# Statistical concepts with R

*Pietro Lesci*

*01 January 2019*

# Contents

# Preface

In this book we will show the mechanics of statistical procedures and the practical intuition behind theoretical concept using R. This is not a book about data analysis, however we will make use of some modern R packages that have been developed for those applications. Thanks to these packages R is living a shining period in the data science universe amongst the best choices to perform data science tasks - and "almost surely" it will give you a job!

# Chapter 1

# Introduction

---

*In this chapter we review the basics of the R programming language and we explore the tidyverse: a universe of interrelated packages that modernize the basic R and simplify operations in a coherent way. It contributed to the estabilishment of R as one of the preferred tools for data science outside the academia.*

---

## 1.1 R

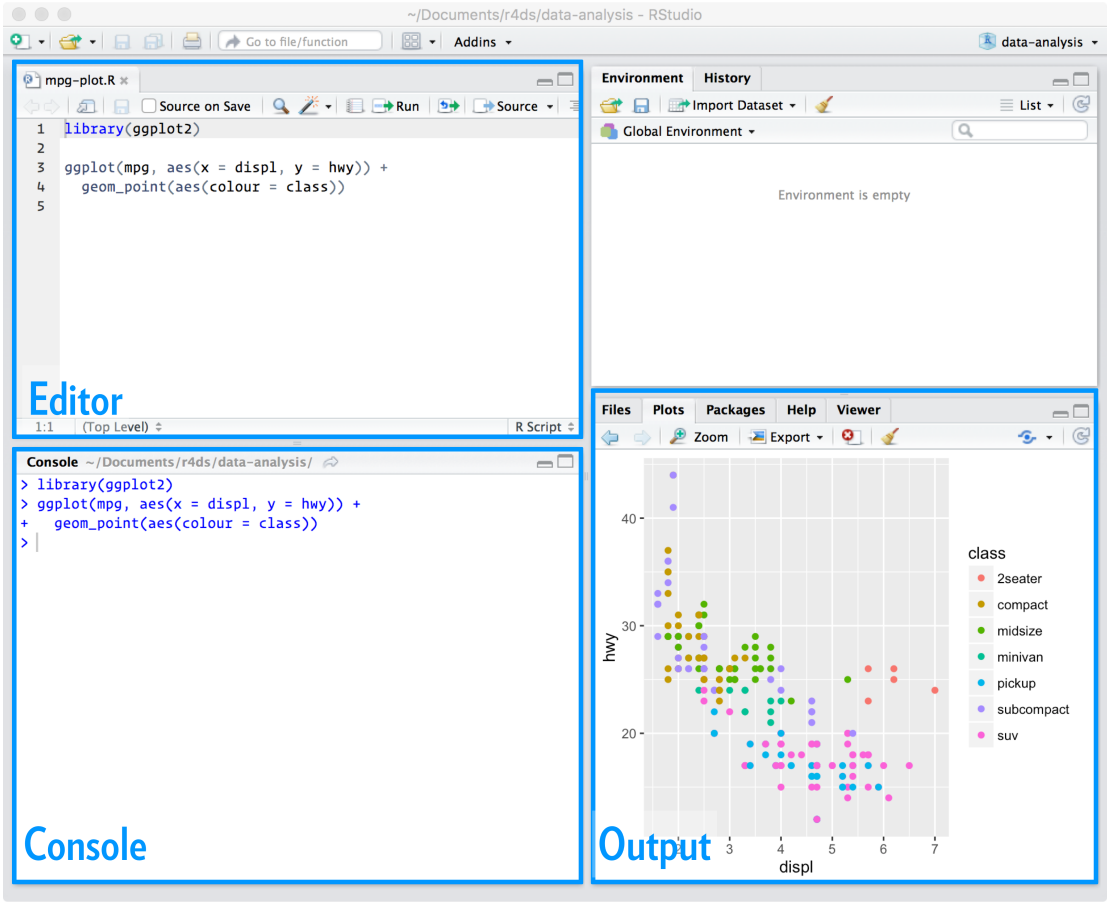
To download R, go to CRAN, the **C**omprehensive **R** **A**rchive **N**etwork. CRAN is composed of a set of mirror servers distributed around the world and is used to distribute R and R packages. There is no need to pick a mirror that is close to the location from where you are downloading: instead use the cloud mirror, https://cloud.r-project.org, which automatically figures it out optimally.

Once a year a new major version of R is released, besides 2-3 minor releases. It is suggested to update it regularly. Upgrading can be a bit of a hassle, especially for major versions, which require the users to reinstall all packages, but putting it off only makes it worse and prevents the user to work with the state-of-the-art tools.

## 1.2 RStudio

RStudio is an integrated development environment, or IDE, for R programming. Download and install it from http://www.rstudio.com/download. RStudio is updated usually twice a year. When a new version is available, RStudio notifies the user and asks for the confirmation to perform the update. It is a good practice to upgrade regularly in order to take advantage of the latest and greatest features. For this book, I am making use of RStudio. Writing any kind of paper that requires to write latex code and R code is easely done within RStudio. Furthermore, thanks to great packages, it is nowadays possible to create and deploy apps, websites, html files completely written in RStudio. Therefore, besides statistical analysis, RStudio is a great companion during your years at university and beyond. The three great packages that enhance RStudio capabilities are `rmarkdown`, `bookdown`, and `rshiny`.

When start RStudio, four key regions will appear in the interface:



R code can be typed in the console pane, and pressing enter will run it. This way of programming is often called interactive programming. For more complicated pieces of code, and to share code with others, it is convenient to write code in the editor. It is still possible to run a single line of code pressing ctrl and enter, but more conveniently it is possible to run (or source) an entire script.

## 1.3 The R language: basics

In this section we will review the building blocks R which are, in general the basic activities that a language should have to be considered a "programming language"[1]:

- data structures
- control flow
- iterations
- functions

### 1.3.1 Data structures

R's base data structures can be organised by their dimensionality (1d, 2d, or nd) and whether they are homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types). This gives rise to the five data types most often used in data analysis:

| | Homogeneous | Heterogeneous |
|---|---|---|
| 1d | Atomic | Vector List |
| 2d | Matrix | Dataframe |
| nd | Array | |

Note that R has no 0-dimensional, or scalar types. Individual numbers or strings are actually vectors of length one.

Given an object, the best way to understand in which data structures it is stored is to use `str()`. `str()` is short for structure and it gives a compact, human readable description of any R data structure.

[1] An example of a language which is not considered a programming language is HTML.

#### 1.3.1.1 Vectors

The basic data structure in R is the vector. Vectors come in two flavours: atomic vectors and lists. They have three common properties:

- Type, `typeof()`, what it is
- Length, `length()`, how many elements it contains
- Attributes, `attributes()`, additional arbitrary metadata

They differ in the types of their elements: all elements of an atomic vector must be the same type, whereas the elements of a list can have different types.

**Atomic vectors** are usually created with `c()`, short for "combine":

```
vector <- c(1, 2, 3)
```

They are always flat, even if you nest `c()`'s

```
c(c(1), c(2, 3))
#> [1] 1 2 3
```

Missing values are specified with `NA`, which is a logical vector of length 1. All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

For example, combining a character and an integer yields a character:

```
c('a', 1)
#> [1] "a" "1"
```

When a logical vector is coerced to an integer or double, `TRUE` becomes 1 and `FALSE` becomes 0. This is very useful in conjunction with `sum()` and `mean()`

```
sum(c(TRUE, FALSE, 1))
#> [1] 2
```

**Lists** are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using `list()` instead of `c()`:

```
x <- list(1:3, 'a', "b", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 5
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : chr "b"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

Note how `''` and `""` can be used interchangeably to define strings, or characters: in R, unlike Python, there is no distinction between the two. As said above, `str()` returns the structure of an object, more info with `?str`. Lists are sometimes called recursive vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
```

`c()` will combine several lists into one. If given a combination of atomic vectors and lists, `c()` will coerce the vectors to lists before combining them. Compare the results of `list()` and `c()`:

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(y)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

#### 1.3.1.2 Arrays and Matrices

Matrices are a special type of array: they are 2-dimensional arrays. They can be defined by creating a list of objects and providing the dimensions

```
matrix(list(1, 2, 3, 'a', 'b', 'c'), nrow = 3, ncol = 2)
#>      [,1] [,2]
#> [1,] 1    "a"
#> [2,] 2    "b"
#> [3,] 3    "c"
matrix(c(1, 2, 3, 'a', 'b', 'c'), nrow = 3, ncol = 2)
#>      [,1] [,2]
#> [1,] "1"  "a"
#> [2,] "2"  "b"
#> [3,] "3"  "c"
```

Note that `matrix()` and `array()` can host any kind of data types: they just store the data in a multidimensional format. Note how the number are coerced to strings when fed through `c()` contrarely at what happens when using `list()`.

#### 1.3.1.3 Dataframes

A dataframe is the most common way of storing data in R, and if used systematically makes data analysis easier. Under the hood, a dataframe is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. A dataframe has `names()`, `colnames()`, and `rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a dataframe is the length of the underlying list and so it is the same as `ncol()`; `nrow()` gives the number of rows. You create a dataframe using `data.frame()`, which takes named vectors (atomic vectors or lists) as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Beware `data.frame()`'s default behaviour which turns strings into factors. Use `stringsAsFactors = FALSE` to suppress this behaviour.

You can combine dataframes using `cbind()` and `rbind()`:

```
df_1 <- data.frame(x = 1:3, y = c("a", "b", "c")) #row:3, col:2
df_2 <- data.frame(z = 3:1)                       #row:3, col:1
df_3 <- data.frame(x = 10, y = "z")               #row:1, col:2
cbind(df_1, df_2)                                 #row:3, col:3
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df_1, df_3)                                 #row:4, col:2
#>    x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number and names of columns must match.

### 1.3.2 Functions

Functions allow the user to automate common tasks in a more powerful and general way than copy-and-pasting the same code. There are three key steps to creating a new function:

1. Pick a **name** for the function
2. List the inputs, or **arguments**, to the function inside `function (...)`
3. Place the code into the **body** of the function, a `{...}` block that immediately follows `function(...)`

```
my_func <- function(argument) {
    value <- do_something(argument)
    return(value)
}
```

Note that the last `return()` can be discarded and we can just write `value`. MOre about this below.

### 1.3.3 Control structures

Control structures allow us to specify the execution of the code based on some conditions. They are extremely useful when we want to run a piece of code multiple times, or when we want to run a piece a code if a certain condition is met.

An `if` statement allows us to conditionally execute code. It looks like this:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

Below a simple function that uses an `if` statement is presented. The goal of this function is to return a logical value describing whether or not a string is equal to the name "mimmo".

```
check_name <- function(string) {
    if (string == 'mimmo') {
        print('YES')
    } else {
        print('NO')
    }
}
check_name('pietro')
#> [1] "NO"
```

This function takes advantage of the standard return rule: a function returns the last value that it computed, thus there is no need to use the `return()` function.

The condition must evaluate to either a single `TRUE` or `FALSE`. If it is a logical vector, we will get a warning message, usually saying that only the first condition has been used; if it is an `NA`, we will get an error. Watch out for these messages in your own code:

```
if (c(TRUE, FALSE)) {}
#> NULL

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

You can use `||` (or) and `&&` (and) to combine multiple logical expressions. These operators are *short-circuiting*: as soon as `||` sees the first `TRUE` it returns `TRUE` without computing anything else. As soon as `&&` sees the first `FALSE` it returns `FALSE`. This greatly improves performances without causing damages. `|` or `&` should be never used in an `if` statement: these are vectorised operations that apply to multiple values.

You can chain multiple if statements together:

```
if (this) {
    # do this
} else if (that) {
    # do that
} else {
    # do something else
}
```

When we have a very long series of chained `if` statements, we can consider to use the `switch()` function. It allows us to evaluate selected code based on position or name.

```
function(x, y, op) {
    switch(op,
           plus = x + y,
           minus = x - y,
           times = x * y,
           divide = x / y,
           stop("Unknown op!")
    )
}
```

Another useful function that can often eliminate long chains of `if` statements is `cut()`. It is used to discretise continuous variables into categories or ranges of values.

### 1.3.4 Iterations

Every for loop has three components:

1. The **output**: Before starting the loop, we must always allocate sufficient space for the output. We will see in all our examples that beore a loop is

performed, a container is initilized. This is very important for efficiency: if we grow the for loop at each iteration using `c()` (for example), our for loop will be very slow; keep in mind that R is already slow in performing for loops compared to other programming languages, that is why, when possible, we should use vectorized function.

2. The **sequence**: This determines what to loop over. Usually in R we loop over indices of vector, such as `i in nrow(df)` to loop over dataframe's columns; the sequence is placed inside `(...)` following the `for` statement

3. The **body**: This is the part of the code that does the work. It's run repeatedly, each time with a different value of the **sequence**

There are some variations that it is important to be aware of. There are three basic ways to loop over a vector. So far we have addressed the most general: looping over the numeric indices with `for (i in 1:length(x))` or equivalently `for (i in seq_along(x)`, and extracting the value of `x` with `x[[i]]` at each iteration. There are two other forms:

1. Loop over the elements: `for (element in x)`

2. Loop over the names: `for (name in names(x))`. This gives us name, which we can use to access the value with `x[[name]]`

```
x <- c('a' = 1, 'b' = 2, 'c' = 3)

# indeces
for (i in seq_along(x)) {
    print(i)
}

# elements
for (i in x) {
    print(i)
}

# names
for (i in names(x)) {
    print(x[[i]])
}
```

Imagine we want to loop until we get a specific event. We cannot do that sort of iteration with the for loop since we do not know, ex-ante, how many iterations we

will need. Instead, we can use a while loop. A while loop is simpler than for loop because it only has two components, a condition and a body:

```
while (condition) {
    # body
}
```

A while loop is also more general than a for loop, because we can rewrite any for loop as a while loop, but you cannot rewrite every while loop as a for loop:

```
for (i in seq_along(x)) {
    # body
}

# Equivalent to
i <- 1
while (i <= length(x)) {
    # body
    i <- i + 1
}
```

Therefore, a while loop is a good choice when we want to find out how many iteration we need to fulfill a specific condition.

## 1.4 The tidyverse

Base R comes with excellent packages, but we also need to install some other R packages. An R **package** is a collection of functions, data, and documentation that extends the capabilities of base R. Using packages is key to the successful use of R. The majority of the packages that we will use in this book are part of the so-called tidyverse. The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.

You can install the complete tidyverse with a single line of code:

```
install.packages('tidyverse')
```

On your own computer, type that line of code in the console, and then press enter to run it. R will download the packages from CRAN and install them on to your computer. If you have problems installing, make sure that you are connected to

the internet, and that https://cloud.r-project.org/ is not blocked by a firewall or a proxy.

To use the functions, objects, and help files in a package until we must load it with `library()` or `require()`; as far as we are concerned we can consider these two ways of loading R packages equivalent. Once installed, load a package, or a collenction of them in this case, as follows:

```
library(tidyverse)
```

This tells you that tidyverse is loading the ggplot2, tibble, tidyr, readr, purrr, and dplyr packages. These are considered to be the **core** of the tidyverse because they are used in almost every analysis.

Packages in the tidyverse change fairly frequently. To check for updates, and optionally install them, run `tidyverse_update()`.

### 1.4.1 Tibbles

Throughout this book we work with "tibbles" instead of R's traditional `data.frame`. Tibbles *are* dataframes, but they tweak some older behaviours to make life a little easier. R is an old language, and some things that were useful 10 or 20 years ago now get in our way. It is difficult to change base R without breaking existing code, so most innovation occurs in packages. Here we will describe the **tibble** package, which "[...] provides opinionated dataframes that make working in the tidyverse a little easier"(Wickham and Grolemund 2017). We will use the term tibble and dataframe interchangeably.

Almost all of the functions that we will use in this book produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most other R packages use regular dataframes, so to coerce a dataframe to a tibble we can use the dplyr function `as_tibble()`:

```
as_tibble(iris)
#> # A tibble: 150 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
```

```
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # ... with 140 more rows
```

We can create a new tibble from individual vectors with `tibble()`. `tibble()` will automatically recycle inputs of length 1, and allows us to refer to variables that we just created, as shown below. For any practical purpose, we can consider the use of `tibble()` equivalent to `data.frame()`.

```
tibble(
    x = 1:5,
    y = 1,
    z = x ^ 2 + y
)
#> # A tibble: 5 x 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

Note that `tibble()` does much less than `data.frame()`: it never changes the type of the inputs (e.g. it never converts strings to factors!), it never changes the names of variables, and it never creates row names.

It is possible for a tibble to *have* column names that are not valid R variable names, known as **non-syntactic** names. For example, they might not start with a letter, or they might contain unusual characters like a space. To refer to these variables, we need to surround them with backticks, `` ` ``:

```
tb <- tibble(
    `:)` = "smile",
    ` ` = "space",
    `2000` = "number"
)
tb
#> # A tibble: 1 x 3
#>   `:)`  ` `   `2000`
#>   <chr> <chr> <chr>
#> 1 smile space number
```

We will also need the backticks when working with these variables in other packages, like ggplot2, dplyr, and tidyr.

Another way to create a tibble is with `tribble()`, short for **tr**ansposed tibble. `tribble()` is customised for data entry in code: column headings are defined by formulas (i.e. they start with `~`), and entries are separated by commas. This makes it possible to lay out small amounts of data in an easy-to-read form.

```
tribble(
    ~x, ~y, ~z,
    "a", 2, 3.6,
    "b", 1, 8.5
)
#> # A tibble: 2 x 3
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1 a         2   3.6
#> 2 b         1   8.5
```

There are two main differences in the usage of a tibble vs a classic `data.frame`: printing and subsetting.

**Printing**: Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `str()`. Tibbles are designed to not accidentally overwhelm the console when printing large dataframes. But sometimes we need more output than the default display. There are a few options that can help. First, we can explicitly `print()` the dataframe and control the number of rows (`n`) and the `width` of the display. `width = Inf` will display all columns:

```
mtcars %>% print(n = 10, width = Inf)
```

You can see a complete list of options by looking at the package help with `package?tibble`. A final option is to use RStudio's built-in data viewer to get a scrollable view of the complete dataset. This is also often useful at the end of a long chain of manipulations.

```
mtcars %>% View()
```

**Subsetting**: To pull out a single variable we can use `$` and `[[`. `[[` can extract by name or position; `$` only extracts by name but is a little less typing.

```
df <- tibble(
    x = runif(5),
    y = rnorm(5)
)

# Extract by name
df$x
#> [1] 0.0989923 0.7828605 0.6136371 0.9571959 0.9554693
df[['x']]
#> [1] 0.0989923 0.7828605 0.6136371 0.9571959 0.9554693

# Extract by position
df[[1]]
#> [1] 0.0989923 0.7828605 0.6136371 0.9571959 0.9554693
```

To use these in a pipe, we need to use the special placeholder `.`:

```
df %>% .$x
#> [1] 0.0989923 0.7828605 0.6136371 0.9571959 0.9554693
df %>% .[["x"]]
#> [1] 0.0989923 0.7828605 0.6136371 0.9571959 0.9554693
```

More on the pipe below. Compared to a `data.frame`, tibbles are more strict: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

### 1.4.2 Magrittr and the Pipe

Pipes are a powerful tool for clearly expressing a sequence of multiple operations. So far, we have been using them without knowing how they work, or what the alternatives are. Now, in this section, we will explore the pipe in more detail.

The pipe, `%>%`, comes from the **magrittr** package by Stefan Milton Bache. Packages in the tidyverse load only `%>%` for us automatically, not the entire package magrittr explicitly. Here, however, we are focussing on piping, and we are not loading any other package, so we will load it explicitly.

```
library(magrittr)
```

The point of the pipe is to help in writing code in a way that is easier to read and understand. To see why the pipe is so useful, we are going to explore a number

of ways of writing the same code. Let's use code to tell a story about a common Bocconi student daily routine:

> Wake up
> Have a coffee
> Have a shower
> Sit at the desk
> Study Sleep

We start by defining an object to represent the Bocconi student:

```
student <- student()
```

And we use a function for each key verb: `wake_up()`, `have_coffee()`, `shower()`, `sit()`, `study()`, and `sleep()`. Using this object and these verbs, there are (at least) four ways we could retell the story in code:

1. Save each intermediate step as a new object
2. Overwrite the original object many times
3. Compose functions
4. Use the pipe

We will implement each approach, showing the code and talking about the advantages and disadvantages.

The simplest approach is to save each step as a new object:

```
student_1 <- wake_up(student)
student_2 <- have_coffee(student_1)
student_3 <- shower(student_2)
student_4 <- sit(student_3)
student_5 <- study(student_4)
student_6 <- sleep(student_5)
```

The main downside of this form is that it forces us to name each intermediate element. That leads to two problems:

- The code is cluttered with unimportant names
- We have to carefully increment the suffix on each line

Instead of creating intermediate objects at each step, we can overwrite the original object:

```
student <- wake_up(student)
student <- have_coffee(student)
student <- shower(student)
student <- sit(student)
student <- study(student)
student <- sleep(student)
```

There are two problems with this approach:

- Difficult debugging: if we make a mistake we will need to re-run the complete pipeline from the beginning
- The repetition of the object being transformed (we have written `student` 12 times!) obscures what is changing on each line.

Another approach is to abandon assignment and just string the function calls together:

```
sleep(
    study(
        sit(
            shower(
                have_coffee(
                    wake_up(student)
                )
            )
        )
    )
)
```

Here the disadvantage is that we have to read from inside-out, from right-to-left, and that the arguments end up spread far apart. In short, this code is hard for a human to consume.

Finally, we can use the pipe:

```
student %>%
    wake_up() %>%
    have_coffee() %>%
    shower() %>%
    sit() %>%
    study() %>%
    sleep()
```

This form focusses on verbs, not nouns. We can read this series of function compositions like it is a set of imperative actions. The downside, of course, is that we need to be familiar with the pipe. The pipe works by performing a "lexical transformation": behind the scenes, magrittr reassembles the code in the pipe to a form that works by overwriting an intermediate object. When we run a pipe like the one above, magrittr does something like this:

```
my_pipe <- function(.) {
    . <- wake_up(.)
    . <- have_coffee(.)
    . <- shower(.)
    . <- sit(.)
    . <- study(.)
    sleep(.)
}
my_pipe(student)
```

### 1.4.3 Other packages

There are many other excellent packages that are not part of the tidyverse because they solve problems in a different domain or are designed with a different set of underlying principles. This does not make them better or worse, just different.

## 1.5 Running R code

The previous sections showed a couple of examples of running R code. Code in the book looks like this:

```
1 + 2
#> [1] 3
```

There are two main differences. In the usual R console, we type after the `>`, called the **prompt**; we do not show the prompt in the book. Furthermore, in the book the output is commented out with `#>`; in the console it appears directly after the code.

## 1.6 Python, Julia, and friends

In this book, we will not learn anything about Python, Julia, or any other programming language useful for data science. This is not because we think these

tools are bad. They are not! And in practice, most data science teams use a mix of languages, often at least R and Python.

However, we strongly believe that it is best to master one tool at a time. We think R is a great place to start any data science journey because it is an environment designed from the ground up to support data science. R is not just a programming language, but it is also an interactive environment for doing data science. To support interaction, R is a much more flexible language than many of its peers, like Python. This flexibility comes with its downsides, but the big upside is how easy it is to evolve tailored grammars for specific parts of the data science process.

# Chapter 2

# Frequentist Inference Concepts

---

*In this chapter we will implement stochastic simulations to empirically validate some simple theoretical propositions. We will do this exploiting the computational capabilities of R. In particular, this chapter will focus on 3 theoretical concepts: unbiasedness, consistency, and the Central Limit Theorem (CLT) in the context of estimators. We will also discuss Maximum Likelihood Estimation.*

---

## 2.1 Introduction

Statistics is the science of learning from experience, particularly experience that arrives a little bit at a time (Efron and Hastie 2016). Statistical inference usually begins with the assumption that some probability model, the "true" data generating process, has produced the observed data $x = (x_1, ..., x_n)$. Indicate with $X = (X_1, ..., X_n)$ the $n$ independent draws from the unknown probability distribution, i.e. our model, $\mathcal{P} = \{P_\theta, \theta \in \Theta\}$. Once a realization $X = x$ has been observed, the statisticians want to infer some property (or properties) of the unknown distribution $P_\theta$, parametrised by $\theta$. In other words, they want to use the data to "discover" the "hidden" properties of the data generating process.

Various approaches to probability lead to different inferencial "philosophies". The **frequentist inference**, the one treated in this book, is associated with the frequentist interpretation of probability. The frequentist approach to probability is based on the possibility of repeating an experiment under equivalent conditions,

producing statistically independent results (Castagnoli, Cigola, and Peccati 2009). The probability is intended as the *frequency* of success over the $n$ experiment performed – to refer to the above paragraph, we may think of the experiment to be the $n$ independent draws from the probability distribution.

The probability of the event of interest $A$, labelled "success", is the ratio:

$$f_n(A) = \frac{\text{successes in } n \text{ experiments}}{n}$$

and is called *frequency* of success over $n$ experiments. Obviously, $f_n(A)$ fluctuates when $n$ changes. The "empirical law of chance" states that as $n$ gets bigger, the frequency becomes more stable, suggesting the existence of the limit that defines the concept of probability in the frequentist framework:

$$P(A) = \lim_{n \to \infty} f_n(A)$$

Empirically, however, infinite experiments (samples) are not possible, but the issue is solved by accepting that for $n$ sufficiently large $f_n(A)$ is a good approximation for $P(A)$.

More generally, we are interested in **understanding** certain features of a reference population – as opposed to *predicting* – based only on a sample (or samples): it is like inferring the objects in a photo based on the few pixels we are able to look at. Building a statistical model serves the purpose of generalizing from small data in a principled way. Having incomplete information, we are forced to make some assumptions. Within the realm of *parametric* statistics, we usually assume that the true data-generating process can be well described by some probability distribution $P_\theta$ parametrized by some parameters $\theta$. Therefore, given that the parametric model we impose on the data is correct, our aim is to "infer" the correct value of $\theta$. Since we are only able to look at a sample (or samples) from the population, inference based on different samples brings to different inferences. Therefore, we also need a machinery to compute the uncertainty of our inference. Importantly, the frequentist approach to statistics assumes that in nature there exists a "true" **fixed** value of $\theta$. If we were able to collect all the information from the population then there would not be any uncertainty. Uncertainty, thus, arises only from the fact that our information set is limited. We will clarify with an example below.

### 2.1.1 Example: the Normal population

We are interested in modelling the height of Italian citizens. First, we need to assume a parametric model that would describe the true data-generating process

well. Since height is a continuous variable, a good parametric family of distributions to describe the phenomenon is the Normal distribution which models variable with support in $\mathbb{R}$[1]

$$\mathcal{N}\left(\mu,\, \sigma^2\right)$$

The problem is that there exists an infinite number of distribution in this family depending on the values of the parameters $(\mu, \sigma^2)$[2].

Suppose we collect data from $n = 6$ individuals randomly selected from all around Italy. We assume the observation are *identically distributed*, namely they come from the same data generating process. Furthermore, the random sampling assures they are also independent

$$X_1, \ldots, X_6 \overset{i.i.d.}{\sim} \mathcal{N}\left(\theta, \sigma^2\right)$$

In `R`, the data we get to observe are stored as a `data.frame` or as its modern counterpart: the `tibble`. Throughout this book we work with "tibbles" instead of R's traditional dataframes. In most places, we will use the term tibble and dataframe interchangeably; to draw particular attention to R's built-in dataframe, we will refer to them as `data.frame`s.

To estimate $\theta$ we would repeatedly extract a sample of size $n$ from the population and take its mean as a guess for the unknown parameter. That is, we compute a function of the data $T(X) = T(X_1, \ldots, X_6) = \hat{\theta}$. The function $T$ is our **estimator** that once "fed" with the data computes the estimate – it is an algorithm that computes the values of interest. In this particular case, our algorithm to estimate $\theta$ is $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$, the sample mean. This is a random variable because depending on the sample we apply this algorithm to its value changes. Thus, **repeated sampling** will give us different values of $\bar{X}$ and from the different results we can compute its *sampling distribution*.

We exploit R to do the computation without fatigue. Firstly, let's pretend we are the Natural hidden force that has produced the data: we know that $\theta = 170$ and $\sigma^2 = 25$. We produce $m = 1000$ i.i.d. samples, each of $n = 6$ observation. The function `rnorm()` produces a sample from the normal distribution[3].

Let's build the tibble with the data of 1000 samples, each with 6 observation

```
library(tidyverse)
```

[1] Well, let's pretend it is: this assumption is not truly justified since negative heights should have zero probability.

[2] Recall that for the Normal model $\mu \in (-\infty, +\infty)$ and $\sigma^2 \in [0, +\infty)$.

[3] Be careful: the function takes the standard deviation as input, not the variance.

```
n <- 6
m <- 1000
mu <- 170
sigma2 <- 25

dataset <- tibble(Individual = 1:6)

for (i in 1:m) {
    sample_name = paste0('sample_', i)
    dataset[sample_name] <- rnorm(n = 6, mean = 170, sd = sqrt(sigma2))
}

dataset[1:5] %>%
    knitr::kable(booktabs = T, align = 'c')
```

| Individual | sample_1 | sample_2 | sample_3 | sample_4 |
|---|---|---|---|---|
| 1 | 167.293 | 159.898 | 180.442 | 168.289 |
| 2 | 174.277 | 170.809 | 174.411 | 177.681 |
| 3 | 172.458 | 167.534 | 175.666 | 170.130 |
| 4 | 171.783 | 176.110 | 158.041 | 173.089 |
| 5 | 166.290 | 173.297 | 168.154 | 169.320 |
| 6 | 167.807 | 163.077 | 175.875 | 170.193 |

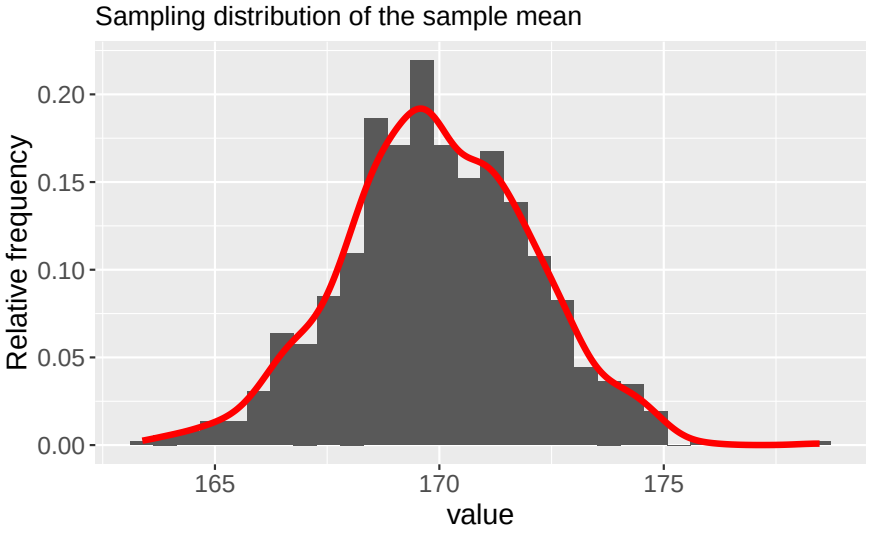
Detailed explanation of the steps:

- Load the required packages
- Set $n$ and $m$
- Set the true values of $\mu$ and $\sigma^2$
- Initialize the tibble with just one column referring to individuals' IDs
- For each iteration:
  - Create the column name of the new column of values of the i-th sample
  - Randomly generate observation from the normal distribution
- For space reasons print only 5 columns of the tibble

Now we come back being the humble statistician willing to discover the inner secrets of Nature. Suppose we observe the $m$ samples of size $n$ – to have $m$

samples is the core idea of repeated sampling. Since we want to estimate the mean parameter, as discussed above, we use the estimator $\bar{X}$. For each sample $m$ we would have a different value of $\bar{X}_n^{(m)}$. After having computed all these values we can study their *empirical distribution*

```
dataset %>%
    select(-Individual) %>%
    map_df(mean) %>%
    gather() %>%
    ggplot(aes(x = value)) +
        geom_histogram(aes(y = ..density..)) +
        geom_line(stat = 'density', colour = 'red', size = 1.5) +
        labs(title = 'Sampling distribution of the sample mean',
             y = 'Relative frequency')
```




Detailed explanation of the steps:

- Exclude column `Individuals` from the selection, select all the others
- Compute the mean by column and return a tibble
- Reshape the tibble from a $1 \times m$ vector to a $m \times 1$ vector for plotting purposes
- Plot the histogram were each bar represents the relative frequency of a small set of values

Based on the idea of repeated sampling we hope that our one sample mean would provide us with some information – consider that we get to observe only one sample, i.e. $m = 1$! It is possible to show that the empirical sampling distribution we have computed and plotted above will tend to some probability density function as the number of samples $m$ grows

$$\bar{X}_n \sim \mathcal{N}\left(\theta, \frac{\sigma^2}{n}\right)$$

Note that $E(\bar{X}) = \theta$.

In the previous example we stored the samples in the columns of a tibble called `dataset` for clarity. With the number of simulation, $m$, increasing it is not efficient to keep all the simulated data and then compute a statistic per simulation. It is more efficient to create a sample, store it into a vector, compute the statistic and throw the vector away keeping in memory only the estimate. This will be the approach henceforth.

## 2.2 Estimators

Suppose that the distribution of $X$ depends on an unknown parameter, so that the statistical model is of the form $\mathcal{P} = \{P_\theta, \theta \in \Theta\}$ for $P_\theta$ the distribution of $X$ if $\theta$ is the true parameter. Based on an observation $x$, we want to estimate the true value of $\theta$, or perhaps the value of a function $g(\theta)$ of $\theta$. Formally,

> **Estimator**: An estimator or statistic is a random vector $T(X)$ that depends only on the observation $X = x$. The corresponding estimate for an observation $x$ is $T(x)$

By this definition, many objects are estimators. What matters is that $T(X)$ is a function of $X$ that does not depend on the parameter $\theta$: we must be able to compute $T(x)$ from the data $x$. Given the observed value $x$, the statistic $T$ is realized as $t = T(x)$, which is used as an estimate of $\theta$ (or $g(\theta)$). Before going into the details of the computations and characteristics of estimators, we define a framework that enable us to compare various estimators.

### 2.2.1 Mean Squared Error

Although every function of the observation is an estimator, not every estimator is a good one. A good estimator of $\theta$, or $g(\theta)$, is a function $T$ of the observed

data such that $T$ is "close" to the estimand, $\theta$ or $g(\theta)$. We could think about a distance measure, informally defined $d$, such that we evaluate the goodness of the estimator by this measure $d(T, g(\theta))$. However, such measure poses two ploblems:

- It depends on $\theta$, which is unknown
- It is stochastic and cannot be computed before seeing the data

To avoid the second difficulty, we can consider the *distribution* of this distance under the assumption that $\theta$ is the true value. We thus look for an estimator whose distribution for the true value is concentrated as much as possible aroud $g(\theta)$ or, equivalently, the distribution of $d(T, g(\theta))$ is concentrated around 0.

It is useful to express concentration as a number. One meausure of concentration is the

**Mean Squared Error**: the MSE of an estimator $T$ for $g(\theta)$ is

$$\mathrm{MSE}(\theta; T) = \mathrm{E}_\theta \|T - g(\theta)\|^2$$

The expectation depends on $\theta$, that is the subscript is essential since the MSE is the expected square deviation of $T$ from $g(\theta)$ under the assumption that $\theta$ is the true value of the parameter.

The first difficulty, i.e. the fact that the measure of quality depends on $\theta$, has not been addressed yet. The mean square error is a function of $\theta$. In principle, it suffices if MSE is as small as possible in the true value of the parameter. As we do not know that value, we try to keep the mean square error (relatively) small for *all* values of $\theta$ at once: hence, the expectation.

The MSE can be decomposed into a *bias* and *variance* component:

$$\mathrm{MSE}(\theta; T) = \mathrm{Var}_\theta(T) + \left[\underbrace{\mathrm{E}_\theta(T) - g(\theta)}_{\text{bias}}\right]^2$$

The second term in the decomposition is therefore the square of the bias. The standard deviation $\mathrm{std}_\theta(T) = \sqrt{\mathrm{Var}_\theta(T)}$ of an estimator is also called the *standard error*. This should not be confused with the standard deviation of the observations. In principle, the standard error depends on the unknown parameter and is therefore itself an unknown quantity.

For an unbiased estimator, the bias term is identically 0. This seems very desirable, but it is not always the case. Namely, the condition that an estimator is

unbiased can lead to the variance being very large. In general, a small variance leads to a large bias, and a small bias to a large variance. We must therefore balance the two terms against each other. It is important to note that, since the bias of reasonable estimators is usually small, the standard error often gives an idea of the quality of the estimator. An estimate of the standard error is often given along with the estimate itself.

The criteria is readily served: between two estimators choose the one with smallest MSE or, equivalently, with smallest standard error and smallest bias.

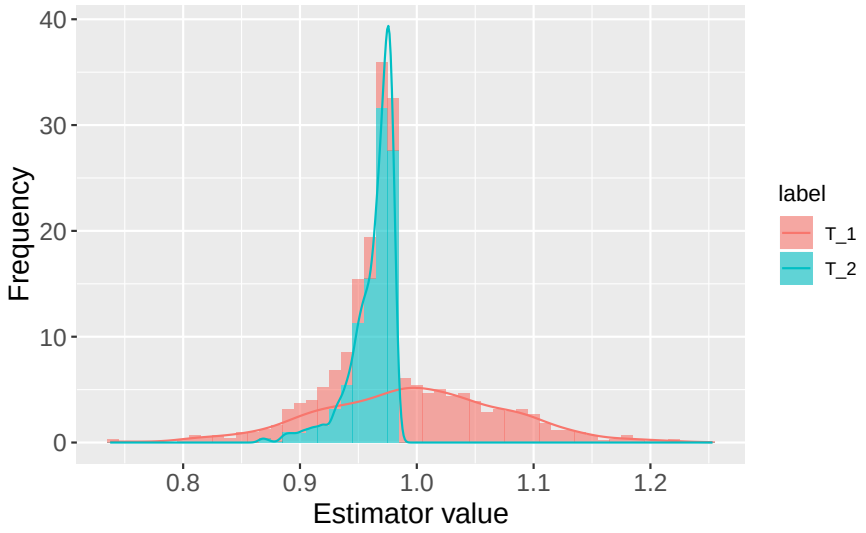
### 2.2.2 Example: the Uniform distribution

Let $X_1, \ldots, X_n$ be independently distributed random variables $U[0, \theta]$. The i.i.d. observations are stored in the vector $X = (X_1, \ldots, X_n)$, and we want to estimate the unknown $\theta$, the upper bound of the interval. Since $E_\theta X_i = \frac{1}{2}(\theta - 0)$, it is reasonable to estimate $\frac{1}{2}\theta$ using the sample mean $\bar{X}$ and $\theta$ using $2\bar{X}$. Suppose that $n = 10$ and that the data have the following values: $x = (3.03, 2.70, 7.00, 1.59, 5.04, 5.92, 9.82, 1.11, 4.26, 6.96)$, so that $2\bar{x} = 9.49$. This estimate is certainly too small. Indeed, one of the observations is 9.82, so that we must have $\theta \geq 9.82$. We can avoid the problem we just mentioned by taking the maximum $X_{(n)}$ of the observations. However, the maximum is certainly also less than the true value, for all observations $x_i$ lying in the interval $[0, \theta]$. An obvious solution is to add a small correction. We could, for example, take $(n+2)/(n+1)X_{(n)}$ as estimator.

So there are several candidates. Which estimator is the best? To gain some insights into this question, we carry out the following simulation. Suppose the true $\theta$ is equal to 1. We choose $n = 50$ and simulated $m = 1000$ i.i.d. samples from the uniform distribution on $[0, 1]$. We compute the estimators $T_1 = 2\bar{X}$ and $T_2 = (n+2)/(n+1)X_{(n)}$ and compare their MSE.

```
n <- 50
m <- 1000
theta <- 1

t_1 <- c()
t_2 <- c()

for (i in 1:m) {
    sample <- runif(n, 0, theta)
    t_1 <- c(t_1, 2*mean(sample))
    t_2 <- c(t_2, ((n+1)/(n+2))*max(sample))
```

```
}

simulation_data <- tibble(estimators = c(t_1, t_2),
                          label = c(rep('T_1', m), rep('T_2', m)))

simulation_data %>%
    ggplot(aes(x = estimators, fill = label)) +
        geom_histogram(aes(y = ..density..),
                       alpha = 0.6,
                       binwidth = 0.01) +
        geom_line(aes(colour = label), stat = 'density') +
        labs(x = 'Estimator value',
             y = 'Frequency')
```



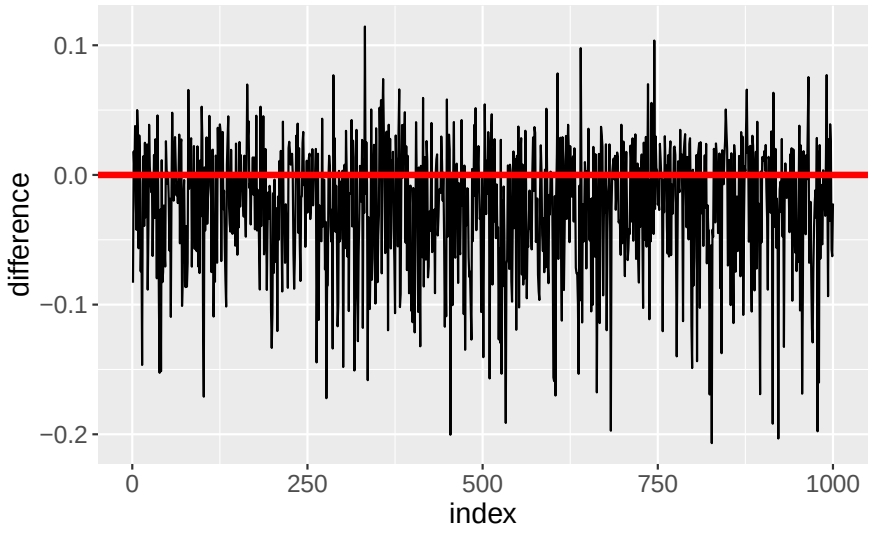

Detailed explanation of the steps:

- Define $n$, $m$, and $\theta$
- Initialize the vectors that will contain the estimates
- Generate samples from the uniform distribution
- Compute the two statistics and store the values into the container defined at the previous step; our notation `c(old_value, new_value)` simply appends to the end of the vector the new value

- Create a 2-column tibble containing the $2m$-dimansional vector of labels $[T_1, \ldots, T_1, T_2, \ldots, T_2]$ and the $2m$-dimansional vector of estimates $[t_1^{(1)}, \ldots, t_1^{(m)}, t_2^{(1)}, \ldots, t_2^{(m)}]$
- Plot the histogram by label where each bar represents the relative frequency of a small set of values

Note that it is not true that the estimator $(n+2)/(n+1)X_{(n)}$ gives the best estimate on each of the 1000 samples. This can be seen in the next figure, where the difference $|(n+2)/(n+1)x_{(n)} - 1| - |2x - 1|$ is set out along the vertical axis. This kind of check is taken from Bijma, Jonker, and Vaart (2017). In general, this difference is negative, but sometimes it is positive, in which case the estimator $2\bar{X}$ gives a value that is closer to the true value $\theta = 1$. Because in practice we do not know the true value, we must use the estimator that is the best *on average*.

```
tibble(difference = abs(t_2 - theta) - abs(t_1 - theta),
       index = 1:m) %>%
    ggplot(aes(x = index, y = difference)) +
        geom_line() +
        geom_hline(yintercept = 0, colour = 'red', size = 1.5)
```




Detailed explanation of the steps:

- Create a tibble with two column: the index representing simply the position in the vector of the values and differences

- Plot these differences

Our simulation experiment only shows that $(n+2)/(n+1)X_{(n)}$ is the better estimator if the true value of $\theta$ is equal to 1. To determine which estimator is better when $\theta$ has a different value, we would have to repeat the simulation experiment with simulated samples from the uniform distribution on $[0, \theta]$, for every $\theta$. This is not something very smart, of course, and that is one of the reasons to study estimation problems mathematically. Another problem with the simulation approach is that instead of ordering pairs of estimators, we would like to find the overall best estimator amongst many. This is the reason we introduced a framework to let us compare multiple estimators without performing any simulation. Let's compute the MSE for the two estimators using the approximation

$$\mathrm{E}_\theta \left[T - g(\theta)\right]^2 \approx \frac{1}{m} \sum_{i=1}^{m} \left(t_n^{(i)} - g(\theta)\right)^2$$

```
simulation_data %>%
    mutate(summand = (estimators - theta)^2) %>%
    group_by(label) %>%
    summarise(mse = mean(summand)) %>%
    knitr::kable(booktabs = T, align = 'c')
```

| label | mse |
|---|---|
| T_1 | 0.006384 |
| T_2 | 0.001790 |

Detailed explanation of the steps:

- Recall the tibble `simulation_data` created above
- Add a new column called `summand` by subtracting $\theta$ from each value (this is called braodcasting: when subtracting a scalar from a vector, R automatically performs the computation elementwise)
- Group by label the computation that will follow
- Compute the mean by group, that is compute the MSE
- Pretty printing of the table

The results above confirm that, on average, estimator $(n+2)/(n+1)X_{(n)}$ performs better than $2\bar{X}$ in MSE sense.

## 2.3 From practise to theory

Using R we can test empirically if some theoretical facts holds. The facts we want to check are:

- Unbiasedness
- Consistency
- Central Limit Theorem

### 2.3.1 Unbiasedness

Suppose we have a sample of size $n$ from a parametric distribution $f_X(x;\theta)$, $X_1, \ldots, X_n \overset{i.i.d.}{\sim} f_X(x;\theta)$. We want to learn about $\theta$ from the data. In order to do this, we use the data (a function of them) to estimate the unknowns. In general, we write this function of the data as $T = T(X_1, \ldots, X_n)$. Obviously, this function of the data will have its own distribution because if the data change, its value changes too – the idea of the *sampling distribution*. So $T \sim f_T(t;\theta)$. Now, we can define

> **Unbiasedness**: T is an unbiased estimator for $g(\theta)$ if $E(T) = g(\theta)$, for any function $g$.

#### 2.3.1.1 Example: Sample Mean

Let's go back to the Normal example and prove empirically that the expected value of the sample mean equals the true mean, that is

$$\mathrm{E}\left(\bar{X}_n\right) = \theta$$

To verify empirically that the above holds, let's verify that the approximation

$$\mathrm{E}\left(\bar{X}_n\right) \approx \frac{1}{m}\sum_{i=1}^{m} \bar{X}_n^{(i)}$$

works. Recall that we simulated $m = 1000$ sample of size $n$ from a Normal population. Therefore, we will compute the mean for each sample, $\bar{X}_n^{(i)},\ i = 1, \ldots, m$, obtaining an $m$-dimensional vector of means. Afterwards, we will compute the mean across these $m$ sample means to obtain an approximation of the expectation above. Hopefully, we will find a value in a small neighbourhood of 170, the true expected value of the distribution.

```
dataset %>%
    select(-Individual) %>%
    map_df(mean) %>%
    gather() %>%
    extract2(2) %>%
    mean()
#> [1] 169.958
```

Description of the steps:

- Recall the tibble created above containing values from 1000 samples
- Select every column except `Individuals` and compute the mean for each column
- Reshape the tibble so that the first column contains the simulation number and the second column contains all means stacked as a column vector
- Extract this second column
- Compute the mean across all samples

Indeed, the value is close to the true expectation, i.e. as the number of repeated samples grows, the approximation will be closer and closer to the true value.

#### 2.3.1.2 Example: Sample Variance

Let's continue with the Normal example and evaluate empirically two different estimator of the variance

$$\tilde{S}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2$$

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2$$

We know that

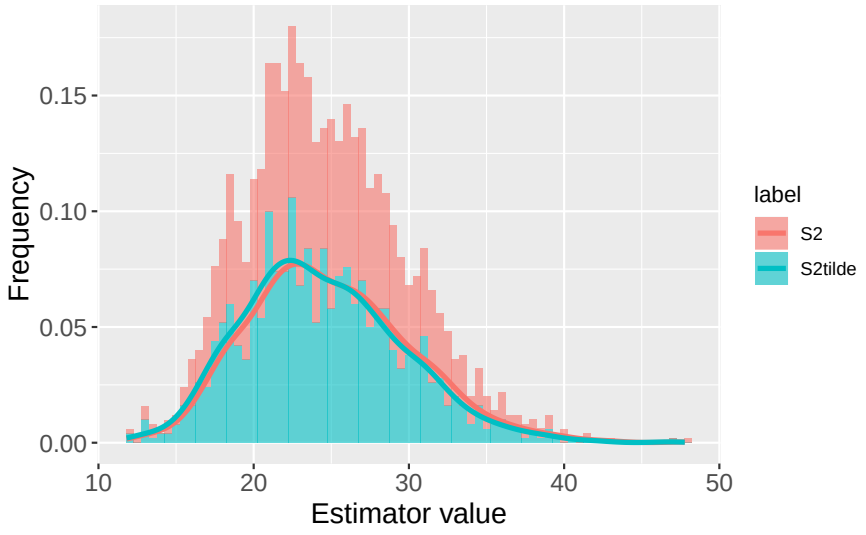
$$\mathrm{E}\left(S^2\right) = \sigma^2$$

that is $S^2$ is unbiased, while

$$\mathrm{E}\left(\tilde{S}^2\right) = \frac{n-1}{n}\sigma^2$$

thus it is biased. Let's check which one is the best in the MSE sense

```
s2 <- c()
s2tilde <-c()

for (i in 1:m) {
    sample <- rnorm(n, mu, sqrt(sigma2))
    sample_mean <- mean(sample)
    summand <- sum((sample - sample_mean)^2)
    s2 <- c(s2, (1/(n-1))*summand)
    s2tilde <- c(s2tilde, (1/n)*summand)
}

var_simulation_data <- tibble(estimators = c(s2, s2tilde),
                              label = c(rep('S2', m), rep('S2tilde', m)))

var_simulation_data %>%
    ggplot(aes(x = estimators, fill = label)) +
    geom_histogram(aes(y = ..density..),
                   alpha = 0.6,
                   binwidth = 0.5) +
    geom_line(aes(colour = label), stat = 'density', size = 1.2) +
    labs(x = 'Estimator value',
         y = 'Frequency')
```




Detailed explanation of the steps:

- Reuse $n$, $m$, $\mu$, and $\sigma^2$ defined above

- Initialize the vectors that will contain the estimates
- Generate samples from the normal distribution
- Compute the two statistics and store the values into the container defined at the previous step
- Create a 2-column tibble containing the $2m$-dimansional vector of labels $[S^2, \dots, S^2, \tilde{S}^2, \dots, \tilde{S}^2]$ and the $2m$-dimansional vector of estimates $[s^2_{(1)}, \dots, s^2_{(m)}, \tilde{s}^2_{(1)}, \dots, \tilde{s}^2_{(m)}]$
- Plot the histogram by label where each bar represents the relative frequency of a small set of values

Using the usual approximation for the expectation, let's compare the MSE of the two estimators and their bias and variance

```
var_simulation_data %>%
    mutate(summand = (estimators - sigma2)^2) %>%
    group_by(label) %>%
    summarise(mse = mean(summand),
              bias = mean(estimators) - sigma2,
              variance = sd(estimators)^2) %>%
    knitr::kable(booktabs = T, align = 'c')
```

| label | mse | bias | variance |
|---|---|---|---|
| S2 | 27.1541 | -0.051640 | 27.1786 |
| S2tilde | 26.3794 | -0.550607 | 26.1023 |

We can see that in this case the MSE is slighlty better for $\tilde{S}^2$. However, note how much the bias of $S^2$ is smaller than the bias of $\tilde{S}^2$: almost 10 times so!

### 2.3.2 Consistency

When simulating, we can move along two dimensions:

- Varying the number of samples $m$
- Varying the sample size $n$

In this section, we fix $m = 1$, that is we confront reality: although fascinating, the idea of repeated sampling is not in general feasible. Usually, we have only one sample and we use the *one* value $\bar{X}_n$ as our guess for $\theta$.

Another good "quality", perhaps the most important, that an estimator $T$ should have is

> **Consistency**: An estimator $T$ is consistent for $g(\theta)$ if $T \xrightarrow{P} g(\theta)$, for any $g$, that is if $T$ converges in probability to $g(\theta)$.

Convergence in probability could be formally expressed as follows

> **Convergence in Probability**: A sequence of random variables $T_n$, with $n \geq 1$, is said to converge in probability to a number $\theta$ as $n \to \infty$ if and only if for any $\varepsilon \geq 0$,
>
> $$P(T \notin [\theta - \varepsilon, \theta + \varepsilon]) \xrightarrow{n \to \infty} 0$$
>
> or, equivalently
>
> $$P(|T - \theta| > \varepsilon) \xrightarrow{n \to \infty} 0$$

We can prove consistency indirectly, proving convergence in quadratic mean.

> **Convergence in quadratic mean**: A sequence of random variables $T_n$, with $n \geq 1$, is said to converge in quadratic mean to a number $\theta$ if and only if
>
> $$\lim_{n \to \infty} \mathrm{E}(T_n) = \theta$$
> $$\lim_{n \to \infty} \mathrm{Var}(T_n) = 0$$

Below we will show, using the usual approximations for the expectations, that this is the case for the sample mean of a normally distributed sample. In other words, we will show that the sample mean converges in quadratic mean to the true mean as the sample size grows.

#### 2.3.2.1 Example: Sample mean

In principle we would write a nested loop, something like

```
# pseudo-code
sizes = (10, 20, ..., 10000)
m = 100

for n in sizes:
    for i in 1:m:
        generate sample of size n
        compute sample mean
        store value of sample mean
    compute mean of the m sample means obtained in the previous loop
    store value of the mean
    compute variance of the m sample means
    store value
end loop

plot everything
```

However, R is not efficient in performing loops! Therefore, make the code run faster we have to use a little bit of theory to come up with an idea.

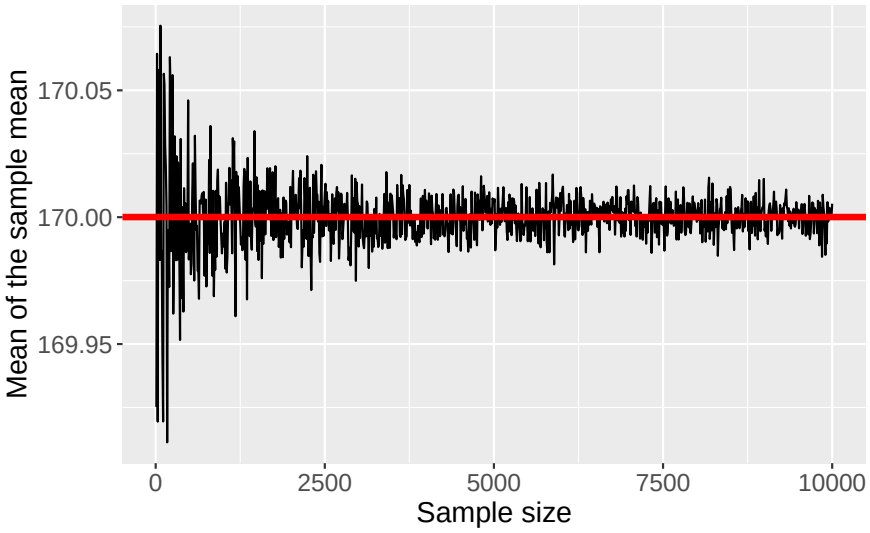
In particular, we will generate for each $n$ a sample of size $n \times m$, and then we will compute its mean:

$$\frac{1}{m}\sum_{j=1}^{m}\left[\frac{1}{n}\sum_{i=1}^{n}X_i\right]_j = \frac{1}{nm}\sum_{i=1}^{nm}X_i$$

and its variance. In other words, for each sample size $n$, we generate $m$ sample of that size. This will give us $m$ sample means. Computing the mean of these sample means approximates the quantity $\mathrm{E}(\bar{X}_n)$.

```
mu <- 170
sigma2 <- 25
m <- 10^2
sizes <- seq(from=10, to=10^4, by = 10)

sample_mean <- c()
var_sample_mean <- c()

for(n in sizes) {
    sample <- rnorm(n*m, mu, sqrt(sigma2))
```

```
    sample_mean <- c(sample_mean, mean(sample))
    var_sample_mean <- c(var_sample_mean, sd(sample)/n*m)
}

tibble(size = sizes, sample_mean = sample_mean) %>%
    ggplot(aes(x = size, y = sample_mean)) +
        geom_line() +
        geom_hline(yintercept = mean(sample_mean),
                   colour = "red",
                   size = 1.5) +
        labs(x = "Sample size",
             y = "Mean of the sample mean")
```

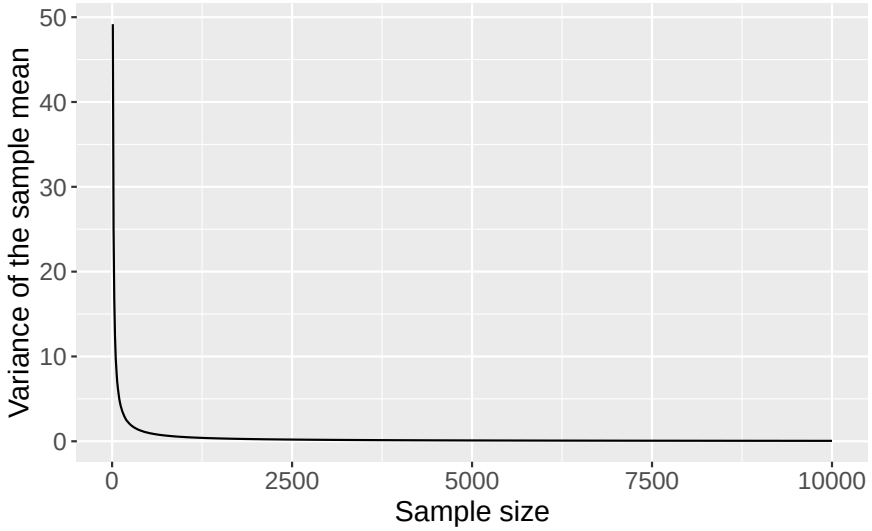



Description of the steps:

- Create a tibble where each row refers to the sample size for which we perform the experiment
- Create an empty vector to store the values of the sample mean for each sample size used
- Randomly generate iid samples from the true normal distribution, each time of different size
- Add the vector of means into the dataframe `samples`
- Plot the resulting sample means associated to the respective sample sizes

Indeed we see that as the sample size grows the mean of the sample mean is more concentrated around the true value.

```
tibble(size = sizes, var_sample_mean = var_sample_mean) %>%
    ggplot(aes(x = size, y = var_sample_mean)) +
        geom_line() +
        labs(x = "Sample size",
             y = "Variance of the sample mean")
```



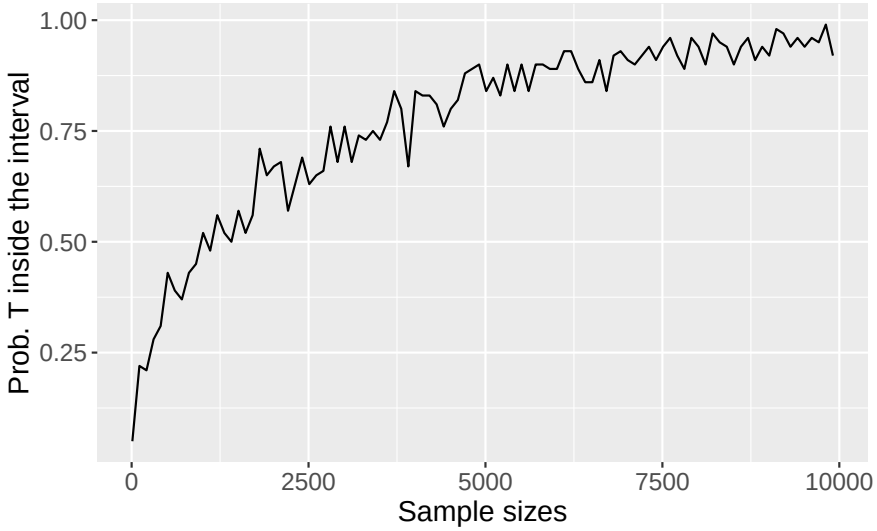

Description of the steps:

- Reuse the tibble `sample` where each row of `size` refers to the sample size for which we perform the experiment
- Create an empty vector to store the values of the variance of the sample mean for each sample size used
- Randomly generate iid samples from the true normal distribution, each time of different size
- Add the vector of variances into the dataframe `samples`
- Plot the resulting sample means associated to the respective sample sizes

Let's prove that the sample mean is consistent using the definition. That is, we draw a sample of size $n$, $m$ times for each sample size $n$, so that we obtain an

$m$-dimensional vector of sample means computed on a sample of size $n$. In this way, for each sample size $n$ we can count how many times the sample mean falls into the neighbourhood of the true value.

```
epsilon <- 0.1
mu <- 170
sigma2 <- 25
m <- 10^2
sizes <- seq(from=10, to=10^4, by = 100)

perc_inside <- c()

for(n in sizes) {
    sample_mean <- c()

    for (i in 1:m) {
        sample <- rnorm(n, mu, sqrt(sigma2))
        sample_mean <- c(sample_mean, mean(sample))
    }

    inside_interval <- abs(sample_mean - mu) < epsilon
    perc_inside <- c(perc_inside,
                     sum(inside_interval)/length(inside_interval))
}

tibble(size = sizes, perc_inside = perc_inside) %>%
    ggplot(aes(x = sizes, y = perc_inside)) +
        geom_line() +
        labs(y = 'Prob. T inside the interval',
             x = 'Sample sizes')
```




Description of the steps:

- Fix $\varepsilon \geq 0$, $n$, $m$, $\mu$, and $\sigma^2$
- Initialize the container for the probabilities of being inside the interval
- For each sample size $n$ in the vector of sample sizes, `sizes`, perform the following steps:
  - Initialize the containers for the sample mean that will be overwritten at each iteration for more efficient memory usage
  - Generate $m$ samples of size $n$
  - Compute the mean of the sample
- Evaluate the logical condition $|T_n - \theta| < \varepsilon$ for each component of the vector of sample means
- Since `TRUE` in R is stored as a 1, summing the component of the vector will give us the total number of sample means, computed on a sample of size $n$, in the vector that are inside the interval; dividing for the total number of sample means in the vector gives us the proportion of sample means inside the interval
- Plot the results

What we observe is that, indeed, as the sample size increases, the probability of the sample mean to be in a neightbourhood of the true value increases. Thus, this experiment confirms the results obtained above.

### 2.3.3 Central limit theorem

The CLT establishes that when independent random variables are added, their properly normalized sum tends toward a normal distribution even if the original variables themselves are not normally distributed. The theorem is a key concept in probability theory because it implies that probabilistic and statistical methods that work for normal distributions can be applicable to many problems involving other types of distributions; that is why the normal distribution is such an important topic in each statistics course.

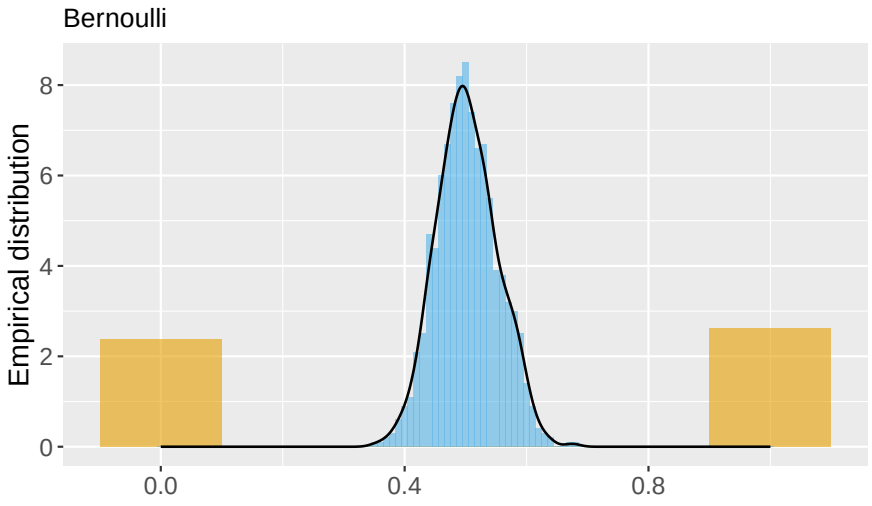
In other words, CLT is a statement about the **shape** of the distribution. A normal distribution is bell shaped so the shape of the distribution of sample means begins to look bell shaped as the sample size increases even if the samples are drawn from an arbitrary distribution.

To empirically validate such fact, we will simulate data from the bernoulli distribution, $Ber(p)$. Recall that a bernoulli distribution describes a random experiment with exactly two possible outcomes, "success" and "failure", in which the probability of success is the same every time the experiment is conducted. If we sample from the bernoulli distribution $n$ times, e.g. $n = 10$, it would look like $X = (0, 1, 1, 1, 1, 0, 1, 1, 1, 0)$, where 1 stands for "success". We will compute their sample means and see how they behave when $n$ grows. Our goal is to make inference on the parameter $p$, the probability of "success" which is also the expected value of a bernoulli random variable, that is if $X_i \sim Ber(p)$ then $E(X_i) = p$. Suppose the true $p$ is $1/2$.

In R we use the function `rbinom(n, size, prob)` with `n` being the number the sample size; `size` can be ignored since it is used to define the binomial distribution; and `prob` being the probability of success in each experiment. Therefore to have a bernoulli sample with $n = 10$ use `n` $= 10$, `size` $= 1$. If you increase `size`, each draw contains 10 bernoulli experiments, and you sum their results to obtain the draw from the binomial distribution. We now prove that as the number of bernoulli trials in each sample grows, the distribution of the bernoulli samples tends to a normal distribution.

```
p <- 1/2
n <- 100
m <- 10^3
sample_mean <- c()

for (i in 1:m) {
    sample <- rbinom(n, 1, p)
    sample_mean <- c(sample_mean, mean(sample))
```

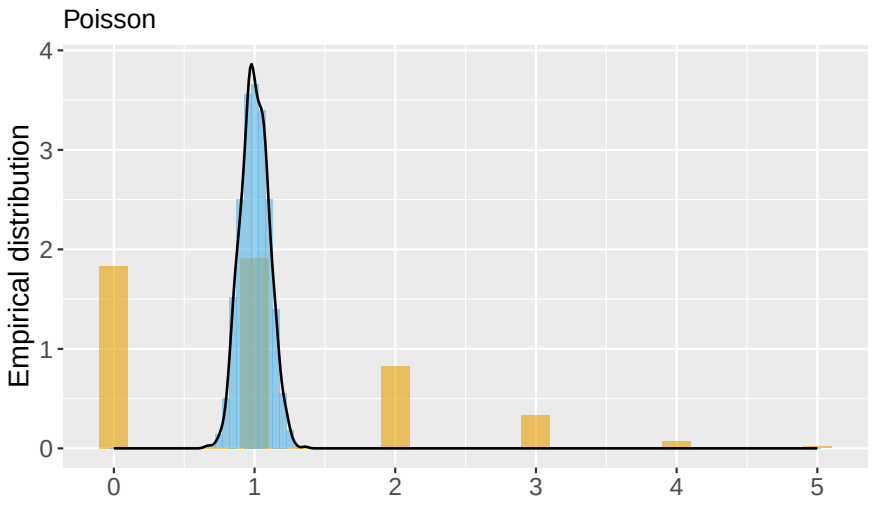
```
}

tibble(sample = rbinom(m, 1, p), sample_mean = sample_mean) %>%
    ggplot() +
        geom_histogram(aes(x = sample, y = ..density..),
                       alpha = 0.6,
                       binwidth = 0.2,
                       fill = '#E69F00') +
        geom_histogram(aes(x = sample_mean, y = ..density..),
                       alpha = 0.6,
                       binwidth = 0.01,
                       fill = '#56B4E9') +
        geom_line(aes(x = sample_mean), stat = 'density', size = 0.6) +
        labs(y = 'Empirical distribution', x = '', title = 'Bernoulli')
```




We perform the same experiment for the Poisson distribution whose parameter is $\lambda$.

```
lambda <- 1
n <- 100
m <- 10^3
sample_mean <- c()

for (i in 1:m) {
    sample <- rpois(n, lambda)
    sample_mean <- c(sample_mean, mean(sample))
```

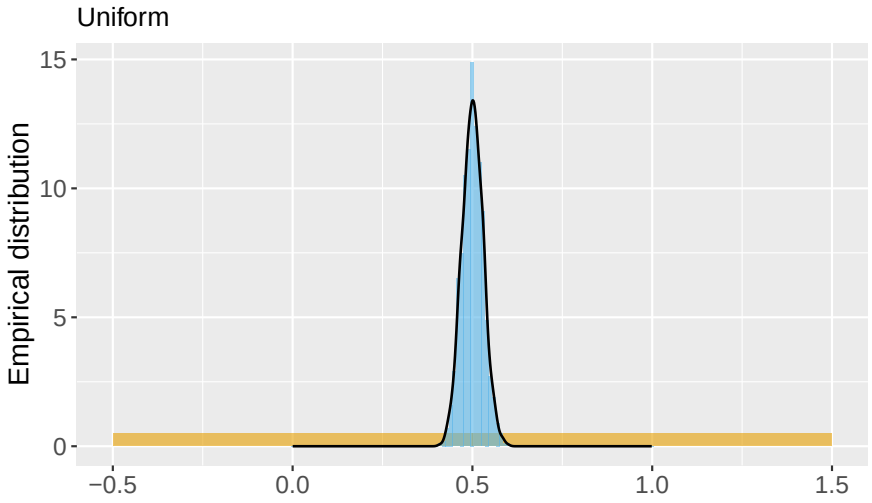
```
}

tibble(sample = rpois(m, lambda), sample_mean = sample_mean) %>%
    ggplot() +
        geom_histogram(aes(x = sample, y = ..density..),
                       alpha = 0.6,
                       binwidth = 0.2,
                       fill = '#E69F00') +
        geom_histogram(aes(x = sample_mean, y = ..density..),
                       alpha = 0.6,
                       binwidth = 0.05,
                       fill = '#56B4E9') +
        geom_line(aes(x = sample_mean), stat = 'density', size = 0.6) +
        labs(y = 'Empirical distribution', x = '', title = 'Poisson')
```




And we perform again the experiment for the uniform distribution

```
m <- 10^3
sample_mean <- c()

for (i in 1:m) {
    sample <- runif(n)
    sample_mean <- c(sample_mean, mean(sample))
}

tibble(sample = runif(m), sample_mean = sample_mean) %>%
```

```
    ggplot() +
        geom_histogram(aes(x = sample, y = ..density..),
                       alpha = 0.6,
                       binwidth = 1,
                       fill = '#E69F00') +
        geom_histogram(aes(x = sample_mean, y = ..density..),
                       alpha = 0.6,
                       binwidth = 0.01,
                       fill = '#56B4E9') +
        geom_line(aes(x = sample_mean), stat = 'density', size = 0.6) +
        labs(y = 'Empirical distribution', x = '', title = 'Uniform')
```




Description of the steps (apart from the distribution we sample from they are equal for all the 3 experiments above):

- Set the parameters
- Initialize the container for the sample means
- Sample $m$ times from the distribution of choice and compute the mean
- Sample from the distribution $m$ observations and bind it with the sample means computed at the previous step to create a tibble
- Plot

As we can assess from the graphs, the sample mean indeed tends to be distributed as a normal random variable.

## 2.4 Maximum Likelihood Estimator

We have now shown in general some properties of an estimator and how to compare and evaluate their goodness. But, how do we actually find an estimator? Imagine we do not know that the sample mean is a "good" estimator for the mean of a normal distribution, how do we find one?

The **maximum likelihood** estimation method is the most common method to find estimators for an unknown parameter. Since, as the name suggests, the way to find the estimator is maximize a likelihood, we need a formal definition of likelihood.

> **Likelihood function**: Let $X$ be a random vector with probability density $p_\theta$ that depends on a parameter $\theta \in \Theta$. For $x$ fixed, the function $L(\theta : x) := p_\theta(x)$, seen as a function of $\theta \in \Theta$ (where $\Theta$ is the parameter space), is called the likelihood function.

Often, $X = (X_1, \ldots, X_n)$ is drawn iid. When this is the case the density of $X = x$ is then equal to the product $\prod_{i=1}^n p_\theta(x_i)$ of the marginal probability densities of $X_1, \ldots, X_n$, and the likelihood function is equal to

$$L(\theta : x_1, \ldots, n) = \prod_{i=1}^{n} p_\theta(x_i)$$

We can now define the maximum likelihood estimator and estimate

> **Maximum Likelihood Estimate**: The maximum likelihood estimate of $\theta$ is the value of $T(x) \in \Theta$ that maximizes the likelihood function $L(\theta; x_1, \ldots, x_n)$. The maximum likelihood estimator is the corresponding estimator $T(X)$.

In the case of a discrete probability distribution, the maximum likelihood estimate can be described as the value of the parameter that assigns the greatest probability to the observed value $x$: we maximize the probability mass $p_\theta(x) = P_\theta(X = x)$ with respect to $\theta$ for fixed $x$. This method is only *a* method to obtain estimates: maximum likelihood estimators are not necessarily the best estimators. By a "best" estimator, we mean an estimator with the smallest possible mean square error. For a given model, computing the maximum likelihood estimators is a matter of applying calculus: differentiate the likelihood function and set the derivatives equal to 0. A trick that limits the necessary calculations (especially with independent observations) is to first take the logarithm of the likelihood.

Because the logarithm is a monotone function, the value $\theta$ that $\log L(\theta; x)$ is the same that maximizes $L(\theta; x)$. (Note that we are speaking only of the value where the maximum is reached, the argmax, not of the maximum value. For fixed $x$, the log-likelihood function is given by

$$\log L(\theta; x) = \log p_\theta(x) = \sum_{i=1}^{n} \log p_\theta(x_i)$$

If $L$ is differentiable in $\theta \in \Theta \subset \mathbb{R}^k$ then

$$\frac{\partial}{\partial \theta_j} \log L(\theta; x)_{|\theta=\hat{\theta}} = 0, \quad j = 1, \ldots, k$$

This system cannot always be solved explicitly. If necessary, an iterative method is used to obtain an approximation of the solution – e.g. Newton-Raphson method. Not only maxima, but also minima and inflection points are solutions of the likelihood equations. To verify whether a solution is indeed a maximum, we must consider the form of the (log-)likelihood function. One way to do this, is to determine the second derivative (or the Hessian matrix if the parameter has dimension greater than 1) of the log-likelihood function in the solution. If the function has a maximum in the solution, the second derivative in that point will be negative. For higher-dimensional parameters, all eigenvalues of the Hessian matrix must be negative. Note that he derivative of $\log L$ is called the *score function* and is the sum of the score functions of the individual observations.

We will now compute "manually" the maximum likelihood estimate of the normal distribution we work with so far. Of course there are many routines already available in R to do this, but to build one from scratch at least once can be very useful. Since R is provided with a very good minimizer called `optim`, we will consider the negative log-likelihood $NLL$, $-L(\theta; x)$. The argmin is what we are after now. Below we define a function that computes the sum of the individual log-likelihoods

$$\sum_{i=1}^{n} \log \left( \frac{1}{\sqrt{2\pi}\sigma} \right) \exp \left\{ -\frac{(x_i - \mu)^2}{2\sigma^2} \right\}$$

```
neg_loglik <- function(data, mu, sd) {
    loglik = 0
    for (x_i in data) {
        loglik = loglik +
            log( 1 / (sqrt(2*pi)*sd) ) *
                exp( -(x_i - mu)^2 / 2*sd^2)
    }
    # note that we take the opposite
```

```
    nll = -loglik
    return(nll)
}
```

Now that we have a way to compute the log-likelihood of the data, i.e. an objective function to optimize, let's perform the optimization. `optim` minimises a function by varying its parameters. The first argument of optim are the parameters with respect to we want to minimize the function – in our case $\mu$. The second argument is the function to be minimised – in our case the negative of `loglik`, i.e. `nll`. The tricky bit is to understand how to apply optim to the data. The solution is the `...` argument in optim, which allows us to pass other arguments, in the specific case our data `data=x` and `variance = sigma2` samples from the normal distribution. To distinguish between the true value of the mean used to generate the sample and the unknown value of the mean that we want to estimate, we will refer to this estimand as $\theta$

```
mu <- 170
sigma2 <- 25
n <- 100

x <- rnorm(n, mu, sqrt(sigma2))
theta <- rnorm(1, mu, sqrt(sigma2))

optimization <- optim(par = theta,
                      fn = neg_loglik,
                      data = x,
                      sd = sqrt(sigma2),
                      method = 'L-BFGS-B',
                      lower = -Inf,
                      upper = +Inf)
```

Description of the steps:

- Define the parameters $n, \mu, \sigma^2$
- Generate the sample
- Initialize the value of $\theta$ to a random value extracted from the same distribution
- Perform the optimization; note how the argument of `neg_loglik` are passed as additional arguments of optim

The two important output of optim are `convergence` that needs to be 0 to signal that the optimizer as converged to a minimum, and `par` which is the estimate. In our case, optim has converged to 177.482616.

# Chapter 3

# Confidence Interval Estimation

---

*In this chapter we implement stochastics simulation to empirically validate some simple theoretical propositions in the context of confidence interval estimation.*

---

## 3.1 Frequentist interpretation of Confidence Intervals

In Chapter 2, we have seen how a parameter $\theta$ could be estimated by the value $t = T(x)$ of an estimator $T$. In the context of this chapter, we will also refer to such estimates as *point estimates*. As a rule, an estimate $t$ differs from the parameter $\theta$ to be estimated: estimator will never be perfect. Using the confidence regions described in this chapter, we can quantify the possible difference between the estimator $T$ and $\theta$. This leads to an interval estimate $[L(x), R(x)]$, with the interpretation that $\theta$ has a high probability of lying in this interval. This interval, however, depending on the sample has a different value, thus it is a random interval. In the context of Bayesian statistic the opposite occurs: the interval is not random, the parameter $\theta$ is.

The definition of a confidence region (interval in case $\Theta \subseteq \mathbb{R}$) is the following

> **Confidence region**: Let $X$ be a random variable with a probability distribution that depends on a parameter $\theta \in \Theta$. A map $X \to G_X$

whose codomain is the set of subsets of $\Theta$ is a confidence region for $\theta$ of confidence level $1-\alpha$ if

$$P_\theta\left(\theta \in G_X\right) \geq 1-\alpha$$

for all $\theta \in \Theta$.

In other words, a confidence region is a "stochastic subset" $G_X$ of $\theta$ that has a high probability of containing the true parameter $\theta$. Because we do not know beforehand which value of $\theta$ is the true value, the condition in the definition holds for all values of $\theta$: under the assumption that $\theta$ is the true value, this true value must have probability at least $1-\alpha$ of being in $G_X$. After $X = x$ has been observed, the stochastic set $G_X$ changes into a nonstochastic subset $G_x$ of $\theta$. However, as said above, $G_x$ is sample-dependent.

Generally, $\alpha$ is taken small. The probability statement that the realization $G_x$ contains the true value $\theta$ with probability at least $1-\alpha$ can easily be interpreted incorrectly. In the frequentist view, the true value of $\theta$ is fixed; the realized confidence region $G_x$ is nonstochastic. Consequently, the true $\theta$ either lies in the confidence region or does not. Unfortunately, we do not know which of the two cases occurs. Therefore, the probability statement can be interpreted in the sense that if we, for example, carry out the experiment that gives $X$ independently 100 times and compute the confidence region $G_x$ 100 times, then we may expect that (at least) approximately $100(1-\alpha)$ of the regions will contain the true $\theta$ (Bijma, Jonker, and Vaart 2017).

Sometimes the center of the confidence interval is exactly the point estimate $T = T(X)$ for $\theta$. We then also write the interval in the form $\theta = T \pm \eta$, with $\eta = 1/2\left(R(X) - L(X)\right)$ half the length of the interval. In other case, the interval is intentionally chosen asymmetric around the used point estimate. Below we will provide an example using the common problem of estimating the unknown mean of normal distribution.

### 3.1.1 Example: Normal distribution

Let $X = (X_1, \ldots, X_n)$ be a sample from the normal $\mathcal{N}(\mu, \sigma^2)$ distribution with unknown $\mu \in \mathbb{R}$ and known variance $\sigma^2 = 9$. The natural estimator (unbiased and consistent) for $\mu$ is $\bar{X}$. We know that $\bar{X} \sim \mathcal{N}(\mu, \sigma^2/n)$, therefore

$$\sqrt{n}\frac{\bar{X}-\mu}{\sigma} \sim \mathcal{N}(0,1)$$

We compute the probabilities as follows:

$$P_\mu\left(z_{\frac{\alpha}{2}} \leq \sqrt{n}\frac{\bar{X}-\mu}{\sigma} \leq z_{1-\frac{\alpha}{2}}\right)$$

where $z_\alpha$ is the $\alpha$-quantile of the standard normal distribution, that is the number $z$ such that $F(z) = \alpha$, with $F$ being the cumulative distribution function of the standard normal distribution. Let $\mu = 10$ and $\alpha = 0.05$. We want to compute the value $z$ such that $F(z) = \alpha/2$ and $F(z) = 1 - \alpha/2$. This can be achieved in R with the function `qnorm()`

```
alpha <- 0.05

# find z
qnorm(c(alpha/2, 1-alpha/2))
#> [1] -1.95996  1.95996
```

Now we have discovered the values to plug-in in the formula in place of $z_{\frac{\alpha}{2}}$ and $z_{1-\frac{\alpha}{2}}$. Now we observe that

$$P_\mu\left(z_{\frac{\alpha}{2}} \leq \sqrt{n}\frac{\bar{X}-\mu}{\sigma} \leq z_{1-\frac{\alpha}{2}}\right) = P_\mu\left(\sqrt{n}\frac{\bar{X}-\mu}{\sigma} \leq z_{1-\frac{\alpha}{2}}\right) - P_\mu\left(\sqrt{n}\frac{\bar{X}-\mu}{\sigma} \leq z_{\frac{\alpha}{2}}\right)$$

We can rearrage the inequalities to obtain an interval for $\mu$:

$$\bar{X} - z_{\alpha/2}\frac{\sqrt{n}}{\sigma} \leq \mu \leq \bar{X} + z_{\alpha/2}\frac{\sqrt{n}}{\sigma}$$

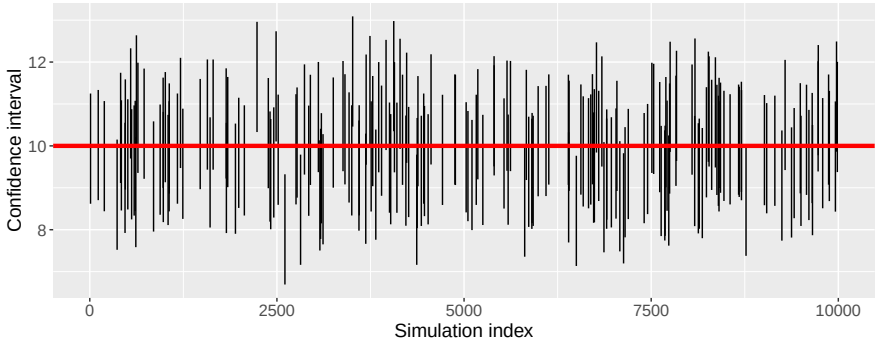
Using the often "abused" notation $z_{\alpha/2}$, which it is interpreted $|z_{\alpha/2}|$ – what we need is to subtract (on the left) and add (to the right) 1.95. The key point is knowing how our estimator is distributed! Once we have a distribution, everything is easily computable. Given that we know $z_{\alpha/2}$, to compute $\alpha/2$ we can use the function `pnorm()` in R

```
z <- 1.95
pnorm(z)
#> [1] 0.974412
```

which is approximately $1 - 0.05/2$.

To prove that indeed the **coverage** of the confidence interval is 95% – that is given 100 iid samples $X$, when we compute the confidence region $G_x$ for each sample, we want to prove that (at least) approximately 95 of the regions contain the true $\theta$. We will generate many samples of size $n = 20$ from the $\mathcal{N}(10, 9)$ distribution, compute the mean, the confidence interval and we will count how many times the interval contains the true value of $\mu$.

```r
alpha <- 0.05
n <- 20
mu <- 10
sigma2 <- 9
z <- qnorm(1 - (alpha/2))
m <- 10^4

right_bound <- c()
left_bound <- c()

for (i in 1:m) {
    sample <- rnorm(n, mu, sqrt(sigma2))
    sample_mean <- mean(sample)
    right_bound <- c(right_bound, sample_mean + z*sqrt(sigma2)/sqrt(n))
    left_bound <- c(left_bound, sample_mean - z*sqrt(sigma2)/sqrt(n))
}

sample_index <- sample(1:m, 200)

tibble(right = right_bound[sample_index],
       left = left_bound[sample_index],
       index = sample_index) %>%
    ggplot(aes(x = index, y = right)) +
        geom_segment(aes(xend = index, yend = left)) +
        geom_hline(yintercept = mu, colour = 'red', size = 1.5) +
        labs(x = 'Simulation index',
             y = 'Confidence interval')

check <- left_bound < mu & mu < right_bound

sum(check)/length(check)
#> [1] 0.9461
```



which is approximately $1 - \alpha$.

Detailed explanation of the steps:

- Define the parameters
- Initialize the container for the upper and lower bounds of the inteval
- For each iteration, generate a sample, compute the mean, compute the right and left bound of the interval and append it to the container
- For a better visualization sample 200 integer between 1 and $m$ to be used as indeces to subset the vectors `right_bound` and `left_bound`
- Put everything into a tibble so that we can feed it to ggplot for plotting

As said above, the most important thing is to know how the estimator is distributed. This allows us to compute the $\alpha$ quantiles. In most cases, thus, we rely on large sample approximation to obtain confidence intervals, especially when the estimator is some fancy function of the data (e.g. a neural network) and we do not know how it is distributed, or it is complicated to obtain even an approximate distribution. We know that for $n$ large $\bar{X}$ will tend to be distributed normally, regardless the distribution which produced the sample. Therefore, in the next example we will exploit this fact to compute the confidence interval for the estimator of the parameter $p$ of a bernoulli distribution.

### 3.1.2 Example: Bernoulli distribution

Assume that $X_1, \dots, X_n \overset{i.i.d.}{\sim} Ber(p)$. Then $\bar{X}$ is an estimator for $p$. How it is distrubuted? We know from theory that

$$\begin{aligned} P\left(\bar{X}_n = \frac{t}{n}\right) &= P\left(\underbrace{X_1 + \cdots + X_n}_{Bin(n,p)} = t\right), \qquad t \in \{0, 1, \dots, n\} \\ &= \frac{n!}{t!(n-t)!} p^t (1-p)^{n-t} \end{aligned}$$

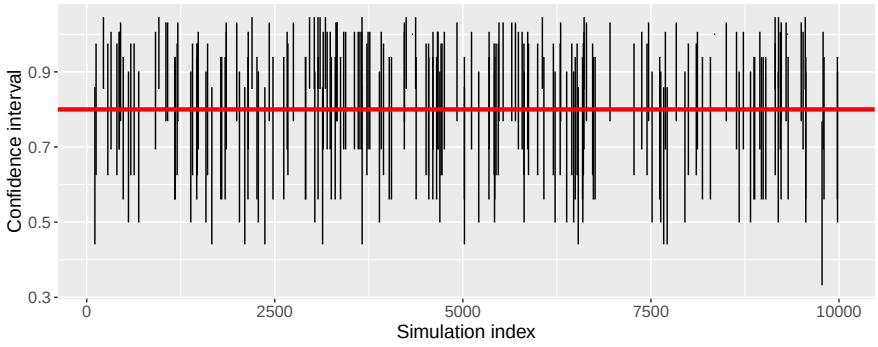
As $n \to \infty$, the sampling distribution of $\bar{X}$ tends to resemble $\mathcal{N}(\mu, \sigma^2/n)$ where $\mu = p$ and $\sigma^2 = p(1-p)$. Therefore, we will approximate the confidence interval using the quantiles from the distribution

$$\bar{X} \sim \mathcal{N}\left(p, \frac{p(1-p)}{n}\right)$$

Therefore the approximate confidence interval becomes

$$p \in \left[\bar{X} - z_{\alpha/2}\sqrt{\frac{\bar{X}(1-\bar{X})}{n}}, \quad \bar{X} + z_{1-\alpha/2}\sqrt{\frac{\bar{X}(1-\bar{X})}{n}}\right]$$

```
alpha <- 0.05
n <- 20
p <- 0.8
z_norm <- qnorm(1 - (alpha/2))
m <- 10^4

right_bound <- c()
left_bound <- c()

for (i in 1:m) {
    sample <- rbinom(n = n, size = 1, prob = p)
    sample_mean <- mean(sample)

    approx_interval <- z_norm * sqrt(sample_mean*(1-sample_mean)/n)

    right_bound <- c(right_bound, sample_mean + approx_interval)
    left_bound <- c(left_bound, sample_mean - approx_interval)
}

sample_index <- sample(1:m, 200)

tibble(right = right_bound[sample_index],
       left = left_bound[sample_index],
       index = sample_index) %>%
    ggplot(aes(x = index, y = right)) +
        geom_segment(aes(xend = index, yend = left)) +
        geom_hline(yintercept = p, colour = 'red', size = 1.5) +
        labs(x = 'Simulation index',
             y = 'Confidence interval')

check <- left_bound < p & p < right_bound

sum(check)/length(check)
#> [1] 0.9147
```




Detailed explanation of the steps:

- Define the parameters
- Initialize the container for the upper and lower bounds of the inteval
- For each iteration, generate a sample, compute the mean, compute the right and left bound of the interval and append it to the container
- For a better visualization sample 200 integer between 1 and $m$ to be used as indeces to subset the vectors `right_bound` and `left_bound`
- Put everything into a tibble so that we can feed it to ggplot for plotting

As we can see, in this case the empirical proportion of intervals that contain the true value $p$ is more distant from $1 - \alpha$ than what we obtained in the example above. This is due to the fact that we are approximating the distribution of the estimator for $p$, $\bar{X}$ using an asymptotic result with $n = 20$. However, we expect that as $n$ grows, the empirical $1 - \alpha$ will converge to the true value.

# Chapter 4

# Hypotheses testing

*In this chapter we will implement stochastic simulations to empirically validate some simple theoretical propositions about test of hypotheses. We will do this exploiting the computational capabilities of R.*

## 4.1 Frequentist Hypotheses Testing

The decision between conflicting hypotheses is based on a suitable statistical model for the observation X. The hypotheses are coded in parameter values that index the probability distributions in the statistical model. Here, we will restrict ourselves to two hypotheses. The parameter $\theta$ belongs either to a set $\Theta_0$ corresponding to the one hypothesis or to the complement $\Theta_1 = \Theta \setminus \Theta_0$. We call the hypothesis $H_0 : \theta \in \Theta_0$ the *null* hypothesis and the hypothesis $H_1 : \theta \in \Theta_1$ the *alternative* hypothesis.

In the standard approach to testing (followed by most "users" of statistics), the null and alternative hypotheses are not treated symmetrically. We, in particular, want to know whether the alternative hypothesis is correct. If the data do not give sufficient indication to support this, this does not necessarily imply that the alternative hypothesis is incorrect (and the null hypothesis correct); it is also possible that there is not sufficient proof for either of the hypotheses. The statistical analysis can thus lead to two conclusions:

- Reject $H_0$ (and accept $H_1$ as being correct)

- Do not reject $H_0$ (but do not accept $H_0$ as being correct)

The first is a strong conclusion, the second is not truly a conclusion. The second should be seen as the statement that more information is needed to reach a conclusion. By basing our statements concerning the hypotheses on our observations, we can make two types of mistakes, corresponding to mistakenly coming to one of the two possible conclusions:

- A *type I error* consists of rejecting $H_0$ when it is correct
- A *type II error* consists of not rejecting $H_0$ when it is incorrect

A type I error corresponds to falsely choosing the strong conclusion. This is very undesirable. A type II error corresponds to falsely choosing the weak conclusion. This is also undesirable, but since the weak conclusion is not truly a conclusion, it is not as bad. Because of the asymmetric handling of the hypotheses $H_0$ and $H_1$ when choosing a test, we should not attach too much value to not rejecting $H_0$. It is therefore of great importance to choose the null hypothesis and the alternative hypothesis wisely. In principle, we choose the statement we want to show as the alternative hypothesis. We then argue for $H_0$: we only reject $H_0$ if there is strong evidence against it.

Let's give the definition of a statistical test:

> **Statistical test**: Given a null hypothesis $H_0$, a statistical test consists of a set $R_T$ (where $T$ identifies the test statistic) of possible values for the observation $X$, the *critical region*. Suppose that we have an observation $x$. If $x \in R_T$, we reject $H_0$; if $x \in \bar{R}_T$ (where the bar represents the complement), we do not reject $H_0$.

When $X = (X_1, \ldots, X_n)$ is a vector of observations, in particular, it is often difficult to decide based on $X$ whether the statement of the alternative hypothesis can be true. We therefore often summarize the data in a test statistic. A test statistic is a real-valued quantity $T = T(X)$ based on the data that gives information on the correctness of the null and alternative hypotheses; so the test statistic does **not** depend on the unknown parameter and, importantly, we know its distribution.

The critical region $R_T$ is often of the form $\{x : T(x) \in R_T\}$, thus it is a subset of the codomain of the test statistic – or, simply, a subset of the space where $X$ "lives". Therefore the critical region depends on the test statistic, which in turn

depends on the sample and thus is random. A test is therefore random. In general, another test statistic $T'$ leads to a different set $R_{T'}$. However, the critical region $R$ can be the same in both cases, i.e. the same critical region $R$ can correspond to two different test statistics.

In testing $H_0 : \theta \in \Theta_0$ against $H_1 : \theta \in \Theta_1$, when the true value of $\theta$ belongs to $\Theta_0$ the null hypothesis is true. If in that case $x \in R_T$ then we falsely reject $H_0$ and make a type I error. For a good test, the probability $P_\theta(X \in R_T)$ for $\theta \in \Theta_0$ must therefore be small. On the other hand, when the null hypothesis is false this probability needs to be large. The quality of a test can therefore be measured using the function $P_\theta(X \in R_T)$. In particular,

$$P_\theta(X \in R_T; \theta \in \Theta_0) = \alpha$$

$$P_\theta(X \notin R_T; \theta \in \Theta_1) = \beta$$

It is impossible to get $\alpha = \beta = 0$. Theory gives the tools to choose a test such that, given $\alpha$, we minimize $\beta$. The quantity

$$P_\theta(X \in R_T; \theta \in \Theta_1) = 1 - \beta$$

is called the **power** of a test. In general,

> **Power function**: The power function of a test with critical region $R_T$ is
>
> $$Q_T(\theta) = P_\theta(X \in R_T)$$

Thus, it is possible to define the goodness of a test based on the power function: we are looking for a critical region for which the power function takes on "small values" (close to 0) when $\theta \in \Theta_0$, and "large values" (close to 1) when $\theta \in \Theta_1$.

We now define another quantity of interest of a test

> **Size**: The size of a test with critical region $R_T$ with power function $Q_T(\theta)$ is the number
>
> $$\alpha = \sup_{\theta \in \Theta_0} Q_T(\theta)$$
>
> A test has significance level $\alpha_0$ if $\alpha \geq \alpha_0$.

This quantity is important because usually we first choose a fixed number $\alpha_0$, the significance level, and then we use tests of level $\alpha_0$. In other words, we only allow tests whose power function $Q_T(\theta)$ under the null hypothesis is at most $\alpha_0$:

$$\sup_{\theta \in \Theta_0} Q_T(\theta) \leq \alpha_0$$

This ensures that the probability of a type I error is at most $\alpha_0$. However, choosing $\alpha_0$ extremely small is rare. We can only achieve this by making $R_T$ very small. In doing this, however, the power function for $\theta \in \Theta_1$ also becomes small. The probability of a type II error, $P_\theta(X \notin R_T) = 1 - Q_T(\theta)$, for $\theta \in \Theta_1$, therefore becomes very large, which is also undesirible. The requirements for making both the type I and type II errors small work against each other. Therefore we usually do not treat the two types of errors symmetrically: usually the type I error is given more importance and thus the goal is to set $\alpha_0$ small and find the test that minimises $\beta$, given that $\alpha_0$.

In practice, $\alpha_0$ is often chosen equal to the magical number 0.05. With this choice 1 out of 20 times we will falsely reject the null hypothesis (making a type I error). If the probability of making a type II error is too large, the test is, of course, not very meaningful, because we then almost never reject $H_0$. More formally, given the level $\alpha_0$, we prefer a test of level $\alpha_0$ with the greatest possible power function $Q_T(\theta)$ for $\theta \in \Theta_1$.

Under this assumption, for a given level $\alpha_0$, we prefer a test with critical region $R_T$ to a test with critical region $R_{T'}$ if both have level $\alpha_0$ and the first has a greater power function than the second for all $\theta \in \Theta_1$:

$$\sup_{\theta \in \Theta_0} Q_i(\theta), \quad i = T, T'$$

**and**

$$Q_T(\theta) \geq Q_{T'}(\theta), \quad \forall \theta \in \Theta_1$$

with strict inequality for at least one $\theta \in \Theta_1$. We call the test with critical region $R_T$ *more powerful* than the test with critical region $R_{T'}$ in some $\theta \in \Theta_1$ if $Q_T > Q_{T'}$. We call the test with critical region $R_T$ *uniformly more powerful* if the inequality holds for all $\theta \in \Theta_1$.

Below we will perform examples of few important kind of tests on the normal distribution. In what follows we will refer to the distribution used in chapter 1, but with a more realistic variance: 225 – it comes from the fact that we want to assign high probability to the range $170 \pm 15$ – and sample size $n = 5$ that is $X_1, \ldots, X_5 \overset{i.i.d.}{\sim} \mathcal{N}(170, 225)$.

### 4.1.1 One-sided tests of Population Mean with Known Variance

The null hypothesis of the lower tail test of the population mean can be expressed as follows:

$$H_0 : \mu \geq \mu_0$$
$$H_1 : \mu < \mu_0$$

where $\mu_0$ is the hypothesized upper bound of the true population mean $\mu$ – we would like to find enough evidence to reject $H_0$. Let us define the test statistic $z$, under the null hypothesis, in terms of the sample mean, the sample size and the population standard deviation

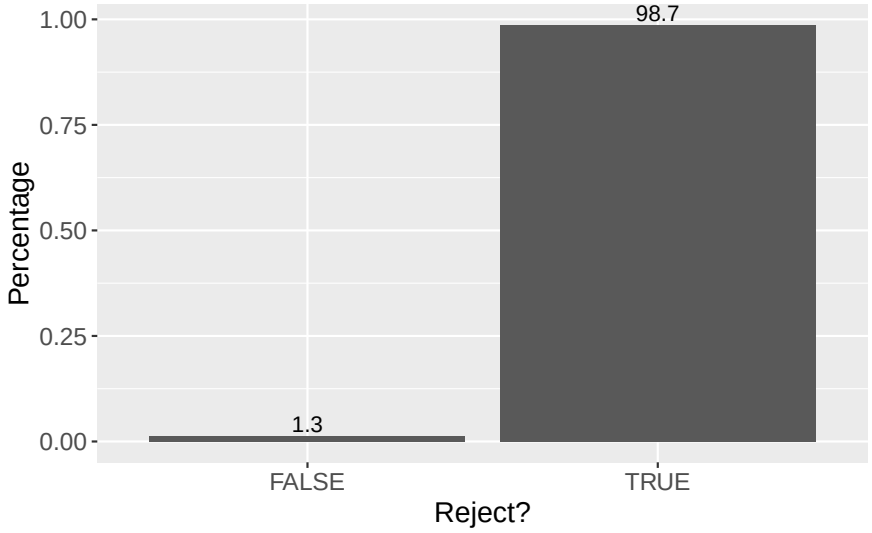
$$z = \sqrt{n}\frac{\bar{X} - \mu_0}{\sigma}$$

Then the null hypothesis of the lower tail test is to be rejected if $z \in R_z := (-\infty, z_\alpha)$ , where $z_\alpha$ is the $100(1-\alpha)$ percentile of the standard normal distribution.

Suppose we want to test if the *average* height of italian has an upper bound at 195cm. We set $\mu_0 = 195$. We will perform the test $m$ times, each time for a sample of size $n = 5$.

```
m <- 1000
n <- 5
mu <- 170
sigma2 <- 225
alpha <- 0.05
mu_0 <- 195

z <- c()

for (i in 1:m) {
    sample <- rnorm(n, mu, sqrt(sigma2))
    sample_mean <- mean(sample)
    z <- c(z, sqrt(n) * (sample_mean - mu_0) / sqrt(sigma2))
}

z_alpha <- qnorm(alpha)
reject <- z < z_alpha
```

```
tibble(rejections = reject) %>%
    ggplot(aes(x = rejections, y = (..count..)/sum(..count..))) +
        geom_bar() +
        geom_text(aes(label = (..count..)/sum(..count..)*100),
                  stat = 'count',
                  vjust = -0.25) +
        labs(x = 'Reject?',
             y = 'Percentage')
```




Detailed explanation of the steps:

- Define the parameters
- Initialize the container for the test statistic
- For each iteration, generate a sample, compute the mean, compute the test statistic and append it to the container
- Compute the quantile 0.05 for the standard normal distribution
- For each iteration we obtained a value of the test statistic, thus we check how many times it falls in the rejection region

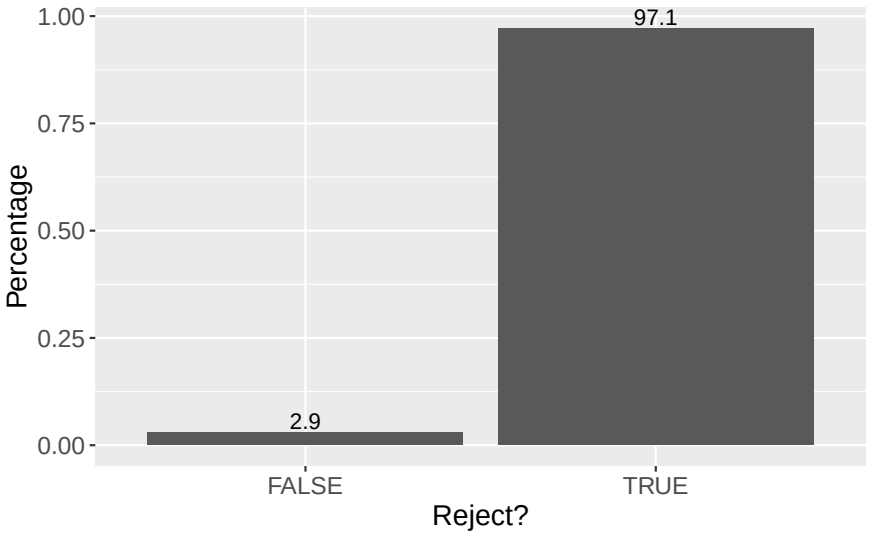
Obviously, we expect that the null hypothesis must be rejected, because we know that the true $\mu = 170$. This is confirmed by the test: indeed, for $m = 1000$ tests we performed, the test rejected $H_0$ 98.7% of the cases. We say that at a $\alpha = 0.05$ significance level we do reject the null – that is, performing the test infinite times, 95% of the times we would not reject the null.

Let's now perform the opposite check, that is let's test that 145cm is a lower bound for the *average* population height in Italy. We hope to have enough evidence to reject the null.

$$H_0 : \mu \leq \mu_0$$
$$H_1 : \mu > \mu_0$$

In the previous case the rejection region was $(-\infty, -z_\alpha)$. In this case it is $(z_{1-\alpha}, +\infty)$ the only three line of code we change are (the rest is not shown)

```
mu_0 <- 145
z_alpha <- qnorm(1-alpha)
reject <- z > z_alpha
```




The description of the algorithm is the same as above with the only difference that now we changed `qnorm(alpha)` into `qnorm(1-alpha)` and swapped the sign that defines the `reject` variable.

As we can see, 97.1% of the times we reject the null hypothesis as expected.

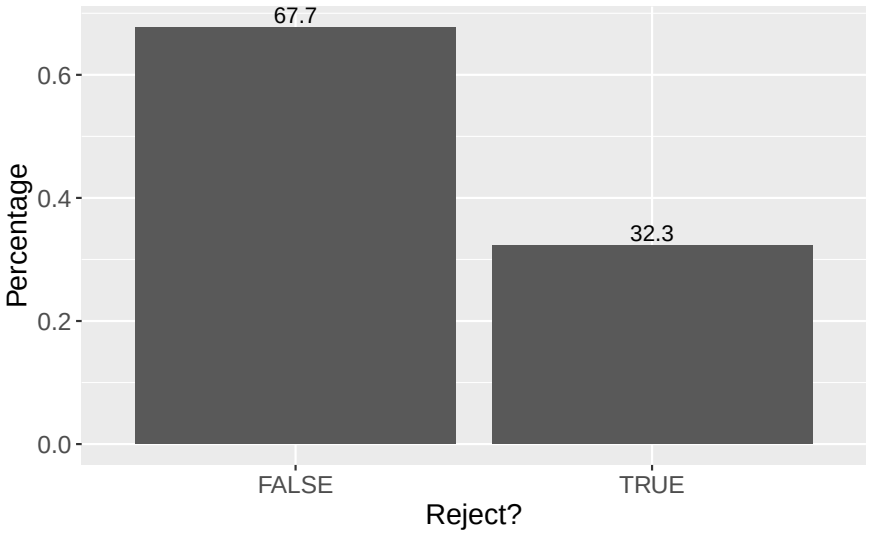
### 4.1.2 Two-sided tests of Population Mean with Known Variance

The null hypothesis of the two-tailed test of the population mean can be expressed as follows:

$$H_0 : \mu = \mu_0$$
$$H_1 : \mu \neq \mu_0$$

where $\mu_0$ is a hypothesized value of the true population mean $\mu$. The definition of the test statistic is the same as above. The rejection region is now defined as the interval $R_z := (-\infty, z_{\alpha/2}) \cup (z_{1-\alpha/2}, +\infty)$. Let's suppose we want to check whether the average height is 160cm. The only three line of code we change are (the rest is not shown)

```
mu_0 <- 160
z_alpha <- qnorm(c(alpha/2, 1-alpha/2))
reject <- z < z_alpha[1] | z > z_alpha[2]
```

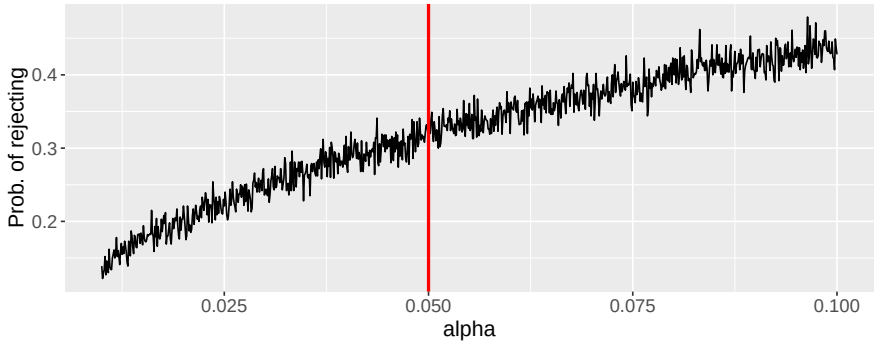



Detailed explanation of the steps:

- Define the parameters
- Initialize the container for the test statistic
- For each iteration, generate a sample, compute the mean, compute the test statistic and append it to the container
- Compute the quantile 0.05 and 0.95 for the standard normal distribution and store it into a two dimensional vector
- For each iteration we obtained a value of the test statistic, thus we check how many times it falls in the rejection region (note the `|` (or) between the two `z_alpha`s)

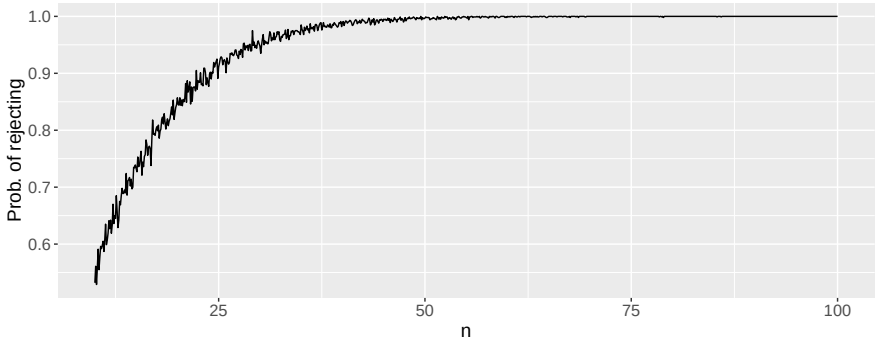
As we may see, there is no clear-cut decision here. What is the problem? What parameters are resposible for this? The answer is $\sigma^2$, $n$, and $\alpha$. Let's now see how the decision changes if we modify $\alpha$

```
m <- 1000
n <- 5
mu <- 170
sigma2 <- 225
alpha <- seq(0.01, 0.10, 0.0001)
mu_0 <- 160

rejection <- c()

for (a in alpha) {
    z <- c()

    for (i in 1:m) {
        sample <- rnorm(n, mu, sqrt(sigma2))
        sample_mean <- mean(sample)
        z <- c(z, sqrt(n) * (sample_mean - mu_0) / sqrt(sigma2))
    }

    z_alpha <- qnorm(c(a/2, 1-a/2))
    reject <- z < z_alpha[1] | z > z_alpha[2]
    rejection <- c(rejection, sum(reject)/length(reject))
}

tibble(alpha = alpha, rejections = rejection) %>%
    ggplot(aes(x = alpha, y = rejection)) +
        geom_line() +
        geom_vline(xintercept = 0.05, colour = 'red', size = 1) +
        labs(x = 'alpha',
             y = 'Prob. of rejecting')
```



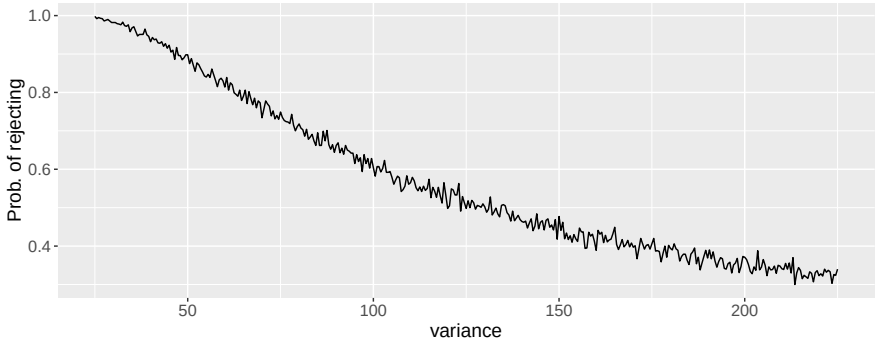
When $\alpha$ becomes bigger we have some improvements, but still we reject the false null less than 50% of the times. Let's see what happens if we modify $n$, given everything remains the same

```
m <- 1000
sizes <- seq(10, 100, 0.1)
mu <- 170
sigma2 <- 225
alpha <- 0.05
mu_0 <- 160

rejection <- c()

for (n in sizes) {
    z <- c()

    for (i in 1:m) {
        sample <- rnorm(n, mu, sqrt(sigma2))
        sample_mean <- mean(sample)
        z <- c(z, sqrt(n) * (sample_mean - mu_0) / sqrt(sigma2))
    }

    z_alpha <- qnorm(c(alpha/2, 1-alpha/2))
    reject <- z < z_alpha[1] | z > z_alpha[2]
    rejection <- c(rejection, sum(reject)/length(reject))
}

tibble(alpha = sizes, rejections = rejection) %>%
    ggplot(aes(x = sizes, y = rejection)) +
        geom_line() +
        labs(x = 'n',
             y = 'Prob. of rejecting')
```



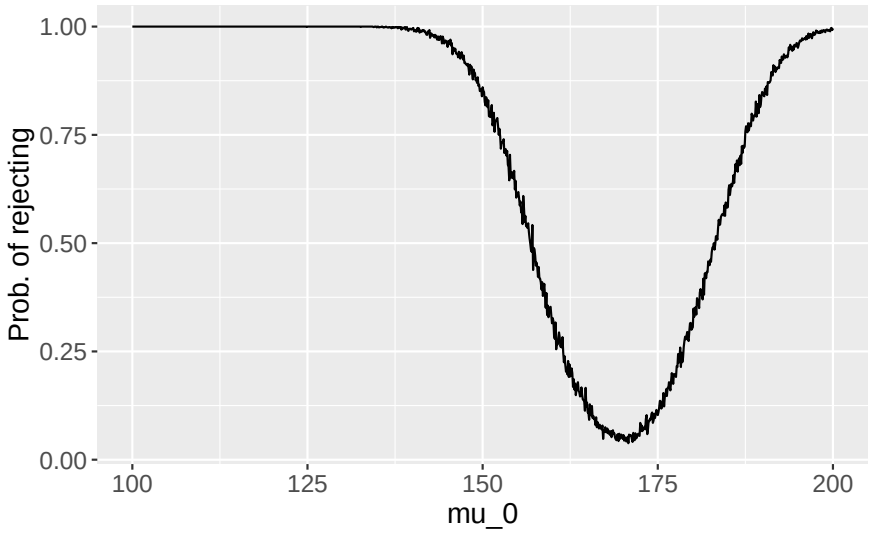
As we may assess from the graph, the sample size has a huge impact on the rejection of the null. If the $\alpha$ is not adjusted for the sample size, it can happen that we reject the null even when it is true! Let's assess what happend when the variances changes

```
m <- 1000
n <- 5
mu <- 170
variances <- seq(25, 225, 0.5)
alpha <- 0.05
mu_0 <- 160

rejection <- c()

for (sigma2 in variances) {
    z <- c()

    for (i in 1:m) {
        sample <- rnorm(n, mu, sqrt(sigma2))
        sample_mean <- mean(sample)
        z <- c(z, sqrt(n) * (sample_mean - mu_0) / sqrt(sigma2))
    }

    z_alpha <- qnorm(c(alpha/2, 1-alpha/2))
    reject <- z < z_alpha[1] | z > z_alpha[2]
    rejection <- c(rejection, sum(reject)/length(reject))
}

tibble(alpha = variances, rejections = rejection) %>%
    ggplot(aes(x = variances, y = rejection)) +
        geom_line() +
        labs(x = 'variance',
             y = 'Prob. of rejecting')
```

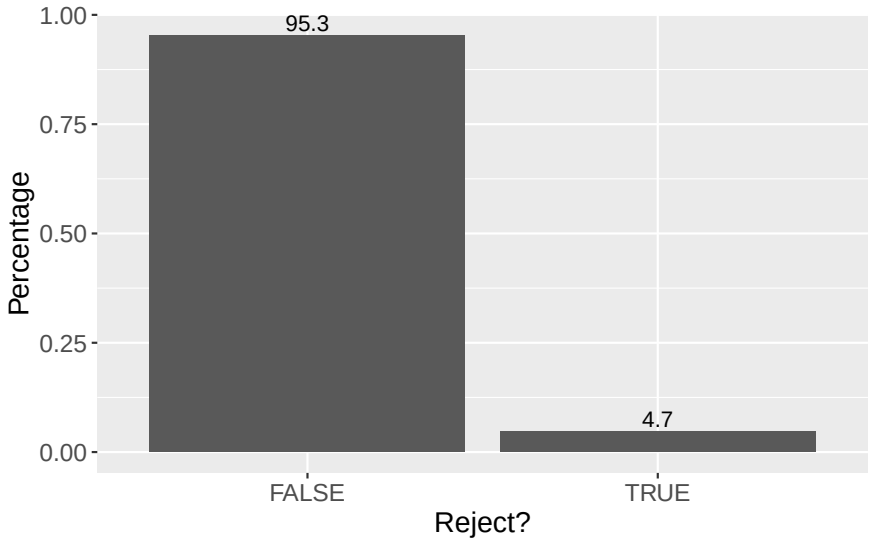


The variance has a huge impact on the rejection of the null: smaller variance brings to an easier rejection of the null. To summarize, the conclusion with this test would be that we do not have enough information to distinguish values around 170cm. We need more evidence: collect more observation from the population! To draw the power function of this test, we will make $\mu_0$ change, given everything remains the same

```
m <- 1000
n <- 5
mu <- 170
sigma2 <- 225
alpha <- 0.05
mu_0s <- seq(100, 200, 0.1)

rejection <- c()

for (mu_0 in mu_0s) {
    z <- c()

    for (i in 1:m) {
        sample <- rnorm(n, mu, sqrt(sigma2))
        sample_mean <- mean(sample)
        z <- c(z, sqrt(n) * (sample_mean - mu_0) / sqrt(sigma2))
    }

    z_alpha <- qnorm(c(alpha/2, 1-alpha/2))
    reject <- z < z_alpha[1] | z > z_alpha[2]
    rejection <- c(rejection, sum(reject)/length(reject))
}

tibble(mu_0 = mu_0s, rejections = rejection) %>%
    ggplot(aes(x = mu_0, y = rejection)) +
        geom_line() +
        labs(x = 'mu_0',
             y = 'Prob. of rejecting')
```




We can assess that the test is not perfect: the minimum probability of rejecting is not achieved at $\mu_0 = 170$ as one would have expected. For curiosity, let's see how it behaves if we plug-in the true value for the mean $\mu_0 = 170$, given everything remains the same (code for the test not reported since it is equal to the one above)




As we may appreciate, the test do **not** reject almost 95% of the times, this confirms the proper functioning of the test, although being weak – in the sense of not very powerful.

## Conclusions

This chapter ends this brief collection of statistical concepts that we tried to visualize and implement with R.

## References

Bijma, Fetsje, Marianne Jonker, and Aad van der Vaart. 2017. *An Introduction to Mathematical Statistics*. Amsterdam University Press.

Castagnoli, Erio, Margherita Cigola, and Lorenzo Peccati. 2009. *Probability, a Brief Introduction*. 2nd ed. Milano, MI, Italy: Egea.

Efron, Bradley, and Trevor Hastie. 2016. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. 1st ed. New York, NY, USA: Cambridge University Press.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science*.