- Reduce dimensionality of a dataset by grouping observations with similar values

Clustering analysis is not too difficult to implement and is meaningful as well as actionable for business.

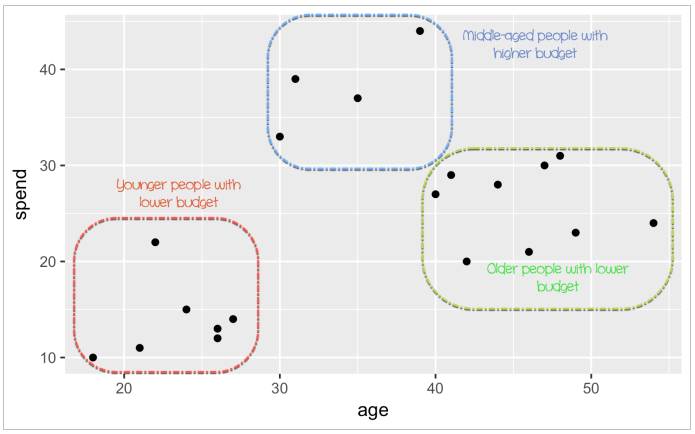
The most striking difference between supervised and unsupervised learning lies in the results. Unsupervised learning creates a new variable, the label, while supervised learning predicts an outcome. The machine helps the practitioner in the quest to label the data based on close relatedness. It is up to the analyst to make use of the groups and give a name to them.

Let's make an example to understand the concept of clustering. For simplicity, we work in two dimensions. You have data on the total spend of customers and their ages. To improve advertising, the marketing team wants to send more targeted emails to their customers.

In the following graph, you plot the total spend and the age of the customers.

A pattern is visible at this point:

- At the bottom-left, you can see young people with a lower purchasing power
- Upper-middle reflects people with a job which can afford more spend
- Finally older people with a lower budget.



In the figure above, you cluster the observations by hand and define each of the three groups. This example is somewhat straightforward and highly visual. If new observations are appended to the data set, you can label them within the circles. You define the circle based on our judgment Instead, you can use Machine Learning to group the data objectively.

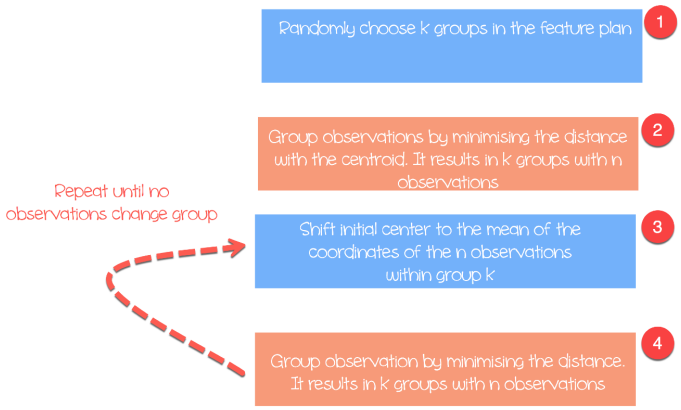
In this tutorial, you will learn how to use the *k-mean* algorithm

5.6.1 K-mean algorithm

Minimize the distances

K-mean is, without doubt, the most popular clustering method. Researchers released the algorithm decades ago, and lots of improvements have been done to *k-mean*.

The algorithm tries to find groups by minimizing the distance between the observations, called **local optimal** solutions. The distances are measured based on the coordinates of the observations. For instance, in a two-dimensional space, the coordinates are simple x_i and y_i.



The algorithm works as follow:

- Step 1: Choose k groups in the feature plan randomly
- Step 2: Minimize the distance between the cluster center and the different observations. It results of k groups with n observations
- Step 3: Compute the mean of the coordinates within a group. It becomes the new **centroid**
- Step 4: Minimize the distance according to the new centroids. New boundaries are created. Thus, observations will move from one group to another
- Repeat until no observation changes groups

K-mean usually takes the Euclidean distance between the feature x and feature y:

$$distance(x, y) = \sum_{i}^{n}(x_i - y_i)^2$$

Different measures are available such as the Manhattan distance or Minlowski distance. Note that, *k-mean* returns different groups each time you run the algorithm. Recall that the first initial guesses are random and compute the distances until the algorithm reaches a homogeneity within groups. That is, *k-mean* is very sensitive to the first choice, and unless the number of observations and groups are small, it is almost impossible to get the same clustering.

5.6.2 Select the number of clusters

Another difficulty found with *k-mean* is the choice of the number of clusters. You can set a high value of k, i.e. a large number of groups, to improve stability but you might end up with **overfit** of data. Overfitting means the performance of the model decreases substantially for new coming data. The machine learnt the little details of the data set and struggle to generalize the overall pattern.

The number of clusters depends on the nature of the data set, the industry, business and so on. However, there is a rule of thumb to select the appropriate number of clusters:

$$cluster = \sqrt{2/n}$$

with n equals to the number of observation in the dataset.

Generally speaking, it is interesting to spend times to search for the best value of k to fit with the business need.

We will use the `Prices of Personal Computers` dataset to perform our clustering analysis. This dataset contains 6259 observations and 10 features. The dataset observes the price from 1993 to 1995 of 486 personal computers in the US. The variables are `price`, `speed`, `ram`, 'screen, 'cd among other.

You will proceed as follow:

- Import data
- Train the model
- Evaluate the model

5.6.3 Import data

K-mean is not suitable for factor variables because it is based on the distance and discrete values do not return meaningful values. You can delete the three categorical variables in our dataset. Besides, there are no missing values in this dataset.

```
library(dplyr)
PATH <-"https://raw.githubusercontent.com/thomaspernet/data_csv_r/master/data/Computers.csv"
df <- read.csv(PATH) %>%
      select(-c(X, cd, multi, premium))
glimpse(df)
```

```
## Observations: 6,259
## Variables: 7
## $ price  <int> 1499, 1795, 1595, 1849, 3295, 3695, 1720, 1995, 2225, 2...
## $ speed  <int> 25, 33, 25, 25, 33, 66, 25, 50, 50, 50, 33, 66, 50, 25,...
## $ hd     <int> 80, 85, 170, 170, 340, 340, 170, 85, 210, 210, 170, 210...
```

```
## $ ram    <int> 4, 2, 4, 8, 16, 16, 4, 2, 8, 4, 8, 8, 4, 8, 8, 4, 2, 4,...
## $ screen <int> 14, 14, 15, 14, 14, 14, 14, 14, 14, 15, 15, 14, 14, 14,...
## $ ads    <int> 94, 94, 94, 94, 94, 94, 94, 94, 94, 94, 94, 94, 94, 94,...
## $ trend  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
```

From the summary statistics, you can see the data has large values. A good practice with *k-mean* and distance calculation is to rescale the data so that the mean is equal to one and the standard deviation is equal to zero.

```
summary(df)
```

```
##      price          speed              hd             ram
##  Min.   : 949   Min.   : 25.00   Min.   :  80.0   Min.   : 2.000
##  1st Qu.:1794   1st Qu.: 33.00   1st Qu.: 214.0   1st Qu.: 4.000
##  Median :2144   Median : 50.00   Median : 340.0   Median : 8.000
##  Mean   :2220   Mean   : 52.01   Mean   : 416.6   Mean   : 8.287
##  3rd Qu.:2595   3rd Qu.: 66.00   3rd Qu.: 528.0   3rd Qu.: 8.000
##  Max.   :5399   Max.   :100.00   Max.   :2100.0   Max.   :32.000
##      screen           ads            trend
##  Min.   :14.00   Min.   : 39.0   Min.   : 1.00
##  1st Qu.:14.00   1st Qu.:162.5   1st Qu.:10.00
##  Median :14.00   Median :246.0   Median :16.00
##  Mean   :14.61   Mean   :221.3   Mean   :15.93
##  3rd Qu.:15.00   3rd Qu.:275.0   3rd Qu.:21.50
##  Max.   :17.00   Max.   :339.0   Max.   :35.00
```

You rescale the variables with the `scale()` function of the `dplyr` library. The transformation reduces the impact of outliers and allows to compare a sole observation against the mean. If a standardize value (or **z-score**) is high, you can be confident that this observation is indeed above the mean (a large z-score implies that this point is far away from the mean in term of standard deviation. A z-score of two indicates the value is 2 standard deviations away from the mean. Note, the z-score follows a Gaussian distribution and is symmetrical around the mean.

```
rescale_df <- df %>%
  mutate(price_scal = scale(price),
        hd_scal = scale(hd),
        ram_scal = scale(ram),
        screen_scal = scale(screen),
        ads_scal = scale(ads),
        trend_scal = scale(trend)) %>%
  select(-c(price, speed, hd, ram, screen, ads, trend))
```

R base has a function to run the *k-mean* algorithm. The basic function of *k-mean* is:

```
kmeans(df, k)
arguments:
- df: dataset used to run the algorithm
- k: Number of clusters
```

You are ready to train the model.

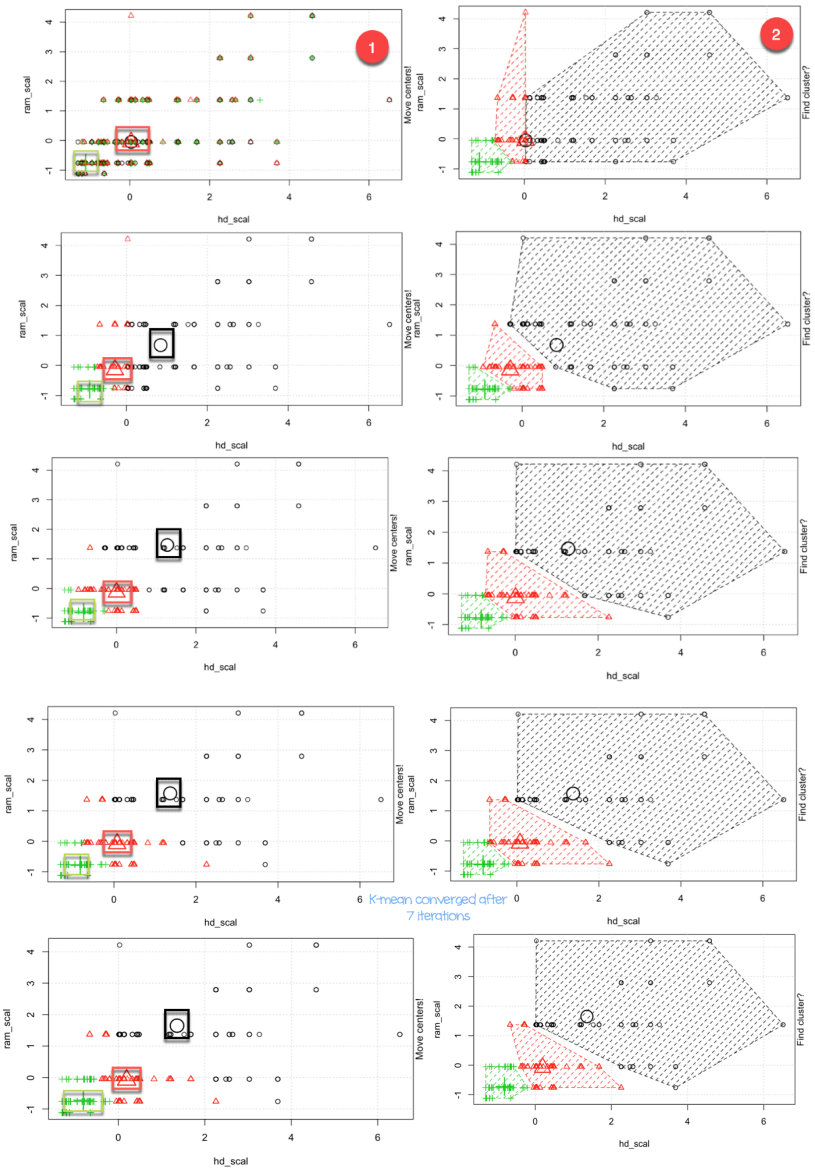
5.6.4 Train the model

In figure three, you detailed how the algorithm works. You can see each step graphically with the great package build by Yi Hui (also creator of Knit for Rmarkdown). The package `animation` is not available in the conda library. You can use the other way to install the package with `install.packages("animation")`. You can check if the package is installed in our Anaconda folder.

```
install.packages("animation")
```

After you loaded the library, you add `.ani` after kmeans and R will plot all the steps. For illustration purpose, you only run the algorithm with the rescaled variables `hd` and `ram` with three clusters.

```
set.seed(2345)
library(animation)
kmeans.ani(rescale_df[2:3], 3)
```



Code Explananation

- `kmeans.ani(rescale_df[2:3], 3)`: Select the columbns 2 and 3 of `rescale_df` data set and run the algorithm with `k` sets to 3. Plot the animation.

You can interpret the animation as follow:

- Step 1: R randomly chooses three points
- Step 2: Compute the Euclidean distance and draw the clusters. You have one cluster in green at the bottom left, one large cluster colored in black at the right and a red one between them.

- Step 3: Compute the centroid, i.e. the mean of the clusters
- Repeat until no data changes cluster

The algorithm converged after seven iterations. You can run the *k-mean* algorithm in our dataset with five clusters and call it `pc_cluster`.

```
pc_cluster <- kmeans(rescale_df, 5)
```

The list `pc_cluster` contains seven interesting elements:

```
- pc_cluster$cluster: Indicates the cluster of each observation
- pc_cluster$centers: The cluster centers
- pc_cluster$totss: The total sum of squares
- pc_cluster$withinss: Within sum of square. The number of components return is equal to `k`
- pc_cluster$tot.withinss: Sum of withinss
- pc_clusterbetweenss: Total sum of square minus Within sum of square
- pc_cluster$size: Number of observation within each cluster
```

You will use the sum of the within sum of square (i.e. `tot.withinss`) to compute the optimal number of clusters `k`. Finding `k` is indeed a substantial task.

5.6.5 Optimal k

One technique to choose the best `k` is called the **elbow method**. This method use within-group homogeneity or within-group heterogeneity to evaluate the variability. In other word, you are interested in the percentage of the variance explained by each cluster. You can expect the variability to increase with the number of clusters, alternatively, heterogeneity decreases. Our challenge is to find the `k` that is beyond the diminishing returns. Adding a new cluster does not improve the variability in the data because very few information is left to explain.

In this tutorial, we find this point using the heterogeneity measure. The Total within clusters sum of squares is the `tot.withinss` in the list return by `kmean()`.

You can construct the elbow graph and find the optimal `k` as follow:

- Step 1: Construct a function to compute the total within clusters sum of squares
- Step 2: Run the algorithm n times
- Step 3: Create a data frame with the results of the algorithm
- Step 4: Plot the results

Step 1

You create the function that run the *k-mean* algorithm and store the total within clusters sum of squares

```
kmean_withinss <- function(k){

  cluster <- kmeans(rescale_df, k)
  return(cluster$tot.withinss)
}
```

Code Explanation

- `function(k)`: Set the number of arguments in the function
- `kmeans(rescale_df, k)`: Run the algorithm `k` times
- `return(cluster$tot.withinss)`: Store the total within clusters sum of squares

You can test the function with k equals 2.

```
## Try with 2 cluster
kmean_withinss(2)
```

```
## [1] 27087.1
```

Step 2

You will use the `sapply()` function to run the algorithm over a range of `k`. This technique is faster than creating a loop and store the value.

```
# Set maximum cluster
max_k <-20
# Run algorithm over a range of k
wss <- sapply(2:max_k, kmean_withinss)
```

Code Explanation

- `max_k <-20`: Set a maximum number of k to 20
- `sapply(2:max_k, kmean_withinss)`: Run the function `kmean_withinss()` over a range `2:max_k`, i.e. 2 to 20.

Step 3

Post creation and testing our function, you can run the *k-mean* algorithm over a range from 2 to 20, store the `tot.withinss` values.

```
# Create a data frame to plot the graph
elbow <- data.frame(2:max_k, wss)
```

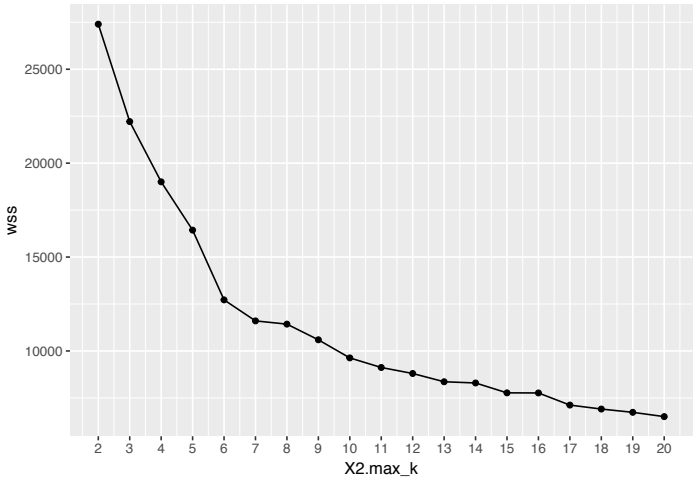
Code Explanation

- `data.frame(2:max_k, wss)`: Create a data frame with the output of the algorithm store in `wss`

Step 4

You plot the graph to visualize where is the elbow point

```
library(ggplot2)
# Plot the graph with gglop
ggplot(elbow, aes(x= X2.max_k, y = wss)) +
  geom_point()+
  geom_line()+
  scale_x_continuous(breaks = seq(1, 20, by = 1))
```



From the graph, you can see the optimal `k` is seven, where the curve is starting to have a diminishing return.

Once you have our optimal `k`, you re-run the algorithm with `k` equals to 7 and evaluate the clusters.

5.6.6 Examining the cluster

```
pc_cluster_2 <- kmeans(rescale_df, 7)
```

As mention before, you can access the remaining interesting information in the list returned by `kmean()`.

```
pc_cluster_2$cluster
pc_cluster_2$centers
pc_cluster_2$size
```

The evaluation part is subjective and relies on the used of the algorithm. Our goal here is to gather computer with similar features. A computer guy can do the job by hand and group computer based on his expertise. However, the process will take lots of time and will be error prone. *K-mean* algorithm can prepare the field for him/her by suggesting clusters.

As a prior evaluation, you can examine the size of the clusters.

```
pc_cluster_2$size
```

```
## [1]  545  599 1232 1003  685  607 1588
```

The first cluster is composed of 608 observations, while the smallest cluster, number 4, has only 580 computers. It might be good to have homogeneity between clusters, if not, a thinner data preparation might be required.

You get a deeper look at the data with the `center` component. The rows refer to the numeration of the cluster and the columns the variables used by the algorithm. The values are the average score by each cluster for the interested column. Standardization makes the interpretation easier. Positive values indicate the z-score for a given cluster is above the overall mean. For instance, cluster 2 has the highest price average among all the clusters.

```
center <- pc_cluster_2$centers
center
```

```
##   price_scal     hd_scal    ram_scal screen_scal     ads_scal trend_scal
## 1  0.6101026  0.02080832 -0.05421596   2.6419582  0.008923609  0.1789176
## 2  0.8477444  2.09804028  2.12868625   0.5521847 -0.958505816  1.2493831
## 3  0.2308749 -0.08246679 -0.18875904  -0.2142829  0.705208528 -0.2369967
## 4 -0.8329420  0.27999135 -0.31653862  -0.3244544 -0.944605756  1.2449639
## 5  1.2606262  0.51366979  1.16854749  -0.2338301  0.593845535 -0.3848922
## 6  0.2203274 -0.72146691 -0.34410480  -0.3867732 -1.328160437 -1.5534402
## 7 -0.8101823 -0.85719917 -0.81050341  -0.4951238  0.659517304 -0.3753275
```

You can create a heat map with `ggplot` to help us highlight the difference between categories.

The default colors of `ggplot` need to be changed with the `RColorBrewer` library.

- `RORC`: Customize the color's palette.
  - Anaconda: `conda install -c r r-rcolorbrewer`

To create a heat map, you proceed in three steps:

- Build a data frame with the values of the centre and create a variable with the number of the cluster
- Reshape the data with the `gather()` function of the `tidyr` library. You want to transform data from wide to long.
- Create the palette of colors with `colorRampPalette()` function

Step 1

Let's create the reshape dataset

```
library(tidyr)
```

```
##
## Attaching package: 'tidyr'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     extract
```

```
# create dataset with the cluster number
cluster <- c(1:7)
center_df <- data.frame(cluster, center)

# Reshape the data
center_reshape <- gather(center_df, features, values, price_scal:trend_scal)
head(center_reshape)
```

```
##   cluster   features     values
## 1       1 price_scal  0.6101026
## 2       2 price_scal  0.8477444
## 3       3 price_scal  0.2308749
## 4       4 price_scal -0.8329420
## 5       5 price_scal  1.2606262
## 6       6 price_scal  0.2203274
```

Step 2

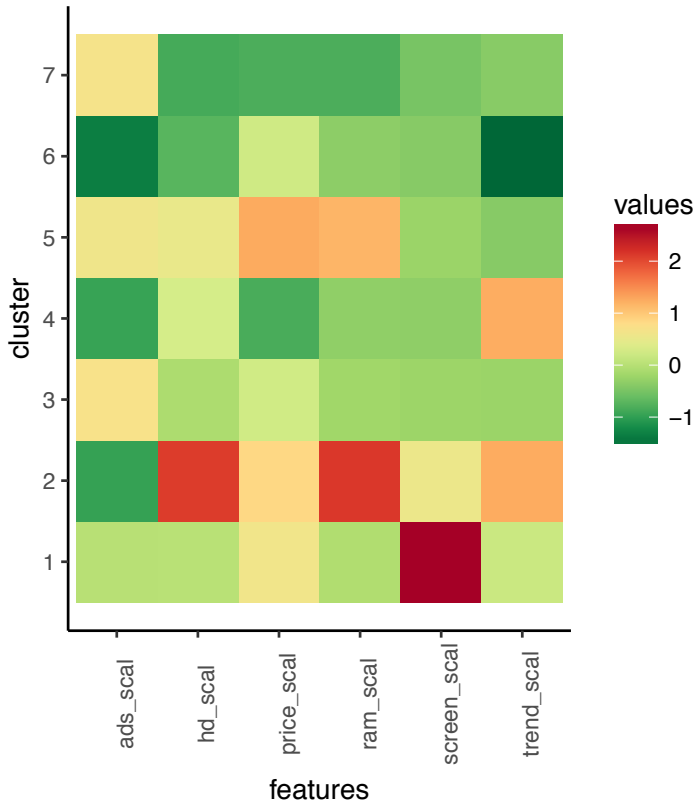
The code below create the palette of colors you will use to plot the heat map.

```
library(RColorBrewer)
# Create the palette
hm.palette <- colorRampPalette(rev(brewer.pal(10, 'RdYlGn')), space='Lab')
```

Step 3

You can plot the graph and see what the clusters look like.

```
# Plot the heat map
ggplot(data = center_reshape, aes(x = features, y = cluster, fill = values)) +
  scale_y_continuous(breaks = seq(1, 7, by = 1))+
  geom_tile()+
  coord_equal()+
  scale_fill_gradientn(colours = hm.palette(90)) +
  theme_classic()+
  theme(axis.text.x = element_text(angle = 90))
```



5.6.7 Summary

We can summarize the *k-mean* algorithm in the table below

Package	Objective	function	argument
base	Train k-mean	kmeans()	df, k
	Access cluster	kmeans()$cluster	
	Cluster centers	kmeans()$centers	
	Size cluster	kmeans()$size	

To create the elbow curve, you can use the following code:

```
# Create tot within sum square function
kmean_withinss <- function(k){

  cluster <- kmeans(rescale_df, k)
  return(cluster$tot.withinss)
}

# Set maximum cluster
max_k <-20
# Run algorithm over a range of k
wss <- sapply(2:max_k, kmean_withinss)
# Create a data frame to plot the graph
elbow <- data.frame(2:max_k, wss)
# Plot the graph with gglop
```

```
ggplot(elbow, aes(x= X2.max_k, y = wss)) +
  geom_point()+
  geom_line()+
  scale_x_continuous(breaks = seq(1, 20, by = 1))
```