ESSENTIALS OF R

BE603 – DATA ANALYTICS III

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

1. R & Statistical Programming

`R` is the statistical programming software used in the Data Analytics course. By *statistical programming*, we mean that `R` is a programming language designed for statistics. It has a human-readable vocabulary and grammar and needs to be interpreted by your computer through the `R` software. In this tutorial, I present the essentials about `R` for Data Analytics.

1.1 Books that are worth reading

In this course, we use R as a tool to perform some analyses. To go further than what we cover, I can only recommend you the following resources:

- Wickham, H., & Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* O'Reilly Media, Inc. https://r4ds.had.co.nz/
  - This book is an introduction to R. It teaches the workflow basics in R, data import, cleaning, types, visualization, and elementary modeling. That is pretty much what we do in this class.
- Wickham, H. (2016). *ggplot2: elegant graphics for data analysis.* Springer. https://ggplot2-book.org/
  - This book is a must-read if you ever want to create beautiful graphs. It tells all you need to know about *ggplot2*, the package to generate plots with R.
- **StackOverflow:** https://stackoverflow.com/questions/tagged/r
  - Part of the Stack Exchange network, StackOverflow is a Q&A community website for people working in programming. It is a great place to search for answers to your questions. Tons of excellent R users and developers interact daily on StackOverflow. The website is also very well designed: if you state a specific problem, you will likely find an example code with a ready-made solution.
- **ChatGPT**: https://chat.openai.com/auth/login
  - ChatGPT is not a website to learn R, but it can help generate example code when you get stuck. ChatGPT works well if you formulate small and precise requests. **WARNING:** For complex requests, the code generated will likely contain mistakes with the current version of ChatGPT.
  - For this course, I encourage you to avoid using ChatGPT and always think of solutions yourself, as a learning process.

2. How to insteall R and RStudio ?

2.1 R interpreter

R is an interpreted language. It means that the R code is readable by a human, and an interpreter will translate it into something readable by a computer when you try to run it. You must install an

interpreter on your computer to run R code. Find below direct download links to the R interpreter software:

- For **Windows**: https://cran.r-project.org/bin/windows/base/R-4.3.1-win.exe
- For macOS 11 (**Big Sur**), with Apple Arm processors:

  https://cran.r-project.org/bin/macosx/big-sur-arm64/base/R-4.3.1-arm64.pkg

2.2 RStudio

The R interpreter is a command line tool. To interact with it, you need need RStudio, an Integrated Development Environment (IDE) for R.

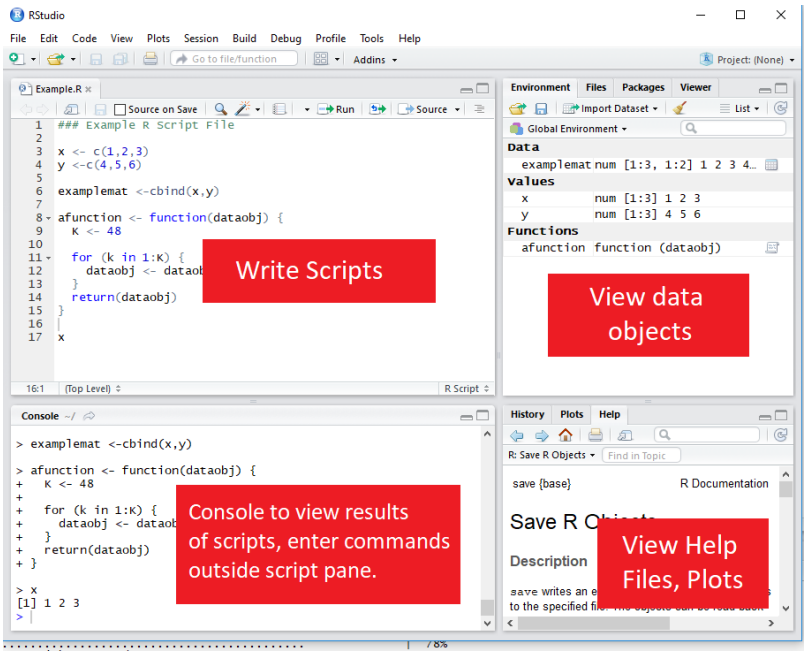
- Download links to RStudio: https://posit.co/download/rstudio-desktop/#download

3. Getting Started in RStudio

3.1 RStudio GUI

RStudio is an integrated development environment (IDE). This means that in addition to a script editor, it also lets you view your environments, data objects, plots, help files, *etc*.

There are four parts in the RStudio screen. They can be rearranged as it pleases you through `Tools` > `Global options` > `Pane layout`.



The RStudio Graphical User Interface (GUI).

1. The **script** pane is a text editor. This is where you read and write the script files. In general, you will work with `.qmd` files, that I describe below.
2. The **console** pane shows the R interpreter, where you see the outputs of the scripts you ran.

3. The two other panes contain multiple tabs, including:
   - The **environment** tab lists all the objects loaded in your environment[1]: vectors, functions, dataframes, etc. (we will introduce them later in this course). In R, you can simultaneously load as many objects and data frames as you want in your environment (Stata is limited to one dataframe per session). In R, the only limit is the memory of your machine.
   - The **Files** tab is a file explorer of your working directory, convenient to find and open the script files you want to read in the script pane.
   - The **Plot** tab shows all the plots you have made since you opened your R session.

3.2 RStudio projects

In R, you work with projects. A project is a folder on your computer containing your scripts (the R code), your data, and your outputs. In this course, we always provide you with the project folder (compressed as zip files on Canvas).

To open the project (containing data, scripts, or anything else), you can choose `File` > `Open project` in RStudio or open the `.Rproj` file created in your project directory from your file explorer.

Warning! Do **NOT** open scripts from your file explorer; always open the `.Rproj` file first. Otherwise, it is as if you were not using the project infrastructure.

3.2.1 Executing code from the script

- To execute a code section, highlight the code and click "**Run**" or use `CTRL`+`ENTER`.
- You can also click on the small green arrow at the top of any code chunk to run it.

3.2.2 A few essential good practices

- Data are stored in a dedicated folder (usually called `data\`). This is the case in all the projects we provide you in this course. If you share your work with others, please be careful of your right to share the data and GDPR compliance.
- You should **NEVER EVER EVER** modify your original data files[2]: if you modify the original files, you may lose precious data. Always save your changes into new data files.

---

[1] The environment contains the objects that are ready to use for your statistical purposes. As you are writing your scripts, and running your model, you may accumulate a lot of objects in your environment, some of which may have taken hours to run. In the global options, you can set it to automatically save your environment on close in an `.RData` file, and load it back where you stopped when reopening your project. If you want to empty your environment from previous objects, you can click on the small wiper in your environment tab, called *Clear objects from the workspace*. Saving or cleaning your environment **does not affect** your data files.

[2] I know this is a common practice among users of Excel, Stata, and others, but this is highly discouraged; you should avoid that.

4. Packages in R

Packages in R are kinds of extensions. They can bring new functions or data to the base R software. In this course, all the projects starts with a few lines of code to install the package you need for that specific projects. Run these lines before joining the seminars. Alternatively, to install a package from RStudio, click Tools > Install packages..., type the package name you want, and click `Install`.

- For example, install `rio` for easy data import, export (saving), and conversion.

```
install.packages("rio")
```

5. Importing & exporting

5.1 Importing using `rio`

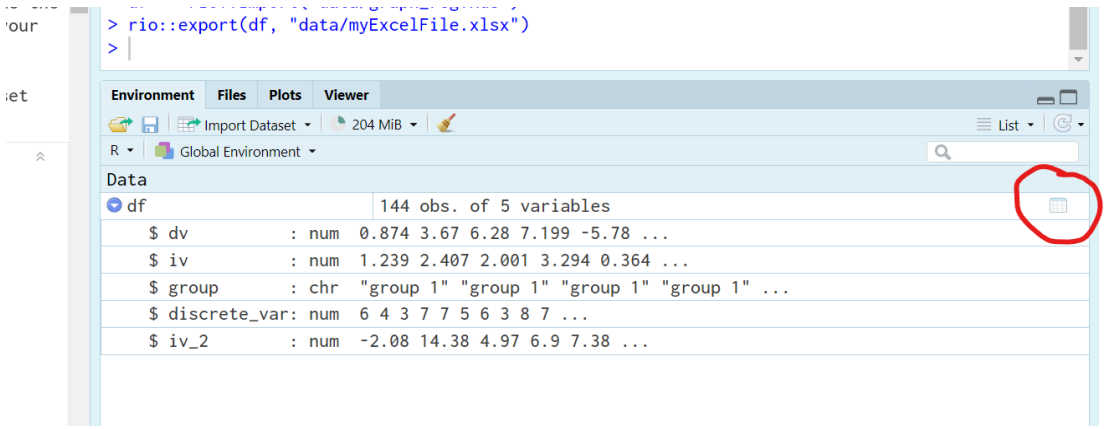
The `rio` package can load more than 30 different data file formats in R, including csv, Excel, SAS, SPSS, Stata, Matlab, JSON, and others.

Use the following command to load the data file named `myExcelFile.xlsx` in your R environment:

```
# Dataset in Excel format
df <- rio::import("data/myExcelFile.xlsx")
```

Notes:

- The arrow `<-` symbol indicates that you want to load the output of the function `rio::import()` into a data.frame called `df`). Then, `df` contains the content of the Excel file you loaded.
- The first part, `data/`, is not part of the filenames: it is the folder where your data files are located in your project directory.
- After you run the two commands above, you should see the data.frame `df` in the `Environment` tab of RStudio: this means that they are loaded correctly in your environment. You can visualize the content of the objects by clicking on the small table at the end of the corresponding line in your `Environment` tab.



Visualize a dataset in RStudio

5.2 Exporting data

If you want to save an object, *e.g.*, `pisa_sas`, into a new file, you can use:

```
# Save data
rio::export(df, "data/test2.xlsx")
```

- The first argument (before the comma `,`) is the name of the object you save. The second argument is the path where you save it on your computer.
- In the example above, the data are saved in an Excel file.

Warning: Some formats will result in a loss of data, for example, when the data types you use are incompatible with the chosen file format.

6. Data as objects

In R, you manipulate objects only. **Everything** in R is an object. An object has:

- **Attribute(s)**, *e.g.*, names, levels, etc.,
- A **type**, *e.g.*, the object is an integer, a character, etc.,
- **Value(s)**, the content of an object

To define an object, use the arrow `<-`. For example, `x <- 4` will assign 4 as the value to an object named x. The type of x will be numeric because 4 is a numeric value.

```
# Define the object named x with a numeric value of 4
x <- 4

# Define the object y, a copy of x
y <- x

# Define the object z, the result of an addition
z <- 2 + 2

# Redefine z as the square of z
z <- z^2

# Print x, y, and z
x
```

```
[1] 4
```

```
y
```

```
[1] 4
```

```
z
```

```
[1] 16
```

Note: In R, there is no distinction between defining and redefining an object. You can always change an object's values, types, and attributes on the fly.

6.1 Data Types

The R objects may be of different types. The most common types are:

1. **logical:** Data that should be interpreted as a logical statement, *i.e.,* `TRUE` or `FALSE`. This is what you use for dichotomous variables.
2. **integer:** `4L` (The `L` tells R to store the 4 as an integer). These are integer, used for discrete variables.
3. **numeric:** `15.5`. Numerics are floating numbers, used for continuous variables.
4. **character:** `"string or text"`. This is a string of characters. You may use it for nominal, and sometimes, ordinal variables.

In R, you do not need to state the type of an object before assigning a value to it (as is the case in C language, for example). Types are defined on the fly. R can also convert some objects to other types on the fly. For example:

6.2 Mathematical operations on R objects

As we have seen with the examples above, applying mathematical operations on R objects defined above is possible. These include additions, subtractions, multiplication and division, exponentiation, logarithms, or any other mathematical function that has a definition in R.

```
# Addition and Subtraction
2 + 2 - 3
```

```
[1] 1
```

```
# Multiplication and Division
2 * 2 + 2 / 2
```

```
[1] 5
```

```
# Exponentiation and Logarithms
2^2 + log(2)
```

```
[1] 4.693147
```

6.3 Logical operations on R objects

You can also evaluate logical expressions in R:

```
# Defining variables
a <- 5
b <- 7

# Less than
a < b
```

```
[1] TRUE
```

```
# Less than or equal
a <= b
```

```
[1] TRUE
```

```
# More than
a > b
```

```
[1] FALSE
```

```
# Greater than or equal
a >= b
```

```
[1] FALSE
```

```
# Equals
a == b
```

```
[1] FALSE
```

```
# Not equals
a != b
```

```
[1] TRUE
```

```
# Another negation
!TRUE == FALSE
```

```
[1] TRUE
```

You can also use `&` (`AND`) and `|` (`OR`) operations with logical expressions:

```
a <- 12
b <- 15
c <- a

# Is 5 equal to 5 OR is 5 equal to 6?
(a == b) | (a == c)
```

```
[1] TRUE
```

```
# Is 5 equal to 5 AND is 5 equal to 6?
(a == b) & (a == c)
```

```
[1] FALSE
```

7. Dataframes

When you do data analytics, you often manipulate tables, called dataframe in R. They are composed of a series of vectors (i.e. variables) of the *same length* and possibly *different types*. The object `df`, that we loaded before with `rio`, is a dataframe.

```
# Dataset in SPSS format loaded into R
df <- rio::import("data/myExcelFile.xlsx")
```

- Each column of a dataframe is a variable. You can rename the dataframe's columns with the `names()` function, using a condition.

```
# Rename the column of df named "column 1" into "COLUMN A"
names(df)[names(df) == "discrete_var"] <- "discrete_variable"
```

7.1 Subset Variables from a `data.frame`

You can subset elements of a dataframe with element positions and with values. Simply keep in mind that dataframes have two dimensions:

- the first dimension (before the comma) is the rows
- the second dimension (after the comma) is the columns of your dataframe

```
# For example, call the 3rd element of the 2nd column in df
df[3, 2]
```

```
[1] 2.001291
```

- To extract all the values of the 3rd row, write (note the position of the comma, after 3):

```
# Extract all the values of the third row
df[3,]
```

```
        dv       iv   group discrete_variable     iv_2     iv_3 gender
3 6.280088 2.001291 group 1                 3 4.966587 95.26099 female
```

- To extract all the values of the 3rd columns, write (note the position of the comma, before 2):

```
# Extract all the values of the 2nd column
df[,2]
```

- As for vectors, you can extract elements (lines or columns of your tables) using row names, column names, or conditions.

```
# Extract one column by name
iv <- df[,"iv"]
```

Note: To simplify the writing, when you call a column of a dataframe, instead of writing the bracket and comma, you can replace the bracket part with a dollar symbol `$`. It does the same, but the writing is simpler for the general case.

```
# Extract one column by name
iv2 <- df$iv
```

Note that `iv` and `df$iv` are two different objects. The first one is a vector in your environment. The second one is a column of the data.frame `df`.

You can also extract multiple columns at a time `"dv"`, `"iv"`, `"group"`, from `df`. In that case, you cannot use the dollar symbol.

```
# Using brackets
df_2 <- df[, names(df) %in% c("dv", "iv", "group")]
```

- You can also extract the first 25 observations, or a random sample, or observations meeting a condition.

```
# Extract the first 25 observations
df_3 <- df[1:25,]

# Extract a random sample of 25 observations
df_4 <- df[sample(nrow(df), 25), ]

# Extract rows when iv <= 2
df_5 <- df[df$iv <= 2, ]

# You can also combine multiple conditions
df_6 <- df[(df$iv > 3) & (df$dv <= 5),]
```

Note: the conditional subsetting keeps the missing values.

7.2 Identify Missing Values

Does one of your variables contain missing values? You can assess if an observation is missing with the function `is.na()`. Missing values in R are coded `NA`. For example:

```
# Count the number of missing values in the gender variables
sum(is.na(df$gender))
```

```
[1] 4
```

To know the number of cases are found in a categorical or a logical variable, use `table`. The table commands counts all the occurrences of each values of a variable. Example:

```
table(is.na(df$gender))
```

```

FALSE  TRUE 
  140     4 
```

7.2.1 Remove Missing Values

You can remove the missing values using:

```
# Keep only the complete cases in your data.frame
df <- df[complete.cases(df), ]
```

The command above remove all the rows containing missing values on any variables of `df`.

Question: Using `is.na()`, how could you remove rows containing missing values on `gender`?

Solution:

```
# Remove rows containing missing value on the gender variable
df_2 <- df[!is.na(df$gender), ]
```

- `!is.na()` means NOT is NA.

7.2.2 Recode Missing Values

Another problem characteristic of observational data is missing data. In R, the way to represent missing data is with the value `NA`. You can recode missing values that *should be* `NA` but are coded using a different schema by using brackets:

```
# Replace prefer not to say missing data with NA
df[df$gender == "prefer not to say", ] <- NA
```

Note: R does not naturally support multiple types of missingness like other languages, although it's possible to use the `sjmisc` package to do this.

7.3 Add Variables in a `data.frame`: New Columns

Creating new variables that are functions of existing variables in a data set can be done with:

```
# var_3 is 2 * var_2
df$var_3 <- df$iv * 2

# We can also write
df[, "var_3"] <- df[, "iv"] * 2

# Recode a dummy variable from the gender variable
```

```r
df$is_female <- as.integer(df$gender == "female")

# Create a log value
df$iv_log <- log(df$iv_3)
```

8. Graphs in R

8.1 Data visualization overview

One of the strong points of R is creating very high-quality data visualization. R is very good at both "static" and interactive data visualization designed for web use. Today, we will cover static data visualization.

8.2 ggplot2 for data visualization

The main package for publication-quality static data visualization in R is `ggplot2`, which is part of the tidyverse collection of packages. With `ggplot2` you control all the elements of your graphs, from the data you visualize and the way you represent it, to the style of the graph and the quality of the output (high-resolution...).

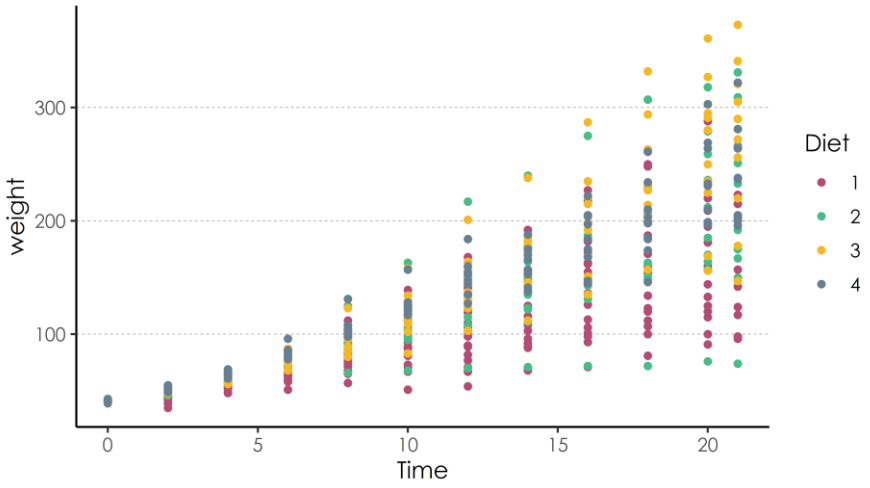


Figure 1: Example of ggplot2 graph, with SSE stylesheet

The workhorse function of ggplot2 is `ggplot2::ggplot()`. The `gg` stands for "*grammar of graphics*". In each `ggplot()` call, the appearance of the graph is determined by specifying:

- the `data`(frame) to be used,
- the `aes`(thetics) of the graph: like size, color, *x* and *y* variables,
- the `geom`(etry) of the graph: the chosen representation (*e.g.* `point`, `histogram`, `bar`, `qq`, `curve`, `density`, `abline`, `boxplot`, `map`, etc., etc.)
- You may then add other functions to your graph, to define, `labs` (the labels), `ggtitle` (the title), `ggtheme` (the theme), etc.

8.3 Scatterplots

First, let's look at a simple scatterplot, which is defined by using the geometry `geom_point()`. Let's use the data from `myExcelFile.xlsx`.

```
# Loading the grammar of graphs
library(ggplot2)
```

```
Warning: package 'ggplot2' was built under R version 4.3.1
```

```
# Loading the data, if not done already
df <- rio::import("data/myExcelFile.xlsx")

# Defining data
ggplot(df) +

  # defining aesthetics: x and y
  aes(x = iv, y = dv) +

  # defining geometry: geom_point is for scatterplot
  geom_point()
```

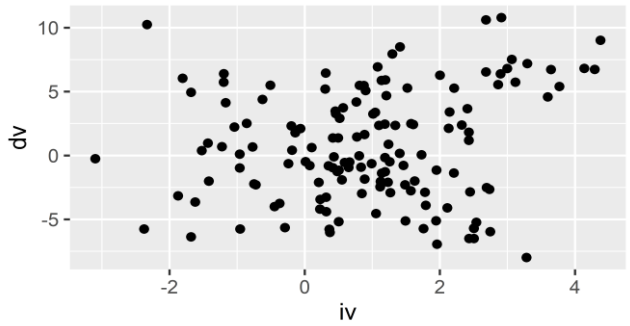


Figure 2: A simple scatterplot

- Each of the graph elements is an R function.
- Functions in `ggplot2` are added up with a `+` symbol.
- Variables names are **NOT** quoted (`""`) in `ggplot2`: they are written directly with their full names (here: `iv`, `dv`)

8.3.1 Colors and shapes

- Graphs can be extensively customized using additional arguments inside of elements, or with additional functions. For example, you may want to **color the element by group,** or you want to combine it with different shapes, in case you print in black and white:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point()
```

Figure 3: Adding colors and shapes

8.4 Axis and titles

Sometimes, you would like to use a customized scale for the axis:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  xlim(-4, 4) + ylim(-10,10)
```

```
Warning: Removed 6 rows containing missing values (`geom_point()`).
```

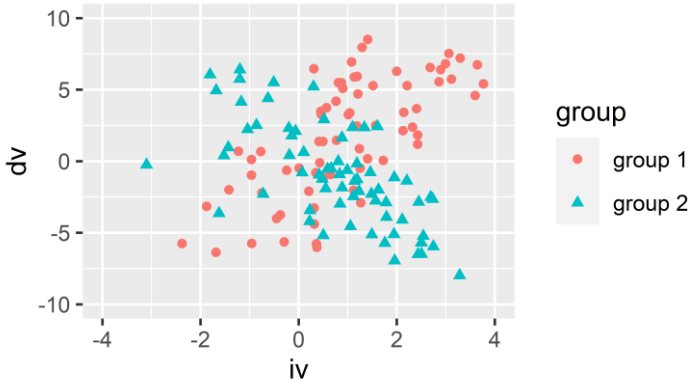


Figure 4: Custom scales

- **Note** the warning message you get: this is because some data points are out of bounds, hence, not plotted on the graph.

You can also change the name of axis, for example, using full labels:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  labs(x = "independent variables",
       y = "dependent variables",
       color = "categories",
       shape = "categories")
```

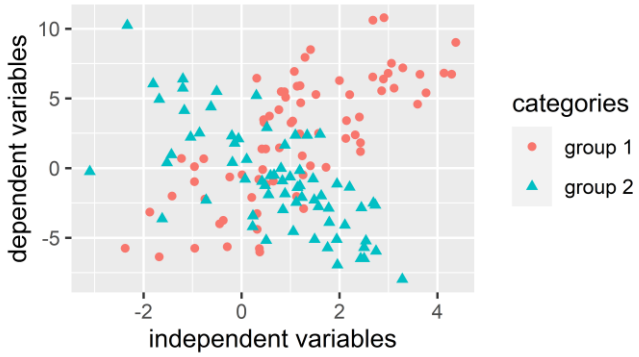


Figure 5: Custom labels

- **Note** that you have to indicate both `color` and `shape` in your graph to change the title of your legend, because you graph include colors and shapes. If you change only one, the legen will be repeated twice: once with the new label you wrote, one with the variable name. Example:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  labs(x = "independent variables",
       y = "dependent variables",
       color = "categories")
```

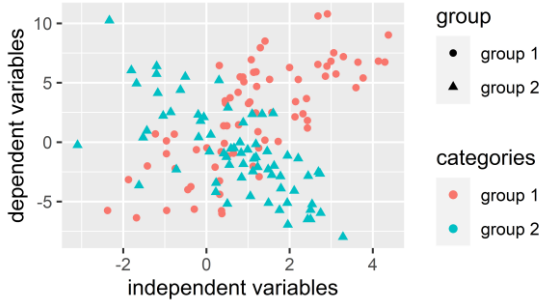


Figure 6: Wrongly assigned shape legend

8.4.1 Lines

To plot a straight line somewhere on your graph, you can use `geom_vline()` (vertical) or `hline` (horizontal):

```
ggplot(df) + aes(x = iv,
                 y = dv) +
  geom_point() +
  geom_vline(xintercept = 1.6,
             color = "#FF5733",
             linetype = "dotted") +
  geom_hline(yintercept = 2.3,
             color = "green",
             linetype = "solid")
```

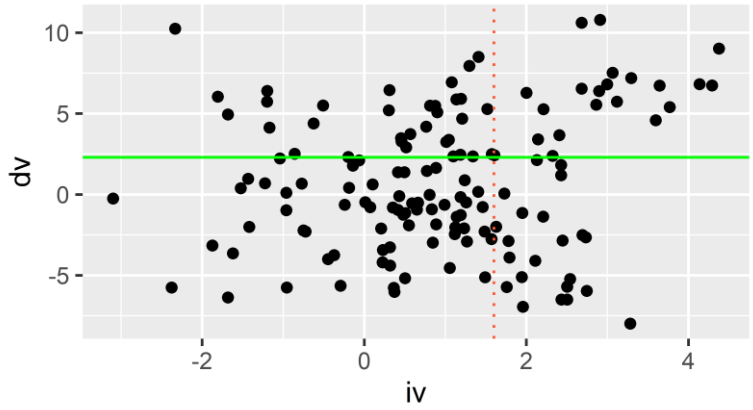


Figure 7: Vertical and horizontal lines

- Here again, you can customize the position, color, or linetype. The seven styles of linetypes are available here.

8.4.2 ablines and regression lines

You can also plot a line across the graph, indicated slope and intercept, using `abline()`, or you can plot a regression lines fitted to your data, using `geom_smooth()`:

```
# With abline (not fitted line)
ggplot(df) +
  aes(x = iv, y = dv) +
  geom_point() +
  geom_abline(intercept = -1,
              slope = 2)

# With geom_smooth (fitted)
ggplot(df) + aes(x = iv,
                 y = dv) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = "y ~ x", se = FALSE)
```

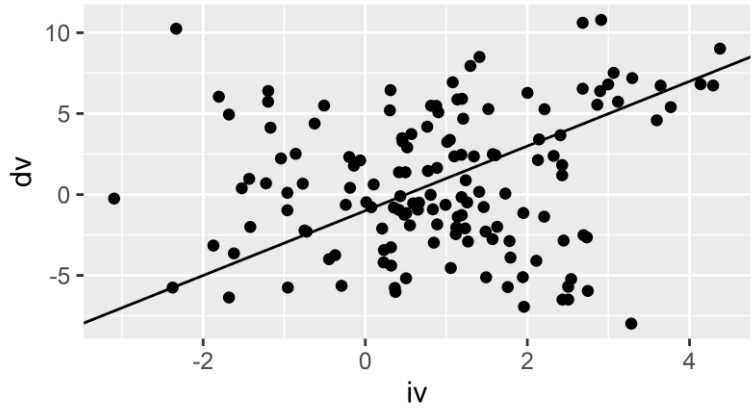


Figure 8: Abline and regression lines

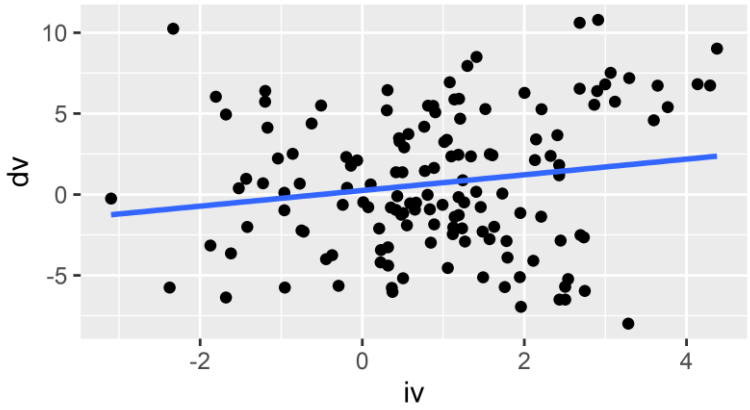


Figure 9: Abline and regression lines

The parameter of `geom_smooth`:

- `method`: the model to use to fit the data (here, OLS model)
- `formula`: the formula used to fit the model (here, y on x, but more complex formula could be written, as we will see)
- `se`: a logical whether standard errors should be plotted or not.
-

8.5 Bars and histograms

For continuous variables, you can plot a histogram with `geom_histogram()`.

```
ggplot(df) +
  aes(x = iv) +
  geom_histogram(bins = 15) +
  labs(x = "a continuous variable")
```

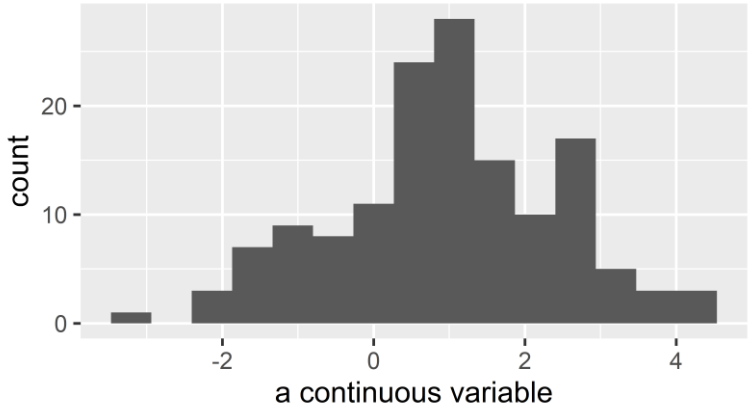


Figure 10: Drawing histograms

`bar_plot()` is the same but for discrete variables. By default, a bar plot uses frequencies for its values, but you can use values from a column by specifying `stat = "identity"` inside `geom_bar()`.

```
# Bar plot with frequency
ggplot(df) +
  aes(x = group) +
  geom_bar() +
  labs(x = "a discrete variable")

tmp <- as.data.frame(table(df$group))
ggplot(tmp) +
  aes(x = Var1, y = Freq) +
  geom_bar(stat = "identity")
```

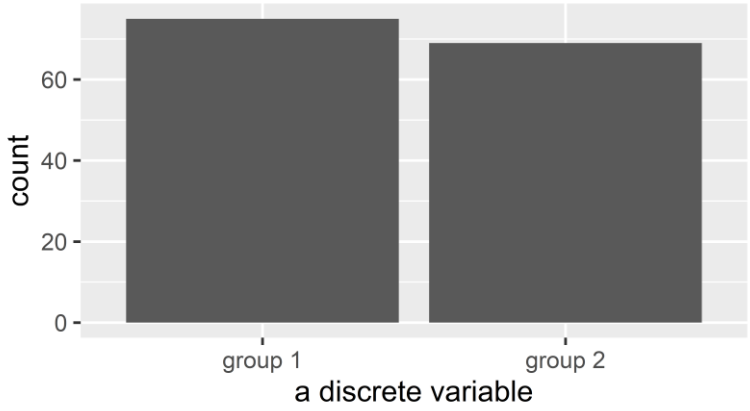


Figure 11: Drawing barplots

8.6 Facet wrap

I placed the two figures above side by side, but they are two different figures. Sometimes, you may want to show a grid of figures, for example, n scatterplots of the same variables, per group (instead of using colors or shapes). You can do this with `facet_wrap`. For example, reusing `graph_1` from above:

```
graph_1 + facet_wrap(~group, ncol = 2)
```

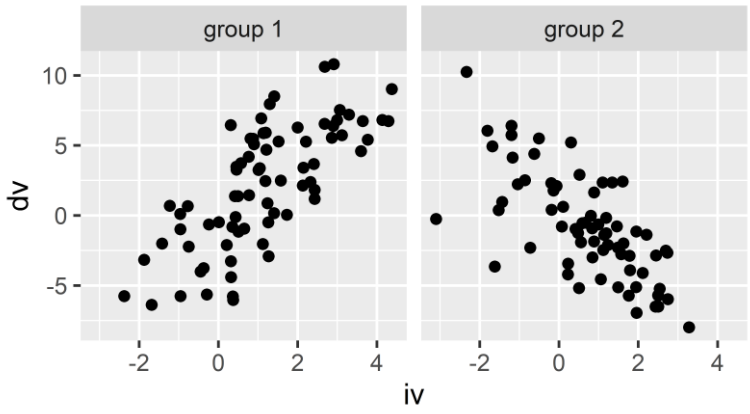


Figure 15: Facet wrap

8.7 Adding themes

Another option to affect the appearance of the graph is to use **themes**, which affect a number of general aspects concerning how graphs are displayed. Dozens of themes are available for ggplot(). For example:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() + theme_classic()
```

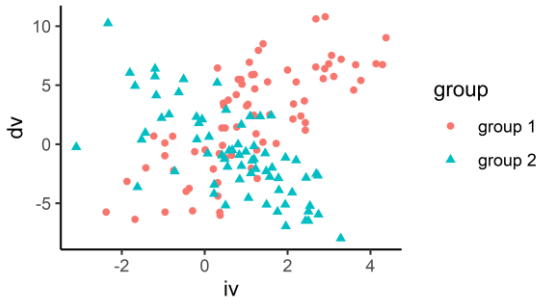


Figure 16: Changing theme

- To know more about ggplot2, read the book in the syllabus about `ggplot2`: https://ggplot2-book.org/.

9. Descriptive statistics

9.1 Summarize data

When exploring your data, or writing your result section in your thesis, you will need descriptive statistics. R includes all the required functions for usual descriptive statistics: `mean()`, `median()`, `sd()`, `min()`, `max()`, `cor()`, `quantile()`, `IQR()` , *etc.*

To get the mean of dv, a continuous variable, simply enter:

```
# Calculating the mean
mean(df$dv)
```

```
[1] 0.7022274
```

- You can also save the multiple descriptive statistics in a dataframe. Example:

```
summary_stat <- data.frame(avg_dv    = mean(df$dv),
                           median_dv = median(df$dv),
                           min_dv    = min(df$dv),
                           max_dv    = max(df$dv))
```

- or you may want to calculate the mean or other measures per group:

```
aggregate(df$dv, list(df$group), FUN = mean)
```

```
  Group.1          x
1 group 1  2.1669233
2 group 2 -0.8898333
```

Warning: If your data contains missing values, the function will send `NA`. To avoid it, use the `na.rm` argument:

```
test_var <- c(2, 5, 5, 4, 8, 9, NA, 12)

# No na.rm
mean(test_var)
```

```
[1] NA
```

```
# With na.rm
mean(test_var, na.rm = TRUE)
```

```
[1] 6.428571
```

9.2 Summary statistics in a nice table

Often, in your report, you simply want to insert a table with the mean and standard deviation on all the variables in your model. You can do this with the command `st()` from the package `vtable`:

```
vtable::st(df, out = "kable")
```

Table 1: Using vtable without options

Variable	N	Mean	Std. Dev.	Min	Pctl. 25	Pctl. 75	Max
dv	144	0.7	4.3	-8	-2.5	4.2	11
iv	144	0.94	1.5	-3.1	0.18	2	4.4
group	144						
... group 1	75	52%					
... group 2	69	48%					
discrete_var	144	5	2	0	4	6	10
iv_2	144	5.5	9.8	-20	-1.1	12	28
iv_3	144	131	116	0.47	48	186	528
gender	140						
... female	55	39%					
... male	76	54%					
... other	5	4%					
... prefer not to say	4	3%					

- This function is very handy. You can use it to generate a table compatible with word, that you can use directly in your thesis.
- `out = "kable"` is here to create a table that you can export in a MS Word document. You can also export tables in html format (for a webbrowser), csv, or in latex.
- By default, it returns the descriptive statistics of all your variables in your dataframe.
  - If you want to select return, for example, only `group` and `dv` variables in your table, use: `vars = c("iv", "dv")` .

- You can assign the labels of each columns with `labels =` followed by a vector of labels. **Warning:** labels MUST be ordered in the same order as the `vars` parameter.
- summ and summ.names to choose what statistics to display and how to name the columns. For example, to display N, mean, standard deviation, min, and max, you would enter the parameter: `summ = c("notNA(x)", "mean(x)", "sd(x)", "min(x)", "max(x)")`. The help page provides more information on what statistics can be displayed.
- `digits =` lets you choose how many decimals to display

```
# Example:
vtable::st(
  # The dataset to describe
  df,

  # Vector of variables to display
  vars = c("iv", "dv"),

  # How to name the variables in the output table
  labels = c("My independent var.", "My dependent var."),

  # Statistics to display
  summ = c("notNA(x)", "mean(x)", "sd(x)", "min(x)", "max(x)"),

  # How to name the columns in the output table
  summ.names = c("N", "Mean", "S.D.", "Min", "Max"),

  # Number of decimals to display
  digits = 3,

  # Table to export for MS Word
  out = "kable"
)
```

Table 2: Summary Statistics

Variable	N	Mean	S.D.	Min	Max
My independent var.	144	0.938	1.47	-3.1	4.38
My dependent var.	144	0.702	4.33	-7.98	10.8

9.3 Correlation matrices

You can return a correlation between two variables with `cor(x, y)`.

Unfortunately, correlation matrices, which you need in most quantitative works, have always been a pain to display easily in R. The following blog posts give you relatively easy solution to display a correlation matrix with an acceptable design for an academic work:

- Correlation matrices according to sthda.
- Correlation matrices on datadream.org.

The simplest option to get a correlation matrix is with the `corrr` package from the tidyverse.

```
# Create a correlation table
x <- corrr::correlate(df)
```

```
Non-numeric variables removed from input: `group`, and `gender`
Correlation computed with
• Method: 'pearson'
• Missing treated using: 'pairwise.complete.obs'
```

```
# Save only the lower triangle
x <- corrr::shave(x)

# Format "nicely", and get ready for MS Word
knitr::kable(corrr::fashion(x))
```

term	dv	iv	discrete_var	iv_2	iv_3
dv					
iv	.16				
discrete_var	.03	-.07			
iv_2	-.04	-.18	.06		
iv_3	-.06	-.20	.05	.94	

That way, you get a correlation matrix without the significance level. You can still get them through the procedures of the blog posts above, but that will be longer code.

10. Getting nice output documents with Quarto

10.1 Reproducible R Reports

Quarto is a module of R that allows you mixing R code and text. With it, you can also generate Word documents containing your results and interpretations. You can find a detailed documentation here: https://quarto.org/docs/guide/.

All the seminars in this course are written with Quarto. The quarto visual editors contains 3 different types of blocks

1. On top, (in the red square on the image), you can see the **YAML metadata**, where you set the properties of your document: title, author, type of output, etc. You may also set multiple types of output here.
2. The part in the green square, below, are just simple text, that you can edit with the buttons on top (Bold, Italic, code, choose a style, add hyperlink, insert images or tables, footnotes, references, etc.) just as a normal text editor.
3. Then, the part in the blue square, starting with `{r}` is a **code chunk** (*i.e.* a portion of R code). *A code chunk* is a portion of R code that will can be interpreted by Quarto, and the output displayed below it, in the output document. For example, the code chunk in blue in the image above contains the code `1+1` (in gray) and you can see bellow the output `[1] 2`. You generate the output of the chunk by clicking on the little green play button (top right corner of the code chunk), or when you `Render` the output document. Options allow you to choose which chunk have to be interpreted, or not, when rendering a document.

On top of the document, if you press `Render`, a Word document will be generated in your project directory.

The content of the .qmd file is in fact a combination or R and Markdown code (the markup language defining what is a header, what is a text body, where are the bullet lists, etc.) that is interpreted in the visual editor. If you prefer to work on the `.qmd` source, you can click on the `source` button (on the top left).

10.2 YAML metadata

The YAML metadata define the general properties of your document, including the appearance and the output format. Bellow, the figure shows the options I used to generate the module 2 course document. The blue parts (in quotes) are the values, the brown parts are possible options. You may want to adjust the title, subtitle or author. You can find more info on the other properties in the Quarto documentation.

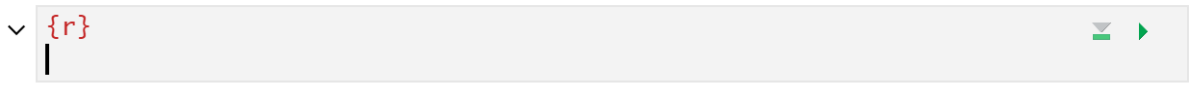
```yaml
---
title: "7316 - Introduction to data analysis with R"
subtitle: "MODULE 2: Playing with data!"
author: "Mickaël Buffart ([mickael.buffart@hhs.se](mailto:mickael
.buffart@hhs.se))"
format:
  docx:
    reference-doc: "./assets/sse-quarto-template.docx"
toc: true
toc-depth: 3
toc-title: "Table of contents"
number-sections: true
number-depth: 5
---
```

Some YAML options in quarto

10.3 Code chunk

Now, if you want to write R code and display the output in your document, you need to insert a new code chunk by clicking on `Insert` > `Code chunk` > `R`. An empty chunk will then appear, as shown bellow:

insert a new code chunk by clicking on `Insert` > `Code chunk` > `R`. An empty chunk will then appear, as shown bellow:



Example of empty code chunk

In the code chunk, you can type and run code, just as in an R script.

10.3.1 Code chunk options

There are several output options you can specify for how R code and the code output are expressed in reports. These options are expressed bellow the red `{r}` declaration at the top of the chunk, on new lines (one line per option).

- `#| echo: false`: the R code will not be displayed in the output document (but potentially the output of the code execution depending on other chunk options).
- `#| eval: false`: the code inside the chunk will not be executed (*i.e.* there won't be an output), but the code is displayed in the output document.
- `#| results: "hide"`: the code will be executed, but the output of the execution will not be displayed in the output document.
- `#| warning: false` / `#| message: false`: do not display warnings or messages associated with the R code.
- `#| fig-width:`, `#| fig-height:`, `#| fig-align: "center"` will be useful options in case the code chunk is generating a figure (with `ggplot2`, for example). `#| fig-cap:` is useful to define the figure title, and `#| dpi:` determines the resolution of the figure (a higher value means a better resolution, but also a more heavy file).

You will find plenty of other useful options in the quarto documentation online: https://quarto.org/docs/guide/

11. Modeling with formulæ

In R, to estimate a model, you need formulæ. A formula is an R object that simply states the equation representing the relationship you are trying to model. As a simple example, let us assume you would like to estimate an OLS model described by the following equation:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

In R, you would write **the formul**a as: `Y ~ X1 + X2`. You can write formulæ for all kinds of mathematical models. Because the formula is an R object, just as any object, you can save it in your environment, change its type on the fly, manipulate it, etc. For example, assign the formula with `myf <- Y ~ X1 + X2`.

11.1 Formula syntax:

- The left-hand side and the right-hand side are separated with `~`. Example: `Y ~ X`
- In a normal linear model, the independent variables are separated with `+`.
  Example: `Y ~ X1 + X2` .
- You can summarize all the variables in a dataframe with a `.`. For example, `Y ~ .` is the formula to regress `Y` on any of the other variables in your dataframe of interest.
- For the rest, I provide bellow some example of formulæ and the equivalent equation:

Example of formulæ

Formula	Equation	Note
`Y ~ X1 + X2`	$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \epsilon$	Linear model
`Y ~ X1 + X2 - 1`	$Y = \beta_1 X_1 + \beta_2 X_2 + \epsilon$	`-` is used to remove something from the equation. `1` stands for the intercept. It is also possible to remove a variable, as below. You could as well force including the intercept in the equation by adding `+ 0`.
`Y ~ X1*X2`	$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon$	Interacting effect and main effects (we will use it in the 4th lecture)
`Y ~ X1:X2`	$Y = \alpha + \beta_1 X_1 X_2 + \epsilon$	Interacting effect without the main effects

12. OLS regression in R

Being able to reproduce equations is not enough. To obtain the output above, you also need to estimate models. For example, to fit an OLS model, you can just call the `lm()` function. Once you have fitted the model, you can visualize it with the `summary()` function, or store it in an object, or use it for prediction, extracting residuals, etc.

To fit the model 1 above, using OLS regressions, write:

```
# Fitting
fit_1 <- lm(dv ~ iv + iv_2, data = df)
```

- Note that we assigned the fitted model (generated with `lm()`) into `fit_1`. You can then save it in an `.Rds` file, display the results, extract residuals, etc.
- To display the fitted model, use:

```
summary(fit_1)
```

- The ouptut will be:

```
Call:
lm(formula = dv ~ iv + iv_2, data = df)

Residuals:
    Min      1Q  Median      3Q     Max
-9.8095 -3.1639 -0.1336  3.4521 11.1923

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.280459   0.491721   0.570   0.5693
iv           0.478603   0.248281   1.928   0.0559 .
iv_2        -0.004964   0.037463  -0.133   0.8948
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.302 on 141 degrees of freedom
Multiple R-squared:  0.02726,   Adjusted R-squared:  0.01346
F-statistic: 1.976 on 2 and 141 DF,  p-value: 0.1425
```

- Note that some measures of diagnostics, such as the R2, the F-statistic, or the distribution of residuals, are also available in the output of `summary()`.
- You could also extract residuals, fitted values, etc.. Bellow, I assign them to new variables into the dataframe.

```
fit_1$coefficients[1]
```

```
(Intercept)
  0.2804595
```

```
# Extract the fitted values
df$fitted <- fit_1$fitted.values

# Extract the residuals of the models
df$resid <- fit_1$residuals
```

12.1 Diagnostic checking for OLS regression

Saving the residuals, the fitted values, or other parameters, may be useful for diagnostic checking of your models.

12.1.1 Plotting diagnostics

- Using what we have see last time, we can plot the actual and the fitted values:

```
library(ggplot2)

ggplot(df) + aes(y = dv, x = iv) +
  geom_point() +
  geom_segment(aes(xend = iv, yend = fitted), size = 0.2) +
  ggthemes::theme_clean()
```

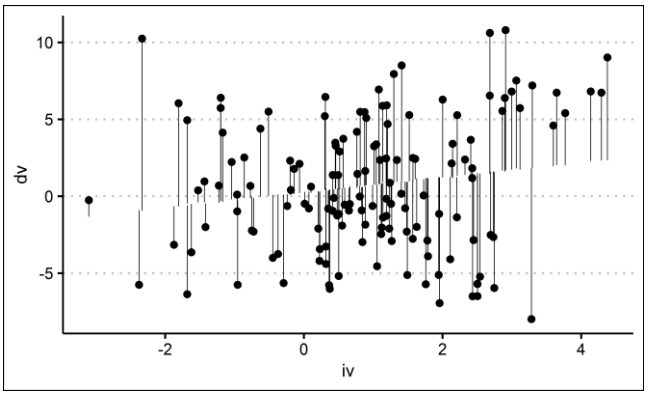


Figure 17: Ploting diagnostic

- The `lm()` function contains also a method for creating diagnostic visuals in base R, that can be displayed as ggplot using `ggfortify`. `ggfortify` is a package to convert automatic plots into plots that are compatible with `ggplot2` functions.
  - The visual diagnostic are useful to assess linearity (*residuals vs fitted*), normality of the residuals (*Normal Q-Q*), homogeneity of variance (*scale-location*), or outliers (*residuals vs leverage*).

```
library(ggfortify)
autoplot(fit_1) + ggthemes::theme_clean()
```

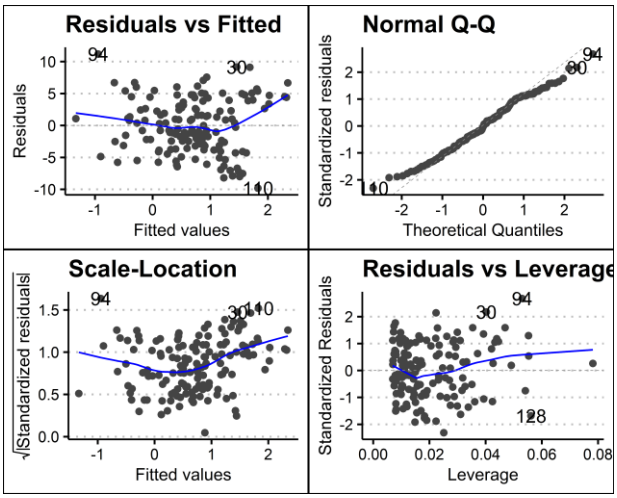


Figure 18: Autoplot diagnostics

12.1.2 Some other measures of diagnostics

Other diagnostics measures can easily be produced in R, either with base R, or with well known packages: cook's distance (`cooks.distance()`), variance inflation factors (`car::vif()`), normality of the residuals, and others:

- Cook's distance:

```
# Extract cooks distance to detect outliers
df$cooks <- cooks.distance(fit_1)
```

- VIFs:

```
# Display variance inflation factors (VIF) to assess multicolinearity issues
car::vif(fit_1)
```

```
      iv     iv_2
1.033011 1.033011
```

- Skewness

```
# Skewness
moments::skewness(df$resid)
```

```
[1] 0.04298591
```

- Kurtosis

```
# Kurtosis
moments::kurtosis(df$resid)
```

```
[1] 2.351579
```

- Jarque-Bera Normality Test

```
# Jarque-Bera Normality Test
moments::jarque.test(df$resid)
```

```
    Jarque-Bera Normality Test
data:  df$resid
JB = 2.567, p-value = 0.2771
alternative hypothesis: greater
```