Introduction to Statistics and R for Neuroscience

Contents

Introduction

As observers, we commonly only have access to a very tiny portion of the world, from which, ideally, we would like to derive something general and universal. This is practically the role of statistics, that, in a world full of uncertainty, aims to derive proper scientific inferences in the face of measurement and systems noise, or of unaccounted for, or unobserved, factors and sources of influence. The rapid advancement of neuroimaging technologies and high-performance computing (HPC) has enabled neuroscience researchers to collect unprecedented amounts of brain data. This wealth of information allows scientists to revisit old problems and explore new ones on a much larger scale. There is, especially in neuroscience, a increasing demand for novel statistical tools, due to the new challenges of simultaneous observations or recordings from an ever-growing set of variables, more complicated data structures, and a rapid growth of information.

The impact of statistics on human neuroscience can be discussed through the three main concepts of statistics defined by Ronald Fisher, father of modern statistics:

- the study of populations,
- the analysis of variance,
- data reduction.

These foundational principles are central to key areas of brain research, such as understanding connectivity between brain regions, modeling the flow of information within and between areas, and extracting meaningful patterns from large-scale neuroimaging data. For example, predictive models are now frequently employed to anticipate disease progression or classify patients based on brain features.

Study of populations   Neuroscience research often involves different types of populations, such as population across individuals, and population across time. Imagine scanning the brains of 100 individuals in an MRI scanner for an hour. Here, the population is formed by the aggregate data across these subjects, capturing inter-individual variability. If we scan one individual every day for a year, the population shifts to reflect temporal changes in the same person's brain over time.

Analysis of variance   Each of these populations introduces different types of variance–between individuals, across time, or spatial–all of which require tailored statistical methods to understand and quantify.

Data reduction   One of the challenges in neuroscience is dealing with the scale of the data. Summarizing this data effectively is critical for ensuring that statistical models and computational tools can handle it efficiently. Moreover, the brain itself has the unique ability of extracting stable, universal patterns, such as color or motion perception, from the ever-changing sensory inputs. Under this perspective, there are statistical methods that aim to extract smaller representative data which summarize patterns of specific brain regions, across individuals or a specific subject, distinguish healthy subjects from patients, and potentially track changes in mental states or disease progression over time. (This aspect is not treated in this course.)

Statistical integration in Neuroscience

Predictive modeling offers exciting opportunities in neuroscience, enabling applications such as brain disease diagnosis and decoding mental states. However, it also brings challenges, especially with causal inference. Causal inference aims to determine whether an observed relationship between two variables reflects a true cause-and-effect relationship. For example, irregular neural signals in a specific brain region may correlate with a mental disorder, but this association does not necessarily imply causation. These signals could be an effect of the disorder, or the relationship could arise from a coincidental correlation caused by unmeasured factors. This challenge is further exacerbated in high-dimensional brain data, where spurious correlations or incidental endogeneity—features unintentionally correlated with the model's error terms—can violate critical assumptions and lead to misleading conclusions.
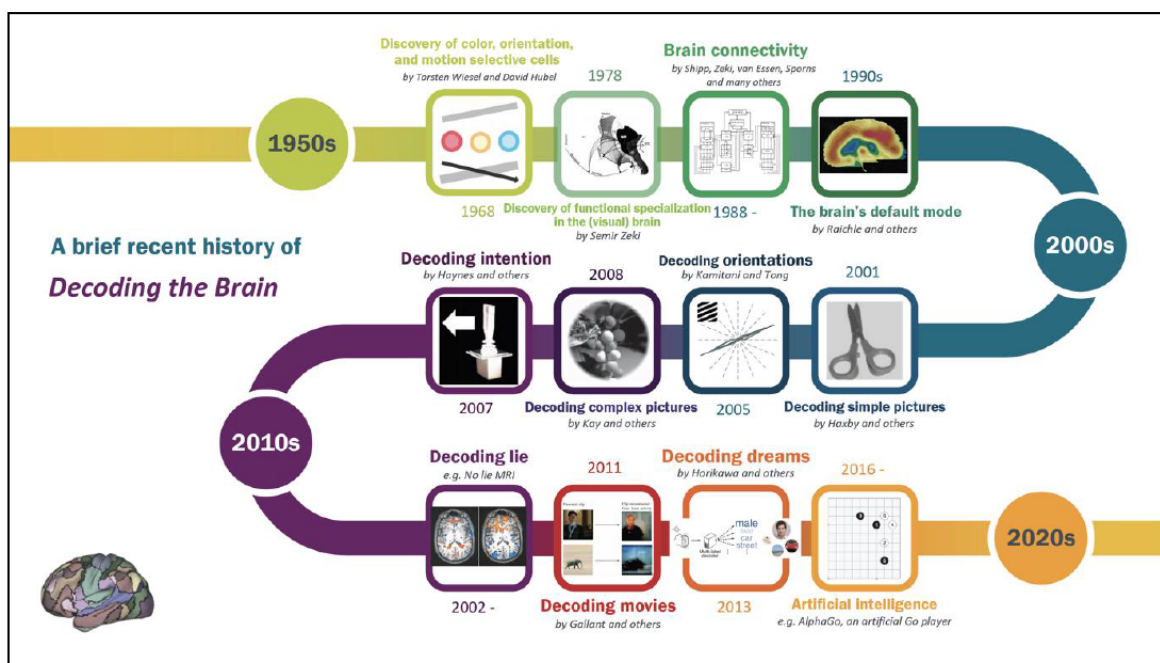
This figure provides a visual overview of how advancements in neuroscience and statistical methods have evolved together to decode the brain.

At its core, statistical inference allows researchers to generalize findings from a small sample to a broader population. This process is essential in neuroscience, where directly observing every aspect of the brain is unfeasible. Careful attention to data validity and robustness ensures that these generalizations are meaningful and scientifically sound.

The integration of statistical analysis, computer science technologies and brain studies is advancing our understanding of brain function and inspiring brain-like computers. Neural networks, for instance, are

designed following the structure and operation of the brain. McCulloch and Pitts laid the groundwork for artificial neural networks by conceptualizing a logical calculus for neural activity, framing the brain as a computational device. Similarly, Hebb's rule, often summarized as "neurons that fire together, wire together," provided a biological basis for learning mechanisms, influencing the design of Rosenblatt's perceptron—the first artificial neural network capable of performing simple tasks. Many neural network-based model have begun to outperform humans in certain tasks, such as AlphaGo which demonstrated unprecedented strategic thinking playing chess. Facilitating collaborative research between statistics and neuroscience in the next future is a key component for pursuing the common goal of charting the functions of the human brain and using them to benefit daily life.

Reproducibility

In science, it is of vital importance that multiple studies corroborate the same result. Researchers therefore need to know all the details of previous experiments in order to implement the procedures as exactly as possible. However, this is becoming a major problem in neuroscience, as animal studies of behavior have proven to be hard to reproduce, and most experiments are never replicated by other laboratories.

such failures undermine the credibility of theories building on them and potentially call into question substantial parts of scientific knowledge

Remark: *Reproducibility* refers to re-examining and validating the analysis of a given set of data. *Replication* refers to repeating the experiment or study to obtain new, independent data with the goal of reaching the same or similar conclusions.

The importance of using appropriate statistical methods for experimental design and data analysis is well recognized across scientific disciplines. The growing concern over reproducibility in biomedical research is often referred to as a "problem of inadequate rigor" (Kilkenny et al., 2009; Prinz et al., 2011). The reproducibility crisis has been attributed to various factors that include lack of adherence to good scientific practices, underdeveloped experimental designs, and the misuse of statistical methods

The proper analysis of Big Data requires up-to-date statistical methodologies that take complex features of data such as explicit and implicit data dependencies into consideration. Better matching of statistical models that take data characteristics into account will allow for better interpretation of data outcomes.

Despite recent advances in statistical methods, current neuroscience research is often conducted using a limited set of well-known statistical tools. Many models and tests assume that the observations are independent of one another. Failure to account for this dependency in the data often leads to an increased number of false positives, a major cause of the irreproducibility crisis. Statisticians have developed effective methods to analyze correlated data.

Introduction to R

What is R   R is a programming language that includes conditionals, loops, and the ability to define custom functions allowing data processing and graphical representation in the statistical domain. R serves as an environment where various statistical techniques are implemented and made available as *packages*. Approximately 25 packages are included in the base `R` environment, referred to as "standard" and "recommended." All other packages are available through the R CRAN website: https://cran.r-project.org/.

Why Use R:

- Free software, available for download at the link above;
- Available for multiple platforms (Windows, Linux, Macintosh);
- Constantly developed and improved;
- Versatile, allowing the development of new procedures.

It is possible to interact with R using **RStudio** (available at https://rstudio.com/). RStudio allows users to write ".R" files containing commands that can be executed in the R command prompt. Additionally, it

provides faster access to plots, the environment (variables and functions defined during the session), and the help documentation.

Rstudio and GitHub Git is an open-source version control system. It means that whenever a developer develops some project (like an app) or something, he/she constantly update it catering to the demands of users, technology, and whatsoever it maybe. Version control systems keep these revisions straight, storing the modifications in a central repository. It allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute. Git is used to storing the source code for a project and track the complete history of all changes to that code, while **GitHub** is a cloud-based platform built around the Git tool.

It is possible to link a *Rstudio project* to a *GitHub repository*. RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents. To create a new project in the RStudio IDE, use the `New Project` command (available on the `Projects` (top right corner) menu or go to `File > New Project`. RStudio creates a project file (with an `.Rproj` extension) within the project directory. This file contains various project options and can also be used as a shortcut for opening the project directly from the filesystem.

For more details:

- https://rfortherestofus.com/2021/02/how-to-use-git-github-with-r/
- https://annakrystalli.me/talks/r-in-repro-research.html#27

Rmarkdown

Rmarkdown (Xie, Allaire, and Grolemund 2018) is a powerful tool for combining analysis and reporting into the same document. A complete guide can be found at https://bookdown.org/yihui/rmarkdown-cookbook/. To use Rmarkdown we need to install the package `rmarkdown`

```r
install.packages("rmarkdown")
```

```r
library(rmarkdown)
```

```
## Warning: il pacchetto 'rmarkdown' è stato creato con R versione 4.3.3
```

A Rmarkdown `.Rmd` document has three components:

1. *yaml*: the document starts with yaml for setting metadata, e.g. title, output format. You can also specify bibliography as `.bib` file (it must be in the same folder of the .Rmd file) or specify the full path where the .bib file is. For example, the yaml part of the current document is

```yaml
---
title: "Introduction to Statistics and R for Neuroscience"
output:
html_document:
    df_print: paged
date: ''
bibliography: notes.bib
---
```

Possible `output` formats include HTML, PDF, WORD, LaTeX.
2. *text*: this is equivalent to writing in a text editor, but more powerful and interesting for how the document is generated. Some simple rules:

- Multiple spaces on a given line are reduced to one;
- Line endings with fewer than two spaces are ignored;
- Two or more spaces at the end of lines introduce hard breaks, forcing a new line;
- Lines starting with the greater-than sign > introduce block quotes;

- One or more blank lines introduce a new paragraph;
- Text with the syntax `<!- - comments - - >` is omitted from output;
- The sign `#` introduces headers; lower levels are created with additional signs - up to total five levels;
- `*italics*` becomes *italics*;
- `**bold**` becomes **bold**;
- `[see this website](website_url)` becomes see this website;
- you can also insert inline equations between a pair of single dollar sign, e.g. `$E=mc^2$` is displayed as $E = mc^2$;
- Block equations go in between a pair of double dollar signs, e.g. `$$E=mc^2$$`

$$E = mc^2$$

- Lines starting with asterisk $*$ as well as plus $+$ or minus $-$ signs introduce lists.

3. *code chunks*: portions where to insert some code, e.g. for R script. A code chunk takes a language engine (R, Pyhton, . . .), and a label, that can be used for navigating through error messages. Duplicated labels lead to errors during compilation. Some of the most used options:

- `include = FALSE`: do not include the output;

- `echo = FALSE`: do not print the code;

- `warning = FALSE`: exclude the warning message;

- `comment = FALSE`: exclude the comments;

- `eval = FALSE`: do not run the code;

- `out.height = 50%, out.width = 50%`: define the size of figures as they appear in the output document;

- `fig.align="center"`: Define the alignment of figures - left, right, or center;

- `fig.caption="my caption"`: Define the caption for figures.
  The complete list of options is available at https://yihui.org/knitr/options.
  It is possible to include a figure in a Rmarkdown document:

- `![Figure Caption](figure.extension)` in the text;

- `knitr::include_graphics("Images/figure.extension")` in a code chuck;

- directly using R functions in a code chuck.

Basics concepts

For a detailed introduction to R sintax and commands please see the book (Long and Teetor 2019) or the Supplemental material with code examples and exercises of the R cookbook.

R has a very simple syntax. It is *case sensitive*, meaning it distinguishes between uppercase and lowercase letters. The basic commands include *assignments* and *expressions*. When an expression, consisting of certain operations, is written in the prompt, it is interpreted as a command, executed, and its result is displayed (unless specified otherwise) but immediately lost. After the expression is executed, the result is stored in a variable and not displayed. For example:

```
x <- 2 + 3
x
```

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

In this case, we need to explicitly request the value to display it.
Commands can be separated by a semicolon (;) if written on the same line. Otherwise, a newline is sufficient. While using semicolons reduces the number of lines, it is discouraged as it affects code readability.

To obtain more information about any function, you can use the help command. For example, to learn about the mean function:

```
help(mean)
```

or alternatively `?mean`. Comments can be added to the code using the `#` symbol. Everything following the `#` until the end of the line is considered a comment.

Basic mathematical operations for addition, subtraction, multiplication, and division use the standard symbols $+$, $-$, $*$, and $/$. For exponentiation, use $**$ or $\hat{}$. The output of a command is preceded by an index enclosed in square brackets. For example:

```
2 + 3 * 4
```

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

Operations follow the standard order of operations.

The following table summarizes the comparison and logical operators in R:

Operator	Description
$<$	Less than
$>$	Greater than
$\leq$	Less than or equal
$\geq$	Greater than or equal
$==$	Equal to
$!=$	Not equal to
$\&$	AND
$!$	NOT

These operators are primarily used to verify conditions in R. For example

```
4 * 2 == 2^3
```

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

Note: In R, TRUE and FALSE are binary values where TRUE corresponds to 1 and FALSE to 0.

At the end of an R session, all created objects can be permanently saved in a file with the .RData extension in the current directory. The command history is saved in a .Rhistory file. When R restarts in the same directory, the workspace and history are loaded automatically. It is good practice to set the working directory at the start of a session using `setwd()`

```
setwd("C:/desired/directory")
```

Types of objects

Vectors

R operates on named data structures. The simplest type of structure is a numeric vector, which is an ordered collection of numbers. You can define a vector using the `c()` (concatenate) command as follows

```
x <- c(1, 2, 3, 4)
x
```

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

Here, we have created a vector with 4 elements. Note that the assignment operator `<-` points directly to the object receiving the assignment. Alternatively, the = symbol can be used. Assignments can also be written in the reverse direction

```r
c(1, 2, 3, 4) -> x
```

You can also define a new vector by concatenating existing ones

```r
y <- c(x, 0, x)
```

In this case, we create a vector of 9 elements by concatenating x, 0, and x again. To determine the length of a vector, use the `length()` function:

```r
length(y)
```

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

Arithmetic Operations on Vectors   Operations between vectors are performed element by element. Vectors involved in an operation do not necessarily need to have the same length. If they differ, the shorter vector is recycled until it matches the length of the longer one. For example:

```r
v <- 2 * x + y
```

```
## Warning in 2 * x + y: la lunghezza più lunga dell'oggetto non è un multiplo
## della lunghezza più corta dell'oggetto
```

```r
v
```

```
## [1]  3  6  9 12  2  5  8 11  6
```

Note: If the lengths of the vectors are not multiples of each other, R will still perform the operation but issue a warning.

In addition to basic arithmetic operations, there are commands for specific mathematical functions, such as `log` for logarithm, `sqrt` for square root, `exp` for exponential, `sin` for sine, `cos` for cosine, and `abs` for absolute value. Given a vector `x`, `min(x)` and `max(x)` return the smallest and largest values, respectively. The function `range(x)` returns a vector of length 2 with the minimum and maximum values. `sum(x)` and `prod(x)` return the sum and product of the elements of `x`, respectively. Lastly, the function `sort(x)` returns a vector of the same length as `x` sorted in ascending order.

Sequences   R can easily generate sequences of numbers. For example

```r
1:10
```

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

The colon operator has precedence over other operations. For instance, to generate the first 10 even numbers, you can use

```r
2 * 1:10
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

You can also create descending sequences

```r
10:1
```

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

To create sequences with a specific step size or length, use the `seq()` function. For example

```r
s1 <- seq(-1, 1, by = 0.2)
s1
```

```
##  [1] -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0
```

```r
s2 <- seq(0, 2, length.out = 5)
s2
```

```
## [1] 0.0 0.5 1.0 1.5 2.0
```

Selecting and Modifying Parts of a Vector   You can select a subset of a vector using square brackets
`[]`. The content inside the brackets specifies the indices to extract or modify. Common methods include:

```
y
```

```
## [1] 1 2 3 4 0 1 2 3 4
```
```r
# extract first element
y[1]
```

```
## [1] 1
```
```r
# extract elements from position 3 to 6
y[3:6]
```

```
## [1] 3 4 0 1
```
```r
# extract element in positions 2, 4, 8
y[c(2, 4, 8)]
```

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

We can excludes elements at the specified positions

```r
y[-(1:4)]
```

```
## [1] 0 1 2 3 4
```

or selects elements which satisfy a certain condition (corresponding to TRUE)

```r
y[y > 2]
```

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

You can also modify elements of a vector by assigning new values:

```r
y[y == 1] <- -2
y
```

```
## [1] -2  2  3  4  0 -2  2  3  4
```

Matrices

In general, a matrix is a two-dimensional generalization of a vector. The base function to create a matrix is
`matrix()`. If no arguments are provided, a matrix with a single column is created:

```r
matrix(1:5)
```

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

Note that this is not equivalent to a column vector, as a second dimension is still defined. The indices shown
in the output indicate the row and column of the element within the matrix. To create a matrix with specific
dimensions, you need to specify them:

```r
matrix(1:6, nrow = 2)
```

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

By default, matrices are filled column-wise. To fill them row-wise, use the `byrow = TRUE` argument:

```
matrix(1:6, nrow = 2, byrow = TRUE)
```

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

You can also specify the number of columns using the `ncol` argument. Given a matrix `A`, you can determine its dimensions using the `dim()` function, which returns a vector with the number of rows and columns.

Constructing Matrices from Existing Vectors A matrix can also be built from an existing vector. Consider the following vector containing data on the area, population, and altitude of three Italian cities:

```
dati <- c(157.88, 120709, 194, 52.29, 107816, 262, 140.86, 395149, 54)
```

Using the `matrix` function, we can create the `province_dati` matrix:

```
province_dati <- matrix(dati, nrow = 3, byrow = TRUE)
province_dati
```

```
##        [,1]   [,2] [,3]
## [1,] 157.88 120709  194
## [2,]  52.29 107816  262
## [3,] 140.86 395149   54
```

```
dim(province_dati)
```

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

Naming Rows and Columns Row and column names can be assigned using the `rownames()` and `colnames()` functions:

```
rownames(province_dati) <- c("Trento", "Bolzano", "Bologna")
colnames(province_dati) <- c("Area", "Population", "Altitude")
province_dati
```

```
##           Area Population Altitude
## Trento  157.88     120709      194
## Bolzano  52.29     107816      262
## Bologna 140.86     395149       54
```

Alternatively, the `dimnames()` function can be used to assign both row and column names simultaneously

```
dimnames(province_dati) <- list(c("Trento", "Bolzano", "Bologna"),
                                c("Area", "Population", "Altitude"))
```

To retrieve row or column names without assigning new ones

```
colnames(province_dati)
```

```
## [1] "Area"       "Population" "Altitude"
```

```
dimnames(province_dati)
```

```
## [[1]]
## [1] "Trento"  "Bolzano" "Bologna"
```

```
##
## [[2]]
## [1] "Area"       "Population" "Altitude"
```

Accessing Matrix Elements   To access specific elements of a matrix, use square brackets `[row, column]`. For example, to retrieve the population of Trento (row 1, column 2):

```
province_dati[1, 2]
```

```
## [1] 120709
```

To retrieve all data for Bolzano (row 2):

```
province_dati[2, ]
```

```
##       Area Population   Altitude
##      52.29  107816.00     262.00
```

You can also use row and column names for indexing:

```
province_dati["Bologna", "Area"]
```

```
## [1] 140.86
```

Operations on Matrices   Matrices support element-wise operations such as addition, subtraction, multiplication, and division. Consider the following matrices

```
A <- matrix(0:5, nrow = 2, ncol = 3)
B <- matrix(seq(0, 10, 2), nrow = 2, ncol = 3)
```

Element-wise addition and subtraction:

```
A + B
```

```
##      [,1] [,2] [,3]
## [1,]    0    6   12
## [2,]    3    9   15
```

```
A - B
```

```
##      [,1] [,2] [,3]
## [1,]    0   -2   -4
## [2,]   -1   -3   -5
```

Element-wise multiplication and division:

```
A * B
```

```
##      [,1] [,2] [,3]
## [1,]    0    8   32
## [2,]    2   18   50
```

```
A / B
```

```
##      [,1] [,2] [,3]
## [1,]  NaN  0.5  0.5
## [2,]  0.5  0.5  0.5
```

Division by zero results in `NaN` (Not a Number), indicating invalid numeric operations.

Adding Rows and Columns   You can add rows or columns to an existing matrix using `rbind()` and `cbind()`. For example, to add a column for population density:

```
Density <- province_dati[, "Population"] / province_dati[, "Area"]
Density <- round(Density, digits = 2)
province_dati <- cbind(province_dati, Density)
province_dati
```

```
##              Area Population Altitude Density
## Trento   157.88     120709      194  764.56
## Bolzano  52.29      107816      262 2061.89
## Bologna 140.86      395149       54 2805.26
```

To add a new row for another city:

```
Trapani <- c(273.13, 65431, 3, 239.56)
province_dati <- rbind(province_dati, Trapani)
province_dati
```

```
##              Area Population Altitude Density
## Trento   157.88     120709      194  764.56
## Bolzano  52.29      107816      262 2061.89
## Bologna 140.86      395149       54 2805.26
## Trapani 273.13      65431        3  239.56
```

Multi-Dimensional Arrays

Beyond matrices, R supports higher-dimensional arrays. For example:

```
x <- array(1:24, c(3, 4, 2))
x
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

This creates a 3x4x2 array. Array dimensions and elements can be accessed in a similar way to matrices.

Qualitative Variables (Factor)

An object of type `factor` is used to indicate a discrete classification (or grouping) of the elements in a vector of the same length. For example, consider the data for the height of 30 individuals whose gender is specified as:

```
genere <- c("F", "M", "M", "F", "M", "F", "F", "F", "M", "F",
            "M", "M", "F", "F", "M", "M", "M", "F", "F", "M")
```

A factor is created using the `factor()` function

```
generef <- factor(genere)
generef
```

```
##  [1] F M M F M F F F M F M M F F M M M F F M
## Levels: F M
```

Factors have a representation slightly different from vectors. They include the possible `levels` that the variable can take. To retrieve the levels of a factor

```
levels(generef)
```

```
## [1] "F" "M"
```

Continuing the example, consider the following height values for individuals, expressed in centimeters

```
Altezza <- c(145.1787, 185.0102, 195.1313, 161.1447, 186.5746, 148.6862,
             166.3729, 150.0860, 175.6791, 157.0512, 167.5121, 188.3428,
             152.6443, 171.8734, 181.6952, 187.6279, 200.6651, 153.4996,
             163.1848, 180.1263)
```

We can perform operations on this vector with respect to the levels of the factor using the `tapply()` function

```
tapply(Altezza, generef, mean)
```

```
##        F        M
## 156.9722 184.8365
```

Here, we computed the average height of individuals grouped by gender (using the factor variable). The `tapply` function requires the vector to operate on, a factor of the same length, and the function to apply. The output displays the results labeled by the levels used. Similarly, you can use the by() function:

```
by(Altezza, generef, mean)
```

```
## generef: F
## [1] 156.9722
## ------------------------------------------------------------
## generef: M
## [1] 184.8365
```

Ordered Factors   By default, factor levels are sorted alphabetically. To specify a custom order for the levels, use the levels argument within the factor() function. For instance, consider a variable `Grado` representing education levels with levels ordered as 'S: diploma superiore' < 'L: laurea' < 'D: dottorato':

```
Grado <- c("S", "D", "D", "S", "D", "S", "L", "S", "S", "L",
           "S", "D", "S", "S", "L", "L", "S", "D", "D", "D")
Grado
```

```
##  [1] "S" "D" "D" "S" "D" "S" "L" "S" "S" "L" "S" "D" "S" "S" "L" "L" "S" "D" "D"
## [20] "D"
```

```
Gradof <- factor(Grado, levels = c("S", "L", "D"), ordered = TRUE)
Gradof
```

```
##  [1] S D D S D S L S S L S D S S L L S D D D
## Levels: S < L < D
```

Alternatively, you can use the ordered() function to create an ordered factor:

```
ordered(Grado, levels = c("S", "L", "D"))
```

```
##  [1] S D D S D S L S S L S D S S L L S D D D
## Levels: S < L < D
```

Data Frames

Suppose we want to add a column to the matrix `province_dati` containing the corresponding province abbreviations

```
Sigla <- c("TN", "BZ", "BO", "TP")
```

Using the `cbind()` function, we can add this vector to the matrix

```
province_dati1 <- cbind(province_dati, Sigla)
province_dati1
```

```
##          Area      Population Altitude Density    Sigla
## Trento  "157.88" "120709"    "194"    "764.56"  "TN"
## Bolzano "52.29"  "107816"    "262"    "2061.89" "BZ"
## Bologna "140.86" "395149"    "54"     "2805.26" "BO"
## Trapani "273.13" "65431"     "3"      "239.56"  "TP"
```

However, adding a column of strings converts all numeric vectors in the matrix to strings (note the quotation marks). This happens because arrays (of any dimension) can only contain objects of the same type. To avoid this issue, we use data frames, which are similar to matrices but can contain vectors of different types. Create a data frame with the `data.frame()` function

```
province_frame <- data.frame(province_dati, Sigla)
province_frame
```

```
##           Area Population Altitude Density Sigla
## Trento  157.88    120709      194  764.56    TN
## Bolzano  52.29    107816      262 2061.89    BZ
## Bologna 140.86    395149       54 2805.26    BO
## Trapani 273.13     65431        3  239.56    TP
```

Additionally, columns can be accessed directly using the `$` operator:

```
province_frame[, 3]
```

```
## [1] 194 262  54   3
```

```
province_frame$Altitude
```

```
## [1] 194 262  54   3
```

Alternatively, you can attach the data frame to make its columns directly accessible by name

```
attach(province_frame)
Population
```

```
## [1] 120709 107816 395149  65431
```

To remove direct access, use the `detach()` function:

```
detach(province_frame)
```

To get an overview of a data frame's structure, use the `str()` function:

```
str(province_frame)
```

```
## 'data.frame':    4 obs. of  5 variables:
##  $ Area      : num  157.9 52.3 140.9 273.1
##  $ Population: num  120709 107816 395149 65431
##  $ Altitude  : num  194 262 54 3
##  $ Density   : num  765 2062 2805 240
```

```
##  $ Sigla     : chr  "TN" "BZ" "BO" "TP"
```

Lists

Data frames generalize matrices, allowing operations on objects that can include numeric vectors. A more flexible structure is the list, which allows combining objects of different types and dimensions, such as matrices, character vectors, and data frames. For example, given the following elements

```
A <- matrix(c(1, 0, 0, 1), nrow = 2)
regioni <- c("Trentino-Alto Adige", "Emilia Romagna", "Sicilia")
```

We can combine them into a single list:

```
lista <- list(A, regioni = regioni, province_frame)
lista
```

```
## [[1]]
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
##
## $regioni
## [1] "Trentino-Alto Adige" "Emilia Romagna"       "Sicilia"
##
## [[3]]
##           Area Population Altitude Density Sigla
## Trento  157.88     120709      194  764.56    TN
## Bolzano  52.29     107816      262 2061.89    BZ
## Bologna 140.86     395149       54 2805.26    BO
## Trapani 273.13      65431        3  239.56    TP
```

List elements can be accessed using double square brackets [[ ]] or their assigned names

```
lista[[1]]
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
lista[["regioni"]]
```

```
## [1] "Trentino-Alto Adige" "Emilia Romagna"       "Sicilia"
```

You can add new elements to a list using the $ operator

```
lista$vettore <- 1:5
lista
```

```
## [[1]]
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
##
## $regioni
## [1] "Trentino-Alto Adige" "Emilia Romagna"       "Sicilia"
##
## [[3]]
##           Area Population Altitude Density Sigla
## Trento  157.88     120709      194  764.56    TN
```

```
## Bolzano  52.29      107816      262 2061.89      BZ
## Bologna 140.86      395149       54 2805.26      BO
## Trapani 273.13       65431        3  239.56      TP
##
## $vettore
## [1] 1 2 3 4 5
```

The functions `lapply()` and `sapply()` can be used to perform operations on all elements of a list. Note: Internally, data frames are a special case of lists where each list element is a vector of the same length.

Special Operators

In R, several special functions and objects are included. If some requested information is unavailable, it is conventionally represented with `NA`. For example:

```
province_frameNA <- province_frame
province_frameNA$Population[2] <- NA
province_frameNA$Population
```

```
## [1] 120709     NA 395149  65431
```

Any operation involving a missing value will result in a missing value:

```
min(province_frameNA$Population)
```

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

Many functions allow you to ask R to remove missing data using the argument `na.rm = TRUE`

```
min(province_frameNA$Population, na.rm = TRUE)
```

```
## [1] 65431
```

To check if a vector contains NA, use the `is.na()` function, which returns `TRUE` or `FALSE` for each element

```
is.na(province_frameNA$Population)
```

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

To determine if there is at least one missing value in a vector, use the `any()` function

```
any(is.na(province_frameNA$Population))
```

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

To find the positions of missing values, use the `which()` function

```
which(is.na(province_frameNA$Population))
```

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

Note: The `which()` function generally takes a vector of `TRUE` and `FALSE` and returns the indices corresponding to `TRUE`. Other special objects (with analogous functions) include:

- `Inf` to indicate infinity (`is.finite()`).
- `NaN` to indicate "not a number (`is.nan()`).

Loops and Conditionals

In R, conditional statements can be used in the following form

```
if (expr1) expr2 else expr3
```

Here, `expr1` is evaluated. If it returns `TRUE`, `expr2` is executed; otherwise, `expr3` is executed. For example:

```r
if (province_frame["Trento", "Population"] > 100000) {
  print("The population of Trento exceeds 100,000 inhabitants")
} else {
  print("The population of Trento does NOT exceed 100,000 inhabitants")
}
```

```
## [1] "The population of Trento exceeds 100,000 inhabitants"
```

You can use logical operators (AND, OR, NOT) in the conditions. Specifically, `&` and `|` operate element-by-element for vectors, while `&&` and `||` evaluate only the first element of each vector.

```r
for (name in expr1) expr2
```

Here, `name` is the loop variable, `expr1` is a vector (usually a sequence like 1:20), and `expr2` is the expression executed during each iteration of the loop. To execute multiple commands in a loop or conditional, group them with curly braces {}. For example

```r
for (i in 1:4) {
  if (!is.na(province_frameNA$Population[i])) {
    print(paste("The city of", rownames(province_frameNA)[i],
                "has", province_frameNA$Population[i], "inhabitants."))
  }
}
```

```
## [1] "The city of Trento has 120709 inhabitants."
## [1] "The city of Bologna has 395149 inhabitants."
## [1] "The city of Trapani has 65431 inhabitants."
```

Another way to create loops is using the while function

```r
while (condition) expr
```

Here, `expr` is executed as long as condition evaluates to `TRUE`.

Functions

In R, it is possible to create custom functions. These objects are saved and can be used in other expressions. Writing useful functions allows for more efficient and productive work in R. So far, we've encountered many built-in functions (e.g., `mean()`, `matrix()`, `cbind()`). In general, a function is defined as:

```r
name <- function(arg1, arg2, ...) {
  expression
  return()
}
```

- `name` is the function's name (used to call the function),
- `(arg1, arg2, ...)` are the required inputs,
- `expression` contains the commands making up the function body,
- `return()` specifies the function's output. For example, let's write a function to calculate the absolute value. The function takes a numeric vector x as input and returns a vector of the same length with the absolute values:

```r
absolute_value <- function(x) {
  # If x > 0, then |x| = x
  result <- x
  # If x < 0, then |x| = -x
  result[x < 0] <- -x[x < 0]
```

```
    return(result)
}

v <- c(-1, 3, -2, 6, -5)
absolute_value(v)
```

[1] 1 3 2 6 5

Note: The `return()` command is not mandatory. Instead, you can simply write the object to be returned as the last command. For example, the previous function can be rewritten as

```
absolute_value <- function(x) {
  result <- x
  result[x < 0] <- -x[x < 0]
  result
}
v <- c(-1, 3, -2, 6, -5)
absolute_value(v)
```

[1] 1 3 2 6 5

Solving a Linear Equation Now let's create a function to solve a first-degree equation $ax + b = 0$. The solution formula is $x = -a/b$. The function accounts for the special case where $a = 0$.

```
linear_eq <- function(a, b) {
  if (a != 0) {
    return(-b / a)
  } else {
    print("No solution exists")
  }
}
linear_eq(2, -4)
```

[1] 2

Note: Functions are sensitive to the order of input arguments. For example:

```
linear_eq(-4, 2)
```

[1] 0.5

To avoid errors due to argument order, specify the input names

```
linear_eq(b = -4, a = 2)
```

You can also assign default values to function arguments when defining the function:

```
linear_eq <- function(a = 1, b = 1) {
  if (a != 0) {
    return(-b / a)
  } else {
    print("No solution exists")
  }
}
linear_eq()
```

[1] -1

Note: Assignments made inside a function are local and temporary, and they are discarded once you exit the function.

Exercise 1   Write a function to solve a second-degree equation.

Data Management

Often, we work with large data sets, and re-executing commands repeatedly can be time-consuming and error-prone. For this reason, it is possible to save data to a file, which can be loaded later as needed. Suppose we want to create a data frame and save it. For example, consider the following variables:

- **X**: marital status (`Nubile/celibe`, `Coniugato`, `Vedovo/a`, `Separato/divorziato`);
- **Y**: level of education (`Analfabeta`, `scuola dell'Obbligo`, `scuola Superiore`, `Laurea o superiore`);
- **Z**: number of dependents (from 0 to 4);
- **W**: weight. To simplify the analysis:
- Define X, Y, and Z as factors, where N, C, V, S are the levels of X and A < O < S < L are the ordered levels of Y.
- Sample values of W from a normal distribution with a mean of 70 and standard deviation of 15.
- Assume 50 individuals.

Note: The functions used to generate this example will be introduced and explained later.

```r
x <- factor(c("N", "C", "V", "S"))
y <- factor(c("A", "O", "S", "L"))
z <- factor(0:4)
n <- 50
set.seed(37)
X <- sample(x, n, replace = TRUE)
Y <- sample(y, n, replace = TRUE)
Z <- sample(z, n, replace = TRUE)
W <- rnorm(n, mean = 70, sd = 15)
dati <- data.frame(X = X, Y = Y, Z = Z, W = W)
```

Since the data frame contains 50 observations, printing all rows to inspect the data set might not be helpful. Besides the `str()` function, we can use `head()` to print only the first few rows of the data frame

```r
str(dati)
```

```
## 'data.frame':    50 obs. of  4 variables:
##  $ X: Factor w/ 4 levels "C","N","S","V": 1 4 3 1 4 3 2 3 2 3 ...
##  $ Y: Factor w/ 4 levels "A","L","O","S": 4 2 1 2 1 4 2 1 2 4 ...
##  $ Z: Factor w/ 5 levels "0","1","2","3",..: 4 4 3 4 2 1 1 4 1 1 ...
##  $ W: num  67.4 92 69.6 62.2 62.8 ...
```

```r
head(dati)
```

```
##   X Y Z        W
## 1 C S 3 67.36363
## 2 V L 3 91.96749
## 3 S A 2 69.55256
## 4 C L 3 62.16716
## 5 V A 1 62.82927
## 6 S S 0 78.02315
```

Saving and Loading Data   The easiest way to save an entire session or parts of it is by using the `save()` function, where you specify the objects to be saved. For example

```r
save(X, Y, Z, W, dati, file = "peso.Rdata")
```

This creates a file named `peso.Rdata` in the working directory, containing the specified variables. If we only want to save the data frame and remove the other variables

```r
save(file = "peso.Rdata", dati, X, Z)
rm(list = ls()) # To remove all object in the working enviroment
```

To load the saved session, use the `load()` function:

```r
load("peso.Rdata")
ls()
```

```
## [1] "dati" "X"    "Z"
```

Note: Ensure that the file is in the working directory before attempting to load it. If it is not, you must specify the full file path.

If the object to save is a data frame or matrix (i.e., a table), you can use `write.table()` to save it as a `.txt` file (a compact format). For example

```r
write.table(dati, file = "peso.txt", sep = ";")
dati2 <- read.table("peso.txt", sep = ";", header = TRUE)
head(dati2)
```

```
##   X Y Z        W
## 1 C S 3 67.36363
## 2 V L 3 91.96749
## 3 S A 2 69.55256
## 4 C L 3 62.16716
## 5 V A 1 62.82927
## 6 S S 0 78.02315
```

The character specified in `sep = ""` is used to separate table elements. The `header = TRUE` argument indicates that the file has a header row identifying variable names. For structured data sets, you can use the following functions, which are variations of `read.table()`: `read.csv` and `read.csv2` for CSV files, the function `readxl` from the package `"readxl"` for XLS and XLSX files.

Basic Tidyverse

Tidyverse is a collection of R packages designed for data science. Tidyverse packages are:

- optimised to run fast, relying on C++ under the hood
- maintained by a paid staff of talented developers

```r
install.packages("tidyverse")
```

```r
library(tidyverse)
```

```
## Warning: il pacchetto 'tidyverse' è stato creato con R versione 4.3.2

## Warning: il pacchetto 'ggplot2' è stato creato con R versione 4.3.3

## Warning: il pacchetto 'stringr' è stato creato con R versione 4.3.3

## Warning: il pacchetto 'lubridate' è stato creato con R versione 4.3.2

## -- Attaching core tidyverse packages ----------------------- tidyverse 2.0.0 --
## v dplyr     1.1.3     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
```

```
## v lubridate 1.9.3     v tidyr      1.3.0
## v purrr      1.0.2
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

Tidyverse is based on the pipe operator i.e.,%>%. It connects two function calls by making the value returned by the first call the first argument of the second call

```r
c(1:3) %>% rep(3)
```

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

returns the same output as

```r
rep(c(1:3), 3)
```

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

We can use the pipe more than one times:

```r
c(1:3) %>% rep(3) %>% length()
```

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

By default the value of the left-hand call is piped into the right-hand call as the first argument. You can make it some other argument by referring to it as the dot ., for example:

```r
3 %>% rep(c(1:3), .)
```

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

dplyr

Inside the `tidyverse` function we have the useful `dplyr` package. We will see only the main functions of this package, but you can use the basic R command proposed before instead of these ones.

- `filter()`: Subset rows using column values
- `select()`: Subset columns using their names and types
- `mutate()`: Create, modify, and delete columns
- `group_by()`: Group by one or more variables
- `summarize()`: Summarise each group to fewer rows

Let see some examples. First of all, we load the R dataset `ChickWeight` from the package `datasets`

```r
library(datasets)
data("ChickWeight")
```

You can see the description of this dataset typing help(ChickWeight). So, we want: 1. select only the weight and chick columns:

```r
ChickWeight %>% select(weight, Chick) %>% head()
```

```
##   weight Chick
## 1     42     1
## 2     51     1
## 3     59     1
## 4     64     1
## 5     76     1
## 6     93     1
```

and compute the mean of the weight for each chick:

```
ChickWeight %>% select(weight, Chick) %>%
  group_by(Chick) %>%
  summarize(weight = mean(weight, na.omit = TRUE))
```

```
## # A tibble: 50 x 2
##    Chick weight
##    <ord>  <dbl>
##  1 18        37
##  2 16      49.7
##  3 15      60.1
##  4 13      67.8
##  5 9       81.2
##  6 20      78.4
##  7 10      83.1
##  8 8         92
##  9 17      92.5
## 10 19      86.8
## # i 40 more rows
```

filter only the chick with weight greater than the global mean

```
ChickWeight %>% select(weight, Chick) %>%
  group_by(Chick) %>%
  summarize(weight = mean(weight, na.omit = TRUE)) %>%
  filter(weight >= mean(weight))
```

```
## # A tibble: 27 x 2
##    Chick weight
##    <ord>  <dbl>
##  1 11      130.
##  2 2       120.
##  3 5       127.
##  4 14      151.
##  5 7        150
##  6 28      130.
##  7 26       131
##  8 25      143.
##  9 29      142.
## 10 21      184.
## # i 17 more rows
```

For more information please see the Tidyverse Cookbook at https://rstudio-education.github.io/tidyverse-cookbook/.

Packages for Neuroscience

Ok all good, but what if I want to analyze EEG data, fMRI or other data from neuroscience? The Neuroconductor Project (https://neuroconductor.org/) is a large repository of R packages for imaging. It includes 86 packages published on CRAN! You can find the complete list with all the documentations at https://neuroconductor.org/list-packages/all. For example:

- `fmri`: perform an fMRI analysis as described in Tabelow et al. (2006), Polzehl et al. (2010), and Tabelow and Polzehl (2011).
- `eegUtils`: EEG processing and visualization tools.

- `RNifti`: Provides very fast read and write access to images stored in the NIfTI-1, NIfTI-2 and ANALYZE-7.5 formats.

and there are other packages not on CRAN, like:

- `eeguana`: flexible manipulation of EEG data with dplyr-like functions. You must install it using `devtools::install_github("bnicenboim/eeguana")`.

References

Chén, Oliver Y. 2019. "The Roles of Statistics in Human Neuroscience." *Brain Sciences* 9 (8). https://doi.org/10.3390/brainsci9080194.

Durstewitz, Daniel. 2017. *Advanced Data Analysis in Neuroscience. Bernstein Series in Computational Neuroscience. Cham: Springer.* Springer.

Kass, R. E., S. I. Amari, K. Arai, E. N. Brown, C. O. Diekman, M. Diesmann, B. Doiron, et al. 2018. "Computational Neuroscience: Mathematical and Statistical Perspectives." *Annu Rev Stat Appl.* 5: 183–214. https://doi.org/10.1146/annurev-statistics-041715-033733.

Long, James D, and Paul Teetor. 2019. *R Cookbook: Proven Recipes for Data Analysis, Statistics, and Graphics.* O'Reilly Media.

Xie, Yihui, Joseph J Allaire, and Garrett Grolemund. 2018. *R Markdown: The Definitive Guide.* Chapman; Hall/CRC.