Practical session materials | granolarr

Stefano De Sabbata

2020-10-20

Contents

Preface

Stefano De Sabbata

This book contains the *practical sessions* component of granolarr, a repository of reproducible materials to teach geographic information and data science in R. Part of the materials are derived from the practical sessions for the module GY7702 Practical Programming in R of the MSc in Geographic Information Science at the School of Geography, Geology, and the Environment of the University of Leicester, by Dr Stefano De Sabbata.

This book was created using R, RStudio, RMarkdown, Bookdown, and GitHub.

Session info

```
sessionInfo()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04 LTS
##
## Matrix products: default
## BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/openblas-openmp/libopenblasp-r0.3.8.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=C
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
```

```
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_4.0.2  magrittr_1.5    bookdown_0.20   htmltools_0.5.0
##  [5] tools_4.0.2     yaml_2.2.1      stringi_1.4.6   rmarkdown_2.3
##  [9] knitr_1.29      stringr_1.4.0   digest_0.6.25   xfun_0.16
## [13] rlang_0.4.7     evaluate_0.14
```

Chapter 1

Introduction to R

Stefano De Sabbata

1.1 The R programming language

As mentioned in Lecture 1, **R** was created in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand. R is a free, open-source implementation of the **S** statistical programming language initially created at the Bell Labs. At its core, R is a functional programming language (its main functionalities revolve around defining and executing functions). However it now supports, and it is commonly used as an imperative (focused on instructions on variables and programming control structures) and object-oriented (involving complex object structures) programming language.

In simple terms, nowadays, programming in R mostly focuses on devising a series of instructions to execute a task – most commonly, loading and analysing a dataset.

As such, R can be used to program by creating sequences of **instructions** involving **variables** – which are named entities that can store values. That will be the main topic of this practical session. Instructions can include control flow structures, such as decision points (*if/else*) and loops, which will be the topic of the next practical session. Instructions can also be grouped in **functions**, which we will also see in the next practical session.

R is **interpreted**, not compiled. Which means that an R interpreter (if you are using R Studio, the R interpreter is simply hidden in the backend and R Studio is the frontend that allows you to interact with the interpreter) receives an instruction you write in R, interprets and executes them. Other programming

languages require their code to be compiled in an executable to be executed on a computer.

1.1.1 Using RStudio

As you open RStudio or RStudio Server, the interface is divided into two main sections. On the left side, you find the *Console* – as well as the R script editor, when a script is being edited. The *Console* in an input/output window into the R interpreter, where instructions can be typed, and the computed output is shown.

For instance, if you type in the *Console*

```
1 + 1
```

the R interpreter understands that as an instruction to sum one to one, and produces the result (as the materials for this module are created in RMarkdown, the output of the computation is always preceded by '##').

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

Note how the output value `2` is preceded by `[1]`, which indicates that the output is constituted by only one element. If the output is constituted by more than one element, as the list of numbers below, each row of the output is preceded by the index of the first element of the output.

```
##  [1]   1   4   9  16  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361
## [20] 400
```

On the right side, you find two groups of panels. On the top-right, the main element is the *Environment* panel, which is a representation of the current state of the interpreter's memory, and as such, it shows all the stored variables, datasets, and functions. On the bottom-right, you find the *Files* panel, which shows file system (file and folders on your computer or the server), as well as the *Help* panel, which shows you the help pages when required. We will discuss the other panels later on in the practical sessions.

1.2 Interpreting values

When a value is typed in the *Console*, the interpreter simply returns the same value. In the examples below, `2` is a simple numeric value, while `"String value"` is a textual value, which in R is referred to as a *character* value and in programming is also commonly referred to as a *string* (short for *a string of characters*).

Numeric example:

```
2
```

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

Character example:

```
"String value"
```

```
## [1] "String value"
```

Note how character values need to start and end with a single or double quote (' or "), which are not part of the information themselves. The Tidyverse Style Guide suggests always to use the double quote ("), so we will use those in this module.

Anything that follows a # symbol is considered a *comment* and the interpreter ignores it.

```
# hi, I am a comment, please ignore me
```

As mentioned above, the interpreter understands simple operations on numeric values.

```
1 + 1
```

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

There are also a large number of pre-defined functions, e.g., square-root: `sqrt`.

```
sqrt(2)
```

```
## [1] 1.414214
```

Functions are collected and stored in *libraries* (sometimes referred to as *packages*), which contains related functions. Libraries can range anywhere from the `base` library, which includes the `sqrt` function above, to the `rgdal` library, which contains implementations of the GDAL (Geospatial Data Abstraction Library) functionalities for R.

1.3 Variables

A variable can be defined using an **identifier** (e.g., `a_variable`) on the left of an **assignment operator** `<-`, followed by the object to be linked to the identifier, such as a **value** (e.g., `1`) to be assigned on the right. The value of the variable can be tested/invoked by simply specifying the **identifier**.

```
a_variable <- 1
a_variable
```

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

If you type `a_variable <- 1` in the *Console* in RStudio, a new element appears in the *Environment* panel, representing the new variable in the memory. The left part of the entry contains the identifier `a_variable`, and the right part contains the value assigned to the variable `a_variable`, that is `1`.

It is not necessary to provide a value directly. The right part of the assignment can be a **call to a function**. In that case, the function is **executed** on the provided input and **the result is assigned to the variable**.

```
a_variable <- sqrt(4)
a_variable
```

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

Note how if you type `a_variable <- sqrt(4)` in the *Console* in RStudio, the element in the *Environment* panel changes to reflect the new value assigned to the variable `a_variable`, which is now the result of `sqrt(4)`, that is 2.

In the example below, another variable named `another_variable` is created and summed to `a_variable`, saving the result in `sum_of_two_variables`. The square root of that sum is then stored in the variable `square_root_of_sum`.

```
another_variable <- 4
another_variable
```

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

```
sum_of_two_variables <- a_variable + another_variable

square_root_of_sum <- sqrt(sum_of_two_variables)
square_root_of_sum
```

```
## [1] 2.44949
```

1.4 Basic types

1.4.1 Numeric

The *numeric* type represents numbers (both integers and reals).

```
a_number <- 1.41
is.numeric(a_number)
```

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

```
is.integer(a_number)
```

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

```
is.double(a_number) # i.e., is real
```

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

Base numeric operators.

Operator	Meaning	Example	Output
+	Plus	`5+2`	7
-	Minus	`5-2`	3
*	Product	`5*2`	10
/	Division	`5/2`	2.5
%/%	Integer division	`5%/%2`	2
%%	Module	`5%%2`	1
^	Power	`5^2`	25

Some pre-defined functions in R:

```
abs(-2) # Absolute value
```

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

```
ceiling(3.475) # Upper round
```

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

```
floor(3.475) # Lower round
```

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

```
trunc(5.99) # Truncate
```

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

```
log10(100) # Logarithm 10
```

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

```
log(exp(2)) # Natural logarithm and e
```

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

Use simple brackets to specify the order of execution. If not specified the default order is: rise to power first, then multiplication and division, sum and subtraction last.

```
a_number <- 1
(a_number + 2) * 3
```

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

```
a_number + (2 * 3)
```

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

```
a_number + 2 * 3
```

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

The object `NaN` (*Not a Number*) is returned by R when the result of an operation is not a number.

```
0 / 0
```

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

```
is.nan(0 / 0)
```

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

That is not to be confused with the object `NA` (*Not Available*), which is returned for missing data.

1.4.2 Logical

The *logical* type encodes two truth values: True and False.

```
logical_var <- TRUE
is.logical(logical_var)
```

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

```
isTRUE(logical_var)
```

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

```
as.logical(0) # TRUE if not zero
```

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

Basic logic operators

Operator	Meaning	Example	Output
==	Equal	`5==2`	FALSE
!=	Not equal	`5!=2`	TRUE
>	Greater than	`5>2`	TRUE
<	Less than	`5<2`	FALSE
>=	Greater or equal	`5>=2`	TRUE
<=	Less or equal	`5<=2`	FALSE
!	Not	`!TRUE`	FALSE
&	And	`TRUE & FALSE`	FALSE
|	Or	`TRUE | FALSE`	TRUE

1.4.3 Character

The *character* type represents text objects, including single characters and character strings (that is text objects longer than one character, commonly referred to simply as *strings* in computer science).

```
a_string <- "Hello world!"
is.character(a_string)
```

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

```
is.numeric(a_string)
```

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

```
as.character(2) # type conversion  (a.k.a. casting)
```

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

```
as.numeric("2")
```

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

```
as.numeric("Ciao")
```

```
## Warning: NAs introduced by coercion
```

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

1.5 Tidyverse

As mentioned in the lecture, libraries are collections of functions and/or datasets. Libraries can be installed in R using the function `install.packages` or using `Tool > Install Packages...` in RStudio.

The meta-library Tidyverse contains the following libraries:

- `ggplot2` is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.
- `dplyr` provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges.
- `tidyr` provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable.
- `readr` provides a fast and friendly way to read rectangular data (like csv, tsv, and fwf). It is designed to flexibly parse many types of data found in the wild, while still cleanly failing when data unexpectedly changes.
- `purrr` enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. Once you master the basic concepts, purrr allows you to replace many for loops with code that is easier to write and more expressive.
- `tibble` is a modern re-imagining of the data frame, keeping what time has proven to be effective, and throwing out what it has not. Tibbles

are data.frames that are lazy and surly: they do less and complain more forcing you to confront problems earlier, typically leading to cleaner, more expressive code.

- `stringr` provides a cohesive set of functions designed to make working with strings as easy as possible. It is built on top of stringi, which uses the ICU C library to provide fast, correct implementations of common string manipulations.
- `forcats` provides a suite of useful tools that solve common problems with factors. R uses factors to handle categorical variables, variables that have a fixed and known set of possible values.

A library can be loaded using the function `library`, as shown below (note the name of the library is not quoted). Once a library is installed on a computer, you don't need to install it again, but every script needs to load all the library that it uses. Once a library is loaded, all its functions can be used.

Important: it is always necessary to load the `tidyverse` meta-library if you want to use the `stringr` functions or the pipe operator `%>%`.

```
library(tidyverse)
```

1.5.1 stringr

The code below presents the same examples used in the lecture session to demonstrate the use of `stringr` functions.

```
str_length("Leicester")
```

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

```
str_detect("Leicester", "e")
```

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

```
str_replace_all("Leicester", "e", "x")
```

```
## [1] "Lxicxstxr"
```

1.5.2 The pipe operator

The pipe operator is useful to outline more complex operations, step by step (see also R for Data Science, Chapter 18). The pipe operator `%>%`

- takes the result from one function
- and passes it to the next function
- as the **first argument**
- that doesn't need to be included in the code anymore

The code below shows a simple example. The number `2` is taken as input for the first pipe that passes it on as the first argument to the function `sqrt`. The

output value `1.41` is then taken as input for the second pipe, that passes it on as the first argument to the function `trunc`. The final output `1` is finally returned.

```
2 %>%
  sqrt() %>%
  trunc()
```

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

The image below graphically illustrates how the pipe operator works, compared to the same procedure executed using two temporary variables that are used to store temporary values.

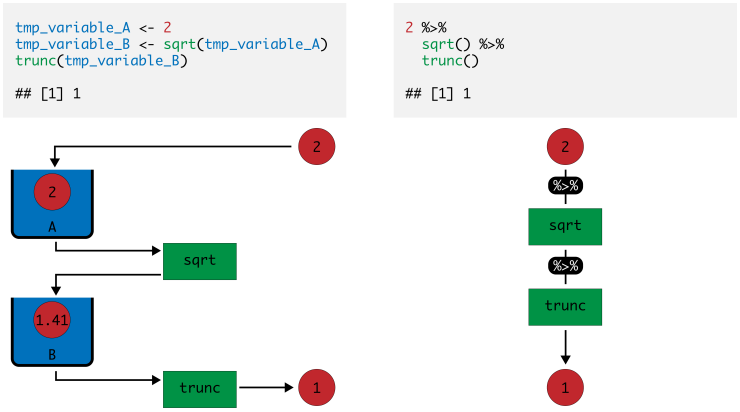


Figure 1.1: Illustration of how the pipe operator works

```
sqrt(2) %>%
 round(digits = 2)
```

The first step of a sequence of pipes can be a value, a variable, or a function including arguments. The code below shows a series of examples of different ways of achieving the same result. The examples use the function `round`, which also allows for a second argument `digits = 2`. Note that, when using the pipe operator, only the nominally second argument is provided to the function `round` – that is `round(digits = 2)`

```
# No pipe, using variables
tmp_variable_A <- 2
tmp_variable_B <- sqrt(tmp_variable_A)
round(tmp_variable_B, digits = 2)

# No pipe, using functions only
round(sqrt(2), digits = 2)

# Pipe starting from a value
```

```
2 %>%
  sqrt() %>%
  round(digits = 2)

# Pipe starting from a variable
the_value_two <- 2
the_value_two %>%
  sqrt() %>%
  round(digits = 2)

# Pipe starting from a function
sqrt(2) %>%
 round(digits = 2)
```

A complex operation created through the use of `%>%` can be used on the right side of `<-`, to assign the outcome of the operation to a variable.

```
sqrt_of_two <- 2 %>%
  sqrt() %>%
  round(digits = 2)
```

1.6 Coding style

Study the Tidyverse Style Guide (style.tidyverse.org) and use it consistently!

1.7 Exercise 104.1

Question 104.1.1: Write a piece of code using the pipe operator that takes as input the number `1632`, calculates the logarithm to the base 10, takes the highest integer number lower than the calculated value (lower round), and verifies whether it is an integer.

Question 104.1.2: Write a piece of code using the pipe operator that takes as input the number `1632`, calculates the square root, takes the lowest integer number higher than the calculated value (higher round), and verifies whether it is an integer.

Question 104.1.3: Write a piece of code using the pipe operator that takes as input the string `"1632"`, transforms it into a number, and checks whether the result is *Not a Number*.

Question 104.1.4: Write a piece of code using the pipe operator that takes as input the string `"-16.32"`, transforms it into a number, takes the absolute value and truncates it, and finally checks whether the result is *Not Available*.

1.8 Exercise 104.2

Answer the question below, consulting the `stringr` library reference (stringr.tidyverse.org/reference) as necessary

Question 104.2.1: Write a piece of code using the pipe operator and the `stringr` library that takes as input the string `"I like programming in R"`, and transforms it all in uppercase.

Question 104.2.2: Write a piece of code using the pipe operator and the `stringr` library that takes as input the string `"I like programming in R"`, and truncates it, leaving only 10 characters.

Question 104.2.3: Write a piece of code using the pipe operator and the `stringr` library that takes as input the string `"I like programming in R"`, and truncates it, leaving only 10 characters and using no ellipsis.

Question 104.2.4: Write a piece of code using the pipe operator and the `stringr` library that takes as input the string `"I like programming in R"`, and manipulates to leave only the string `"I like R"`.

Chapter 2

R programming

Stefano De Sabbata

2.1 R Scripts

The RStudio Console is handy to interact with the R interpreter and obtain results of operations and commands. However, moving from simple instructions to an actual program or scripts to conduct data analysis, the Console is usually not sufficient anymore. In fact, the Console is not a very comfortable way of providing long and complex instructions to the interpreter and editing past instructions when you want to change something. A better option to create programs or data analysis script of any significant size is to use the RStudio integrated editor to create an *R script*.

To create an R script, select from the top menu *File* > *New File* > *R Script*. That opens the embedded RStudio editor and a new empty R script folder. Copy the two lines below into the file. The first loads the `tidyverse` library, whereas the second simply calculates the square root of two.

```
# Load the Tidyverse
library(tidyverse)

# Calculate the square root of two
2 %>% sqrt()
```

```
## [1] 1.414214
```

From the top menu, select *File* > *Save*, type in *My_first_script.R* (make sure to include the underscore and the *.R* extension) as *File name*, and click *Save*. That is your first R script, congratulations!

New lines of code can be added to the file, and the whole script can then be executed. Edit the file by adding the line of code shown below, and save it. Then click the *Source* button on the top-right of the editor to execute the file. What happens the first time? What happens if you click *Source* again?

```
# First variable in a script
a_variable <- "This is my first script"
```

Alternatively, you can click on a specific line or select one or more lines, and click *Run* to execute only the selected line(s).

Delete the two lines calculating the square root of two and defining the variable `a_variable` from the script, leaving only the line loading the Tidyverse library. In the following sections, add the code to the script to execute it, rather than using the Console.

2.2 Vectors

Vectors can be defined in R by using the function `c`, which takes as parameters the items to be stored in the vector – stored in the order in which they are provided.

```
east_midlands_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")
length(east_midlands_cities)
```

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

Once the vector has been created and assigned to an identifier, elements within the vector can be retrieved by specifying the identifier, followed by square brackets, and the *index* (or indices as we will see further below) of the elements to be retrieved – remember that indices start from 1.

```
# Retrieve the third city
east_midlands_cities[3]
```

```
## [1] "Lincoln"
```

To retrieve any subset of a vector (i.e., not just one element), specify an integer vector containing the indices of interest (rather than a single integer value) between square brackets.

```
# Retrieve first and third city
east_midlands_cities[c(1, 3)]
```

```
## [1] "Derby"   "Lincoln"
```

The operator `:` can be used to create integer vectors, starting from the number specified before the operator to the number specified after the operator.

```
# Create a vector containing integers between 2 and 4
two_to_four <- 2:4
two_to_four
```

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

```
# Retrieve cities between the second and the fourth
east_midlands_cities[two_to_four]
```

```
## [1] "Leicester"  "Lincoln"    "Nottingham"
```

```
# As the second element of two_to_four is 3...
two_to_four[2]
```

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

```
# the following command will retrieve the third city
east_midlands_cities[two_to_four[2]]
```

```
## [1] "Lincoln"
```

```
# Create a vector with cities from the previous vector
selected_cities <- c(east_midlands_cities[1], east_midlands_cities[3:4])
```

The functions `seq` and `rep` can also be used to create vectors, as illustrated below.

```
seq(1, 10, by = 0.5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
## [16]  8.5  9.0  9.5 10.0
```

```
seq(1, 10, length.out = 6)
```

```
## [1]  1.0  2.8  4.6  6.4  8.2 10.0
```

```
rep("Ciao", 4)
```

```
## [1] "Ciao" "Ciao" "Ciao" "Ciao"
```

The logical operators `any` and `all` can be used to test conditional statements on the vector. The former returns `TRUE` if at least one element satisfies the statement, the second returns `TRUE` if all elements satisfy the condition

```
any(east_midlands_cities == "Leicester")
```

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

```
my_sequence <- seq(1, 10, length.out = 7)
my_sequence
```

```
## [1]  1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

```
any(my_sequence > 5)
```

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

```
all(my_sequence > 5)
```

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

All built-in numerical functions in R can be used on a vector variable directly. That is, if a vector is specified as input, the selected function is applied to each element of the vector.

```
one_to_ten <- 1:10
one_to_ten
```

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

```
one_to_ten + 1
```

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

```
sqrt(one_to_ten)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

2.3 Filtering

As seen in the first practical session, a conditional statement entered in the Console is evaluated for the provided input, and a logical value (`TRUE` or `FALSE`) is provided as output. Similarly, if the provided input is a vector, the conditional statement is evaluated for each element of the vector, and a vector of logical values is returned – which contains the respective results of the conditional statements for each element.

```
minus_three <- -3
minus_three > 0
```

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

```
minus_three_to_three <- -3:3
minus_three_to_three
```

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

```
minus_three_to_three > 0
```

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

A subset of the elements of a vector can also be selected by providing a vector of logical values between brackets after the identifier. A new vector returned,

containing only the values for which a `TRUE` value has been specified correspondingly.

```
minus_two_to_two <- -2:2
minus_two_to_two
```

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

```
minus_two_to_two[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
```

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

As the result of evaluating the conditional statement on a vector is a vector of logical values, this can be used to filter vectors based on conditional statements. If a conditional statement is provided between square brackets (after the vector identifier, instead of an index), a new vector is returned, which contains only the elements for which the conditional statement is true.

```
minus_two_to_two > 0
```

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

```
minus_two_to_two[minus_two_to_two > 0]
```

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

2.4 Conditional statements

Conditional statements are fundamental in (procedural) programming, as they allow to execute or not execute part of a procedure depending on whether a certain condition is true. The condition is tested and the part of the procedure to execute in the case the condition is true is included in a *code block*.

```
temperature <- 25

if (temperature > 25) {
  cat("It really warm today!")
}
```

A simple conditional statement can be created using `if` as in the example above. A more complex structure can be created using both `if` and `else`, to provide not only a procedure to execute in case the condition is true, but also an alternative procedure, to be executed when the condition is false.

```
temperature <- 12

if (temperature > 25) {
  cat("It really warm today!")
} else {
```

```
  cat("Today is not warm")
}
```

```
## Today is not warm
```

Finally, conditional statements can be **nested**. That is, a conditional statement can be included as part of the code block to be executed after the condition is tested. For instance, in the example below, a second conditional statement is included in the code block to be executed in the case the condition is false.

```
temperature <- -5

if (temperature > 25) {
  cat("It really warm today!")
} else {
  if (temperature > 0) {
    cat("There is a nice temperature today")
  } else {
    cat("This is really cold!")
  }
}
```

```
## This is really cold!
```

Similarly, the first example seen in the lecture should be coded as follows.

```
a_value <- -7

if (a_value == 0) {
  cat("Zero")
} else {
  if (a_value < 0) {
    cat("Negative")
  } else {
    cat("Positive")
  }
}
```

```
## Negative
```

2.5 Loops

Loops are another core component of (procedural) programming and implement the idea of solving a problem or executing a task by performing the same set of steps a number of times. There are two main kinds of loops in R - **deterministic** and **conditional** loops. The former is executed a fixed number of times, specified at the beginning of the loop. The latter is executed until a specific condition is met. Both deterministic and conditional loops are extremely

important in working with vectors.

2.5.1 Conditional Loops

In R, conditional loops can be implemented using `while` and `repeat`. The difference between the two is mostly syntactical: the first tests the condition first and then execute the related code block if the condition is true; the second executes the code block until a `break` command is given (usually through a conditional statement).

```
a_value <- 0
# Keep printing as long as x is smaller than 2
while (a_value < 2) {
  cat(a_value, "\n")
  a_value <- a_value + 1
}
```

```
## 0
## 1
```

```
a_value <- 0
# Keep printing, if x is greater or equal than 2 than stop
repeat {
  cat(a_value, "\n")
  a_value <- a_value + 1
  if (a_value >= 2) break
}
```

```
## 0
## 1
```

2.5.2 Deterministic Loops

The deterministic loop executes the subsequent code block iterating through the elements of a provided vector. During each iteration (i.e., execution of the code block), the current element of the vector (in the definition below) is assigned to the variable in the statement (in the definition below), and it can be used in the code block.

```
for (<VAR> in <VECTOR>) {
    ... code in loop ...
    }
```

It is, for instance, possible to iterate over a vector and print each of its elements.

```
east_midlands_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")
for (city in east_midlands_cities){
  cat(city, "\n")
}
```

```
## Derby
## Leicester
## Lincoln
## Nottingham
```

It is common practice to create a vector of integers on the spot (e.g., using the : operator) to execute a certain sequence of steps a pre-defined number of times.

```
for (iterator in 1:3) {
  cat("Exectuion number", iterator, ":\n")
  cat("    Step1: Hi!\n")
  cat("    Step2: How is it going?\n")
}
```

```
## Exectuion number 1 :
##     Step1: Hi!
##     Step2: How is it going?
## Exectuion number 2 :
##     Step1: Hi!
##     Step2: How is it going?
## Exectuion number 3 :
##     Step1: Hi!
##     Step2: How is it going?
```

2.6 Exercise 114.1

Question 114.1.1: Use the modulo operator `%%` to create a conditional statement that prints `"Even"` if a number is even and `"Odd"` if a number is odd.

Question 114.1.2: Encapsulate the conditional statement written for *Question 114.1.1* into a `for` loop that executes the conditional statement for all numbers from 1 to 10.

Question 114.1.3: Encapsulate the conditional statement written for *Question 114.1.1* into a `for` loop that prints the name of cities in odd positions (i.e., first, third, fifth) in the vector `c("Birmingham", "Derby", "Leicester", "Lincoln", "Nottingham", "Wolverhampton")`.

Question 114.1.4: Write the code necessary to print the name of the cities in the vector `c("Birmingham", "Derby", "Leicester", "Lincoln", "Nottingham", "Wolverhampton")` as many times as their position in the vector (i.e., once for the first city, two times for the second, and so on and so forth).

2.7 Function definition

Recall from the lecture that *an* **algorithm** *or effective procedure is a mechanical rule, or automatic method, or programme for performing some mathematical operation* (Cutland, 1980). A **program** is a specific set of instructions that implement an abstract algorithm. The definition of an algorithm (and thus a program) can consist of one or more **function**s, which are sets of instructions that perform a task, possibly using an input, possibly returning an output value.

The code below is a simple function with one parameter. The function simply calculates the square root of a number. Add the code below to your script and run that portion of the script (or type the code into the Console).

```
cube_root <- function (input_value) {
   result <- input_value ^ (1 / 3)
   result
}
```

Once the definition of a function has been executed, the function becomes part of the environment, and it should be visible in the Environment panel, in a subsection titled *Functions*. Thereafter, the function can be called from the Console, from other portions of the script, as well as from other scripts.

If you type the instruction below in the *Console*, or add it to the script and run it, the function is called using `27` as an argument, thus returning `3`.

```
cube_root(27)
```

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

2.7.1 Functions and control structures

One issue when writing functions is making sure that the data that has been given to the data is the right kind. For example, what happens when you try to compute the cube root of a negative number?

```
cube_root(-343)
```

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

That probably wasn't the answer you wanted. As you might remember `NaN` (*Not a Number*) is the value return when a mathematical expression is numerically indeterminate. In this case, this is actually due to a shortcoming with the `^` operator in R, which only works for positive base values. In fact -7 is a perfectly valid cube root of -343, since (-7)x(-7)x(-7) = -343.

To work around this limitation, we can state a conditional rule:

- If `x < 0`: calculate the cube root of x 'normally'.
- Otherwise: work out the cube root of the positive number, then change it to negative.

Those kinds of situations can be dealt with in an R function by using an `if` statement, as shown below. Note how the operator `-` (i.e., the symbol minus) is here used to obtain the inverse of a number, in the same way as `-1` is the inverse of the number `1`.

```
cube_root <- function (input_value) {
    if (input_value >= 0){
        result <- input_value^(1 / 3)
    }else{
        result <- -( (-input_value)^(1/3) )
    }
    result
}

cube_root(343)
cube_root(-343)
```

However, other things can go wrong. For example, `cube_root("Leicester")` would cause an error to occur, `Error in x^(1 / 3) : non-numeric argument to binary operator`. That shouldn't be surprising because cube roots only make sense for numbers, not character variables. Thus, it might be helpful if the cube root function could spot this and print a warning explaining the problem, rather than just crashing with a fairly cryptic error message such as the one above, as it does at the moment.

The function could be re-written to making use of `is.numeric` in a second conditional statement. If the input value is not numeric, the function returns the value `NA` (*Not Available*) instead of a number. Note that here there is an `if` statement inside another `if` statement, as it is always possible to nest code blocks – and `if` within a `for` within a `while` within an `if` within … etc.

```
cube_root <- function (input_value) {
    if (is.numeric(input_value)) {
        if (input_value >= 0){
            result <- input_value^(1/3)
        }else{
            result <- -(-input_value)^(1/3)
        }
        result
    }else{
        cat("WARNING: Input variable must be numeric\n")
        NA
    }
}
```

Finally, `cat` is a printing function, that instructs R to display the provided argument (in this case, the phrase within quotes) as output in the Console. The `\n` in `cat` tells R to add a *newline* when printing out the warning.

2.8 Exercise 114.2

Question 114.2.1: Write a function that calculates the areas of a circle, taking the radius as the first parameter.

Question 114.2.2: Write a function that calculates the volume of a cylinder, taking the radius of the base as the first parameter and the height as the second parameter. The function should call the function defined above and multiply the returned value by the height to calculate the result.

Question 114.2.3: Write a function with two parameters, a vector of numbers and a vector of characters (text). The function should check that the input has the correct data type. If all the numbers in the first vector are greater than zero, return the elements of the second vector from the first to the length of the first vector.

Chapter 3

Data wrangling Pt. 1

Stefano De Sabbata

3.1 R Projects

RStudio provides an extremely useful functionality to organise all your code and data, that is **R Projects**. Those are specialised files that RStudio can use to store all the information it has on a specific project that you are working on – *Environment*, *History*, working directory, and much more, as we will see in the coming weeks.

In RStudio Server, in the *Files* tab of the bottom-left panel, click on *Home* to make sure you are in your home folder – if you are working on your own computer, create a folder for these practicals wherever most convenient. Click on *New Folder* and enter *Practicals* in the prompt dialogue, to create a folder named *Practicals*.

Select *File > New Project …* from the main menu, then from the prompt menu, *New Directory*, and then *New Project*. Insert *Practical_204* as the directory name, and select the *Practicals* folder for the field *Create project as subdirectory of*. Finally, click *Create Project*.

RStudio has now created the project, and it should have activated it. If that is the case, the *Files* tab in the bottom-right panel should be in the *Practical_204* folder, which contains only the *Practical_204.Rproj* file. The *Practical_204.Rproj* stores all the *Environment* information for the current project and all the project files (e.g., R scripts, data, output files) should be stored within the *Practical_204* folder. Moreover, the *Practical_204* is now your working directory, which means that you can refer to a file in the folder by using

only its name and if you save a file that is the default directory where to save it.

On the top-right corner of RStudio, you should see a blue icon representing an R in a cube, next to the name of the project (*Practical_204*). That also indicates that you are within the *Practical_204* project. Click on *Practical_204* and select *Close Project* to close the project. Next to the R in a cube icon, you should now see *Project: (None)*. Click on *Project: (None)* and select *Practical_204* from the list to reactivate the *Practical_204* project.

With the *Practical_204* project activated, select from the top menu *File > New File > R Script*. That opens the embedded RStudio editor and a new empty R script folder. Copy the two lines below into the file. The first loads the `tidyverse` library, whereas the second loads another library that the code below uses to produce well-formatted tables.

```
library(tidyverse)
library(knitr)
```

From the top menu, select *File > Save*, type in *My_script_Practical_204.R* (make sure to include the underscore and the *.R* extension) as *File name*, and click *Save*.

3.2 Install libraries

RStudio and RStudio Server come with a number of libraries already pre-installed. However, you might find yourself in the position of wanting to install additional libraries to work with.

The remainder of this practical requires the library `nycflights13`. To install it, select *Tools > Install Packages...* from the top menu. Insert `nycflights13` in the *Packages (separate multiple with space or comma)* field and click install. RStudio will automatically execute the command `install.packages("nycflights13")` (so, no need to execute that yourself) and install the required library.

As usual, use the function `library` to load the newly installed library.

```
library(nycflights13)
```

The library `nycflights13` contains a dataset storing data about all the flights departed from New York City in 2013. The code below, loads the data frame `flights` from the library `nycflights13` into the variable `flights_from_nyc`, using the `::` operator to indicate that the data frame `flights` is situated within the library `nycflights13`.

```
flights_from_nyc <- nycflights13::flights
```

Add both lines above to your R script, as well as the code snippets provided as an example below.

3.2.1 Loading R scripts

It is furthermore possible to load the function(s) defined in one script from another script – in a fashion similar to when a library is loaded.

```
cube_root <- function (input_value) {
   result <- input_value ^ (1 / 3)
   result
}
```

Create a new R script named `Practical_204_RS_main.R` and copy the code below in that second R script and save the file.

```
source("Practical_204_RS_functions.R")

cube_root(27)
```

Executing the `Practical_204_RS_main.R` instructs the interpreter first to run the `Practical_204_RS_functions.R` script, thus creating the `cube_root` function, and then invoke the function using `27` as an argument, thus returning again `3`. That is a simple example, but this can be an extremely powerful tool to create your own library of functions to be used by different scripts.

3.3 Data manipulation

The analysis below uses the `dplyr` library (also part of the Tidyverse), which it offers a grammar for data manipulation.

For instance, the function `count` can be used to count the number rows of a data frame. The code below provides `flights_from_nyc` as input to the function `count` through the pipe operator, thus creating a new `tibble` with only one row and one column.

As discussed in the previous lecture, a `tibble` is data type similar to data frames, used by all the Tidyverse libraries.

All Tidyverse functions output `tibble` rather than `data.frame` objects when representing a table. However, `data.frame` object can be provided as input, as they are automatically converted by Tidyverse functions before proceeding with the processing steps.

In the `tibble` outputted by the `count` function below, the column `n` provides the count. The function `kable` of the library `knitr` is used to produce a well-formatted table.

```
flights_from_nyc %>%
  dplyr::count() %>%
  knitr::kable()
```

n
336776

The example above already shows how the **pipe operator** can be used effectively in a multi-step operation.

The function `count` can also be used to count the number rows of a table that have the same value for a given column, usually representing a category.

In the example below, the column name `origin` is provided as an argument to the function `count`, so rows representing flights from the same origin are counted together – EWR is the Newark Liberty International Airport, JFK is the John F. Kennedy International Airport, and LGA is LaGuardia Airport.

```
flights_from_nyc %>%
  dplyr::count(origin) %>%
  knitr::kable()
```

origin	n
EWR	120835
JFK	111279
LGA	104662

As you can see, the code above is formatted in a way similar to a code block, although it is not a code block. The code goes to a new line after every `%>%`, and space is added at the beginning of new lines. That is very common in R programming (especially when functions have many parameters) as it makes the code more readable.

3.3.1 Summarise

To carry out more complex aggregations, the function `summarise` can be used in combination with the function `group_by` to summarise the values of the rows of a data frame. Rows having the same value for a selected column (in the example below, the same origin) are grouped together, then values are aggregated based on the defined function (using one or more columns in the calculation).

In the example below, the function `sum` is applied to the column `distance` to calculate `distance_traveled_from` (the total distance travelled by flights starting from each airport).

```
flights_from_nyc %>%
  dplyr::group_by(origin) %>%
  dplyr::summarise(
    distance_traveled_from = sum(distance)
  ) %>%
  knitr::kable()
```

origin	distance_traveled_from
EWR	127691515
JFK	140906931
LGA	81619161

3.3.2 Select and filter

The function `select` can be used to select some **columns** to output. For instance in the code below, the function `select` is used to select the columns `origin`, `dest`, and `dep_delay`, in combination with the function `slice_head`, which can be used to include only the first `n` rows (`5` in the example below) to output.

```
flights_from_nyc %>%
  dplyr::select(origin, dest, dep_delay) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

origin	dest	dep_delay
EWR	IAH	2
LGA	IAH	4
JFK	MIA	2
JFK	BQN	-1
LGA	ATL	-6

The function `filter` can instead be used to filter **rows** based on a specified condition. In the example below, the output of the `filter` step only includes the rows where the value of `month` is `11` (i.e., the eleventh month, November).

```
flights_from_nyc %>%
  dplyr::select(origin, dest, year, month, day, dep_delay) %>%
  dplyr::filter(month == 11) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

origin	dest	year	month	day	dep_delay
JFK	PSE	2013	11	1	6
JFK	SYR	2013	11	1	105
EWR	CLT	2013	11	1	-5
LGA	IAH	2013	11	1	-6
JFK	MIA	2013	11	1	-3

Notice how `filter` is used in combination with `select`. All functions in the `dplyr` library can be combined, in any other order that makes logical sense. However, if the `select` step didn't include `month`, that same column couldn't have been used in the `filter` step.

3.3.3 Mutate

The function `mutate` can be used to add a new column to an output table. The `mutate` step in the code below adds a new column `air_time_hours` to the table obtained through the pipe, that is the flight air time in hours, dividing the flight air time in minutes by `60`.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::mutate(
    air_time_hours = air_time / 60
  ) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

flight	origin	dest	air_time	air_time_hours
1545	EWR	IAH	227	3.783333
1714	LGA	IAH	227	3.783333
1141	JFK	MIA	160	2.666667
725	JFK	BQN	183	3.050000
461	LGA	ATL	116	1.933333

3.3.4 Arrange

The function `arrange` can be used to sort a tibble by ascending order of the values in the specified column. If the operator `-` is specified before the column name, the descending order is used. The code below would produce a table showing all the rows when ordered by descending order of air time.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::arrange(-air_time) %>%
  knitr::kable()
```

In the examples above, we have used `slice_head` to present only the first `n` (in the examples `5`) rows in a table, based on the existing order. The `dplyr` library also provides the functions `slice_max` and `slice_min` which incorporate the sorting functionality (see `slice` reference page).

As such, the following code uses `slice_max` to produce a table including only the 5 rows with the *highest* air time.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::slice_max(air_time, n = 5) %>%
  knitr::kable()
```

flight	origin	dest	air_time
15	EWR	HNL	695
51	JFK	HNL	691
51	JFK	HNL	686
51	JFK	HNL	686
51	JFK	HNL	683

The following code, instead, uses `slice_min`, thus producing a table including only the 5 rows with the *lowest* air time.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::slice_min(air_time, n = 5) %>%
  knitr::kable()
```

flight	origin	dest	air_time
4368	EWR	BDL	20
4631	EWR	BDL	20
4276	EWR	BDL	21
4619	EWR	PHL	21
4368	EWR	BDL	21
4619	EWR	PHL	21
2132	LGA	BOS	21
3650	JFK	PHL	21
4118	EWR	BDL	21
4276	EWR	BDL	21
4276	EWR	BDL	21
4276	EWR	BDL	21
4276	EWR	BDL	21
4577	EWR	BDL	21
6062	EWR	BDL	21
3847	EWR	BDL	21

In both cases, if the table contains ties, all rows containing a value that is present among the maximum or minimum selected values are presented, as it is the case with the rows containing the value `21` in the example above.

3.4 Data manipulation example

Finally, the code below illustrates a more complex, multi-step operation using all the functions discussed above.

1. Start from the `flights_from_nyc` data.
2. Select origin, destination, departure delay, year, month, and day.
3. Filter only rows referring to flights in November.
4. Filter only rows where departure delay is not (notice that the negation operator `!` is used) `NA`.

- That is necessary because the function `mean` would return `NA` as output if any of the values in the column is `NA`.

5. Group by destination.
6. Calculated the average delay per destination.
7. Add a column with the delay calculated in hours (minutes over 60).
8. Sort the table by *descending* delay (note that `-` is used before the column name).
9. Only show the first 5 rows.
10. Create a well-formatted table.

```
flights_from_nyc %>%
  dplyr::select(origin, dest, year, month, day, dep_delay) %>%
  dplyr::filter(month == 11) %>%
  dplyr::filter(!is.na(dep_delay)) %>%
  dplyr::group_by(dest) %>%
  dplyr::summarize(
    avg_dep_delay = mean(dep_delay)
  ) %>%
  dplyr::mutate(
    avg_dep_delay_hours = avg_dep_delay / 60
  ) %>%
  dplyr::arrange(-avg_dep_delay_hours) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

dest	avg_dep_delay	avg_dep_delay_hours
SBN	67.50000	1.1250000
BDL	26.66667	0.4444444
CAK	19.70909	0.3284848
BHM	19.61905	0.3269841
DSM	16.14815	0.2691358

3.5 Exercise 204.1

Extend the code in the script `My_script_Practical_204.R` to include the code necessary to solve the questions below.

Question 204.1.1: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the average air time in hours, calculated grouping flights by carrier, but only for flights starting from the JFK airport.

Question 204.1.2: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the average arrival delay compared to the overall air time (**tip**: use manipulate to create a new column that takes the result of `arr_delay / air_time`) calculated grouping flights by carrier, but only for flights starting from the JFK airport.

Question 204.1.3: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the average arrival delay compared to the overall air time calculated grouping flights by origin and destination, sorted by destination.

Chapter 4

Data wrangling Pt. 2

Stefano De Sabbata

This section illustrates the re-shape and join functionalities of the Tidyverse libraries using simple examples. The following sections instead present a more complex example, loading and wrangling with data related to the 2011 Output Area Classification and the Indexes of Multiple Deprivation 2015.

```
library(tidyverse)
library(knitr)
```

4.1 Table manipulation

4.1.1 Long and wide formats

Tabular data are usually presented in two different formats.

- **Wide**: this is the most common approach, where each real-world entity (e.g. a city) is represented by *one single row* and its attributes are represented through different columns (e.g., a column representing the total population in the area, another column representing the size of the area, etc.).

City	Population	Area	Density
Leicester	329,839	73.3	4,500
Nottingham	321,500	74.6	4,412

- **Long**: this is probably a less common approach, but still necessary in

many cases, where each real-world entity (e.g. a city) is represented by *multiple rows*, each one reporting only one of its attributes. In this case, one column is used to indicate which attribute each row represent, and another column is used to report the value.

City	Attribute	Value
Leicester	Population	329,839
Leicester	Area	73.3
Leicester	Density	4,500
Nottingham	Population	321,500
Nottingham	Area	74.6
Nottingham	Density	4,412

The `tidyr` library provides two functions that allow transforming wide-formatted data to a long format, and vice-versa. Please take your time to understand the example below and check out the `tidyr` help pages before continuing.

```
city_info_wide <- data.frame(
  City = c("Leicester", "Nottingham"),
  Population = c(329839, 321500),
  Area = c(73.3, 74.6),
  Density = c(4500, 4412)
)

kable(city_info_wide)
```

City	Population	Area	Density
Leicester	329839	73.3	4500
Nottingham	321500	74.6	4412

```
city_info_long <- city_info_wide %>%
  gather(
    # exclude IDs (city names) from gathering
    -City,
    # name for the new key column
    key = "Attribute",
    # name for the new value column
    value = "Value"
  )

kable(city_info_long)
```

City	Attribute	Value
Leicester	Population	329839.0
Nottingham	Population	321500.0
Leicester	Area	73.3
Nottingham	Area	74.6
Leicester	Density	4500.0
Nottingham	Density	4412.0

```
city_info_back_to_wide <- city_info_long %>%
  spread(
    # specify key column
    key = "Attribute",
    # specify value column
    value = "Value"
  )

kable(city_info_back_to_wide)
```

City	Area	Density	Population
Leicester	73.3	4500	329839
Nottingham	74.6	4412	321500

4.1.2 Join

A join operation combines two tables into one by matching rows that have the same values in the specified column. This operation is usually executed on columns containing identifiers, which are matched through different tables containing different data about the same real-world entities. For instance, the table below presents the telephone prefixes for two cities. That information can be combined with the data present in the wide-formatted table above through a join operation on the columns containing the city names. As the two tables do not contain all the same cities, if a full join operation is executed, some cells have no values assigned.

City	TelephonePrefix
Leicester	0116
Birmingham	0121

City	Population	Area	Density	TelephonePrefix
Leicester	329,839	73.3	4,500	0116
Nottingham	321,500	74.6	4,412	
Birmingham				0121

As discussed in the lecture, the `dplyr` library offers different types of join operations, which correspond to the different `SQL` joins illustrated in the image below. The use and implications of these different types of joins will be discussed in more detail in the GY7708 module next semester.

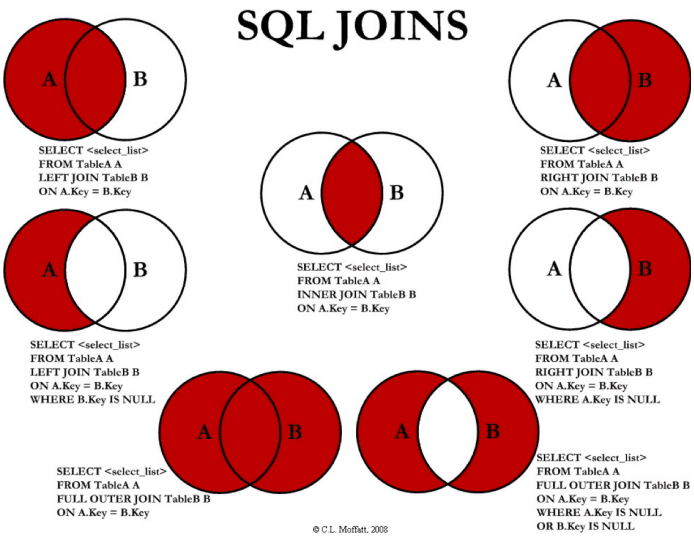


Figure 4.1: by C.L. Moffatt, licensed under The Code Project Open License (CPOL)

Please take your time to understand the example below and check out the related `dplyr` help pages before continuing. Note how the result of the join operations is *not* saved to a variable. The function `kable` is added after each join operation through a pipe `%>%` to display the resulting table in a nice format.

```
city_telephone_prexix <- data.frame(
  City = c("Leicester", "Birmingham"),
  TelephonPrefix = c("0116", "0121")
)

kable(city_telephone_prexix)
```

City	TelephonPrefix
Leicester	0116
Birmingham	0121

```
full_join(
    city_info_wide, city_telephone_prexix,
    by = c("City" = "City")
  ) %>%
  kable()
```

City	Population	Area	Density	TelephonPrefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA
Birmingham	NA	NA	NA	0121

```
left_join(
    city_info_wide, city_telephone_prexix,
    by = c("City" = "City")
  ) %>%
  kable()
```

City	Population	Area	Density	TelephonPrefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA

```
right_join(
    city_info_wide, city_telephone_prexix,
    by = c("City" = "City")
  ) %>%
  kable()
```

City	Population	Area	Density	TelephonPrefix
Leicester	329839	73.3	4500	0116
Birmingham	NA	NA	NA	0121

```
inner_join(
    city_info_wide, city_telephone_prexix,
    by = c("City" = "City")
  ) %>%
  kable()
```

City	Population	Area	Density	TelephonPrefix
Leicester	329839	73.3	4500	0116

4.2 Read and write data

The `readr` library (also part of the Tidyverse) provides a series of functions that can be used to load from and save data to different file formats.

Download from the data folder of the repository the following files:

- `2011_OAC_Raw_uVariables_Leicester.csv`
- `IndexesMultipleDeprivation2015_Leicester.csv`

Create a *Practical_202* project and make sure it is activated and thus the *Practical_202* showing in the *File* tab in the bottom-right panel. Upload the two files to the *Practical_202* folder by clicking on the *Upload* button and selecting the files from your computer (at the time of writing, Chrome seems to upload the file correctly, whereas it might be necessary to change the names of the files after upload using Microsoft Edge).

Create a new R script named `Data_Wrangling_Example.R` in the *Practical_202* project, and add `library(tidyverse)` as the first line. Use that new script for this and the following sections of this practical session.

The 2011 Output Area Classification (2011 OAC) is a geodemographic classification of the census Output Areas (OA) of the UK, which was created by Gale et al. (2016) starting from an initial set of 167 prospective variables from the United Kingdom Census 2011: 86 were removed, 41 were retained as they are, and 40 were combined, leading to a final set of 60 variables. Gale et al. (2016) finally used the k-means clustering approach to create 8 clusters or supergroups (see map at datashine.org.uk), as well as 26 groups and 76 subgroups. The dataset in the file `2011_OAC_Raw_uVariables_Leicester.csv` contains all the original 167

variables, as well as the resulting groups, for the city of Leicester. The full variable names can be found in the file `2011_OAC_Raw_uVariables_Lookup.csv`.

The Indexes of Multiple Deprivation 2015 (see map at cdrc.ac.uk) are based on a series of variables across seven distinct domains of deprivation which are combined to calculate the Index of Multiple Deprivation 2015 (IMD 2015). That is an overall measure of multiple deprivations experienced by people living in an area. These indexes are calculated for every Lower layer Super Output Area (LSOA), which are larger geographic unit than the OAs used for the 2011 OAC. The dataset in the file `IndexesMultipleDeprivation2015_Leicester.csv` contains the main Index of Multiple Deprivation, as well as the values for the seven distinct domains of deprivation, and two additional indexes regarding deprivation affecting children and older people. The dataset includes scores, ranks (where 1 indicates the most deprived area), and decile (i.e., the first decile includes the 10% most deprived areas in England).

The `read_csv` function reads a *Comma Separated Values (CSV)* file from the path provided as the first argument. The code below loads the 2011 OAC dataset. The `read_csv` instruction throws a warning that shows the assumptions about the data types used when loading the data. As illustrated by the output of the last line of code, the data are loaded as a tibble 969 x 190, that is 969 rows – one for each OA – and 190 columns, 167 of which represent the input variables used to create the 2011 OAC.

```
leicester_2011OAC <- read_csv("2011_OAC_Raw_uVariables_Leicester.csv")

leicester_2011OAC %>%
  select(OA11CD,LSOA11CD, supgrpcode,supgrpname,Total_Population) %>%
  slice_head(n = 3) %>%
  kable()
```

OA11CD	LSOA11CD	supgrpcode	supgrpname	Total_Population
E00069517	E01013785	6	Suburbanites	313
E00069514	E01013784	2	Cosmopolitans	323
E00169516	E01013713	4	Multicultural Metropolitans	341

The code below loads the IMD 2015 dataset.

```
# Load Indexes of Multiple deprivation data
leicester_IMD2015 <- read_csv("IndexesMultipleDeprivation2015_Leicester.csv")
```

The function `write_csv` can be used to save a dataset as a `csv` file. For instance, the code below uses `tidyverse` functions and the pipe operator `%>%` to:

1. **read** the 2011 OAC dataset again directly from the file, but without storing it into a variable;
2. **select** the OA code variable `OA11CD`, and the two variables representing the code and name of the supergroup assigned to each OA by the 2011 OAC (`supgrpcode` and `supgrpname` respectively);

3. **filter** only those OA in the supergroup *Suburbanites* (code `6`);
4. **write** the results to a file named *Leicester_Suburbanites.csv* in your home folder.

```
read_csv("2011_OAC_Raw_uVariables_Leicester.csv") %>%
  select(OA11CD, supgrpcode, supgrpname) %>%
  filter(supgrpcode == 6) %>%
  write_csv("~/Leicester_Suburbanites.csv")
```

4.3 data wrangling example

4.3.1 Re-shaping

The IMD 2015 data are in a *long* format, which means that every area is represented by more than one row: the column `Value` presents the value; the column `IndicesOfDeprivation` indicates which index the value refers to; the column `Measurement` indicates whether the value is a score, rank, or decile. The code below illustrates the data format for the LSOA including the University of Leicester (feature code `E01013649`).

```
leicester_IMD2015 %>%
  filter(FeatureCode == "E01013649") %>%
  select(FeatureCode, IndicesOfDeprivation, Measurement, Value)
```

```
## # A tibble: 30 x 4
##    FeatureCode IndicesOfDeprivation                          Measurement  Value
##    <chr>       <chr>                                         <chr>        <dbl>
##  1 E01013649   Income Deprivation Domain                      Score        0.07
##  2 E01013649   Employment Deprivation Domain                 Score        0.075
##  3 E01013649   Income Deprivation Affecting Children Index (~ Score        0.087
##  4 E01013649   Income Deprivation Affecting Older People Ind~ Score        0.153
##  5 E01013649   Health Deprivation and Disability Domain       Score        0.272
##  6 E01013649   Index of Multiple Deprivation (IMD)            Score       19.7
##  7 E01013649   Education, Skills and Training Domain          Score        2.19
##  8 E01013649   Barriers to Housing and Services Domain        Score       14.3
##  9 E01013649   Living Environment Deprivation Domain          Score       57.2
## 10 E01013649   Crime Domain                                   Score        1.16
## # ... with 20 more rows
```

In the following section, the analysis aims to explore how certain census variables vary in areas with different deprivation levels. Thus, we need to extract the `Decile` rows from the IMD 2015 dataset and transform the data in a *wide* format, where each index is represented as a separate column.

To that purpose, we also need to change the name of the indexes slightly, to exclude spaces and punctuation, so that the new column names are simpler than the original text, and can be used as column names. That part of the

manipulation is performed using `mutate` and functions from the `stringr` library.

```
leicester_IMD2015_decile_wide <- leicester_IMD2015 %>%
  # Select only Socres
  filter(Measurement == "Decile") %>%
  # Trim names of IndicesOfDeprivation
  mutate(IndicesOfDeprivation = str_replace_all(IndicesOfDeprivation, "\\s", "")) %>%
  mutate(IndicesOfDeprivation = str_replace_all(IndicesOfDeprivation, "[:punct:]", "")) %>%
  mutate(IndicesOfDeprivation = str_replace_all(IndicesOfDeprivation, "\\(", "")) %>%
  mutate(IndicesOfDeprivation = str_replace_all(IndicesOfDeprivation, "\\)", "")) %>%
  # Spread
  spread(
    key = IndicesOfDeprivation,
    value = Value
  ) %>%
  # Drop columns
  select(-DateCode, -Measurement, -Units)
```

Let's compare the columns of the original *long* IMD 2015 dataset with the *wide* dataset created above.

```
colnames(leicester_IMD2015)
```

```
## [1] "FeatureCode"          "DateCode"             "Measurement"
## [4] "Units"                "Value"                "IndicesOfDeprivation"
```

```
colnames(leicester_IMD2015_decile_wide)
```

```
##  [1] "FeatureCode"
##  [2] "BarrierstoHousingandServicesDomain"
##  [3] "CrimeDomain"
##  [4] "EducationSkillsandTrainingDomain"
##  [5] "EmploymentDeprivationDomain"
##  [6] "HealthDeprivationandDisabilityDomain"
##  [7] "IncomeDeprivationAffectingChildrenIndexIDACI"
##  [8] "IncomeDeprivationAffectingOlderPeopleIndexIDAOPI"
##  [9] "IncomeDeprivationDomain"
## [10] "IndexofMultipleDeprivationIMD"
## [11] "LivingEnvironmentDeprivationDomain"
```

Thus, we now have only one row representing the LSOA including the University of Leicester (feature code `E01013649`) and the main Index of Multiple Deprivations is now represented by the column `IndexofMultipleDeprivationIMD`. The value reported is the same – that is 5, which means that the selected LSOA is estimated to be in the range 40-50% most deprived areas in England – but we changed the data format.

```
# Original long IMD 2015 dataset
leicester_IMD2015 %>%
```

```
  filter(
    FeatureCode == "E01013649",
    IndicesOfDeprivation == "Index of Multiple Deprivation (IMD)",
    Measurement == "Decile"
  ) %>%
  select(FeatureCode, IndicesOfDeprivation, Measurement, Value)
```

```
## # A tibble: 1 x 4
##   FeatureCode IndicesOfDeprivation                Measurement Value
##   <chr>       <chr>                               <chr>       <dbl>
## 1 E01013649   Index of Multiple Deprivation (IMD) Decile          5
```

```
# New wide IMD 2015 dataset
leicester_IMD2015_decile_wide %>%
  filter(FeatureCode == "E01013649") %>%
  select(FeatureCode, IndexofMultipleDeprivationIMD)
```

```
## # A tibble: 1 x 2
##   FeatureCode IndexofMultipleDeprivationIMD
##   <chr>                               <dbl>
## 1 E01013649                               5
```

4.3.2 Join

As discussed above, two tables can be joined using a common column of identifiers. We can thus join the 2011 OAC and the IMD 2015 datasets into a single table. The LSOA code included in the 2011 OAC table is used to match that information with the corresponding row in the IMD 2015. The resulting table provides all the information from the 2011 OAC for each OA, plus the Index of Multiple Deprivations decile for the LSOA containing each OA.

That operation can be carried out using the function `merge`, and specifying the common column (or columns, if more than one is to be used as identifier) as argument of `by`.

```
leicester_2011OAC_IMD2015 <- leicester_2011OAC %>%
  left_join(leicester_IMD2015_decile_wide, by = c("LSOA11CD" = "FeatureCode"))
```

Once the result is stored into the variable `leicester_2011OAC_IMD2015`, further analysis can be carried out. For instance, `count` can be used to count how many OAs fall into each 2011 OAC supergroup and decile of the Index of Multiple Deprivations.

```
leicester_2011OAC_IMD2015 %>%
  count(supgrpname, IndexofMultipleDeprivationIMD)
```

```
## # A tibble: 46 x 3
##    supgrpname                IndexofMultipleDeprivationIMD     n
```

```
##    <chr>                                          <dbl> <int>
##  1 Constrained City Dwellers                          1    30
##  2 Constrained City Dwellers                          2     3
##  3 Constrained City Dwellers                          3     2
##  4 Constrained City Dwellers                          6     1
##  5 Cosmopolitans                                      2    25
##  6 Cosmopolitans                                      3    15
##  7 Cosmopolitans                                      4    15
##  8 Cosmopolitans                                      5     8
##  9 Cosmopolitans                                      6    10
## 10 Cosmopolitans                                      8    10
## # ... with 36 more rows
```

As another example, the code below can be used to group OAs based on the decile and then calculate the percentage of adults not in employment using the `u074` (*No adults in employment in household: With dependent children*) and `u075` (*No adults in employment in household: No dependent children*) variables from the 2011 OAC dataset.

```
leicester_2011OAC_IMD2015 %>%
  group_by(IndexofMultipleDeprivationIMD) %>%
  summarise(
    adults_not_empl_perc = (sum(u074 + u075) / sum(Total_Population)) * 100
  ) %>%
  kable()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

IndexofMultipleDeprivationIMD	adults_not_empl_perc
1	17.071876
2	14.191205
3	10.405029
4	9.966309
5	11.337036
6	10.710509
7	10.641026
8	9.686658
9	9.898140

4.4 Exercise 3.1

Extend the code in the script `Data_Wrangling_Example.R` to include the code necessary to solve the questions below.

Question 3.1.1: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the percentage of EU citizens over total population, calculated grouping OAs by the related decile of the Index of Mul-

tiple Deprivations, but only accounting for areas classified as Cosmopolitans or Ethnicity Central or Multicultural Metropolitans.

Question 3.1.2: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the percentage of EU citizens over total population, calculated grouping OAs by the related supergroup in the 2011 OAC, but only accounting for areas in the top 5 deciles of the Index of Multiple Deprivations.

Question 3.1.3: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the percentage of people aged 65 and above, calculated grouping OAs by the related supergroup in the 2011 OAC and decile of the Index of Multiple Deprivations, and ordering the table by the calculated value in a descending order.

4.5 Exercise 3.2

Extend the code in the script `Data_Wrangling_Example.R` to include the code necessary to solve the questions below.

Question 3.2.1: Write a piece of code using the pipe operator and the `dplyr` and `tidyr` libraries to generate a long format of the `leicester_2011OAC_IMD2015` table only including the values (census variables) used in *Question 3.1.3*.

Question 3.2.2: Write a piece of code using the pipe operator and the `dplyr` and `tidyr` libraries to generate a table similar to the one generated for *Question 3.2.1*, but showing the values as percentages over total population.

4.6 Solutions

A full R Script including the code for the analysis presented in this practical session and the solutions to the 5 questions above is available in the Exercises folder (`docs/exercises`) of the repository (`202_X_Data_Wrangling2_Example.R`). Upload the prepared script to your *Practical_202* project folder, click on the uploaded file to open it in a new editor tab and compare it to your script.

Chapter 5

Reproducibility

Stefano De Sabbata

5.1 Markdown

A essential tool used in creating these materials is RMarkdown That is an R library that allows you to create scripts that mix the Markdown mark-up language and R, to create dynamic documents. RMarkdown script can be compiled, at which point, the Markdown notation is interpreted to create the output files, while the R code is executed and the output incorporated in the document.

For instance the following markdown code

```
[This is a link to the University of Leicester](http://le.ac.uk) and **this is in bold**.
```

is rendered as

This is a link to the University of Leicester and **this is in bold**.

The core Markdown notation used in this session is presented below. A full RMarkdown *cheatsheet* is available here.

```
# Header 1
## Header 2
### Header 3
#### Header 4
##### Header 5

**bold**
*italics*
```

```
[This is a link to the University of Leicester](http://le.ac.uk)

- Example list
    - Main folder
        - Analysis
        - Data
        - Utils
    - Other bullet point
- And so on
    - and so forth

1. These are
    1. Numeric bullet points
    2. Number two
2. Another number two
3. This is number three
```

5.1.1 R Markdown

R code can be embedded in RMarkdown documents as in the example below. That results in the code chunk be displayed within the document (as *echo=TRUE* is specified), followed by the output from the execution of the same code.

````
```{r, echo=TRUE}
a_number <- 0
a_number <- a_number + 1
a_number <- a_number + 1
a_number <- a_number + 1
a_number
```
````

```r
a_number <- 0
a_number <- a_number + 1
a_number <- a_number + 1
a_number <- a_number + 1
a_number
```

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

5.2 Exercise 5.1

Create a new R project named *Practical_301* as the directory name. Create an RMarkdown document in RStudio by selecting *File > New File > R Markdown …* – this might prompt RStudio to update some packages. On the RMarkdown

document creation menu, specify "Practical 05" as title and your name as the author, and select *PDF* as default output format.

The new document should contain the core document information, as in the example below, plus some additional content that simply explains how RMarkdown works.

```
---
title: "Practical 05"
author: "A. Student"
date: "7 October 2018"
output: pdf_document
---
```

Delete the contents below the document information and copy the following text below the document information.

````
# Pipe example

This is my first [RMarkdown](https://rmarkdown.rstudio.com/) document.

```{r, echo=TRUE}
library(tidyverse)
```

The code uses the pipe operator:

- takes 2 as input
- calculates the square root
- rounds the value
    - keeping only two digits

```{r, echo=TRUE}
2 %>%
  sqrt() %>%
  round(digits = 2)
```
````

The option `echo=TRUE` tells RStudio to include the code in the output document, along with the output of the computation. If `echo=FALSE` is specified, the code will be omitted. If the option `message=FALSE` and `warning=FALSE` are added, messages and warnings from R are not displayed in the output document.

Save the document by selecting *File* > *Save* from the main menu. Enter *Square_root* as file name and click *Save*. The file is saved using the *Rmd* (RMarkdown) extension.

Click on the *Knit* button on the bar above the editor panel (top-left area) in RStudio, on the left side. Check the resulting *pdf* document. Try adding some

of your own code (e.g., using some of the examples above) and Markdown text, and compile the document again.

5.3 Exercise 5.2

Create an analysis document based on RMarkdown for each one of the two analyses seen in the practical sessions 3 and 4. For each of the two analyses, within their respective R projects, first, create an RMarkdown document. Then, add the code from the related R script. Finally add additional content such as title, subtitles, and most importantly, some text describing the data used, how the analysis has been done, and the result obtained. Make sure you add appropriate links to the data sources, as available in the practical session materials.

5.4 Git

Git is a free and opensource version control system. It is commonly used through a server, where a master copy of a project is kept, but it can also be used locally. Git allows storing versions of a project, thus providing file synchronisation, consistency, history browsing, and the creation of multiple branches. For a detailed introduction to Git, please refer to the Pro Git book, written by Scott Chacon and Ben Straub.

As illustrated in the image below, when working with a git repository, the most common approach is to first check-out the latest version from the main repository before start working on any file. Once a series of edits have been made, the edits to stage are selected and then committed in a permanent snapshot. One or more commits can then be pushed to the main repository.

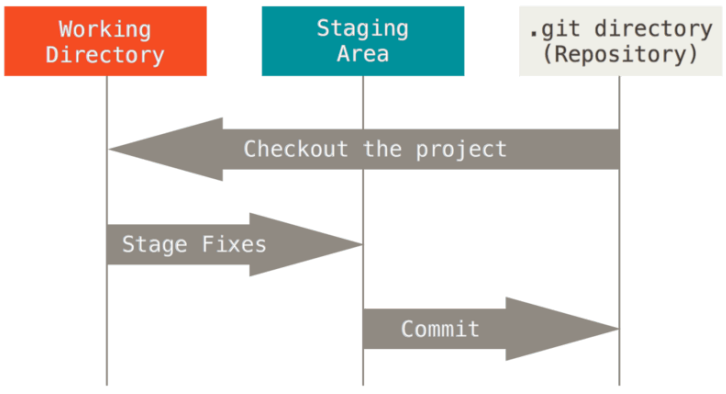


Figure 5.1: by Scott Chacon and Ben Straub, licensed under CC BY-NC-SA 3.0

5.4.1 Git and RStudio

In RStudio Server, in the *Files* tab of the bottom-left panel, click on *Home* to make sure you are in your home folder – if you are working on your own computer, create a folder for granolarr wherever most convenient. Click on *New Folder* and enter *Repos* (short for repositories) in the prompt dialogue, to create a folder named *Repos*.

Create a GitHub account at github.com, if you don't have one, and create a new repository named *my-granolarr*, following the instructions available on the GitHub help pages. Make sure you tick the box next to *Initialize this repository with a README*, which adds a `README.md` markdown file to your repository.

Once the repository has been created, GitHub will take you to the repository page. Copy the link to the repository `.git` file by clicking on the green *Clone or download* button and copying the `https` URL there available. Back to RStudio Server, select *File > New Project…* from the top menu and select *Version Control* and then *Git* from the *New Project* panel. Paste the copied URL in the *Repository URL* field, select the *Repos* folder created above as folder for *Create project as subdirectory of*, and click *Create Project*. RStudio might ask you for your GitHub username and password at this point.

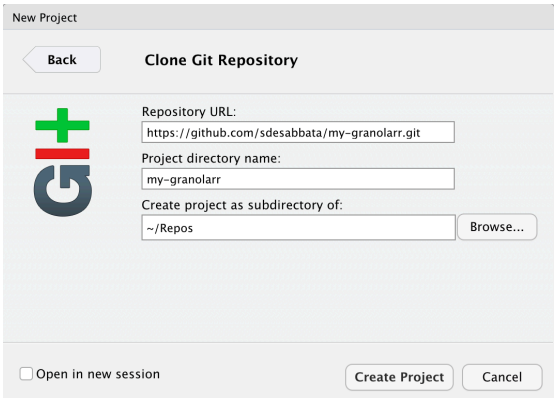


Figure 5.2: Cloning my-granolarr as R project in RStudio

Before continuing, you need to record your identity with the Git system installed on the RStudio Server, for it to be able to communicate with the GitHub's server. Open the *Terminal* tab in RStudio (if not visible, select *Tools > Terminal > New Terminal* from the top menu). First, paste the command below substituting `you@example.com` with your university email (make sure to maintain the double quotes) and press the return button.

```
git config --global user.email "you@example.com"
```

Then, paste the command below substituting `Your Name` with your name (make sure to maintain the double quotes) and press the return button.

```
git config --global user.name "Your Name"
```

RStudio should have now switched to a new R project linked to you *my-granolarr* repository. In the RStudio *File* tab in the bottom-right panel, navigate to the file created for the exercises above, select the files and copy them in the folder of the new project (in the *File* tab, *More* > *Copy To…*).

Check the now available *Git* tab on the top-right panel, and you should see at least the newly copied files marked as untracked. Tick the checkboxes in the *Staged* column to stage the files, then click on the *Commit* button.

In the newly opened panel *Commit* window, the top-left section shows the files, and the bottom section shows the edits. Write `My first commit` in the *Commit message* section in the top-right, and click the *Commit* button. A pop-up should notify the completed commit. Close both the pop-up panel, and click the *Push* button on the top-right of the *Commit* window. Another pop-up panel should ask you for your GitHub username and password and then show the executed push operation. Close both the pop-up panel and the *Commit* window.

Congratulations, you have completed your first commit! Check the repository page on GitHub. If you reload the page, the top bar should show *2 commits* on the left and your files should now be visible in the file list below. If you click on *2 commits*, you can see the commit history, including both the initial commit that created the repository and the commit you just completed.

5.4.2 Cloning granolarr

You can follow the steps listed below to clone the granolarr repository.

1. Create a folder named `Repos` in your home directory. If you are working on RStudio Server, in the *Files* panel, click on the `Home` button (second bar, next to the house icon), then click on `New Folder`, enter the name `Repos` and click `Ok`.
2. In RStudio or RStudio Server select `File > New Project...`
3. Select `Version Control` and then `Git` (you might need to set up Git first if you are working on your own computer)
4. Copy `https://github.com/sdesabbata/granolarr.git` in the `Repository URL` field and select the Repos folder for the field `Create project as subdirectory of`, and click on `Create Project`.
5. Have a look around
6. Click on the project name `granolarr` on the top-left of the interface and select `Close Project` to close the project.

As granolarr is a public repository, you can clone it, edit it as you wish and push it to your own copy of the repository. However, contributing your edits to

the original repository would require a few further steps. Check out the GitHub help pages if you are interested.

5.5 Exercise 5.3

Create a new repository for an R project exploring the presence of the different living arrangements in Leicester among both the different categories of the 2011 Output Area Classification and deciles of Index of Multiple Deprivations. Create the repository, clone it to RStudio Server as a new R project, and copy the required data in the project folder. Create an RMarkdown document and write your analysis, including:

- an introduction to the data and the aims of the project;
- a justification of the analysis methods;
- the code and related results;
- and a discussion of the results within the same document.

Chapter 6

Exploratory data analysis

Stefano De Sabbata

6.1 Introduction

This practical showcases an exploratory analysis of the distribution of people aged 20 to 24 in Leicester, using the `u011` variable from the 2011 Output Area Classification (2011OAC) dataset. Create a new R project for this practical session and create a new RMarkdown document to replicate the analysis in this document.

Once the document is set up, start by adding the first R code snipped including the code below, which is loads the 2011OAC dataset and the libraries used for the practical session.

```
library(tidyverse)
library(knitr)
leicester_2011OAC <- read_csv("2011_OAC_Raw_uVariables_Leicester.csv")
```

6.2 GGlot2 recap

As seen in the practical session 401, the `ggplot2` library is part of the Tidyverse, and it offers a series of functions for creating graphics **declaratively**, based on the concepts outlined in the Grammar of Graphics. While the `dplyr` library offers functionalities that cover *data manipulation* and *variable transformations*, the `ggplot2` library offers functionalities that allow to specify elements, define guides, and apply scale and coordinate system transformations.

- **Marks** can be specified in `ggplot2` using the `geom_` functions.
- The mapping of variables (table columns) to **visual variables** can be specified in `ggplot2` using the `aes` element.
- Furthermore, the `ggplot2` library:
  - automatically adds all necessary **guides** using default table column names, and additional functions can be used to overwrite the defaults;
  - provides a wide range of `scale_` functions that can be used to control the **scales** of all visual variables;
  - provides a series of `coord_` fucntions that allow transforming the **coordinate system**.

Check out the `ggplot2` reference for all the details about the functions and options discussed below.

6.3 Data visualisation

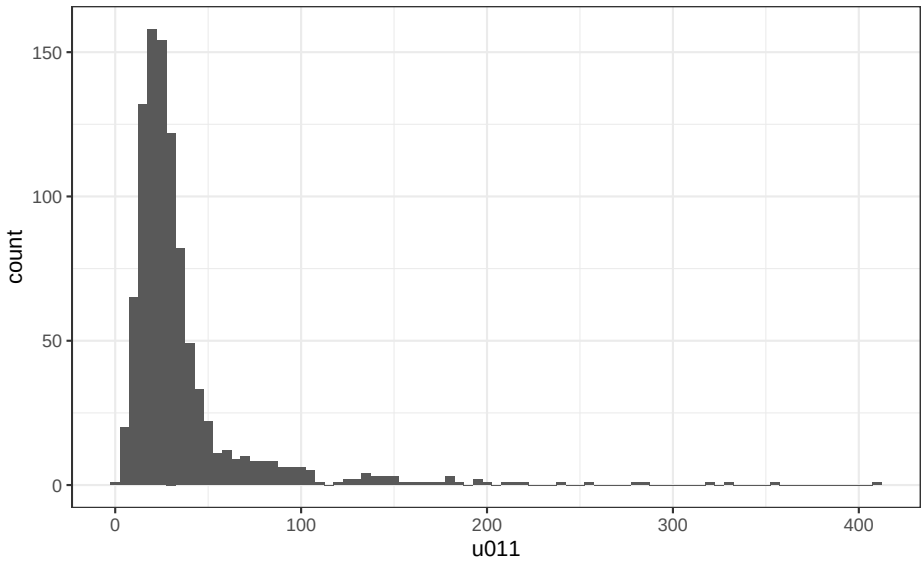
6.3.1 Distributions

We start the analysis with a simple histogram, to explore the distribution of the variable `u011`. RMarkdown allows specifying the height (as well as the width) of the figure as an option for the R snipped, as shown in the example typed out in plain text below.

````
```{r, echo=TRUE, message=FALSE, warning=FALSE, fig.height = 4}
leicester_2011OAC %>%
  ggplot(
    aes(
      x = u011
    )
  ) +
  geom_histogram(binwidth = 5) +
  theme_bw()
```
````

The snipped and barchart is included in output documents, as shown below.

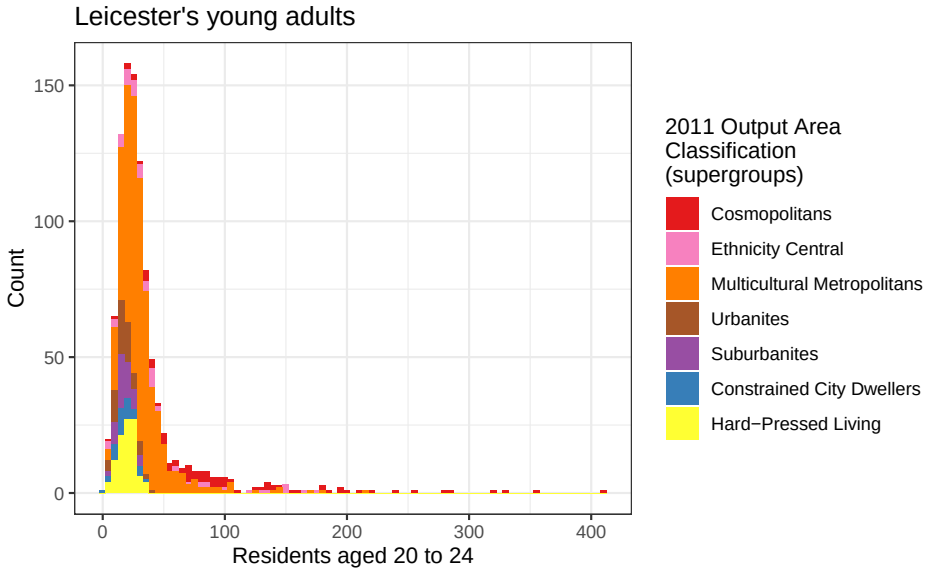
```r
leicester_2011OAC %>%
  ggplot(
    aes(
      x = u011
    )
  ) +
  geom_histogram(binwidth = 5) +
  theme_bw()
```



If we aim to explore how that portion of the population is distributed among the different supergroups of the 2011OAC, there are a number of charts that would allow us to visualise that relationship.

For instance, the barchart above can be enhanced through the use of the visual variable colour and the `fill` option. The graphic below uses a few options seen in the practical session 401 to create a stacked barchart, where sections of each bar are filled with the colour associated with a 2011OAC supergroup.

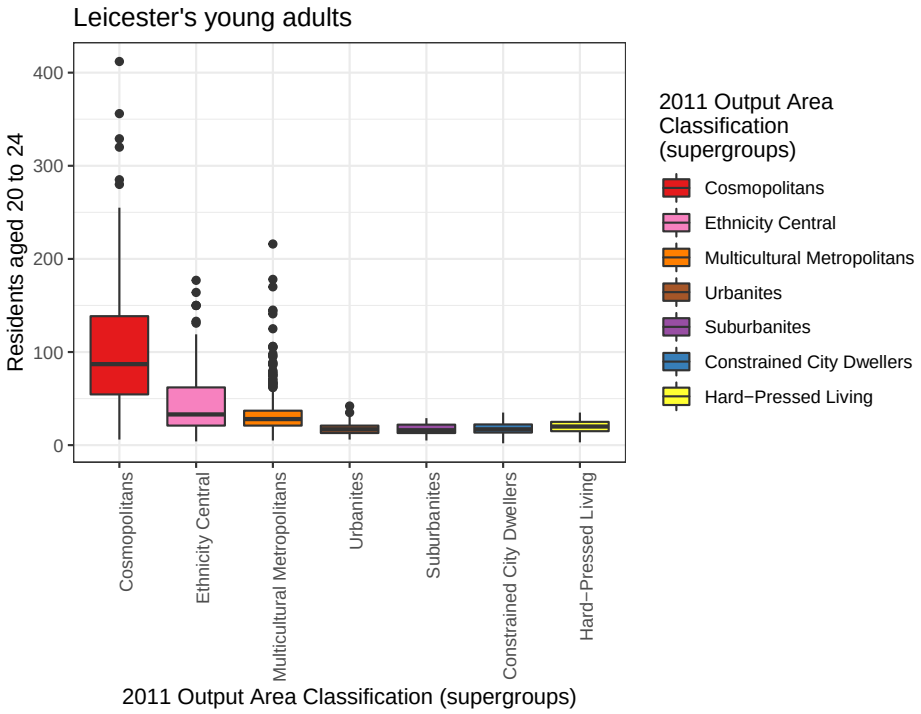
```
leicester_2011OAC %>%
  ggplot(
    aes(
      x = u011,
      fill = fct_reorder(supgrpname, supgrpcode)
    )
  ) +
  geom_histogram(binwidth = 5) +
  ggtitle("Leicester's young adults") +
  labs(
    fill = "2011 Output Area\nClassification\n(supergroups)"
  ) +
  xlab("Residents aged 20 to 24") +
  ylab("Count") +
  scale_fill_manual(
    values = c("#e41a1c", "#f781bf", "#ff7f00", "#a65628", "#984ea3", "#377eb8", "#ffff33")
  ) +
  theme_bw()
```



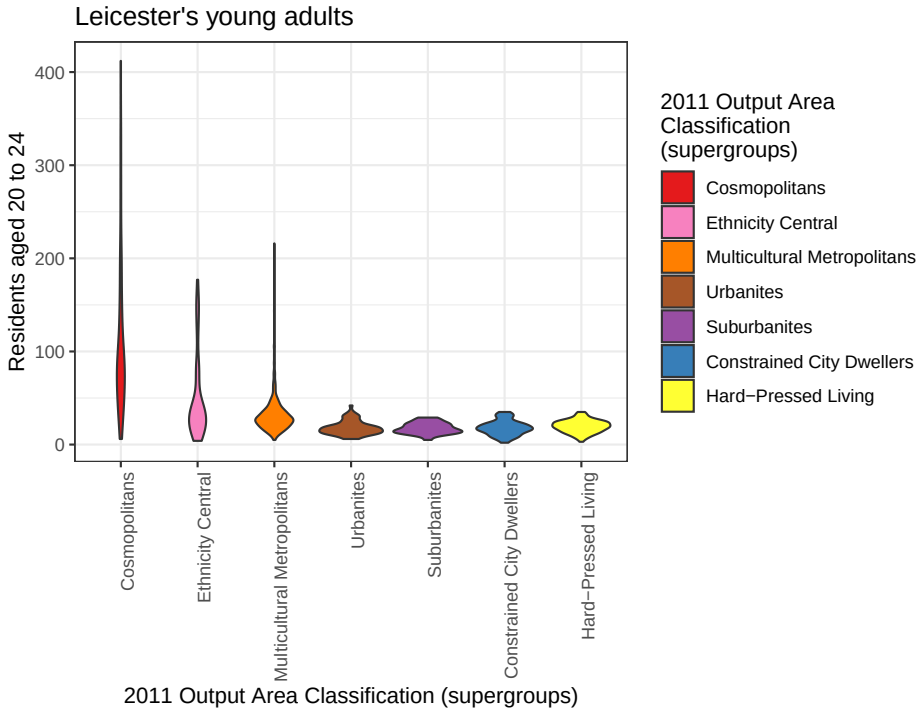
However, the graphic above is not extremely clear. A boxplot and a violin plot created from the same data are shown below. In both cases, the parameter `axis.text.x` of the function theme is set to `element_text(angle = 90, hjust = 1)` in order to orientate the labels on the x-axis vertically, as the supergroup names are rather long, and they would overlap one-another if set horizontally on the x-axis. In both cases, the option `fig.height` of the R snippet in RMarkdown should be set to a higher value (e.g., `5`) to allow for sufficient room for the supergroup names.

```
leicester_2011OAC %>%
  ggplot(
    aes(
      x = fct_reorder(supgrpname, supgrpcode),
      y = u011,
      fill = fct_reorder(supgrpname, supgrpcode)
    )
  ) +
  geom_boxplot() +
  ggtitle("Leicester's young adults") +
  labs(
    fill = "2011 Output Area\nClassification\n(supergroups)"
  ) +
  xlab("2011 Output Area Classification (supergroups)") +
  ylab("Residents aged 20 to 24") +
  scale_fill_manual(
    values = c("#e41a1c", "#f781bf", "#ff7f00", "#a65628", "#984ea3", "#377eb8", "#fff
  ) +
  theme_bw() +
```

```
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```




```
leicester_2011OAC %>%
  ggplot(
    aes(
      x = fct_reorder(supgrpname, supgrpcode),
      y = u011,
      fill = fct_reorder(supgrpname, supgrpcode)
    )
  ) +
  geom_violin() +
  ggtitle("Leicester's young adults") +
  labs(
    fill = "2011 Output Area\nClassification\n(supergroups)"
  ) +
  xlab("2011 Output Area Classification (supergroups)") +
  ylab("Residents aged 20 to 24") +
  scale_fill_manual(
    values = c("#e41a1c", "#f781bf", "#ff7f00", "#a65628", "#984ea3", "#377eb8", "#ffff33")
  ) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

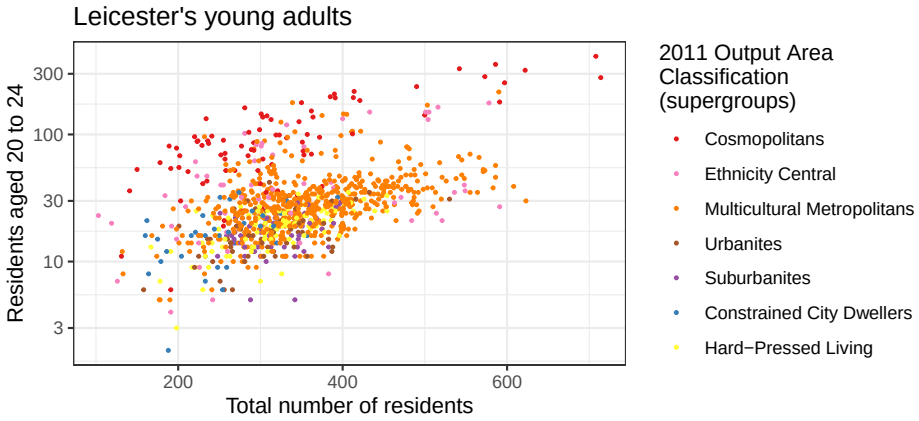


6.3.2 Relationships

The first barchart above seems to illustrate that the distribution might be skewed towards the left, with most values seemingly below 50. However, that tells only part of the story about how people aged 20 to 24 are distributed in Leicester. In fact, each Output Area (OA) has a different total population. So, a higher number of people aged 20 to 24 living in an OA might be simply due to the OA been more populous than others. Thus, the next step is to compare `u011` to `Total_Population`, for instance, through a scatterplot such as the one seen in the practical session 401, reported below.

```
leicester_2011OAC %>%
  ggplot(
    aes(
      x = Total_Population,
      y = u011,
      colour = fct_reorder(supgrpname, supgrpcode)
    )
  ) +
  geom_point(size = 0.5) +
  ggtitle("Leicester's young adults") +
  labs(
    colour = "2011 Output Area\nClassification\n(supergroups)"
```

```
  ) +
  xlab("Total number of residents") +
  ylab("Residents aged 20 to 24") +
  scale_y_log10() +
  scale_colour_brewer(palette = "Set1") +
  scale_colour_manual(
    values = c("#e41a1c", "#f781bf", "#ff7f00", "#a65628", "#984ea3", "#377eb8", "#ffff33")
  ) +
  theme_bw()
```



6.4 Exercise 7.1

Question 7.1.1: Which one of the boxplot or violin plot above do you think better illustrate the different distributions, and what do the two graphics say about the distribution of people aged 20 to 24 in Leicester? Write a short answer in your RMarkdown document (max 200 words).

Question 7.1.2: Create a jittered points plot (see `geom_jitter`) visualisation illustrating the same data shown in the boxplot and violin plot above.

Question 7.1.3: Create the code necessary to calculate a new column named `perc_age_20_to_24`, which is the percentage of people aged 20 to 24 (i.e., `u011`) over total population per OA `Total_Population`, and create a boxplot visualising the distribution of the variable per 2011OAC supergroup.

6.5 Exploratory statistics

The graphics above provide preliminary evidence that the distribution of people aged 20 to 24 might, in fact, be different in different 2011 supergroups. In the remainder of the practical session, we are going to explore that hypothesis further. First, load the necessary statistical libraries.

The code below calculates the percentage of people aged 20 to 24 (i.e., `u011`) over total population per OA, but it also recodes (see recode) the names of the 2011OAC supergroups to a shorter 2-letter version, which is useful for the tables presented further below.

Only the OA code, the recoded 2011OAC supergroup name, and the newly created `perc_age_20_to_24` are retained in the new table `leic_2011OAC_20to24`. Such a step is sometimes useful as stepping stone for further analysis and can make the code easier to read further down the line. Sometimes it is also a necessary step when interacting with certain libraries, which are not fully compatible with Tidyverse libraries, such as `leveneTest`.

```
leic_2011OAC_20to24 <- leicester_2011OAC %>%
  mutate(
    perc_age_20_to_24 = (u011 / Total_Population) * 100,
    supgrpname = dplyr::recode(supgrpname,
      `Suburbanites` = "SU",
      `Cosmopolitans` = "CP",
      `Multicultural Metropolitans` = "MM",
      `Ethnicity Central` = "EC",
      `Constrained City Dwellers` = "CD",
      `Hard-Pressed Living` = "HP",
      `Urbanites` = "UR"
    )
  ) %>%
  select(OA11CD, supgrpname, perc_age_20_to_24)

leic_2011OAC_20to24 %>%
  slice_head(n = 5) %>%
  kable()
```

OA11CD	supgrpname	perc_age_20_to_24
E00069517	SU	4.153355
E00069514	CP	30.650155
E00169516	MM	12.316716
E00169048	MM	6.956522
E00169044	MM	6.211180

6.5.1 Descriptive statistics

The first step of any statistical analysis or modelling should be to explore the *"shape"* of the data involved, by looking at the descriptive statistics of all variables involved. The function `stat.desc` of the `pastecs` library provides three series of descriptive statistics.

- `base`:
  - `nbr.val`: overall number of values in the dataset;

  - `nbr.null`: number of `NULL` values – NULL is often returned by expressions and functions whose values are undefined;
  - `nbr.na`: number of `NA`s – missing value indicator;
- `desc`:
  - `min` (see also `min` function): **minimum** value in the dataset;
  - `max` (see also `max` function): **minimum** value in the dataset;
  - `range`: difference between `min` and `max` (different from `range()`);
  - `sum` (see also `sum` function): sum of the values in the dataset;
  - `median` (see also `median` function): **median**, that is the value separating the higher half from the lower half the values
  - `mean` (see also `mean` function): **arithmetic mean**, that is `sum` over the number of values not `NA`;
  - `SE.mean`: **standard error of the mean** – estimation of the variability of the mean calculated on different samples of the data (see also *central limit theorem*);
  - `CI.mean.0.95`: **95% confidence interval of the mean** – indicates that there is a 95% probability that the actual mean is within that distance from the sample mean;
  - `var`: **variance** (σ^2), it quantifies the amount of variation as the average of squared distances from the mean;
  - `std.dev`: **standard deviation** (σ), it quantifies the amount of variation as the square root of the variance;
  - `coef.var`: **variation coefficient** it quantifies the amount of variation as the standard deviation divided by the mean;
- `norm` (default is `FALSE`, use `norm = TRUE` to include it in the output):
  - `skewness`: **skewness** value indicates
    - positive: the distribution is skewed towards the left;
    - negative: the distribution is skewed towards the right;
  - `kurtosis`: **kurtosis** value indicates:
    - positive: heavy-tailed distribution;
    - negative: flat distribution;
  - `skew.2SE` and `kurt.2SE`: skewness and kurtosis divided by 2 standard errors. If greater than 1, the respective statistics is significant ($p < .05$);
  - `normtest.W`: test statistics for the **Shapiro–Wilk test** for normality;
  - `normtest.p`: significance for the **Shapiro–Wilk test** for normality.

The Shapiro–Wilk test compares the distribution of a variable with a normal distribution having the same mean and standard deviation. The null hypothesis of the Shapiro–Wilk test is that the sample is normally distributed, thus if `normtest.p` is lower than `0.01` (i.e., $p < .01$), the test indicates that the distribution is most probably not normal. The threshold to accept or reject a hypothesis is arbitrary and based on conventions, where $p < .01$ is the most commonly accepted threshold, or $p < .05$ for relatively small data sample (e.g., 30 cases).

The next step is thus to apply the `stat.desc` to the variable we are currently exploring (i.e., `perc_age_20_to_24`), including the `norm` section.

```
leic_2011OAC_20to24_stat_desc <- leic_2011OAC_20to24 %>%
  select(perc_age_20_to_24) %>%
  stat.desc(norm = TRUE)

leic_2011OAC_20to24_stat_desc %>%
  kable(digits = 3)
```

	perc_age_20_to_24
nbr.val	969.000
nbr.null	0.000
nbr.na	0.000
min	1.064
max	60.751
range	59.687
sum	10238.502
median	7.514
mean	10.566
SE.mean	0.304
CI.mean.0.95	0.596
var	89.386
std.dev	9.454
coef.var	0.895
skewness	2.710
skew.2SE	17.249
kurtosis	7.707
kurt.2SE	24.549
normtest.W	0.645
normtest.p	0.000

The table above tells us that all 969 OA in Leicester have a valid value for the variable `perc_age_20_to_24`, as no `NULL` nor `NA` value have been found. The values vary from about 1% to almost 61%, with an average value of 11% of the population in an OA aged between 20 and 24.

The short paragraph above is reporting on the values on the table, taking advantage of two features of RMarkdown. First, the output of the `stat.desc` function in the snippet further above is stored in the variable `leic_2011OAC_20to24_stat_desc`, which is then a valid variable for the rest of the document. Second, RMarkdown allows for in-line R snippets, that can also refer to variables defined in any snippet above the text. As such, the source of the paragraph above reads as below, with the in-line R snipped opened by a single grave accent (i.e., `` ` ``) followed by a lowercase `r` and closed by another single grave accent.

Having included all the code above into an RMarkdown document, copy the text below verbatim into the same RMarkdown document and make sure that you understand how the code in the in-line R snippets works.

```
The table above tells us that all `r leic_2011OAC_20to24_stat_desc["nbr.val",
"perc_age_20_to_24"] %>% round(digits = 0)` OA in Leicester have a valid
value for the variable `perc_age_20_to_24`, as no `NULL` nor `NA` value have
been found.The values vary from about `r leic_2011OAC_20to24_stat_desc["min",
"perc_age_20_to_24"] %>% round(digits = 0)`% to almost `r
leic_2011OAC_20to24_stat_desc["max", "perc_age_20_to_24"] %>% round(digits =
0)`%, with an average value of `r leic_2011OAC_20to24_stat_desc["mean",
"perc_age_20_to_24"] %>% round(digits = 0)`% of the population in an OA aged
between 20 and 24.
```

If the data described by statistics presented in the table above was a random sample of a population, the 95% confidence interval `CI.mean.0.95` would indicate that we can be 95% confident that the actual mean of the distribution is somewhere between 10.566 - 0.596 = 9.97% and 10.566 + 0.596 = 11.162%.

However, this is not a sample. Thus the statistical interpretation is not valid, in the same way that the `sum` values doesn't make sense, as it is the sum of a series of percentages.

Both `skew.2SE` and `kurt.2SE` are greater than `1`, which indicate that the `skewness` and `kurtosis` values are significant ($p < .05$). The `skewness` is positive, which indicates that the distribution is skewed towards the left (low values). The `kurtosis` is positive, which indicates that the distribution is heavy-tailed.

As such, `perc_age_20_to_24` having a heavy-tailed distribution skewed towards low values, it is not surprising that the `normtest.p` value indicates that the Shapiro–Wilk test is significant, which indicates that the distribution is not normal.

The code below present the output of the `shapiro.test` function, which only present the outcome of a Shapiro–Wilk test on the values provided as input. The output values are the same as the values reported by the `norm` section of `stat.desc`. Note that the `shapiro.test` function require the argument to be a numeric vector. Thus the `pull` function must be used to extract the `perc_age_20_to_24` column from `leic_2011OAC_20to24` as a vector, whereas using `select` with a single column name as the argument would produce as output a table with a single column.

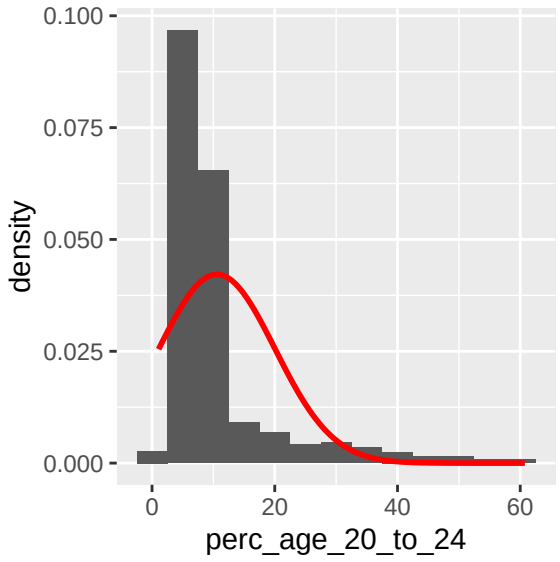
```
leic_2011OAC_20to24 %>%
  pull(perc_age_20_to_24) %>%
  shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
```

```
## data:  .
## W = 0.64491, p-value < 2.2e-16
```

The two code snippets below can be used to visualise a density-based histogram including the shape of a normal distribution having the same mean and standard deviation, and a Q-Q plot, to visually confirm the fact that `perc_age_20_to_24` is not normally distributed.

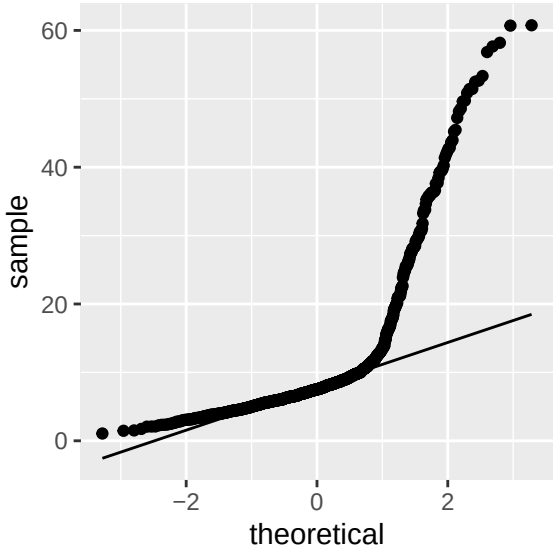
```
leic_2011OAC_20to24 %>%
  ggplot(
    aes(
      x = perc_age_20_to_24
    )
  ) +
  geom_histogram(
    aes(
      y =..density..
    ),
    binwidth = 5
  ) +
  stat_function(
    fun = dnorm,
    args = list(
      mean = leic_2011OAC_20to24 %>% pull(perc_age_20_to_24) %>% mean(),
      sd = leic_2011OAC_20to24 %>% pull(perc_age_20_to_24) %>% sd()
    ),
    colour = "red", size = 1
  )
```



A Q-Q plot in R can be created using a variety of functions. In the example below, the plot is created using the `stat_qq` and `stat_qq_line` functions of the `ggplot2` library. Note that the `perc_age_20_to_24` variable is mapped to a particular option of `aes` that is `sample`.

If `perc_age_20_to_24` had been normally distributed, the dots in the Q-Q plot would be distributed straight on the line included in the plot.

```
leic_2011OAC_20to24 %>%
  ggplot(
    aes(
      sample = perc_age_20_to_24
    )
  ) +
  stat_qq() +
  stat_qq_line()
```



6.6 Exercise 7.2

Create a new RMarkdown document, and add the code necessary to recreate the table `leic_2011OAC_20to24` used in the example above. Use the code below to re-shape the table `leic_2011OAC_20to24` by spreading the `perc_age_20_to_24` column to multiple columns using `supgrpname` as key.

```
leic_2011OAC_20to24_supgrp <- leic_2011OAC_20to24 %>%
  spread(
    key = supgrpname,
    value = perc_age_20_to_24
```

```
  )
```

That manipulation creates one column per supergroup, containing the `perc_age_20_to_24` if the OA is part of that supergroup, or an `NA` value if the OA is not part of the supergroup. The transformation is illustrated in the two tables below. The first shows an extract from the original `leic_2011OAC_20to24` dataset, followed by the wide version `leic_2011OAC_20to24_supgrp`.

```
leic_2011OAC_20to24 %>%
  slice_min(OA11CD, n = 10) %>%
  kable(digits = 3)
```

OA11CD	supgrpname	perc_age_20_to_24
E00068657	HP	6.053
E00068658	MM	6.964
E00068659	MM	8.383
E00068660	MM	4.643
E00068661	MM	10.625
E00068662	MM	8.284
E00068663	MM	8.357
E00068664	MM	3.597
E00068665	MM	7.068
E00068666	MM	5.864

```
leic_2011OAC_20to24_supgrp %>%
  slice_min(OA11CD, n = 10) %>%
  kable(digits = 3)
```

OA11CD	CD	CP	EC	HP	MM	SU	UR
E00068657	NA	NA	NA	6.053	NA	NA	NA
E00068658	NA	NA	NA	NA	6.964	NA	NA
E00068659	NA	NA	NA	NA	8.383	NA	NA
E00068660	NA	NA	NA	NA	4.643	NA	NA
E00068661	NA	NA	NA	NA	10.625	NA	NA
E00068662	NA	NA	NA	NA	8.284	NA	NA
E00068663	NA	NA	NA	NA	8.357	NA	NA
E00068664	NA	NA	NA	NA	3.597	NA	NA
E00068665	NA	NA	NA	NA	7.068	NA	NA
E00068666	NA	NA	NA	NA	5.864	NA	NA

Question 7.2.1: The code below uses the newly created `leic_2011OAC_20to24_supgrp` table to calculate the descriptive statistics calculated for the variable `leic_2011OAC_20to24` for each supergroup. Is `leic_2011OAC_20to24` normally distributed in any of the subgroups? If yes, which supergroups and based on which values do you justify that claim? (Write up to 200 words)

```
leic_2011OAC_20to24_supgrp %>%
  select(-OA11CD) %>%
  stat.desc(norm = TRUE) %>%
  kable(digits = 3)
```

	CD	CP	EC	HP	MM	SU	UR
nbr.val	36.000	83.000	57.000	101.000	573.000	54.000	65.000
nbr.null	0.000	0.000	0.000	0.000	0.000	0.000	0.000
nbr.na	933.000	886.000	912.000	868.000	396.000	915.000	904.000
min	1.064	3.141	2.066	1.515	2.490	1.462	2.256
max	12.963	60.751	36.299	11.261	52.507	9.562	13.505
range	11.899	57.609	34.233	9.746	50.018	8.100	11.249
sum	252.108	2646.551	838.415	619.266	5214.286	295.867	372.010
median	6.854	30.457	10.881	6.053	7.880	5.476	5.380
mean	7.003	31.886	14.709	6.131	9.100	5.479	5.723
SE.mean	0.471	1.574	1.373	0.172	0.230	0.233	0.264
CI.mean.0.95	0.956	3.131	2.751	0.341	0.452	0.467	0.528
var	7.983	205.556	107.523	2.980	30.285	2.929	4.545
std.dev	2.825	14.337	10.369	1.726	5.503	1.712	2.132
coef.var	0.403	0.450	0.705	0.282	0.605	0.312	0.372
skewness	0.322	0.067	0.633	0.124	3.320	0.005	1.042
skew.2SE	0.410	0.127	1.001	0.258	16.266	0.008	1.753
kurtosis	-0.142	-0.825	-1.009	0.220	15.143	-0.391	1.441
kurt.2SE	-0.093	-0.789	-0.810	0.231	37.156	-0.306	1.229
normtest.W	0.965	0.980	0.889	0.993	0.684	0.991	0.937
normtest.p	0.310	0.239	0.000	0.886	0.000	0.954	0.002

Question 7.2.2: Write the code necessary to test again the normality of `leic_2011OAC_20to24` for the supergroups where the analysis conducted for question 7.2.1 indicated they are normal, using the function `shapiro.test`, and draw the respective Q-Q plot.

Question 7.2.3: Observe the output of the Levene's test executed below. What does the result tell you about the variance of `perc_age_20_to_24` in supergroups?

```
leveneTest(leic_2011OAC_20to24$perc_age_20_to_24, leic_2011OAC_20to24$supgrpname)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value    Pr(>F)
## group   6  62.011 < 2.2e-16 ***
##       962
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Chapter 7

Comparing data

Stefano De Sabbata

To-do

Chapter 8

Regression analysis

Stefano De Sabbata

8.1 Introduction

The first part of this practical guides you through the ANOVA (analysis of variance) and regression analysis seen in the lecture, the last part showcases a multiple regression analysis. Create a new R project for this practical session and create a new RMarkdown document to replicate the analysis in this document and a separate RMarkdown document to work on the exercises.

```
library(tidyverse)
library(magrittr)
library(knitr)
```

Many of the functions used in the analyses below are part of the oldest libraries developed for R, they have not been developed to be easily compatible with the Tidyverse and the `%>%` operator. Fortunately, the `magrittr` library (loaded above) does not only define the `%>%` operator seen so far, but also the exposition pipe operator `%$%`, which exposes the columns of the data.frame on the left of the operator to the expression on the right of the operator. That is, `%$%` allows to refer to the column of the data.frame directly in the subsequent expression. As such, the lines below expose the column `Petal.Length` of the data.frame `iris` and to pass it on to the `mean` function using different approaches, but they are all equivalent in their outcome.

```
mean(iris$Petal.Length) # Classic R approach
```

```
## [1] 3.758
```

```
iris$Petal.Length %>% mean() # Using %>% pipe
```

```
## [1] 3.758
```

```
iris %$% Petal.Length %>% mean() # Using %>% pipe and %$% exposition pipe
```

```
## [1] 3.758
```

8.2 ANOVA

The ANOVA (analysis of variance) tests whether the values of a variable (e.g., length of the petal) are on average different for different groups (e.g., different species of iris). ANOVA has been developed as a generalised version of the t-test, which has the same objective but allows to test only two groups.

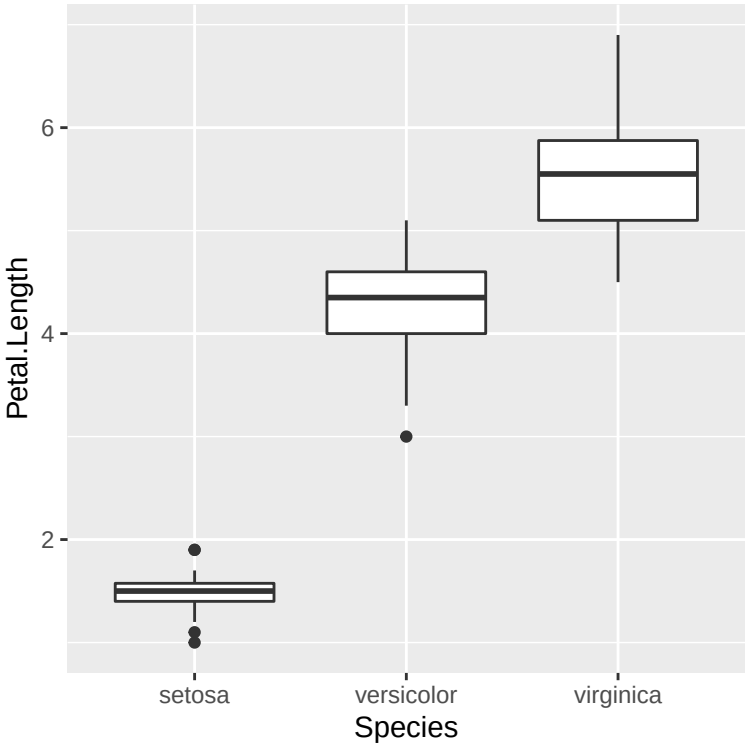
The ANOVA test has the following assumptions:

- normally distributed values in groups
  - especially if groups have different sizes
- homogeneity of variance of values in groups
  - if groups have different sizes
- independence of groups

8.2.1 Example

The example seen in the lecture illustrates how ANOVA can be used to verify that the three different species of iris in the `iris` dataset have different petal length.

```
iris %>%
  ggplot(
    aes(
      x = Species,
      y = Petal.Length
    )
  ) +
  geom_boxplot()
```



ANOVA is considered a robust test, thus, as the groups are of the same size, there is no need to test for the homogeneity of variance. Furthermore, the groups come from different species of flowers, so there is no need to test the independence of the values. The only assumption that needs testing is whether the values in the three groups are normally distributed. The three Shapiro–Wilk tests below are all not significant, which indicates that all three groups have normally distributed values.

```
iris %>% filter(Species == "setosa") %>% pull(Petal.Length) %>% shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.95498, p-value = 0.05481
```

```
iris %>% filter(Species == "versicolor") %>% pull(Petal.Length) %>% shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.966, p-value = 0.1585
```

```
iris %>% filter(Species == "virginica") %>% pull(Petal.Length) %>% shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.96219, p-value = 0.1098
```

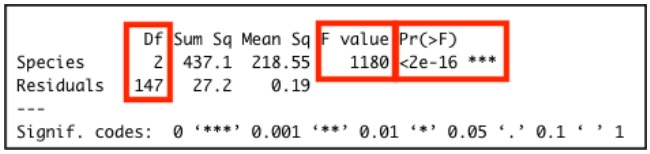
We can thus conduct the ANOVA test using the function `aov`, and the function `summary` to obtain the summary of the results of the test.

```
# Classic R coding approach (not using %$%)
# iris_anova <- aov(Petal.Length ~ Species, data = iris)
# summary(iris_anova)

iris %$%
  aov(Petal.Length ~ Species) %>%
  summary()
```

```
##              Df Sum Sq Mean Sq F value Pr(>F)
## Species       2  437.1  218.55    1180 <2e-16 ***
## Residuals   147   27.2    0.19
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The difference is significant $\mathrm{F}(2, 147) = 1180.16$, $p < .01$.

The image below highlights the important values in the output: the significance value `Pr(>F)`; the F-statistic value `F value`; and the two degrees of freedom values for the F-statistic in the `Df` column.



```
            Df Sum Sq Mean Sq F value Pr(>F)
Species      2  437.1  218.55    1180 <2e-16 ***
Residuals  147   27.2    0.19
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

8.3 Simple regression

The simple regression analysis is a supervised machine learning approach to creating a model able to predict the value of one outcome variable Y based on one predictor variable X_1, by estimating the intercept b_0 and coefficient (slope) b_1, and accounting for a reasonable amount of error ϵ.

$$Y_i = (b_0 + b_1 * X_{i1}) + \epsilon_i$$

Least squares is the most commonly used approach to generate a regression model. This model fits a line to minimise the squared values of the **residuals** (errors), which are calculated as the squared difference between observed values the values predicted by the model.

$$residual = \sum (observed - model)^2$$

A model is considered **robust** if the residuals do not show particular trends, which would indicate that *"something"* is interfering with the model. In particular, the assumption of the regression model are:

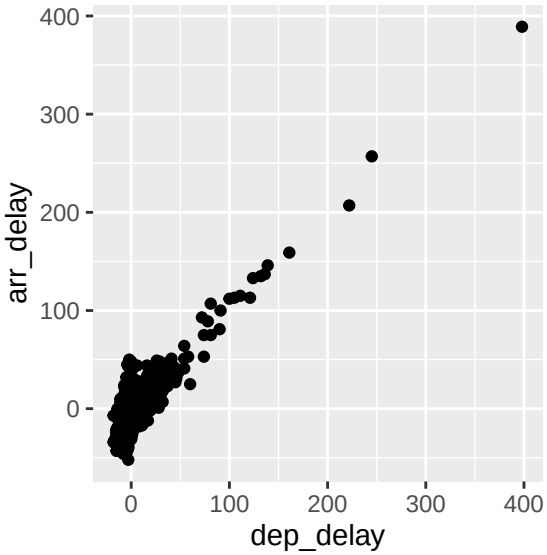
- **linearity:** the relationship is actually linear;
- **normality** of residuals: standard residuals are normally distributed with mean `0`;
- **homoscedasticity** of residuals: at each level of the predictor variable(s) the variance of the standard residuals should be the same (*homo-scedasticity*) rather than different (*hetero-scedasticity*);
- **independence** of residuals: adjacent standard residuals are not correlated.

8.3.1 Example

The example that we have seen in the lecture illustrated how simple regression can be used to create a model to predict the arrival delay based on the departure delay of a flight, based on the data available in the `nycflights13` dataset for the flight on November 20th, 2013. The scatterplot below seems to indicate that the relationship is indeed linear.

$$arr\_delay_i = (Intercept + Coefficient_{dep\_delay} * dep\_delay_{i1}) + \epsilon_i$$

```
# Load the library
library(nycflights13)

# November 20th, 2013
flights_nov_20 <- nycflights13::flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay), month == 11, day ==20)
```



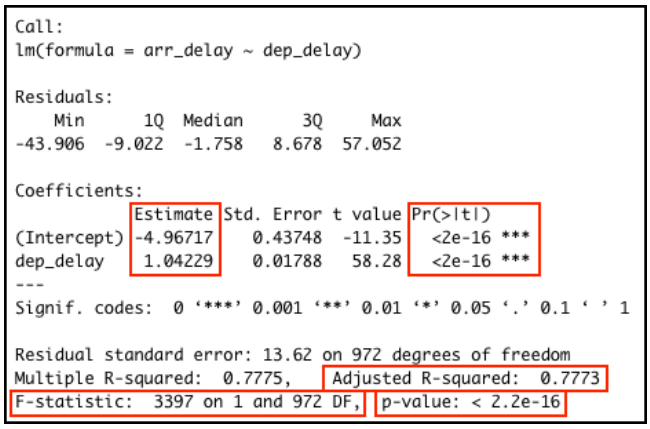
The code below generates the model using the function `lm`, and the function `summary` to obtain the summary of the results of the test. The model and summary are saved in the variables `delay_model` and `delay_model_summary`, respectively, for further use below. The variable `delay_model_summary` can then be called directly to visualise the result of the test.

```
# Classic R coding version
# delay_model <- lm(arr_delay ~ dep_delay, data = flights_nov_20)
# delay_model_summary <- summary(delay_model)

delay_model <- flights_nov_20 %$%
  lm(arr_delay ~ dep_delay)

delay_model_summary <- delay_model %>%
  summary()

delay_model_summary
```

```
##
```

```
## Call:
## lm(formula = arr_delay ~ dep_delay)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -43.906  -9.022  -1.758   8.678  57.052
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.96717    0.43748  -11.35   <2e-16 ***
## dep_delay    1.04229    0.01788   58.28   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.62 on 972 degrees of freedom
## Multiple R-squared:  0.7775,Adjusted R-squared:  0.7773
## F-statistic:  3397 on 1 and 972 DF,  p-value: < 2.2e-16
```

The image below highlights the important values in the output: the adjusted R^2 value; the model significance value `p-value` and the related F-statistic information `F-statistic`; the intercept and `dep_delay` coefficient estimates in the `Estimate` column and the related significance values of in the column `Pr(>|t|)`.

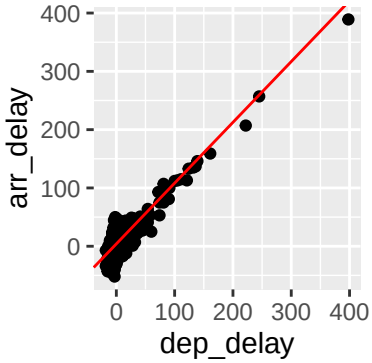


The output indicates:

- **p-value: < 2.2e-16**: $p < .001$ the model is significant;
  - derived by comparing the calulated **F-statistic** value to F distribution 3396.74 having specified degrees of freedom (1, 972);
  - Report as: $F(1, 972) = 3396.74$
- **Adjusted R-squared: 0.7773**: the departure delay can account for 77.73% of the arrival delay;
- **Coefficients**:
  - Intercept estimate -4.9672 is significant;

– `dep_delay` coefficient (slope) estimate 1.0423 is significant.

```
flights_nov_20 %>%
  ggplot(aes(x = dep_delay, y = arr_delay)) +
  geom_point() + coord_fixed(ratio = 1) +
  geom_abline(intercept = 4.0943, slope = 1.04229, color="red")
```



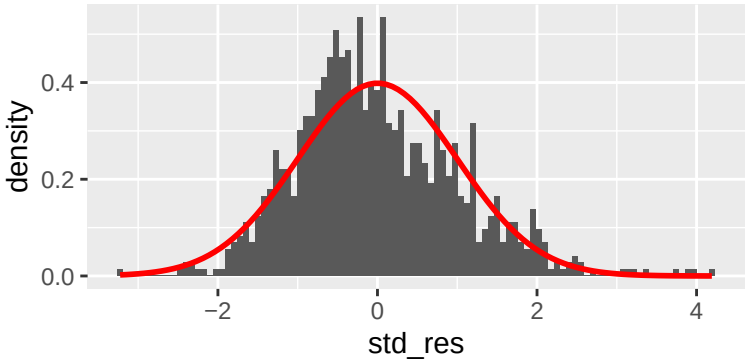
8.3.2 Checking assumptions

8.3.2.1 Normality

The Shapiro-Wilk test can be used to check for the normality of standard residuals. The test should be not significant for robust models. In the example below, the standard residuals are *not* normally distributed. However, the plot further below does show that the distribution of the residuals is not far away from a normal distribution.

```
delay_model %>%
  rstandard() %>%
  shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.98231, p-value = 1.73e-09
```



8.3.2.2 Homoscedasticity

The Breusch-Pagan test can be used to check for the homoscedasticity of standard residuals. The test should be not significant for robust models. In the example below, the standard residuals are homoscedastic.

```
library(lmtest)

delay_model %>%
  bptest()
```

```
##
##  studentized Breusch-Pagan test
##
## data:  .
## BP = 0.017316, df = 1, p-value = 0.8953
```

8.3.2.3 Independence

The Durbin-Watson test can be used to check for the independence of residuals. The test should be statistic should be close to 2 (between 1 and 3) and not significant for robust models. In the example below, the standard residuals might not be completely independent. Note, however, that the result depends on the order of the data.

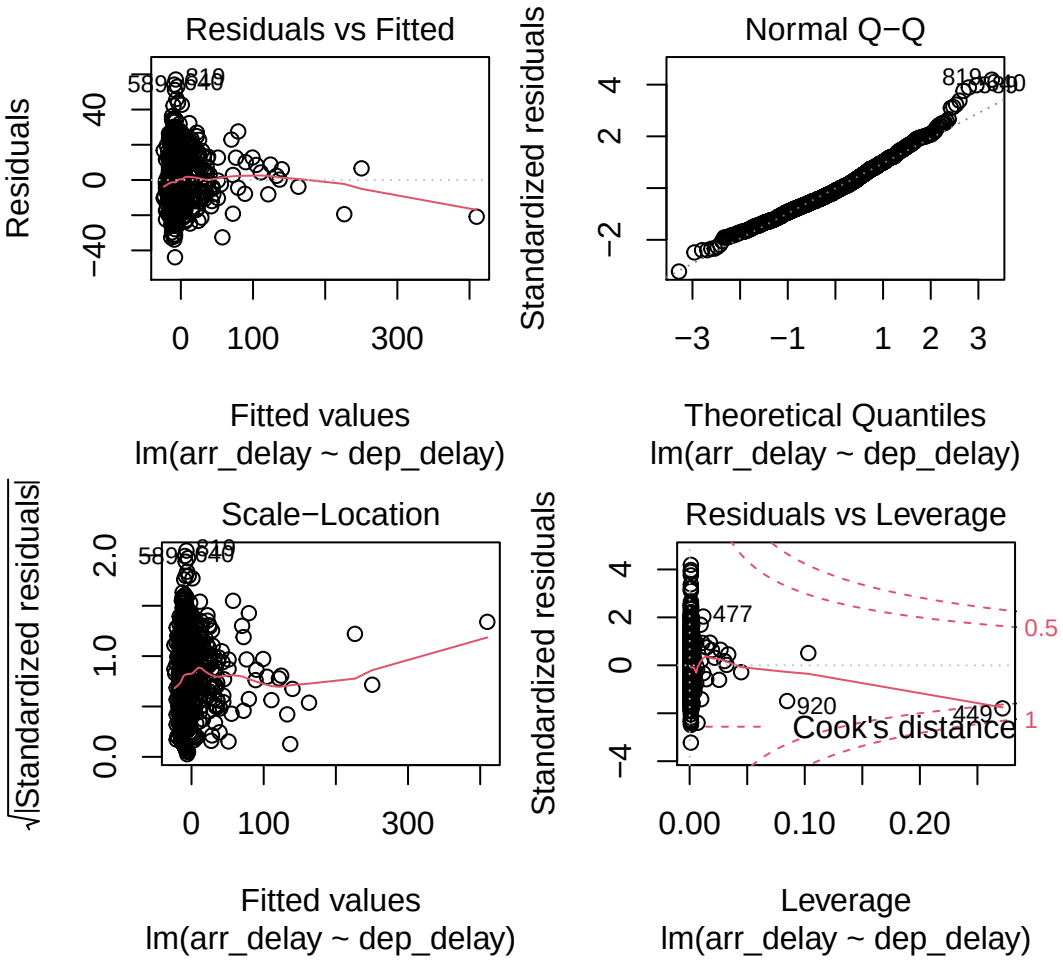
```
# Also part of the library lmtest
delay_model %>%
  dwtest()
```

```
##
##  Durbin-Watson test
##
## data:  .
## DW = 1.8731, p-value = 0.02358
## alternative hypothesis: true autocorrelation is greater than 0
```

8.3.2.4 Plots

The `plot.lm` function can be used to further explore the residuals visuallly. Usage is illustrated below. The *Residuals vs Fitted* and *Scale-Location* plot provide an insight into the homoscedasticity of the residuals, the *Normal Q-Q* plot provides an illustration of the normality of the residuals, and the *Residuals vs Leverage* can be useful to identify exceptional cases (e.g., Cook's distance greater than 1).

```
delay_model %>%
  plot()
```



8.3.3 How to report

Overall, we can say that the delay model computed above is fit ($F(1, 972) = 3396.74$, $p < .001$), indicating that the departure delay might account for 77.73%

of the arrival delay. However the model is only partially robust. The residuals satisfy the homoscedasticity assumption (Breusch-Pagan test, $BP = 0.02$, $p = 0.9$), and the independence assumption (Durbin-Watson test, $DW = 1.87$, $p = 0.02$), but they are not normally distributed (Shapiro-Wilk test, $W = 0.98$, $p < .001$).

The `stargazer` function of the `stargazer` library can be applied to the model `delay_model` to generate a nicer output in RMarkdown PDF documents by including `results = "asis"` in the R snippet option.

```
# Install stargazer if not yet installed
# install.packages("stargazer")

library(stargazer)

# Not rendered in bookdown
stargazer(delay_model)
```

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu % Date and time: Tue, Oct 20, 2020 - 11:24:00 AM

Table 8.1:

	Dependent variable:
	arr_delay
dep_delay	1.042***
	(0.018)
Constant	−4.967***
	(0.437)
Observations	974
R^2	0.778
Adjusted R^2	0.777
Residual Std. Error	13.618 (df = 972)
F Statistic	3,396.742*** (df = 1; 972)
Note:	*p<0.1; **p<0.05; ***p<0.01

8.4 Multiple regression

The multiple regression analysis is a supervised machine learning approach to creating a model able to predict the value of one outcome variable Y based on two or more predictor variables $X_1 \ldots X_M$, by estimating the intercept b_0

and the coefficients (slopes) $b_1 ... b_M$, and accounting for a reasonable amount of error ϵ.

$$Y_i = (b_0 + b_1 * X_{i1} + b_2 * X_{i2} + \cdots + b_M * X_{iM}) + \epsilon_i$$

The assumptions are the same as the simple regression, plus the assumption of **no multicollinearity**: if two or more predictor variables are used in the model, each pair of variables not correlated. This assumption can be tested by checking the variance inflation factor (VIF). If the largest VIF value is greater than 10 or the average VIF is substantially greater than 1, there might be an issue of multicollinearity.

8.4.1 Example

The example below explores whether a regression model can be created to estimate the number of people in Leicester commuting to work using public transport (`u120`) in Leicester, using the number of people in different occupations as predictors.

For instance, occupations such as skilled traders usually require to travel some distances with equipment, thus the related variable `u163` is not included in the model, whereas professional and administrative occupations might be more likely to use public transportation to commute to work.

A multiple regression model can be specified in a similar way as a simple regression model, using the same `lm` function, but adding the additional predictor variables using a `+` operator.

```
leicester_2011OAC <- read_csv("2011_OAC_Raw_uVariables_Leicester.csv")
```

```
# u120: Method of Travel to Work, Public Transport
# u159: Employment, Managers, directors and senior officials
# u160: Employment, Professional occupations
# u161: Employment, Associate professional and technical occupations
# u162: Employment, Administrative and secretarial occupations
# u163: Employment, Skilled trades occupations
# u164: Employment, Caring, leisure and other service occupations
# u165: Employment, Sales and customer service occupations
# u166: Employment, Process, plant and machine operatives
# u167: Employment, Elementary occupations
public_transp_model <- leicester_2011OAC %$%
  lm(u120 ~ u160 + u162 + u164 + u165 + u167)

public_transp_model %>%
  summary()
```

```
##
```

```
## Call:
## lm(formula = u120 ~ u160 + u162 + u164 + u165 + u167)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -16.8606  -4.0247  -0.1084   3.7912  24.6359
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.19593    0.75048   4.258 2.26e-05 ***
## u160         0.06912    0.01416   4.881 1.24e-06 ***
## u162         0.17000    0.03328   5.108 3.93e-07 ***
## u164         0.28641    0.03589   7.979 4.17e-15 ***
## u165         0.21311    0.03107   6.858 1.25e-11 ***
## u167         0.32008    0.02156  14.846  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.977 on 963 degrees of freedom
## Multiple R-squared:  0.436,Adjusted R-squared:  0.4331
## F-statistic: 148.9 on 5 and 963 DF,  p-value: < 2.2e-16
```

```
# Not rendered in bookdown
stargazer(public_transp_model)
```

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu % Date and time: Tue, Oct 20, 2020 - 11:24:00 AM

```
public_transp_model %>%
  rstandard() %>%
  shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.9969, p-value = 0.05628
```

```
public_transp_model %>%
  bptest()
```

```
##
##  studentized Breusch-Pagan test
##
## data:  .
## BP = 45.986, df = 5, p-value = 9.142e-09
```

Table 8.2:

	Dependent variable:
	u120
u160	0.069***
	(0.014)
u162	0.170***
	(0.033)
u164	0.286***
	(0.036)
u165	0.213***
	(0.031)
u167	0.320***
	(0.022)
Constant	3.196***
	(0.750)
Observations	969
R^2	0.436
Adjusted R^2	0.433
Residual Std. Error	5.977 (df = 963)
F Statistic	148.884*** (df = 5; 963)
Note:	*p<0.1; **p<0.05; ***p<0.01

```
public_transp_model %>%
  dwtest()
```

```
##
##  Durbin-Watson test
##
## data:  .
## DW = 1.8463, p-value = 0.007967
## alternative hypothesis: true autocorrelation is greater than 0
```

```
library(car)

public_transp_model %>%
  vif()
```

```
##     u160     u162     u164     u165     u167
## 1.405480 1.486768 1.163760 1.353682 1.428418
```

The output above suggests that the model is fit ($F(5, 963) = 148.88$, $p < .001$), indicating that a model based on the number of people working in the five selected occupations can account for 43.31% of the number of people using public transportation to commute to work. However the model is only partially robust. The residuals are normally distributed (Shapiro-Wilk test, $W = 1$, $p = 0.06$) and there seems to be no multicollinearity with average VIF 1.37, but the residuals don't satisfy the homoscedasticity assumption (Breusch-Pagan test, $BP = 45.99$, $p < .001$), nor the independence assumption (Durbin-Watson test, $DW = 1.85$, $p < .01$).

The coefficient values calculated by the `lm` functions are important to create the model, and provide useful information. For instance, the coefficient for the variable `u165` is `0.21`, which indicates that if the number of people employed in sales and customer service occupations increases by one unit, the number of people using public transportation to commute to work increases by `0.21` units, according to the model. The coefficients also indicate that the number of people in elementary occupations has the biggest impact (in the context of the variables selected for the model) on the number of people using public transportation to commute to wor0, whereas the number of people in professional occupations has the lowest impact.

In this example, all variables use the same unit and are of a similar type, which makes interpretating the model relatively simple. When that is not the case, it can be useful to look at the standardized β, which provide the same information but measured in terms of standard deviation, which make comparisons between variables of different types easier to draw. For instance, the values calculated below using the function `lm.beta` of the library `QuantPsyc` indicate that if the number of people employed in sales and customer service occupations increases by one standard deviation, the number of people using public transportation to commute to work increases by `0.19` standard deviations, according to the

model.

```
# Install lm.beta library if necessary
# install.packages("lm.beta")
library(lm.beta)

lm.beta(public_transp_model)
```

```
##
## Call:
## lm(formula = u120 ~ u160 + u162 + u164 + u165 + u167)
##
## Standardized Coefficients::
## (Intercept)        u160        u162        u164        u165        u167
##   0.0000000   0.1400270   0.1507236   0.2083107   0.1931035   0.4293988
```

8.5 Exercise 9.1

Question 9.1.1: Is mean age (u020) different in different 2011OAC supergroups in Leicester?

Question 9.1.2: Is the number of people using public transportation to commute to work statistically, linearly related to mean age (u020)?

Question 9.1.3: Is the number of people using public transportation to commute to work statistically, linearly related to (a subset of) the age structure categories (u007 to u019)?

Chapter 9

Clustering

Stefano De Sabbata

9.1 Introduction

Chapter 10

Support vector machines

Stefano De Sabbata

To-do.