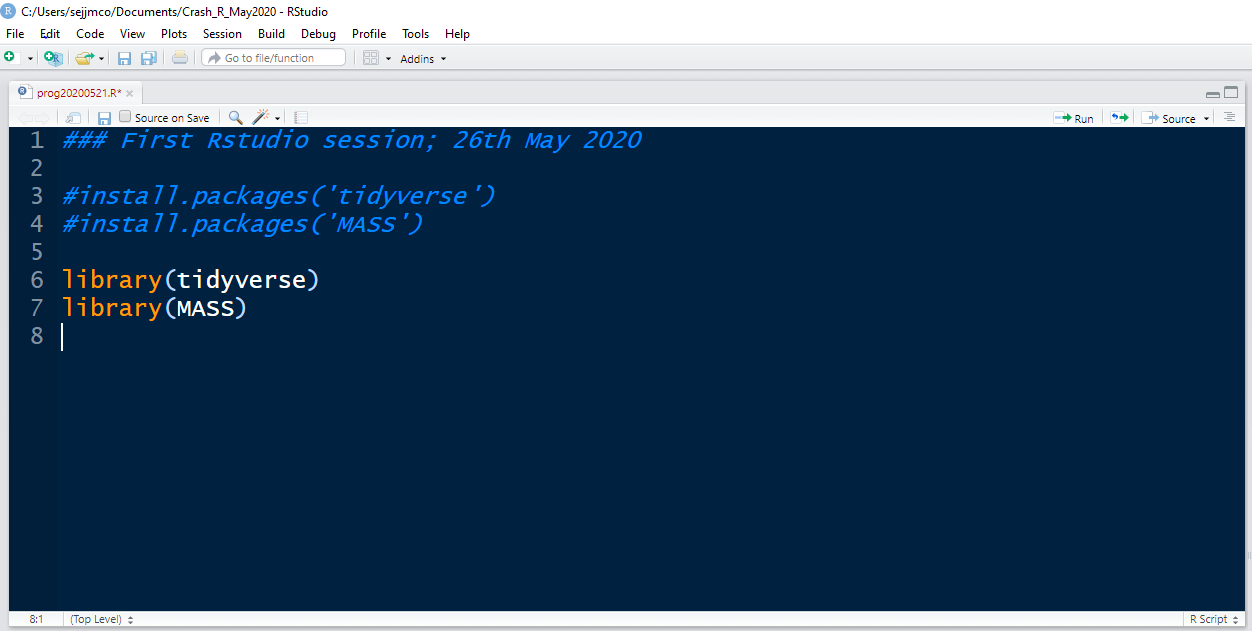
Basic R

# Mario Cortina Borja, PPP/ICH/UCL

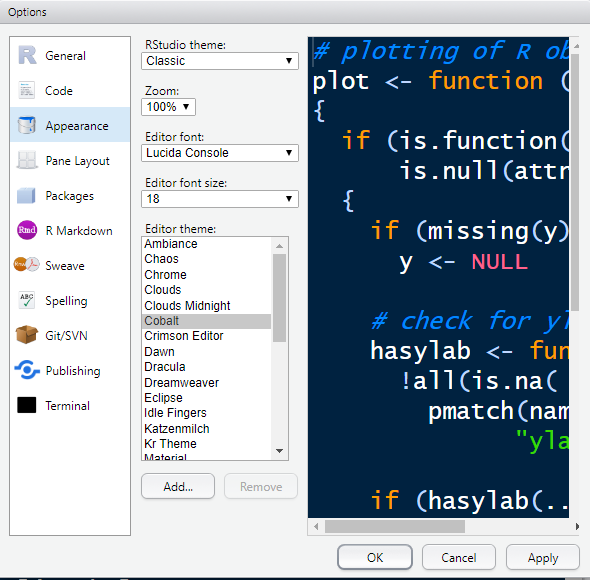
# Tuesday 26th May 2020, 2pm BST

# RStudio layout

Once you’ve installed R and RStudio, and packages tidyverse and MASS, and assuming that you have saved the code in the file prog2002005.R, your Code Editor should look something like this:



There may be variations in the colours of your Code Editor. On the top menu, click **Tools -> General Options**; you should see a window like this:



Try different combinations of **Editor theme**, **Editor font size, Editor font**; you should see the sample code changing in this window. Once you like one, click **OK**. Don’t worry about the other options, you can change them any time from the **Tools** menu. **Editor theme** provides different colours for different aspects of the code. For example in the **Cobalt** theme, comments appear in *blue*, strings in green, reserved words in orange, values in pink, and code in white. You may read more about RStudio themes [here](https://support.rstudio.com/hc/en-us/articles/115011846747-Using-RStudio-Themes).

**As we’ve seen, there are four windows (also called panes) in an RStudio session:**

1. **Bottom left: R Console window** (also called command window). This is the most important window, because this is where R executes your code. The numerical output of your code is also displayed in this window.
2. **Top left: Code editor window** (also called script window). This is where sets of R commands (also called **scripts**, or **programmes**) are edited and saved. You can open a new code editor window from the top menu in RStudio: click **File → New file → R script**. It’s not enough just to type a command in this window: to execute it you have to transfer it to the **R Console window**, where it is interpreted. To run a line or a set of lines or the whole file, you can click **Run** from the **Code editor window** menu, or press CTRL+ENTER to run these commands in the **R Console window**.
3. **Top right: workspace / history window**. There are two windows: the **workspace** **window** lists the objects (and their values) which are in the memory of your RStudio session, whilst the **history** **window** shows what has been typed before in the **Code editor window**.
4. **Bottom right: files / plots / packages / help window**. Here you can open files which are present in your Windows directory, view graphical output from your code (including current and previous plots), install and load packages, and use the help function.

The size of the four main windows can be changed using the bars between them. You can change the RStudio layout by clicking **View -> Panes** from the top RStudio menu bar.

# Computing in R

*To understand computations in R, two slogans are helpful:*

*• Everything that exists is an object.  
• Everything that happens is a function call.*

*John Chambers*

These two slogans, defined by John Chambers a long time ago, are in fact the axioms upon which R is constructed. Please take 20 seconds of your time to read them again.

…

OK. For now, let’s assume that in the first slogan *exists* means that objects somehow “live” in your RStudio session, and that the slogan slogan means that objects are always generated by calling functions.

Recall that in mathematics a *function* can be thought of as a rule (or an algorithm) defining a relation between two sets which are sometimes called domain and range. The domain is the set whose elements the function is applied to; the range is the set where the function maps the elements of the domain. For example, the function *f(x) = x2* can be applied to any real number, and it maps it to its squared value. Thus the domain of this function is all real numbers, and its range is the set of non-negative real numbers because the square of any number, positive or negative, is always non-negative. For example, a *function call* applying this function to the number -3, results in the number 9. This is a central idea in R, and Chambers’ second slogan means that you can construct more objects by applying functions (which are objects themselves!) to some objects. Don’t be confused by this formalism: you’ve been using these ideas implicitly since you were taught that 2 + 3 = 5. Just be aware that they are the most fundamental principles upon which R is built.

# Objects, methods, and brackets

For now, let’s think of any object in R as a data structure having some attributes and methods which may act on its attributes.A data structure can be loosely defined as format in which data can be organised, managed, and stored in order to enable efficient access, modification, and processing. We’ll see more of this in section 5.

There are three basic methods in R: print, summary, and plot. Method print is the default method – if no method is specified for an object, R will simply print it, as we’ll see.

Attributes refer to additional information about an object, for instance the numbers of rows and columns of a matrix. **We should first distinguish between an object and its name, and its value.** For example, suppose we want to create a very simple object consisting of a vector with the first five positive integer numbers. We can use the R function c whose purpose is to combine elements into a vector or a list. Note that the name of the function c is “c” and its value will be the resulting vector combining the elements of interest. To create this very simple vector with five elements we need to talk about the most important function in R: the assignment operator, which can be written in two forms: <- and =. Some people prefer using the second version but I prefer the first version. You can use whichever you want. The purpose of the assignment operator is to assign a value to a name. For example, type in your R console, next to the cursor, this command:

x <- c(1,2,3,4,5)

This can be read aloud as “x gets c of 1 2 3 4 5” – we are **assigning** to the **name** “x” the **value** of applying the function c to the list of the first five positive integer numbers. Note that we could have written this statement as:

x = c(1,2,3,4,5)

Both forms of assignation are equivalent. Note also that the number of spaces anywhere doesn’t matter, apart from the operator <- being required to have no spaces between its two characters. Any of these forms are also valid and equivalent to the first one mentioned above:

x<-c(1,2,3,4,5) x=c(1,2,3,4,5)

x<- c(1, 2, 3, 4, 5) x=c(1, 2, 3, 4, 5)

x <- c(1, 2, 3, 4, 5) x = c(1, 2, 3, 4, 5)

R is case-sensitive so for example X<-c(1,2,3,4,5)assigns the output of the function c in this case to the object named “X” which has a different name than that of object x.

Note that there are two ways to run (or execute, or interpret) this operation:

x <- c(1,2,3,4,5)

1. Typing it in the **Code editor window**, selecting it and then either pressing CTRL+ENTER or pressing **Run** from the menu in the **Code editor window**: any of these actions executes this command in the **R Console window**
2. Typing it directly in the **R console window** where it is executed instantly

Form 1 is the preferred way if you’re programming and have a list of commands to run; usually they are saved as a programme file with extension \*.R. Form 2 is normally used if you just want to see the output of a function immediately.

Recall that print is the default method in R. What happens when you simply type x in the **R console window** is that the method print (which is of course a function, which is of course an object) is applied to the object named “x” (which happens to be the object x in our environment). Try it, please. You should see something like this:

> x<- c(1,2,3,4,5)

> x

[1] 1 2 3 4 5

Note that you don’t have to type “>”. When you type the name of the object x, R prints its value: in this case it’s a vector of five numbers. The symbol [1] that you see next to that vector indicates that this is the first line of that output. You should be careful about what you print! If you ask R to print a vector with 1’000,000 elements… well, better not try it now.

Try this example now

> x<-1:50

> x

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50

Note how the **square brackets** indicate the index of the first element of each row of output.

We mentioned that the three basic methods in R are print, summary and plot, and you may have noticed we read c(1,2,3,4,5) as “c of 1 2 3 4 5”. Let’s focus on the word “of” and the round brackets: this means that c is applied to the vector of numbers defined in the round brackets. This can also be interpreted as this: the argument of the function c is the list of numbers 1, 2, 3, 4, 5, and its value is a vector of length 5 consisting of numbers; this value is then assigned to the object x, whose name is, well, “x”. Note that a vector is a particular, one-dimensional case of an array, that is a set of numbers identified through an index, so we can identify, for example, its first element. Vectors are a data structure, and we’ll talk about this in section 5.

We can now apply the three basic methods to x; try these commands on the **R console window** and note that anything after the symbol # is a comment and won’t be executed.

x <- c(1,2,3,4,5) ### this changes the value of x

x <- 1:5 ### same as above line using operator “:”

x

print(x)

summary(x)

plot(x)

What do you see?

R can be used as a calculator with the usual algebraic operations: +, -, \*, /, and ^. Try, for example:

2 + 2

### note that this means

### “print the output of applying the operator “+” to arguments 2, 2”

6^2

(5 + 6/7) / 3

### note that round brackets can be used to group output before

### applying another function to that output

You can apply an algebraic operation to vectors, for example try:

x + 500

x^2

There are many mathematical and statistical functions built-in, for instance try, and note that “;” has the same effect (separate commands) as return.

sqrt(x); sqrt(x + 100)

mean(x); mean(x + 100)

max(x); max(x + 100)

sd(x); sd(x+100)

You can create more objects using the assignment operator, for example:

mean\_x <- mean(x)

summary\_x <- summary(x)

print them:

mean\_x

summary\_x

What are the lengths of these objects? (answer: 1 and 5).

We have seen that the round brackets,(), can be used in R to define the arguments a function is applied to, as in sqrt(x + 100) and (5 + 6/7) / 3. R uses square brackets, []to subset or select or extract parts of an array. For example try:

x <- c(1:5, 10,11,12)

x ### prints the whole vector

x[6] ### selects the sixth element of the vector x

x[5:8] ### selects elements indexed 5, 6, 7, 8 of x

x[c(1,6,8)] ### selects the first, sixth, and eight elements of x x[c(6,8,1)] ### selects the sixth, eighth, and first elements of x

You can assign the output of selecting parts of an array to another object, for instance:

x1 <- x[c(1,1,3,5,8)]

As we mentioned, round brackets define the arguments passed to the function when we execute it. You can interpret this in the sense that the arguments are the input values passed to a function in order to produce its output value. Remember Chambers’ 2nd slogan? Everything that happens in R is a function call. Precisely. When you say sqrt(x) you’re calling the function sqrt, which computes the square root of the argument you used in the call (in this case, the object x), and then produces the square root values of that argument as output. For example sqrt(100) returns (and prints) the value 10. If you pass an argument which is not appropriate for the function, R will report an error and stop executing that command. Try, for instance

sqrt(-100)

and you’ll get this:

[1] NaN

Warning message:

In sqrt(-100) : NaNs produced

NaN means “Not a Number”, and that’s the output you get by running sqrt on an object outside its domain; you also get a warning message telling you how you called the function and the problem resulting for calling it with an argument outside its domain.

Apart from NaN, R uses NA (Not Available) to denote a missing value. This is an important aspect, as missing values occur too often in data analysis. For example, suppose that the vector ages has the ages in completed months of eight children, but we don’t know this for child number 6 so we input it as NA:

ages <- c(35, 40, 44, 33, 38, NA, 35, 39)

What is the mean of this vector of ages? mean(ages)produces, quite reasonably, an NA value because its algorithm specifies adding up the elements of the vector and dividing this sum by the length of the vector but what do you get adding, for instance 2 + NA? Surely an NA. We can, however, compute this mean after excluding the NA observations by using the argument na.rm=TRUE in the function mean:

mean(ages, na.rm=TRUE)

which yields the value 37.71429; in this case, na.rm refers to removing NA’s, and the previous call to mean passes the logical value TRUE to this argument thus allowing computing the mean of the vector after excluding missing values.

# Environments, working directory, and packages

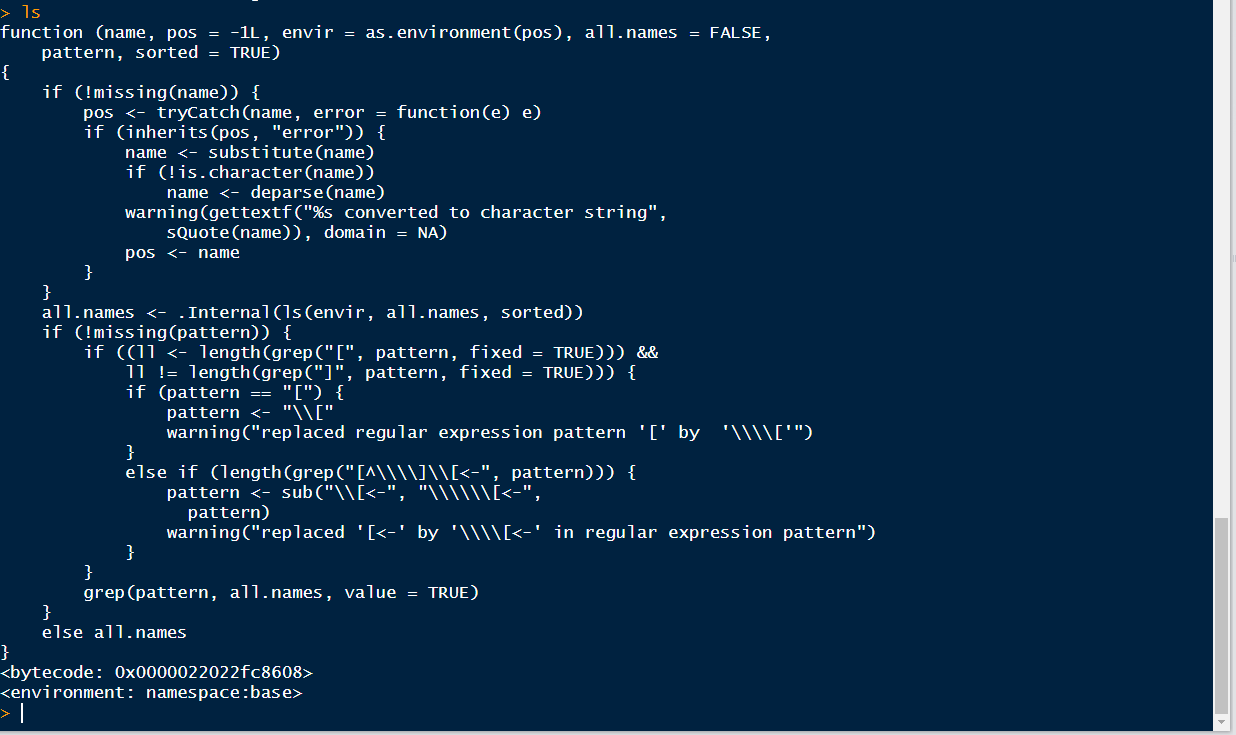
An **environment** is just a place to store objects in R. The main environment in your current RStudio session is called .GlobalEnv, this is where the objects you’re working with in the session are kept. You can see the objects in this environment at the top right-hand side window in RStudio, or you can execute the function ls which lists objects in an environment. If you type ls()you run the function without specifying which arguments you’re running it with. See what happens if you type

ls

and

ls()

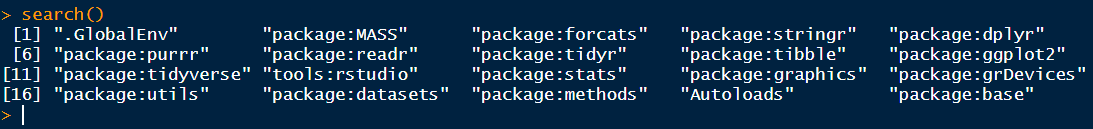
the first command prints the function ls and what you get is the R code which define the function. You should see something like this:



The second command, ls(), executes the function ls with the default arguments: R interprets this as (i) running, not printing the function, because it has round brackets associated to it, and (ii) as using the default arguments as input, because there are no arguments being passed explicitly. You should have something similar to this in your .GlobalEnv:

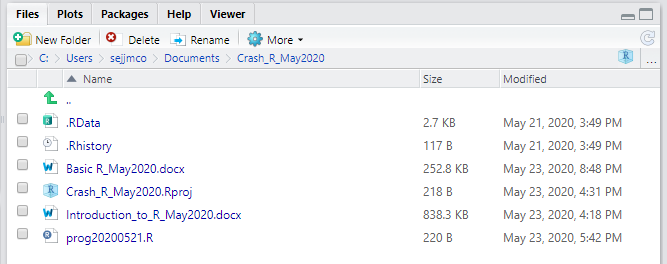


In this example, the default arguments for ls include .GlobalEnv. To see other environments in your RStudio session type search()after the cursor (“>”) in the **R console window**. You should be able to understand the difference between search and search(). This results in:



There are 20 environments in this session, and it’s important to realise that their order is important. The first environment is the .GlobalEnv, and to list the objects stored there it suffices to type ls() because it is a default argument. To see this, type ls(pos=1)and ls(pos=2): the former lists objects in .GlobalEnv, which occupies the first position in the list of environments, as found by function search; the latter lists the objects in the environment in positon 2, in this case library:MASS.

The .GlobalEnv is saved in your working directory, which is the folder on your computer in which you are currently working. Normally, this is the folder in Windows from which you started your RStudio session. To find out this, run getwd() or just look at the top menu in the **R Console window**. An image of the files in your working directory appears in the Files tab at the **Plot and Files window**. It should look like this:



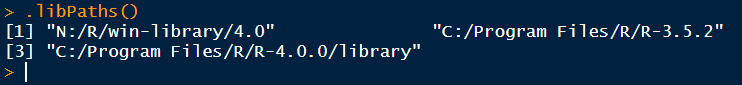
The .Rdata file is where your objects in the .GlobalEnv are stored, file .Rhistory stores the commands you ran in your RStudio session, and the file with extension .Rproj is the file from where you start RStudio. Note that we have a file with extension .R – this is a text file with R code.

Now type save.image(): this updates the .RData file, effectively, it saves all the objects in your RStudio session. You may do this from time to make sure the objects will be there next time you open the .Rproj file.

One of the strengths of R is the large amount of packages freely available. **A package is a collection of R functions, data and compiled code.** The location where the packages are stored is called a library. You can install packages in the directories which are automatically specified when you installed R. To find out where are these directories, type

.libPaths()

in the R console; this gets the library trees within which packages are looked for. For example, this is what I get when I execute .libPaths():



By default, libraries are installed into the first of these directories. To use a library in an R session you must download it from a **repository**. The most widely used is the Comprehensive R Archive Network (CRAN). The CRAN master site is at Wirtschaftsuniversität Wien, in Austria and can be found at the URL <https://CRAN.R-project.org/>. This site is mirrored daily to many sites around the world. See <https://CRAN.R-project.org/mirrors.html> for a complete list of mirrors. Please use the CRAN site closest to you to reduce network load. If you don’t specify a CRAN mirror in the install.packages command, R will choose one for you. In the UK you may use Imperial College London’s <https://cran.ma.imperial.ac.uk/> and in Southern Africa, The University of Cape Town’s <http://r.adu.org.za/>. For example, to install the package MASS the following are equivalent (note that it doesn’t matter if you use single or double quotation marks as long as you’re consistent):

install.packages('MASS', repos='https://cran.ma.imperial.ac.uk')

install.packages('MASS', repos='http://r.adu.org.za/')

install.packages('MASS', repos=' 'http://cran.us.r-project.org')

install.packages('MASS')

A package is installed in the corresponding Windows directory only once; however, you may need to update it if there is a newer version you want to use, or if you change your version of R. After installing a package, you have to load it into memory so you can use it in your RStudio session: this is done with the functions library or require, they are basically equivalent. For example:

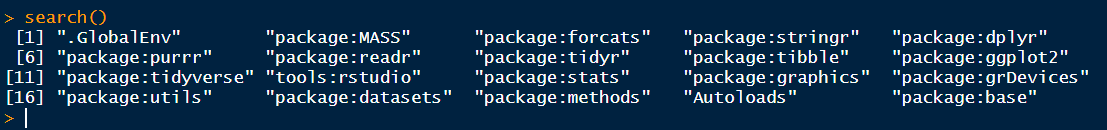
library (MASS)

calls the MASS package into your session. Note the lack of inverted commas: this is because the argument for the function library is the actual package, not its name as it was in the function install.packages.

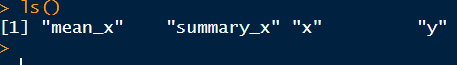
Now recall that packages are environments, that is, spaces where R objects are stored, and that we briefly mentioned the function search as a way to list all the environments present in an RStudio session. The positions in which the environments are listed as the output of search() is important: it determines the order in which R searches for an object that you asked for via a command. The .GlobalEnv environment, which is where the objects present in the .Rdata file in windows is always in position 1. For example, suppose that after typing

library(tidyverse); library(MASS)

the output of search() looks like this:



and that the output of ls() looks similar to this:



To illustrate how R hierarchically searches for objects in the environments present, create an object called Animals consisting of a single number in the .GlobalEnv:

Animals <- 60 ### or any number you like

If you type ls() you should see Animals added to that list of objects. You can of course print(Animals) by simply typing its name; you can also apply to it the other two basic methods in R though that is boring:

summary(Animals)

plot(Animals)

Please remember that R is case sensitive!

Now verify that package(MASS) is loaded in your RStudio session, and make a note of the position it is in the environment list using search(). Assuming MASS is in position 2, list its objects using the explicit argument pos with value 2:

ls(pos=2)

There are 165 objects in library (MASS), and, in alphabetical order, the first ten are:



Note that MASS has also an object with the same name as the singled-number object we just created: Animals. Suppose we want to inspect the Animals object in library MASS: if we call the print method by typing its name, R will search sequentially in the environments present in the RStudio session until it finds it. In this example, such search will end in the first environment, that is .GlobalEnv and R will print the object whose value is the number you assigned to in the previous page. If, however, there were no object named Animals in the .GlobalEnv, R would continue searching in the next environments until it finds it (in this case, in library MASS). If we knew that there’s an object with a particular name in a library present in the session, we can bypass this search by specifying the library, this is done with the following syntax:

MASS::Animals

which prints the object Animals from library MASS. As you see, this is a more interesting object than the one in the .GlobalEnv and we’ll return to it. Meanwhile, try applying the other two basic R methods to this Animals object:

summary(MASS::Animals)

plot(MASS::Animals)

# Data structures in R: vectors, matrices, and lists

We mentioned data structures in section 3; they are a basic concept in computer science. For the purpose of this notes, think of them as formats in which we organise data. You may have worked with data structures – an example is a spreadsheet where the columns represent variables, the rows represent individuals, and the entries are the values of those variables for each individual. In excel, for example, you columns are identified with letters, and rows with numbers.

The difference between working with excel spreadsheets and R is that in R everything is an object: it doesn’t matter if it’s a number, a vector, a matrix, a statistical model, a plot, a list, etc. That is, everything can be programmed, and defined, accessed, modified, processed, … explicitly.

Let’s start with a simple data structure: a vector. We already created vectors in section 3 using the function c to concatenate numbers. For example running this line:

x <- c(1,2,3,4,5); y<-11:15

creates two vectors, x and y, each containing five numerical elements. In section 3 we mentioned that round brackets, (), are used in R to limit the arguments with which a function is called, and that square brackets, [ ], are used to subset, or select elements of a data structure. For example

y[3]; y[1:3]

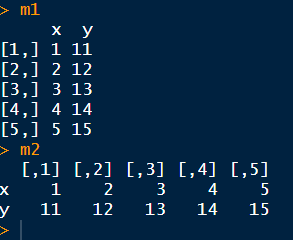
return the third element and the first three elements of the vector y, respectively. As we have seen, we can apply the three basic R methods to a vector, try, for example summary(y) or plot(y). A useful function for vectors is length; try, for instance, to predict the value of

length (c(x,y)).

**Vectors** can be thought of as one-dimensional arrays, and **matrices**, as two-dimensional ones. For example, we can combine x and y column- or row-wise into a matrix using the R functions cbind and rbind:

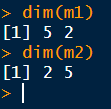
m1 <- cbind(x,y); m2<- rbind(x,y)

print them:



To see the size of a matrix, use the function dim which returns the number of rows and columns (always in that order) of the object it’s applied to.

For instance, dim(m1) and dim(m2) yield vectors of length 2:



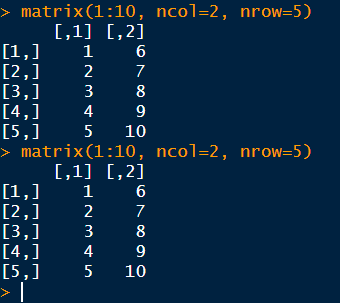
Note that dim(x) yields NULL because x is a vector and doesn’t have dimensions, it has length, which measurable with the function length. Try to see what happens with length(m1) and length(m2).

We mentioned that [] can be used to subset matrices, for instance:

m1[1,1]; m1[1:3, 1:3]; m1[c(1,3,5), c(1,2)]; m1[c(1,3,5), ]

return a number (1); a 3×3 matrix; a 3×2 matrix; the same 3×2 matrix. In the last case, note that the second dimension of m1 has been left without specification of indices, and all (two in this example) columns are chosen. Similarly, m1[, 1] selects all the rows of m1, and only the first column.

We showed how to construct matrices using rbind or cbind; another way is to define them explicitly by positioning the elements of a vector into a matrix with dimensions specified with arguments ncol and nrow. For example



Another data structure commonly used in R is a list, which, unlike a matrix, allows to join elements of different dimensions. A list also allows more flexibility than a matrix in the sense that it can contain elements of different classes: for example, logical values, integer and real numbers, and characters. Consider:

list1<- list('Blue',

c(1,56,18,60,pi),

TRUE,

matrix(1:10, nrow=2, ncol=5)

)

This is a list with four elements separated with commas: the first one is a character string with value 'Blue'; the second is a vector with 5 elements; the third is a single logical variable with value TRUE, and the last one is a matrix with 2 rows and 5 columns, with elements 1,2,3,…,10. Note that R code allows flexibility in writing it by splitting the lines and including spaces as we wish; the important thing is to close the brackets properly. Code should have a sense of order.

We can name the elements of a list, for example:

list1<- list(colour = 'Blue',

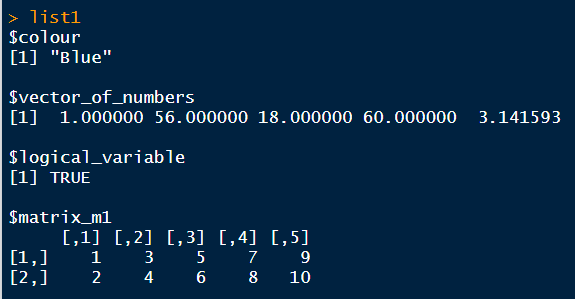
vector\_of\_numbers = c(1,56,18,60,pi),

logical\_variable = TRUE,

matrix\_m1 = matrix(1:10, nrow=2, ncol=5)

)

defines the names of the elements of the list as colour, vector\_of\_numbers, logical\_variable, and matrix\_m1. We can print list1



Elements of a list can be subsetted or accessed in two ways: using indices within the double square bracket, [[ ]], or using the $ operator if the elements of the list have names. For example:

list1[[1]] and list$colour

result in the same value.

# data.frames and tibbles

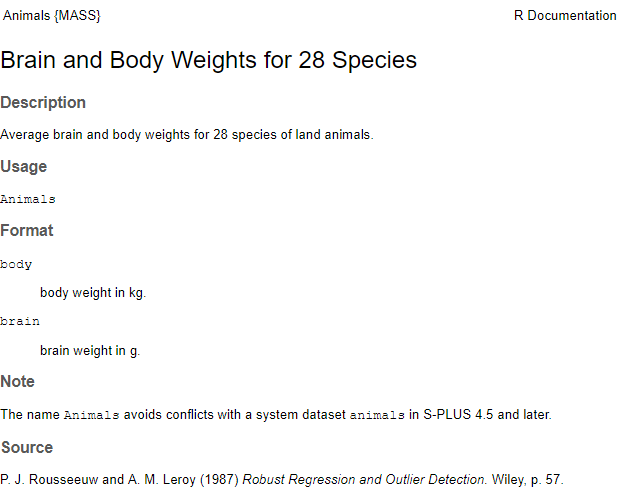
## Definition

data.frames are perhaps the most widely used data structure in R. For now think of them as equivalent to spreadsheets, that is, matrices whose rows represent individuals, and (named) columns represent variables. In R, a data.frame is technically a list, and this allows us to access variables (columns) by their names, as well as having variables with different classes: numerical, categorical, logical, string, etc.

For example, recall the object MASS::Animals which is part of the package MASS. Objects in R often have help files associated with them, and MASS::Animals does have one, try

?MASS::Animals

This opens up a window in the help pane, it looks like this:



To find whether or not this is a data.frame we can ask

is.data.frame(MASS::Animals)

which returns TRUE. We can examine it using the three basic R methods:

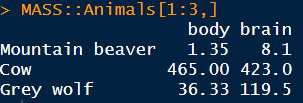
print(MASS::Animals) ### or simply MASS::Animals#

summary(MASS:Animals)

plot(MASS::Animals)

See what happens when you execute these commands. Also, apply the function str, which shows a the structure of a data.frame.

We can select the elements of a data.frame thinking of it as a matrix, for example



or accessing them by name, for instance, MASS::Animals$brain. We’ll return to this data.frame in sections 7 and 8.

## tibbles

Recall that we loaded library tidyverse into our RStudio session. You can (and should!) read more about this library [here](https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/); we only present a definition taken from that webpage:

The tidyverse is a coherent system of packages for data manipulation, exploration and visualization that share a common design philosophy. These were mostly developed by Hadley Wickham himself, but they are now being expanded by several contributors. Tidyverse packages are intended to make statisticians and data scientists more productive by guiding them through workflows that facilitate communication, and result in reproducible work products. Fundamentally, the tidyverse is about the connections between the tools that make the workflow possible.

Having the library tidyverse loaded means that some data.frames will behave like tibbles. A tibble **is** a data.frame; they are sometimes described as “opinionated data.frames”. You can always transform a data.frame into a tibble using the function as\_tibble

In my opinion using the tidyverse is the best way available to work with R. In this workshop I attempted to present the basic concepts of the language but much remains to be done.

## Reading spreadsheets as tibbles

Often data are stored as spreadsheets in an Excel spreadsheet which may be organised in different sheets. Suppose that we have a spreadsheet called “My\_data.xlsx” in our working directory, and that it contains a sheet, called “Data1”. Each of these sheets can be thought of as a matrix in which the first row has the names of the variables, and the first column refers to the names of the individuals.

To read this, we need to install (if it hasn’t been installed before) and load into memory the R library readxl:

install.packages('readxl',repos='https://cran.ma.imperial.ac.uk')

library(readxl)

and use the function read\_xlsx in that library to create a data.frame named My\_data.Data1 (you could have chosen any other name, of course):

My\_data.Data1 <- readxl::read\_xlsx("My\_data.xlsx", sheet="Data1”)

The class of My\_data.Data1 is tibble as it’s specified by read\_xlsx. We’ll return to this class later on.

## Data classes

You may be familiar with the following types (also called classes) of variables: binary, categorical, ordinal, and numeric. A similar system to classify variables operates in R: we have, among others, the following types:

|  |  |
| --- | --- |
| character | refers to strings, for exacple unstructured text, e.g. string <- “This is an object of class character” |
| factor | defines a categorical variable with some structure: could be binary, polytomic, or ordered; usually it includes a reference level, e.g. gender<- c(‘Male’,’Female’, ‘Male’) |
| date | states time-related variables; it can include dates and times of the day |
| numeric | implies numerical values, they can be integer or real numbers |
| formula | implies a statistical model following the [Wilkinson-Rogers notation](https://www.mathworks.com/help/stats/wilkinson-notation.html) e.g.  Y ~ X1 + X2 specifies Y as the outcome (also called response, dependent) variable, and X1, X2 as the explanatory (also called covariates, or independent variables); the W-R notation means that this model implicitly contains an intercept and an error term: |

Note that we normally combine data structures with classes: for instance, we can form a vector of class character. In the next example, we will create a data.frame with data from some of the participants in this course:

sample\_of\_names <- c('Milagros', 'Emma', 'Lorena', 'Lucy', 'Simba', 'Tim', 'Erin', 'Julia', 'Rebecca')

class(sample\_of\_names)returns character. We can operate on this vector, for example:

sort(sample\_of\_names)

[1] "Emma" "Erin" "Julia" "Lorena" "Lucy" "Milagros" "Rebecca" "Simba" "Tim"

Now, define another character vector:

gender<- c(rep('Female', 5), 'Male', 'Male', rep('Female',2))

### note the use of the rep function with two arguments

gender ### prints this vector, which has class character

[1] "Female" "Female" "Female" "Female" "Female" "Male" "Male" "Female" "Female"

Now declare gender as a factor – note that we’re keeping the same name but are changing the class of the object:

gender <- as.factor(gender) ### note the use of function as.factor

Note the difference when we print gender after changing it to a factor

gender ### prints this object

[1] Female Female Female Female Female Male Male Female Female

Levels: Female Male

The elements of gender are not in inverted commas – they are NOT of class character but are levels of a factor; the second line specifies all levels, by default, the reference level (also known as reference category) is found alphabetically, in this case it is Female, and it can be changed easily later on.

To find the frequencies of the levels of this factor, we use the function table:

table(gender)

gender

Female Male

7 2

We get a similar result using summary:

summary(gender)

Female Male

7 2

but note the slightly different format in printing.

As an example of using formula, let’s first generate a numerical variable from the object sample\_of\_names

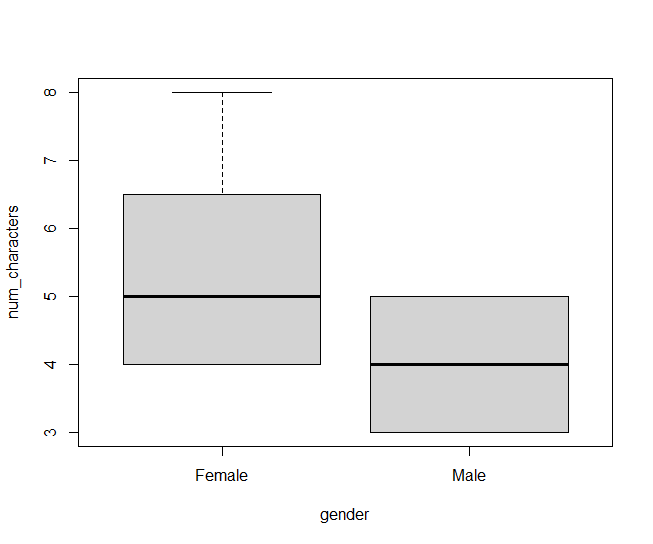
nchar(sample\_of\_names)

#### nchar counts the number of characters in a string

[1] 8 4 6 4 5 3 4 5 7

num\_characters<- nchar(sample\_of\_names) ### assign it to an object

boxplot( num\_characters ~ gender) ### note the formula



Note that R recognises the explanatory variable (gender) in the first argument of function boxplot, which is a formula, as a factor, and acts accordingly.

In the next section we’ll illustrate the elements we’ve seen so far applied to one of the most dramatic data sets in the history of statistics…

# Example: data from the Titanic

The Titanic disaster in April l1912 has attracted the attention of historians, filmmakers, naval engineers and statisticians! For example[, this one](https://rss.onlinelibrary.wiley.com/doi/epdf/10.1111/j.1740-9713.2019.01229.x), in Significance about visualising data from the Titanic; that article has many interesting references to statistical analyses on this dataset. We will use the data.frame titanic3 from R library Hmisc. Firstly, install it:

install.packages(‘Hmisc’) ### use the default repository

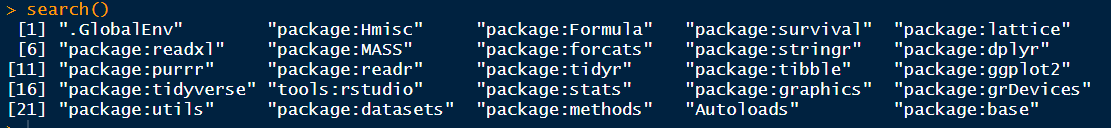
This should take a couple of minutes, whilst all the packages required by Hmisc are downloaded into your windows directory containing R libraries. You should see the R cursor once this is finished. Now load the library into you RStudio session:

library(Hmisc)

Check that it’s been loaded:

search()

should yield something like this, showing package:Hmisc in position 2:



Now use function getHdata[[1]](#footnote-1) from this library to bring the dataset to your .GlobalEnv – this is not always necessary but that’s the easiest way to get these data.

getHdata(titanic3)

Now check that it’s in the .GlobalEnv using ls() to list objects in this environment; titanic3 should be there.

Alternatively, you could read the file titanic3.csv if it is in your working directory:

titanic3 <- read.csv(‘titanic3.csv’)

Let’s explore these data – firstly, are they structured in a data.frame? Ask

is.data.frame(titanic3)### Answer: TRUE

its size, and the names of its variables:

dim(titanic3) ### answer: 1309 14, that is, 1309 individuals, 14 variables

names(titanic3)

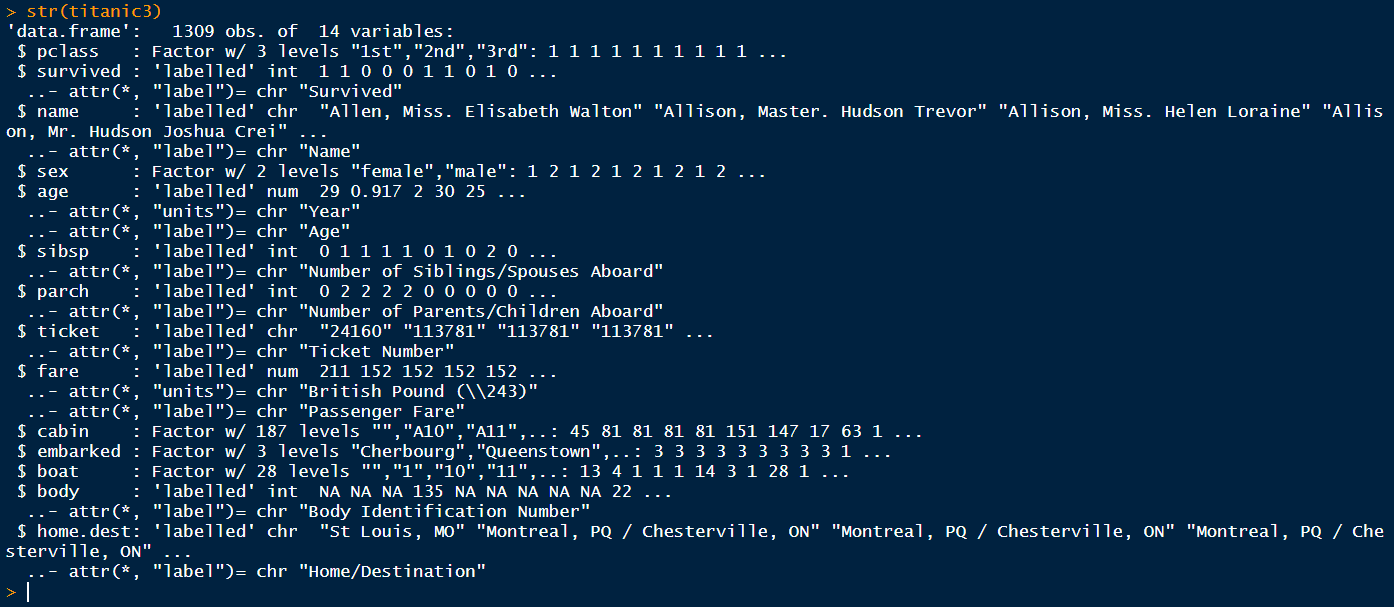
[1] "pclass" "survived" "name" "sex" "age" "sibsp" "parch" "ticket" "fare"

[10] "cabin" "embarked" "boat" "body" "home.dest"

More information on the data.frame: ?titanic3

Use funciton str to describe its structure:

str(titanic3)



Compute frequency tables – note the use of the $ operator to extract variables from the data.frame

table(titanic3$sex)

#female male

# 466 843

#### analysis of titanic3 ####

# note that finishing a line with #### creates an index entry

# at the bottom left corner of the Code Editor pane

table(titanic3$survived)

# 0 1

#809 500

tab1<- table(titanic3$sex, titanic3$survived)

### assign the output of a two-variable table to object tab1

tab1 ### print tab1

# 0 1

# female 127 339

# male 682 161

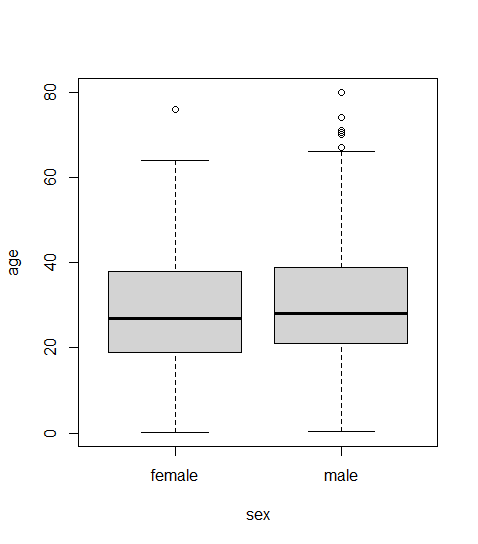
chisq.test(tab1) #### very significant association - why?

# Pearson's Chi-squared test with Yates' continuity correction

#data: tab1

#X-squared = 363.62, df = 1, p-value < 2.2e-16

boxplot(age ~ sex, data=titanic3)



table(titanic3$pclass)

#1st 2nd 3rd

#323 277 709

tab2<- table(titanic3$pclass, titanic3$survived)

tab2 # print table tab2

# 0 1

# 1st 123 200

# 2nd 158 119

# 3rd 528 181

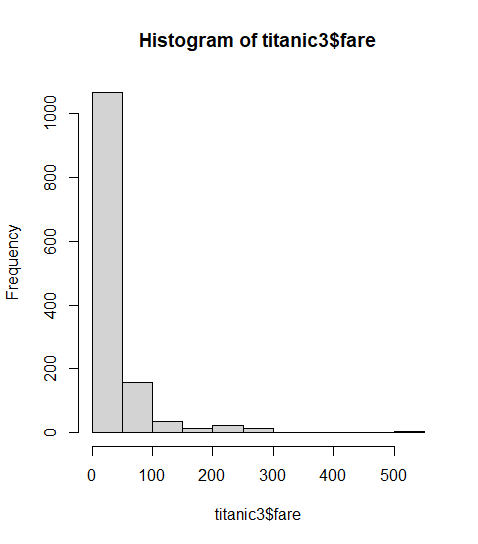
#chisq.test(tab2) ### very significant association

# Pearson's Chi-squared test

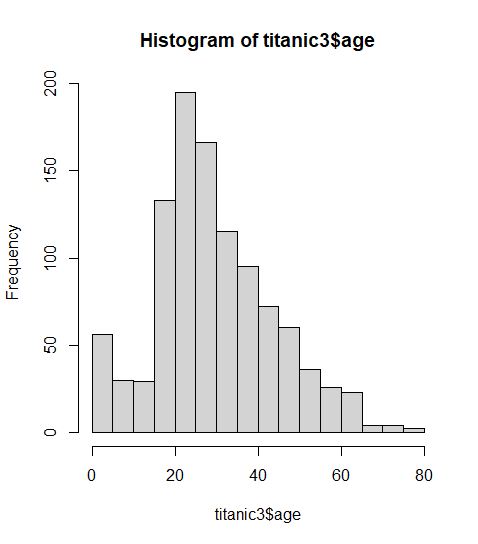
#data: tab2

#X-squared = 127.86, df = 2, p-value < 2.2e-16

hist(titanic3$fare) ### histogram of fares

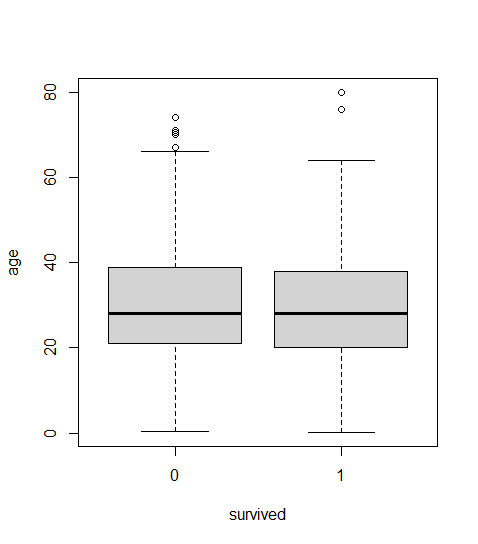


hist(titanic3$age) ### note the mixture of children and adults



boxplot(age ~ survived, data=titanic3)

## not a lot of difference in ages by survivorship



wilcox.test(age ~ survived, data=titanic3)

# Wilcoxon rank sum test with continuity correction

#data: age by survived

#W = 138624, p-value = 0.178

#alternative hypothesis: true location shift is not equal to 0

### p = 0.178, no significant difference in location

### of age by survivorship

mod1<- glm(survived ~ sex + pclass + age,

family=binomial(logit), data=titanic3)

### modelling odds of survivorship by sex, passenger class, and age

summary(mod1)

#Call:

#glm(formula = survived ~ sex + pclass + age,

# family = binomial(logit), data = titanic3)

#Deviance Residuals:

# Min 1Q Median 3Q Max

#-2.6399 -0.6979 -0.4336 0.6688 2.3964

#Coefficients:

# Estimate Std. Error z value Pr(|z|)

#(Intercept) 3.522074 0.326702 10.781 < 2e-16 \*\*\*

#sexmale -2.497845 0.166037 -15.044 < 2e-16 \*\*\*

#pclass2nd -1.280570 0.225538 -5.678 1.36e-08 \*\*\*

#pclass3rd -2.289661 0.225802 -10.140 < 2e-16 \*\*\*

#age -0.034393 0.006331 -5.433 5.56e-08 \*\*\*

#---

#Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

#(Dispersion parameter for binomial family taken to be 1)

# Null deviance: 1414.62 on 1045 degrees of freedom

#Residual deviance: 982.45 on 1041 degrees of freedom

# (263 observations deleted due to missingness)

#AIC: 992.45

#Number of Fisher Scoring iterations: 4

## odds of survivorship are decreased for males vs females

## (reference category = females)

## odds decrease by passenger class (refernce category = 1st)

## odds of survivorship decrease by age: exp(-0.034393) = 0.9662:

## for every year of age, the probability of surviving decreases 4%

## perhaps a linear trend for survivorship vs age is not appropriate

## try quadratic

mod2<- glm(survived ~ sex + pclass + age + I(age^2),

family=binomial(logit), data=titanic3)

### there are better forms of specifying a quadratic term

### but this is the simplest, note the use of function I (identity)

summary(mod2)

#Call:

#glm(formula = survived ~ sex + pclass + age + I(age^2), family = #binomial(logit),

# data = titanic3)

#Deviance Residuals:

# Min 1Q Median 3Q Max

#-2.7161 -0.7018 -0.4232 0.6565 2.4008 #

#Coefficients:

# Estimate Std. Error z value Pr(>|z|)

#(Intercept) 3.7745417 0.3843681 9.820 < 2e-16 \*\*\*

#sexmale -2.4979903 0.1663553 -15.016 < 2e-16 \*\*\*

#pclass2nd -1.2464675 0.2262463 -5.509 3.6e-08 \*\*\*

#pclass3rd -2.2663569 0.2261266 -10.023 < 2e-16 \*\*\*

#age -0.0562897 0.0183243 -3.072 0.00213 \*\*

#I(age^2) 0.0003450 0.0002693 1.281 0.20010

#---

#Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

#(Dispersion parameter for binomial family taken to be 1)

# Null deviance: 1414.62 on 1045 degrees of freedom

#Residual deviance: 980.83 on 1040 degrees of freedom

# (263 observations deleted due to missingness)

#AIC: 992.83

#Number of Fisher Scoring iterations: 4

### The models are nested, so a likelihoood ratio test is OK

anova(mod1, mod2, test='Chisq')

# Analysis of Deviance Table

#Model 1: survived ~ sex + pclass + age

#Model 2: survived ~ sex + pclass + age + I(age^2)

#Resid. Df Resid. Dev Df Deviance Pr(Chi)

#1 1041 982.45

#2 1040 980.83 1 1.6224 0.2028

### Analysis of deviance

### indicates that the goodness-of-fit doesn't significantly

### improved with a quadratic term in age

### so it seems linear trend with age is enough

# Further resources

There are many books and videos to learn the basics of R, for example:

<https://www.r-bloggers.com/learning-r-for-free-free-online-resources/>

An excellent text on Statistical modelling with R is here (though note that it’s rather old and has nothing on tidyverse):

<https://www.academia.edu/34820741/An_Introduction_to_R_Software_for_Statistical_Modelling_and_Computing_Course_Materials_and_Exercises>

I believe using R via the tidyverse is the way to go. However, I don’t think you can start working on the tidyverse without knowing the basics of base R.

Why learn the tidyverse? See this: <https://www.r-bloggers.com/why-learn-the-tidyverse/>

What is the tidyverse? See this: <https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/>

1. Incidentally, the H in Hmisc and getHdata refer to Professor Frank E Harrell, who wrote package::Hmisc [↑](#footnote-ref-1)