Faculty of Engineering

Data Science and Engineering

Semester 1 2023–2024

Jose Alberto ESPINO PITTI, *UID:3035946813*

2023

Course Code

COMP

2501

Class Contents

Introduction

Check out Credits

These notes contain information from several sources, but mainly from the lecture notes from the class COMP2501A at the University of Hong Kong and the book *Introduction to Data Science* by Rafael A. Irizarry. I claim no autorship over any of the contents herein. Feel free to contact me at *jespigno (at) connect (dot) hku (dot) hk* if you have any questions or concerns.

This course will use the **R** language, which is a programming language and free software environment for statistical analysis and data visualisation. This language, as opposed to other mainstream ones, was created by statisticians for the express purpose of data analysis. This means that it is uniquely dynamic and powerful for this purpose.
The "launching pad" for our data science projects is *RStudio*, which is an integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management. In RStudio, you have access to a console and a script editor. You can either type your own command directly into the console, run the entire script (`CTRL+SHIFT+ENTER`), or just run the current line in the script (`CTRL+ENTER`). The console is a powerful component; it is what allows us to perform interactive data analysis. Similarly, there is a tab in the program named *environment*, which shows the variables that are currently stored in the memory. You can also see the history of commands that you have typed in the console. Lastly, there is a tab named *files*, which shows the files in the current working directory. Two important packages we will need to install are *tidyverse* and *dslabs* You can install them by typing `install.packages(c("tidyverse", "dslabs"))` in the console.
Data science is the centre of this class. It is the process of extracting knowledge from data, which more often than not is messy and unstructured. R will be the tool that allows us to find patterns in data.

Introduction to R

In R, an **object** is a variable that contains data. There are different types of objects, such as vectors, matrices, data frames, etc. We can create an object by using the assignment operator `<-`. For example, `x <- 5` creates an object named `x` and assigns it the value `5`. R makes a distinction between `<-` and `=`. the former is used for assignment, while the latter is used for function arguments. For example,

```
x <- 888
log(x, base = 50)
# This is the same as log(x, 50)
# but log(x, base <- 50) would be an error
```

By default, R has a lot of functions that require libraries in other languages, such as `sqrt()` and `log()`. An important function that you need to keep in mind is `help("function_name")` or `?function_name`, which displays information about the function, such as what inputs it expects or what it is computing. The function `rm(variable_name)` allows us to delete an object.
R has many, many datasets included by default on your installation of the language. The way you can access them is by the command `data()`. For example, `data("Titanic")` loads the dataset `Titanic` into the memory. You can then see the contents of the dataset by typing `View(Titanic)`.
The syntax for **if–else statements** is similar to that of languages like C++; you use the keyword `if` followed by the condition in parentheses, and then you use curly braces to indicate the code that will be executed if the condition is true. You can also use the keyword `else` to indicate the code that will be executed if the condition is false. For example,

```
a <- 888
if (x > 0) {
    print("x is positive")
} else {
    print("x is negative")
}
```

R also has a predefined function called `ifelse()`, which is a vectorised version of the `if-else` statement. For example,

```
x <- c(1, 2, 3, 4, 5)
ifelse(x > 3, "big", "small")
#OUTPUT
#[1] "small" "small" "small" "big" "big"
```

As you can see above, the first parameter of the function is the condition, the second the return value if the condition is met, and the last the return value if the condition is not met. Other two predefined functions for control are `any()` and `all()`. The former returns `TRUE` if any of the elements of the vector are `TRUE`, while the latter returns `TRUE` if all of the elements of the vector are `TRUE`. For example,

```
x <- c(TRUE, FALSE, TRUE)
any(x)
#OUTPUT
#[1] TRUE
all(x)
#OUTPUT
#[1] FALSE
```

For–loops can be created by using the sintax `for (variable in vector) {code}`. For example,

```
for (i in 1:10) {
    print(i)
}
```

```
#OUTPUT
#[1] 1
#[1] 2
#[1] 3
#until ten (inclusive)
```

For–loops are not as common in R because by default, all predefined functions will also work on vectors.

1.0.1 Data Types

At any moment, you can find out the data type of a given variable using the command `class()`. The most common data type for R is a **data frame**. Data frames are conceptually a table with rows and column, where each row represents an observation and each column represents a variable. You can analyse the contents of a data frame by using the command `head()`, which shows the first 6 rows of the data frame. Similarly, the command `tail()` shows the last 6 rows of the data frame. You can also use the command `str()`, which shows the structure of the data frame.
If you want to **access** a specific element of a data frame, you can use the command `df[row, column]`. For example, `Titanic[1, 2]` returns the element in the first row and second column of the data frame `Titanic`. If you want to access an entire row or column, you can use the command `df[row, ]` or `df[, column]`. For example, `Titanic[1, ]` returns the first row of the data frame `Titanic`. Similarly, you can access a specific column by using the $ symbol followed by the name of the column. For example, *Titanic$Survived* returns the column `Survived` of the data frame `Titanic`. The way you find out what the name of each column is is by using the command `names()`. For example, `names(Titanic)` returns the names of the columns of the data frame `Titanic`.
Every column and row in a data frame is a **vector**, a series of values of the same type. You can create a vector by using the command `c()`. For example, `c(1, 2, 3)` creates a vector with the values `1, 2, 3`. The command `seq()` creates a **numeric** vector containing numbers in a range. For example, `seq(1, 10)` creates a vector with the values `1, 2, 3, 4, 5, 6, 7, 8, 9, 10`. Furthermore, the command `rep()` creates a vector where every entry is the first parameter passed into the command, repeated as many times as specified in the second parameter. For example, `rep(1, 10)` creates a vector with the values `1, 1, 1, 1, 1, 1, 1, 1, 1, 1`.

Check out Beware!

When creating a vector using the command `c()`, you need to be careful with the data types of the entries. For example, `c(1, "a")` will create a character vector with the values `"1", "a"`. This is because R will coerce the numeric value `1` into a character value `"1"`. In general, when different types are passed into `c`, lower types will be coerced into higher types. This also happens in the case of `c(1, TRUE)`, where the output will be numeric vector `1, 1`.

The data type of a vector matches that of its entries. For example, `c(1, 2, 3)` is a numeric vector, while `c("a", "b", "c")` is a character vector. Most of the types are pretty self–explanatory; however, you might not have heard of the **factor** type. Usually used to store categorical data, this data type consists of certain labels that in the background are stored as integers for memory efficiency purposes. On execution, R will map these integers to the labels. You can create a factor by using the command `factor()`. For example, `factor(c("Hong Kong", "Panama", "Germany"))` creates a factor with the labels `Hong Kong, Panama, Germany`.
You can create a data frame by using the command `data.frame()`.

```
signature_dish <- c("Butter Chicken", "Beef Nihari", "Char Siu Rice", "Pad Thai",
 "Pho")
michelin_star <- c(TRUE, TRUE, TRUE, FALSE, TRUE)
```

```
location <- c("India", "Pakistan", "Hong Kong", "Thailand", "Vietnam")
rating <- c(8.7, 8.7, 8.8, 6.9, 7.7)
sample_frame <- data.frame(signature_dish,michelin_star,location,rating,
stringsAsFactors = TRUE)
sample_frame

#OUTPUT
#signature_dish michelin_star  location rating
#1 Butter Chicken          TRUE     India    8.7
#2    Beef Nihari          TRUE  Pakistan    8.7
#3  Char Siu Rice          TRUE Hong Kong    8.8
#4       Pad Thai         FALSE  Thailand    6.9
#5            Pho          TRUE   Vietnam    7.7
```

Check out Vector Operations

In R, arithmetic operations on vectors occur element–wise. For example,

```
x <- c(1, 2, 3)
x <- x * 8 + 2
print(x)
#OUTPUT
#[1] 10 18 26
```

Vector–on–vector operations also occur element–wise. This implies however, that the vectors must be of the same length.
A useful operator is `%in%`. This operator returns a boolean vector that indicates whether each element of the first vector is present in the second vector. For example,

```
x <- c(1, 2, 999)
y <- c(1, 2, 3, 4, 5)
x %in% y
#OUTPUT
#[1] TRUE TRUE FALSE
```

In a way, data frames are a special type of **list**—lists in R can contain any type of object, including other lists. You can create a list by using the command `list()`. For example, `list(1, 2, 3)` creates a list with the values `1, 2, 3`.

```
pet_info <- list(name = "Cookie".
                 type = "Cat",
                 age = 9,
                 is_cute = TRUE
                 favourite_toys = c("ball", "string", "mouse")
                 )
print(pet_info$name) #output = [1] "Cookie"
#double brackets also output the value
print(pet_info[["favourite_toys"]]) #output = [1] "ball" "string" "mouse"
print(pet_info[["favourite_toys"]][2]) #output = [1] "string"
#single brackets output a list with key and value
print(pet_info["favourite_toys"]) #output = \$favourite_toys
                                  #[1] "ball" "string" "mouse"
```

It is also possible for a list not to have any keys. When this is the case, it is not possible to index it using the $ operator, but you can use the square brackets operator and specify the index of the element you want to access. For example, `list(1, 2, 3)[2]` returns the value `2`.

Check out Indexing

As opposed to other programming languages, like Python and C++, R starts indexing at 1 instead of 0. A possible reason for this is the fact that in math, vectors are usually indexed starting at 1. Be careful!

Another common object type in R is the **matrix**. Similarly to a data frame, a matrix is a table with rows and columns. However, unlike a data frame, a matrix can only contain one type of data (since you can consider them as stacks of vectors). Matrices are defined using the `matrix()` function. The number of rows and columns need to be specified. For example, `matrix(1:6, nrow = 2, ncol = 3)` creates a matrix with the values `1, 2, 3, 4, 5, 6` and dimensions 2×3. You can access specific entries in the matrix using square brackets, where the first index is for the row and the second for the column. `sample_matrix[8,7]` returns the element in the eigth row and seventh column of matrix `sample_matrix`. If you want the entire row or column, just leave the corresponding index blank. It is possible for you to access more than one column or row at a time. This will output a new matrix. The command `sample_matrix[,2:3]` will output a new matrix with the second and third columns of `sample_matrix`. Matrices can also be converted into data frames using the function `as.data.frame()`. You can assign a **name** to each entry of a vector, functionally giving an index to each entry. This is called a **named vector**. You can create one by using the command `c("name1" = value1, "name2" = value2, ...)`. For example, `c("a" = 1, "b" = 2)` creates a named vector with the values `1, 2` and the names `"a", "b"`. You can access the values of a named vector by using the dollar sign operator. Names can also be assigned retroactively. For instance, given the vectors `c(1, 2, 3)` and `c("a", "b", "c")`, you can assign the names `"a", "b", "c"` to the first vector by using the command `names(vector) <- c("a", "b", "c")`. Vectors can be used to access elements within another vector or data frame. For instance,

```
x <- c("a", "b", "c")
y <- c(1, 2, 3)
x[y]
#OUTPUT
#"a" "b" "c"
```

Check out On the issue of coercion

Coercion is R's tendency to convert data types into higher level equivalents to avoid errors. The aforementioned `numeric --> character` coercion is an example of this. Whenever R does not have any valid guesses as to what the value of a particular entry should be, it will coerce it into a `NA` value. For example, `c(1, 2, "a")` will coerce the character value `"a"` into a `NA` value. As you code with R, you will get used to spotting `NA` as you go.

1.0.2 Commands

Some essential commands will be introduced in this section. The section is not meant to be read in one go, but rather to be used as a reference when needed.

- `seq(a:b:s)`
  Creates a vector with a sequence of numbers in the range a to b (inclusive). s determines the **step**. Unless the step is a non–integer, the data type of a sequence is integer.

- `sort(v)`
  Sorts the vector v in ascending order. If you only want the maximum/minimum value of the vector, you can use the command `max(v)` or `min(v)` respectively.

- `order(v)`
  Returns the indices of the vector that make it sorted. For example,

```
v <- c(1, 3, 2)
order(v)
#OUTPUT
#1 3 2
```

  This has useful applications when managing dataframes.

```
df <- data.frame(name = c("a", "b", "c"),
                 age = c(1, 3, 2))
index = order(df$age)
df$name[index]
#OUTPUT
#[1] "a" "c" "b"
```

  If you only want to access the index of the maximum or minimum element, you can use the command `which.max(v)` or `which.min(v)`

- `rank(v)`
  Returns the rank of each element in the vector v. For example,

```
v <- c(88, 91, 238, 19, 7)
rank(v)
#OUTPUT
#4 5 3 2 1
```

Defining your own functions is pretty straightforward: you only need to use the command `function()`. For example,

```
compute_avg <- function(x, arithmetic = TRUE) {
    ifelse(arithmetic, sum(x) / length(x), prod(x) ^ (1 / length(x)))
}
compute_avg(c(1, 2, 3))
#OUTPUT
#[1] 2
compute_avg(c(1, 2, 3), arithmetic = FALSE)
#OUTPUT
#[1] 1.817121
```

1.0.3 Plots

R has a powerful plotting system. The main way to access it is through the command `plot(x,y)`, where both `x` and `y` are vectors. For example,

```
x <- c(1, 2, 3)
y <- c(1, 4, 9)
plot(x, y)
#plot the points $(1, 1), (2, 4), (3, 9)$ inplane
```

Furthermore, you can add a title to the plot by using the command `title("title")` and labels to the x and y axes by using the commands `xlab("label")` and `ylab("label")` respectively.
Histograms can be created by using the command `hist(x)`, where `x` is a vector. **Boxplots**, which demonstrate the locality, spread, and skewness groups of numerical data can be created by using the command `boxplot(x, y)`, where `x` and `y` are vectors.

The Tidyverse

The Tidyverse is a collection of core R open source packages that share an underlying design philosophy, grammar, and data structures. Some of the features all the packages share are non–standard evaluation and piping. If you want to use it in your program, you only need to add the command `library(tidyverse)`. The core of the Tidyverse is **tidy** data. A data table is in tidy format if every row is an observation and every column is a feature. For example (our running example in this section),

```
signature_dish <- c("Butter Chicken", "Beef Nihari", "Char Siu Rice", "Pad Thai",
 "Pho")
michelin_star <- c(TRUE, TRUE, TRUE, FALSE, TRUE)
location <- c("India", "Pakistan", "Hong Kong", "Thailand", "Vietnam")
rating <- c(8.7, 8.7, 8.8, 6.9, 7.7)
sample_frame <- data.frame(signature_dish,michelin_star,location,rating,
stringsAsFactors = TRUE)
sample_frame

#OUTPUT
#signature_dish michelin_star  location rating
#1 Butter Chicken          TRUE     India    8.7
#2    Beef Nihari          TRUE  Pakistan    8.7
#3  Char Siu Rice          TRUE Hong Kong    8.8
#4       Pad Thai         FALSE  Thailand    6.9
#5            Pho          TRUE   Vietnam    7.7
```

Often, the tidy format is not immediately intuitive and can be clumsy to work with. However, it is the most efficient way to work with data in R.
An important library part of the Tidyverse is the **dplyr** library. It contains functions that allow us to manipulate data frames. Some of these are:

- `mutate()`
  It creates a new column in the data frame. For example,

```
        sample_frame <- mutate(sample_frame, rating_squared = rating ^ 2)

```

- `filter()`
  It filters out rows based on certain conditions. For example,

```
        sample_frame <- filter(sample_frame, rating > 8)
        sample_frame
        #OUTPUT
        #signature_dish michelin_star  location rating
        #1 Butter Chicken          TRUE     India    8.7
        #2    Beef Nihari          TRUE  Pakistan    8.7
        #3  Char Siu Rice          TRUE Hong Kong    8.8


```

- `select()`
  It selects certain columns of the data frame. For example,

```
        sample_frame <- select(sample_frame, signature_dish, location)
        sample_frame
        #OUTPUT
        #signature_dish  location
        #1 Butter Chicken     India
```

```
#2    Beef Nihari  Pakistan
#3  Char Siu Rice Hong Kong
#4       Pad Thai  Thailand
#5            Pho   Vietnam

```

Check out On the issue of tidy data

Tidy data is not always the best format for data analysis. For example, if you want to create a scatter plot, you need to have two variables in the same column. This is not possible in tidy data. In this case, you need to use the command `gather()`, which takes a data frame and gathers all the columns into two columns: one for the variable names and one for the values. For example,

```
        sample_frame %>%
            gather(key = "variable", value = "value", -signature_dish, -michelin
    _star, -location)

```

The command `gather()` takes the data frame `sample_frame` and gathers all the columns except `signature_dish, michelin_star, location` into two columns: `variable` and `value`. The `%>%` operator is called the **pipe** operator. It takes the output of the previous command and uses it as the first parameter of the next command. For example, `a %>% b()` is the same as `b(a)`.
The opposite of `gather()` is `spread()`. This command takes a data frame and spreads two columns into multiple columns. For example,

```
        sample_frame %>%
            gather(key = "variable", value = "value", -signature_dish, -michelin
    _star, -location) %>%
            spread(key = "variable", value = "value")

```

The command `spread()` takes the data frame `sample_frame`, gathers all the columns except `signature_dish, michelin_star, location` into two columns: `variable` and `value`, and then spreads the column `variable` into multiple columns.
Other useful commands are `separate()` and `unite()`. The former takes a column and splits it into multiple columns, while the latter takes multiple columns and unites them into one.

A strong operator in the Tidyverse is the **pipe**, represented by either the `%>%` or || > operators. It takes the output of the previous command and uses it as the first parameter of the next command. For example, `a %>% b()` is the same as `b(a)`. We can use this operator to chain manipulations of data frames together without the need for intermediate variables. For example,

```
sample_frame |> select(signature_dish, rating) |> filter(rating > 8)
sample_frame
#OUTPUT
#signature_dish rating
#1 Butter Chicken    8.7
#2    Beef Nihari    8.7
#3  Char Siu Rice    8.8

#is the same as
small_sf <- select(sample_frame, signature_dish, rating)
small_sf <- filter(small_sf, rating > 8)
small_sf
```

The `pull()` function allows us to extract a column from a data frame as a vector. For example,

```
sample_frame |> pull(signature_dish)
#OUTPUT
```

```
#[1] "Butter Chicken" "Beef Nihari"    "Char Siu Rice"  "Pad Thai"       "Pho"
```

The **Tibbles Library** is also included in the Tidyverse macropackage. A tibble is an extension of the data frame that is easier to use because of extra functionality, such as them displaying better, giving better error messages, and being able to hold more information about the data. Tibbles are also able to hold more complex entries. In data frames, columns need to be vectors of numbers, strings, or logical values; tibbles, on the other hand, can hold lists and functions. You can create a tibble by using the command `tibble()`. For example,

```
tibble(x = 1:3, y = c("a", "b", "c"))
#OUTPUT
# A tibble: 3 x 2
#       x y
#   <int> <chr>
#1      1 a
#2      2 b
#3      3 c
```

A common function that returns a tibble is the `group_by()` function. It takes a data frame and groups it internally by a certain column — which makes other functions like `summarize()` operate on these groups considering each a single entry. For example,

```
sample_frame |> group_by(michelin_star) |> summarise(mean_rating = mean(rating))
#OUTPUT
# A tibble: 2 x 2
#  michelin_star mean_rating
#  <lgl>               <dbl>
#1 FALSE                6.90
#2 TRUE                 8.07
```

The `summarise()` function, which takes a data frame and a function, returns a tibble with the result of applying the function to each column of the data frame. For example,

```
sample_frame |> group_by(location) |> summarise(integer_rating = as.integer(
rating))
#OUTPUT
#location integer_rating
#1     India              8
#2  Pakistan              8
#3 Hong Kong              8
#4  Thailand              6
#5   Vietnam              7
```

Conditionals in the tidyverse are different to those in base R. The `case_when()` function takes a vector of conditions and a vector of values and returns a vector with the values that correspond to the first condition that is met. For example,

```
x <- c(-88, 19, 22, 69, 77)
case_when(x < 0 ~ "negative",
          x == 0 ~ "zero",
          x > 0 ~ "positive")
#OUTPUT
#[1] "negative" "positive" "positive" "positive" "positive"
```

Another useful function is `between(a,b,c)`. It returns a boolean vector that indicates whether each element of the vector a is within the range from b to c. For example,

```
x <- c(1, 2, 3, 4, 5)
between(x, 2, 4)
#OUTPUT
#[1] FALSE  TRUE  TRUE  TRUE FALSE
```

Importing Data

When importing data to use in an R script, it is important to make sure that R knows where to find the file in question. This can be achieved by having the file in the default folder looked at by R or by specifying the full path to the file. Look at the following example, where we instruct R to extract information from the file `cheese_guide.csv` stored in the folder `data`:

```
filename <- "cheese_guide.csv"
directory <- "data"
full_path <- file.path(directory, filename)
file.copy(full_path, filename, overwrite = TRUE)
#once the file is in the default folder, we can import it
cheese_guide <- read_csv("cheese_guide.csv")
#This function is defined in the tidyverse
```

A function essential when it comes to handling files is `list.files()`, which returns a vector with the names of all the files in a given directory. Similarly, the function `getwd()` returns the name of the current working directory. We can always read files from the current working directory by using the command `read_csv("filename")`.
Readr amd **readrxl** are two libraries contained in the Tidyverse that have functions to read different types of data files. Take a look at the summary of such functions below:

Function	Format	Typical Suffix
`read_csv()`	Comma–separated values	`.csv`
`read_tsv()`	Tab–separated values	`.tsv`
`read_delim()`	Delimited files	`.txt`
`read_fwf()`	Fixed–width files	`.txt`
`read_excel()`	Excel spreadsheets	`.xlsx`
`read_table()`	Tabular data	`.txt`
`read_xls`	Legacy Excel spreadsheets	`.xls`

Table 1.1: Summary of readr and readxl functions

1.0.4 data.table

`data.table` is perhaps the most prominent package to handle data frames, also known as tables. This section will cover some useful functions that are found in the package.

- `setDT(d)`
  This command allows us to convert a data frame into a data table, the argument d thus being a data frame. For example,

```
        setDT(sample_frame)

```

- `datatable[, a:= f]`
  This method allows us to easily add column a to the data table `datatable` by applying function f to the data table. For example,

```
        sample_frame[, rating_squared := rating ^ 2]

```

  This function can take in several arguments, where each column must be delimited with the `:=` operator.

- `datatable[condition, (a,b)]`
  This method allows us to select certain rows in columns a, b of the data table `datatable` based on a condition. For example,

```
sample_frame[rating > 8, (signature_dish, location)]
#OUTPUT
#signature_dish  location
#1 Butter Chicken     India
#2    Beef Nihari  Pakistan
#3  Char Siu Rice Hong Kong

```

- `datatable[condition, (a,b), by = c]`
  This method allows us to select certain rows in columns a, b of the data table `datatable` based on a condition, grouping the data by column c.

- `datatable[a, .(b, c)]`
  This method allows us to summarise the data contained in column a of the data table `datatable` by applying functions b and c to it. For example,

```
sample_frame[, .(mean_rating = mean(rating), max_rating = max(rating))]
#OUTPUT
#mean_rating max_rating
#1:    8.36          8.8

```

- `datatable[order(a, decreasing = BOOLEAN_VALUE)]`
  This method allows us to order the data table `datatable` by column a in either ascending or descending order.

Data Visualisation

The most popular package employed to create graphs is `ggplot2`. The power of this package lies in its capacity to generate graphs by **layers**, where each layer defines geometries, computes summary statistics, determines what scales to use, and other parameters, something known as a **grammar of graphics**(gg). An important limitation of this package is that the data that is input to it must be in tidy form. The typical syntax we follow to achieve this is `DATA` > ggplot() + LAYERA + LAYER B + ... | Here is an example of this syntax in action:

```
sample_frame |> ggplot() + geom_point(aes(x = location, y = rating)) + geom_text(
aes(location, rating, label = abb))
```

What this code does is take the data frame `sample_frame` and create a scatter plot with the location of the restaurant on the x–axis and the rating on the y–axis. The first step to instantiate a plot is to define a `ggplot` object, which is initialised with the function `ggplot()`. After creating this object, it is time to add layers. The first layer we add is the `geom_point()` layer, which creates a scatter plot. The argument of this function is the `aes()` function, which stands for aesthetic. The `aes()` function is used to map variables to aesthetics, which are visual properties of the graph. For example, the `aes()` function in the previous example maps the variable `location` to the x–axis and the variable `rating` to the y–axis. Other properties of the data that connect with features of the graph are size and colour. The possible arguments of the `aes()` function depend on the type of geeometry being used in the first

place. You can use the `help()` function to find out more about the possible arguments of a function. In the above example, we added another layer to the plot, the `geom_text()` layer. This layer adds text to the graph. The argument of this function is the `aes()` function, which maps the variable `location` to the x–axis, the variable `rating` to the y–axis, and the variable `abb` to the label of the text.

Check out Anatomy of a graph

For easier understanding, we can breakdown the components of a plot into

1. Data: The data that is being plotted.
2. Geometry: The type of plot that is being created. Some examples of geometry are scatterplot, barplot, histogram, smooth densities, qqplot, and boxplot.
3. Aesthetic mapping: The mapping of variables to visual properties of the graph. Some examples of aesthetic mapping are x–axis, y–axis, size, and colour. The way we define the mapping depends on the geometry we have chosen.

You can also scale the axis by adding the `scale_x_continuous()` and `scale_y_continuous()` layers. For example,

```
sample_frame |> ggplot() + geom_point(aes(x = location, y = rating, label = abb))
 + scale_x_continuous(breaks = seq(0, 10, 1))
```

The argument of the `scale_x_continuous` function in this example specifies that the `x` axis should be scaled from 0 to 10 with a step of 1. Similarly, the argument could be `trans = "log10"`, which has the effect of scaling the axis logarithmically. This transformation is so common that actually `ggplot2` provides two functions that perform this automatically: `scale_x_log10()` and `scale_y_log10()`. There are specific layers you can add for colour (`gem_point(size = int, color = "colour_name")`), axis labels (`xlab()` and `ylab()`), and graph title (`ggtitle()`).
The function `qplot()` allows you to create graphs more directly and concisely — the q stands for quick! For example,

```
qplot(x = location, y = rating, data = sample_frame)
```

In a single line of code, we have created a scatter plot with the location of the restaurant on the x–axis and the rating on the y–axis.

Visualising Data Distributions

When it comes to data distributions, we can divide the possible data types in two categories:

- Categorical Type: The data is divided into small groups with many data points in each group, for example `male or female`. For example, the location of the restaurant in our running example. The data in this category can be divided in **ordinal** groups, where they have a natural order (e.g. low, medium, high), or **non–ordinal** groups, where they do not have a natural order. Bar plots are effective for this types of data.
- Numeric Type: In general, the data is divided in many groups with few data points in each group. The groups can be discrete or continuous. Histograms are good for this kind of data.

The most basic statistical summary of a list of objects or numbers is its distribution, a description of a list with many entries. When the data is numeric, the task of displaying its distribution is not trivial. When data is not categorical, its distribution is often defined in terms of a mathematical function; generally, the one used is the *empirical cumulative distribution function* (eCDF). This function can be plotted using the `ecdf()` function. Barplots can be generated by using the `geom_bar()` geometry. For example,

```
sample_frame |> ggplot() + geom_bar(aes(x = location))
```

This function is however not as popular because it does not easily represent features of the data such as the symmetry of the distribution, the modes, or the presence of outliers. An alternative to this kind of graph is the **histogram**, which is a graph that represents the distribution of data by grouping it into bins and plotting the number of entries in each bin — suitable for the numeric type of data. Histograms are generated using `geom_histogram()`. The only required argument for a histogram is `x`, the variable for which we will build a histogram. For example,

```
sample_frame |> ggplot() + geom_histogram(aes(x = rating), binwidth = 0.5)
```

Check out The Normal Distribution

Also known as the bell curve or the Gaussian distribution, it is a function that represents the distribution of many random variables as a **symmetrical** bell–shaped curve about the **mean**. The mean determines the location of the centre of the graph, while the standard deviation determines the height and width of the graph. Data near the centre is more likely to occur, while data far from the centre is less likely to occur. The normal distribution is a good approximation for many real–world phenomena, such as the distribution of heights or weights of a population. Mathematically, it can be defined as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Another approach to the representation of numeric data is the **smooth density** plot, which is a graph that represents the distribution of data by smoothing it out and plotting its density. For any interval, the area under the curve of that interval gives us an approximation of what proportion of the data is in the interval. An advantage of smooth density plots is that they make it easier to compare several distributions, particularly because of its smoothness — as opposed to the rugged edges of a histogram. Density plots are created using the geometry `geom_density()`. For example,

```
sample_frame |> filter(michelin_star == TRUE) |> ggplot(aes(x = rating, fill =
michelin_star)) + geom_density(fill = "red")
```

The most classical way to summarise a numeric dataset is through displaying its **normal distribution**. If our data is *approximately* normally distributed, we can match our data to a normal distribution by matching the average and standard deviation of the data to that of the normal distribution. For a list of numbers contained in vector `v`, the average is defined as `m <- sum(x) / length(x)` and the standard deviation is defined as `sd <- sqrt(sum((x-mu)^2) / length(x))`. R has two pre–build functions that achieve this, given a numeric input as an input: `mean(x)` and `sd(x)`. An example of representing data through its normal distribution is:

```
sample_frame |> ggplot(aes(x = rating)) + geom_histogram(aes(y = ..density..),
binwidth = 0.5) + stat_function(fun = dnorm, args = list(mean = mean(sample_frame
$rating), sd = sd(sample_frame$rating)))
```

The **boxplot** is a powerful way to summarise the distribution of data. It is a graph that represents the distribution of data by dividing it into quartiles. The boxplot is defined by five values: the minimum,

the first quartile, the median, the third quartile, and the maximum. The boxplot is a good way to compare several distributions, particularly because of its simplicity. Boxplots are created using the geometry `geom_boxplot()`. For example,

```
sample_frame |> ggplot(aes(x = location, y = rating)) + geom_boxplot()
```

Notice that in the case of boxplots, we need two arguments: `x`, the categories, and `y`, the values.

Statistics in R

Although both probability and statistics handle data, the first field tries to predict information from this data, whereas statistics tries to **extract** information from it.
Consider a **histogram**, where the data is put into bins according to its value. The smaller the bin is, the *smoother* the histogram curve will be. When we arrive at a bin width of zero, the histogram becomes a smooth curve that provides the most accurate information about certain continuous data — this is known as the **distribution** of the data.

Using the knowledge we have acquired of normal distributions, assume we have a data set `heights` containing the heights of different students. Now, assume that we want to find out the percentage of students whose height `<=175`. This can directly be obtained by

```
library(dslabs)
data("heights")
str(heights)
mean(heights$height <= 175)
```

R provides us a tool to find a probability of a distribution using the `pnorm()` function. For example, if we want to find the probability of a student being shorter than 175cm, we can use the command `pnorm(175, mean = mean(heights$height), sd = sd(heights$height))`. The first argument is the value we want to find the probability of, while the second and third arguments are the mean and standard deviation of the distribution respectively. Specifically for the case of the normal distribution, this function would be called via `pnorm(x, mean=0, sd=1)`, where `x` is the value we want to find the probability of.
Using the function `qnorm()`, we can achieve the converse of what we did with the previous function; namely, we can find the data point with $P(x \leq a) = p$ for a given p. For example, if we want to find the height of a student such that $P(x \leq a) = 0.5$, we can use the command `qnorm(0.5, mean = mean(heights$height), sd = sd(heights$height))`.
Statisticians obtain new information from data through **inference**, a process that consists in taking a sample of a population, computing its parameters, and making probable propositions about fixed parameters of that population that are unknown to statisticians.

Check out Standard Deviation

The standard deviation is a measure of the amount of variation or dispersion of a set of values. A low standard deviation indicates that the values tend to be close to the mean of the set, while a high standard deviation indicates that the values are spread out over a wider range. Mathematically, it is defined as follows:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

An important concept to keep in mind during inference is that of a **confidence interval**. A confidence interval is a range of values that is likely to contain an unknown population parameter, with a certain level of confidence. For example, if we want to find the confidence interval of the mean of a population, we can use the command `t.test(heights$height)`. The output of this command is a list of values, the most important of which are `conf.int`, which is the confidence interval, and `p.value`, which is the probability of the mean being equal to the value we have found. A pretty interesting concept is that of the 95% confidence interval. This is the interval that contains the true mean of the population 95% of the time. It can be computed as follows:

```
s = sample(heights$height, 100)
mean(s)
#Assume it is 0.58
sd(s)
#Assume it is 0.496045
mean(s) - 1.96 * sd(s) / sqrt(100)
#0.4827752
mean(s) + 1.96 * sd(s) / sqrt(100) #100 = sample size
#0.6772248
#This means with probability 95% the population %p is between those two values
```

Data Wrangling

Data wrangling is the process of cleaning and unifying messy and complex data sets for easy access and analysis. It is a fundamental part of the data science process. A common technique to reshape a data frame is the `pivot_longer()` method. This method will select a subset of columns and combine them into two columns — one for the variable names and one for the values. This is useful in cases in which the input data is not in a tidy format. For example,

```
library(tidyverse)
library(dslabs)
path <- system.file("extdata", package = "dslabs")
filename <- file.path(path, "fertility-two-countries.csv")
wide_data <- read_csv(filename)
#This contains:
#country '1960'  '1961'  '1962'  '1963'  '1964'  '1965'  '1966'  ...
#<chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   ...
#1   Canada  3.923   3.909   3.873   3.826   3.772   3.716   3.662   ...
#2   Mexico  6.775   6.817   6.858   6.898   6.936   6.973   7.009   ...
#We can use the pivot_longer() method to reshape this data frame
long_data <- pivot_longer('1960':'2015', names_to = 'year', values_to = '
fertility')
#This contains:
#country year    fertility
#<chr>   <chr>   <dbl>
#1   Canada  1960    3.923
#2   Canada  1961    3.909
#3   Canada  1962    3.873
#4   Canada  1963    3.826
#5   Canada  1964    3.772
#6   Canada  1965    3.716
#etc
```

The reverse operation of `pivot_longer()` is `pivot_wider()`. This method will select a column and then break it into columns whose names are the values of the original column. Another useful command is `separate()`. This command takes a column and splits it into multiple columns. For example,

```
#Consider table3
## # A tibble: 6 x 3
##       country  year              rate
##        <fctr> <int>             <chr>
## 1 Afghanistan  1999      745/19987071
## 2 Afghanistan  2000     2666/20595360
## 3      Brazil  1999   37737/172006362
## 4      Brazil  2000   80488/174504898
## 5       China  1999 212258/1272915272
## 6       China  2000 213766/1280428583
table3 %>%
separate(rate, c("cases", "population"), sep="/")
# A tibble: 6 x 4
##       country  year  cases population
## *      <fctr> <int>  <chr>      <chr>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3      Brazil  1999  37737  172006362
## 4      Brazil  2000  80488  174504898
## 5       China  1999 212258 1272915272
## 6       China  2000 213766 1280428583
```

The inverse of `separate()` is `unite()`. This command takes multiple columns and unites them into one. For example,

```
#Consider table4a
# A tibble: 3 x 3
##       country  year      cases
##        <fctr> <int>      <int>
## 1 Afghanistan  1999        745
## 2      Brazil  1999      37737
## 3       China  1999     212258
table4a %>%
unite(new, country, year, sep = "_")
# A tibble: 3 x 2
##          new  cases
##       <chr>  <int>
## 1 Afghanistan_1999    745
## 2      Brazil_1999  37737
## 3       China_1999 212258
```

The method family `join()` allows us to combine multiple data frames into one. For example,

```
#Consider table1
# A tibble: 6 x 3
##       country  year  cases
##        <fctr> <int>  <int>
## 1 Afghanistan  1999    745
## 2 Afghanistan  2000   2666
## 3      Brazil  1999  37737
## 4      Brazil  2000  80488
## 5       China  1999 212258
## 6       China  2000 213766
#Consider table2
# A tibble: 6 x 3
##       country  year population
##        <fctr> <int>      <int>
## 1 Afghanistan  1999   19987071
```

```
## 2 Afghanistan  2000   20595360
## 3      Brazil  1999  172006362
## 4      Brazil  2000  174504898
## 5       China  1999 1272915272
## 6       China  2000 1280428583
table1 %>%
left_join(table2)
## Joining, by = c("country", "year")
# A tibble: 6 x 4
##       country  year  cases population
##         <fctr> <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3      Brazil  1999  37737  172006362
## 4      Brazil  2000  80488  174504898
## 5       China  1999 212258 1272915272
## 6       China  2000 213766 1280428583
```

The `left_join()` method takes two data frames and combines them into one, keeping all the rows of the first data frame and adding the columns of the second data frame. The `right_join()` method does the same, but keeping all the rows of the second data frame. The `inner_join()` method keeps only the rows that are common to both data frames. The `full_join()` method keeps all the rows of both data frames.

Web Scraping

Web scraping is the process of extracting data from websites. It is a useful tool for data scientists because it allows them to collect data from the internet and use it in their projects. The `rvest` package is a powerful tool for web scraping. It allows us to extract data from HTML and XML documents. The `read_html()` function allows us to read HTML documents. For example,

```
library(rvest)
url <- "https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations)"
page <- read_html(url)
page
#OUTPUT
#{html_document}
#<html lang="en">
#[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">\n<title>List of countries by population (United Nations) - Wikipedia</title>\n<
...
#[2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt ns-0 ns-subject page-List_of_countries_by_population_United_Nations rootpage-List_of_countries_by_popul ...
class(page)
#OUTPUT
#[1] "xml_document" "xml_node"
```

The `html_nodes()` function allows us to extract nodes from HTML documents. For example,

```
page <- read_html("https://en.wikipedia.org/wiki/List_of_countries_by_total_wealth")
page |> html_nodes("table") |> head()
#OUTPUT
```

```
{xml_nodeset (6)}
#[1] <table class="wikitable sortable">\n<caption>Countries by total wealth (billions USD), Credit Suisse (2019)<sup id="cite_ref-1" class="reference"><a href="#cite ...
#[2] <table class="wikitable sortable">\n<caption>Countries by total wealth (billions USD), Credit Suisse (2018)<sup id="cite_ref-1" class="reference"><a href="#cite ...
#[3] <table class="wikitable sortable">\n<caption>Countries by total wealth (billions USD), Credit Suisse (2017)<sup id="cite_ref-1" class="reference"><a href="#cite ...
#[4] <table class="wikitable sortable">\n<caption>Countries by total wealth (billions USD), Credit Suisse (2016)<sup id="cite_ref-1" class="reference"><a href="#cite ...
#[5] <table class="wikitable sortable">\n<caption>Countries by total wealth (billions USD), Credit Suisse (2015)<sup id="cite_ref-1" class="reference"><a href="#cite ...
#[6] <table class="wikitable sortable">\n<caption>Countries by total wealth (billions USD), Credit Suisse (2014)<sup id="cite_ref-1" class="reference"><a href="#cite ...
page <- page |> setNames(c("Country", "Total Wealth", "Share of Global Wealth"))
head(page)
#OUTPUT
#  Country Total Wealth Share of Global Wealth
#1   USA         360.6                  25.0%
#2   China          63.8                  17.7%
```

String Patterns

A **string** is a sequence of characters. A **pattern** is a sequence of characters that define a string. A **regular expression** is a sequence of characters that define a pattern. Regular expressions are used to search for patterns in strings. You can reference the table below for a summary of the most common regular expressions:

stringr	Task	Description	R-b
`str_detect()`	Detect presence	Does a string contain the pattern?	`grep`
`str_which()`	Locate pattern	Returns the index of all entries that contain the pattern	`gre`
`str_subset()`	Detect	Extract elements that match the pattern	`gre`
`str_locate()`	Locate pattern	Returns position of the first occurrence of pattern in the string	`grege`
`str_locate_all()`	Locate pattern	Returns position of all occurrences of pattern in the string	`grege`
`str_view()`	Visualise	View the matches	`N`

Table 1.2: Summary of stringr functions

Let us look at an example. In this table, a column we expect to be numeric is actually a character vector due to the presence of a comma. We can eliminate this using `str_replace_all` as

```
library(tidyverse)
library(dslabs)
data("murders")
murders |> mutate(rate = str_replace_all(rate, ".", " ") |> as.numeric()) |> head()
#OUTPUT
```

```
#  state abb region population total non_citizens rate
#1 Alabama   AL  South    4779736   135           4 3
#2 Alaska    AK   West     710231    19           0 2
#3 Arizona   AZ   West    6392017   232           5 4
#4 Arkansas AR  South    2915918    93           4 3
#5 California CA   West   37253956  1257          27 3
#6 Colorado CO   West    5029196    65           3 1
```

Regular expressions are a powerful tool for searching and manipulating strings. They are a sequence of characters that define a search pattern. They are used to search for patterns in strings. The `stringr` package provides a set of functions that allow us to work with regular expressions. The `str_detect()` function allows us to detect the presence of a pattern in a string. For example,

```
library(stringr)
example <- c("abc", "1ab", "888", "a1b", "a1b2c3")
str_detect(example, "(a)(b)")
#OUTPUT
#[1] TRUE TRUE FALSE FALSE FALSE
```

Regular expressions are explained in detail in this cheatsheet.

Machine Learning

In Machine Learning, our goal is to build an algorithm that takes feature values as inputs and returns a prediction for the outcome. In this section, we will cover the *supervised learning* approach, in which we:

1. Prepare a data set of <input, label> pairs
2. Feed this dataset to the model building algorithm as input

The model building algorithm is a function that takes the data set as input and returns a model as output. The model is a function that takes the input as input and returns the label as output. There is only a handful of these models available on R and they are straightforward to implement. Often, you can pick several, train them, and compare their performance to pick the best one.
As opposed to traditional programming, machine learning programming requires approximately correct answers rather than exact answers. This is because machine learning handles *predictions*. If our answers are too exact, a phenomenon called **overfit** occurs, where the model is too specific to the training data and does not generalise well to new data.
When preparing the dataset to feed into the model, we need to divide it into two: **training** dataset and **testing** dataset. The training dataset is used to train the model, while the testing dataset is used to evaluate the performance of the model. We do not need to manually do the splitting; instead, we use the `createDataPartition()` function from the `caret` package. For example,

```
library(caret)
library(dslabs)
data("murders")
set.seed(1)
index <- createDataPartition(murders$state, times = 1, p = 0.7, list = FALSE)
#times k means we want to create k partitions
#p is the proportion of the data we want to use for training
#list = FALSE means we want to return a vector of indices instead of a list of
indices
```

The accuracy of a trained model is the number of correct predictions divided by the total number of predictions; this is not the only metric you should have: if the negative class only appears 1% of the time, a model that always predicts the positive class will have 99% accuracy, but it is not a good model. When we are performing **binary classification**, there are four possibilities:

- **True Positive**: The model predicts the positive class and the prediction is correct.
- **False Positive**: The model predicts the positive class and the prediction is incorrect.
- **True Negative**: The model predicts the negative class and the prediction is correct.
- **False Negative**: The model predicts the negative class and the prediction is incorrect.

These four cases are summarised in the **confusion matrix**, where the rows represent the actual class and the columns represent the predicted class. Using these data, we can use two new metrics:

- **Sensitivity**: The proportion of positive cases that are correctly identified. It is defined as:

$$\text{Sensitivity} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **Specificity**: The proportion of negative cases that are correctly identified. It is defined as:

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$$

  Specificity can also be measured as **precision**, which is defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- **F1 Score**: A combination of sensitivity and precision. It is defined as:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Sensitivity}}{\text{Precision} + \text{Sensitivity}}$$

Besides these, another way of evaluating the performance of a model is through its **receiver operative characteristic (ROC) curve**. This curve plots the sensitivity against the specificity of a model. The area under the ROC curve is a measure of the performance of the model. The closer the area is to 1, the better the model is.
Let us study some commonly–used machine learning algorithms:

- Linear regression: its goal is to find the line $y = mx + c$ that best fits the data. This is achieved by minimising the sum of the squared errors. Let us see an example:

```
1       install.packages("caret","HistData")
2       library(caret)
3       library(HistData)
4       library(tidyverse)
5       galton_heights <- GaltonFamilies |> filter(gender == 'male') |> group_by
    (family) |> sample_n(1) |> ungroup() |> select(father, childHeight) |>
    rename(son = childHeight)
6       #this produces a table with two columns: father and son
7       y <- galton_heights$son
8       test_index <- createDataPartition(y, times = 1, p = 0.7, list = FALSE)
9       train_set <- galton_heights |> slice(-test_index)
10      test_set <- galton_heights |> slice(test_index)
```

```
        #slice selects rows by indexes.
        fit <- lm(son ~ father, data = train_set)
        #son ~ father means son is the dependent variable and father is the
    independent variable
        #the predict function is then
        predict(fit, test_set)

```

- Logistic regression: its goal is to find the line $y = mx + c$ that best fits the data. This is achieved by minimising the sum of the squared errors. Let us see an example:

```
        library(caret)
        library(dslabs)
        data("murders")
        set.seed(1)
        index <- createDataPartition(murders$state, times = 1, p = 0.7, list =
    FALSE)
        train_set <- murders |> slice(-index)
        test_set <- murders |> slice(index)
        fit <- glm(state ~ population, data = train_set, family = "binomial")

```