Big data in Systems Medicine

# "Introduction into R"



Department of Epidemiology, UMCG, Groningen, the Netherlands

# Contents

* Day 1: Introduction into R
* Day 2: Basic statistics in R
* Day 3: Data visualization in R

# Background

With over two million users worldwide R is becoming the leading software package for statistics and data science. It is freely available and has many utilities and possibilities for e.g. basic and advanced statistical analysis; creating sophisticated graphs; data handling; and writing software. As such R is a very convenient software package since it allows you to create scripts for data handling, analysis, and visualization and to have your results and figures for your paper in one run, which cannot be done with statistical packages like SPSS, SAS, or Stata.

However for many researchers it is not directly clear how to use R, because of the R language and the way of reasoning. We would therefore like to introduce the students to and familiarize them with R. After finishing this introductory R course, you will master some very valuable R skills and will be ready to do your own data analyses.

At this course, you will learn the basics of R through short lectures and many computer exercises. The following topics will be treated: the R language, R variables (objects), R data structures, reading and writing data files, manipulating datasets, making graphs, performing basic statistics, and creating functions.

Have fun!

**Course coordination & organization**:

Dr.Jingyuan Fu & Dr. Serena Sanna;

Department of Pediatrics & Department of Genetics

University Medical Center Groningen

University of Groningen, The Netherlands

**Tutorial Instructors:**

Dr. Alexander Kurilshikov alexa.kur@gmail.com

Drs. Lianmin Chen lianminchen@yeah.net

Drs. Adriaan vd Graaf adriaan.vd.graaf@gmail.com

**Location:**

The course will be given in rooms 3219.0167

**Further information:**

# A R reference card and e-book “Data Analysis for Life Sciences” can be found in Ndestor.

# Acknowledgement:

# Thanks for the help of Dr. Ilja M. Nolte in preparing this tutorial.

# Day 1: Introduction

# Downloading and installing R and R-Studio

R and R-Studio are already installed on the computers in the rooms that are used for this course. However if you want to use R for your research or private work, do the following.

1) Go to <http://cran.r-project.org/> to download R. There you will be asked to click on the operating system that you are using. If you select, 'Windows' you will be confronted with three choices; 'base', 'contrib', and 'Rtools'. You only need the 'base'. Later from within R you can install packages, which allow you to enjoy all the functioning of R. Once you click on 'base' you will be confronted with a page showing (in larger font) a link to 'Download R x.x.x for Windows' where the series of x indicate the current version (e.g. 3.2.0). When you click on the download link, you will be prompted to save the file somewhere on your computer. Save it to your preferred location. Note that this is only the installation file; there is no need to keep it after you have installed R.

2) Install R by double clicking on the installation file. The default options/settings as specified during installation will be fine. The program can very easily be customized after installation.

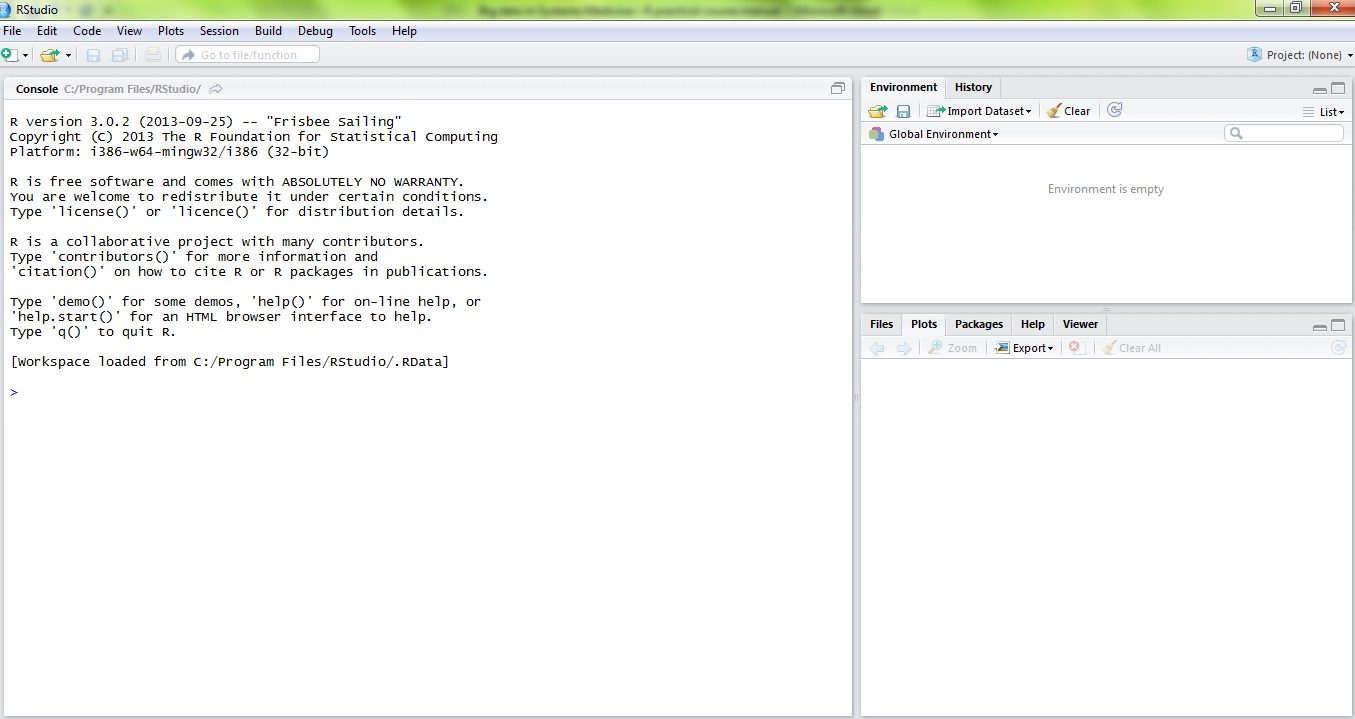
3) Now you could open R by double clicking on the desktop icon or through the menu. You will then be confronted with a window in a window. This is the graphical user interface or console window, which is the core of the program and acts as both input and output display. However, it is preferred to work with R using R-Studio. This is more user-friendly than just R.

4) To download R-Studio go to <http://www.rstudio.com/products/rstudio/download/>. Select from the Installers the operating system that you are using. You can either directly select 'Run' or select 'Save' when you want to save the installation file. Follow the instructions on the screen for installation of R-Studio.

## Let's now start using R!

Today you will be made familiar with the programme R-Studio and with the very basics of the R language.

Open R-Studio by selecting from the Start Menu. You will be faced with a window that is subdivided in three panes (see Figure 1).



*Figure 1: R-Studio on start*

The pane on the left is the **Console**. Here you can type in commands and let R do things for you. With the arrows ↑ and ↓ you can scroll back and forth to previous commands that were executed. On the top right, in the pane with tabs **Environment** and **History** you have an overview of all the variables, objects, datasets, and functions that are present in the working space (tab **Environment**) or you can recall the latest commands (tab **History**). A third pane on the bottom right is the output window and shows the tabs **Files**, **Plots**, **Packages**, **Help**, **Viewer**. The tab **Files** is a file browser; on the tab **Plots** all graphs that you will make during a session will be shown; on the tab **Packages** all packages that have been opened are shown; on the tab **Help** information on the packages, functions, and datasets is provided; on the tab **Viewer**. A fourth pane appears on the top left if you view a dataset or if you use a script.

If you have already worked in R-Studio, it will have saved your last project (*working space*). Upon reopening R-Studio it will automatically open the last working space, scripts and history of commands from your last session. If you don't like this, you can change this in 'Tools' > 'Global options' from the menu.

There are some key terms you will need to become familiar with. First, R is an object oriented system. Anything can be an object; a score, a series of scores (called a vector), a named variable (also called a vector), a matrix (made up of rows and columns), an array (more than two dimensions), a data frame, or a list (a larger group of objects). More details on the different data types will be discussed on day 2. Second, R is an open source program. This means that everyone can add new functions. The addition of new functions is possible through installation and loading of packages. A package provides a mechanism for loading optional code, data and documentation as needed. The R installation already contains ~30 packages by default. During this course we will install and load various packages. It is also possible to create new R packages, but this is a very advanced use of R that most of you will never do. Therefore it is also not part of this course.

## 

## Install and load R packages

You can install packages in multiple ways. First, through the menu: Tools 🡪 Install packages. Or you can install packages from the console:

install.packages(“package name”)

Once you have installed a package, you need to load it before you can use it:

library(“package name”)

Then you can further make use of the package and its functions/variables.

You can remove packages again via the Packages tab (uncheck the check box) or via the function remove.packages(“package name”).

For instance, you can install R tutorial package ‘swirl” and run it the following way:

Install.packages(“swirl”)

Libarary(swirl)

swirl()

## Finding help

From within R you can search for help in multiple ways. For instance if you want to get some help on creating a matrix (one of the data types in R) you can type the following commands in the console: help(matrix), help(“matrix”) or ?matrix. The help page for the function matrix will then appear in the Help tab of the output pane. This help page provides information on how to use the matrix function, which arguments it needs and on the bottom of the page it also shows some examples. Of course, you can also type your question in Google and usually you will find some answers there.

Some useful website about R:

Quick-R: <http://www.statmethods.net/>

CRAN introduction to R: <https://cran.r-project.org/doc/manuals/R-intro.pdf>

CRAN R reference card: <https://cran.r-project.org/doc/contrib/Short-refcard.pdf>

**1. Variable and Operators**

You will learn the very basics of the R language, how to assign values to variables, various data types, and, very importantly, how to search for help. In this manual commands that can be given in R, are indicated in Consolas 10 font. You can try the commands to see what happens.

First create a new object called 'x'. Our object x will initially be something as simple as an individual score, say 5. To communicate this in R, type one of the following commands in the console and hit the enter key:

x<-5

x=5

Notice how both = and <- refer to the same operation and the presence of spaces before and after are not necessary. The operation = and <- perform is assignment; in other words, we have assigned 5 to x. These two commands are the most used ones to assign a value to a variable. It can also be done in the following ways

assign("x",5)

or

5->x

but these are hardly ever used.

Commands can obtain spaces : R will internally 'delete' those :

x <- 5

x = 5

It is up to you to decide upon your own style, i.e. whether you prefer spaces or not.

To see what x is type in the console:

x

or

print(x)

Alternatively, you can look in the right upper pane 'Environment' under **Values**.

By typing x <- 5 the variable x will automatically become a numeric value in R, that is a real number. In order to reduce used memory it is also possible to explicitly tell R that this number is an integer. To do so, you have to add L after the number, e.g.

x <- 5L

Commands can be combined in one line using the separator ';'. E.g. the command

x <- 5; print(x)

will assign the value 5 to x (first command) and print it (second command).

*Question 1: Check out the command*

*(x <- 5)*

*What is the consequence of putting brackets around an assignment?*

*Question 2: Check out the commands*

*x <-*

*5*

*What is the consequence of a hard entry in the command?*

*Hint: note that the prompt changed after the first command.*

## Arithmetic operators

Now we can perform simple arithmetic or complex algebra using our assigned value for x. For example,

x+3 (addition)

x-3 (substraction)

x/6 (devision)

x\*6 (multiplication)

x^2 (exponentiation)

x\*\*2 (exponentiation) x\*\*2 == x^2

x^1/2

x^(1/2)

You can assign the outcome to a new object by e.g.

y <- x+3

and with y or print(y)tell R to show you what the value of the new variable is. Or you can do the assignment and let R tell you the answer in one command by typing

(y <- x+3)

or

print(y <- x+3)

## Logical operators

In addition to arithmetic operators you can also apply logical operators to the objects. For instance,

x<5 (smaller than)

x<=5 (smaller than or equal to)

x>5 (larger than)

x>=5 (larger than or equal to)

x==5 (equal to)

x!=5 (not equals)

! (x==5) (not)

x>3 & x<10 (and)

x==5 | x==6 (or)

Like before it is also possible to assign the outcome to a new variable. This variable would then be of the logical format and can only take values of TRUE or FALSE. E.g.

y <- (x==5)

You can also directly assign a value of TRUE or FALSE to a variable:

y <- FALSE

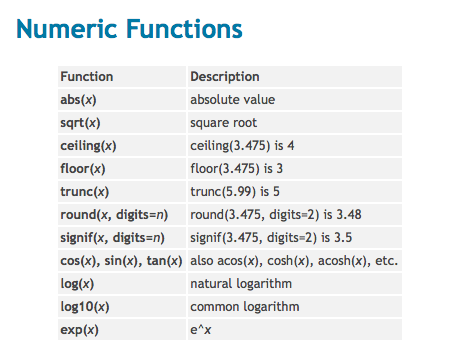
or

y <- F

Instead of TRUE or FALSE, R also recognizes the first letters T and F, respectively.

## (Predefined) Functions

There are also many predefined functions in the base library. For example,



Also the outcomes of the function can directly be assigned to a variable, e.g.

f <- sqrt(x)

### String variable Operators

Apart from numbers and logical values (TRUE and FALSE) R can also handle character and string values. You can store them in a variable, just as numbers, but you need to use quotation marks:

> x<- “Hello”

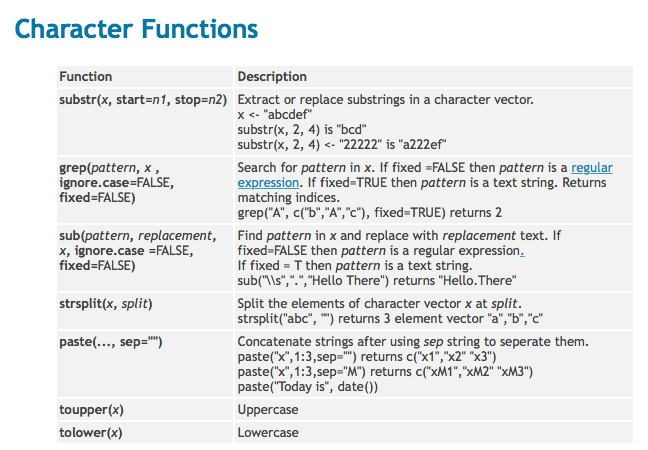
> y <- “Maria”

You can also join two strings by using the paste() command:

> paste(“Hello”,”Maria”, sep= “)

[1] “Hello Maria”

You can also extract or replace a substring, search for a pattern, split the strings, etc.



*Try: What happens when you convert x to a numerical vector?*

*What happens when you convert y to a numerical vector?*

**>>>>>>>>>>>>>>>>>>> Exercises 1: Operators**

**2. Data Structures:**

There are five basic data structures that are available in R.

|  |  |
| --- | --- |
| Type | Description |
| **Vector** | A collection of values that all have the same data type. The elements of a vector are all numbers, giving a numeric vector, or all character values, giving a character vector.  A vector can be used to represent a single variable in a data set. |
| **Factor** | A collection of values that all come from a fixed set of possible values. A factor is similar to a vector, except that the values within a factor are limited to a fixed set of possible values. A factor can be used to represent a categorical variable in a data set, e.g., gender, case-control, etc |
| **Matrix** | A two-dimensional collection of values that all have the same type. The values are arranged in rows and columns. |
| **Data frame** | A collection of vectors that all have the same length. This is like a matrix, except that each column can contain a different data type.A data frame can be used to represent an entire data set. |
| **List** | A collection of data structures. The components of a list can be simply vectors--similar to a data frame, but with each column allowed to have a different length. However, a list can also be a much more complicated structure. This is a very flexible data structure. Lists can be used to store any combination of data values together. |

## Vectors in R

As mentioned earlier, R can handle a lot of data structures. So far we used objects that contained a single value. You can create vectors, which are collections of values, by e.g.

age <- c(18, 21, 22, 43, 23, 54, 60, 20, 33, 72)

The c() function is used to concatenate values and yields a one-dimensional object. Type print(x) to see the result of this command. With the function length(), you can find out the length of the elements in a vector.

> length(age)

[1] 10

Elements can be extracted from a vector by using the index between square brackets:

age[1] is the first element of the vector and is equal to 18.



You can also change an item in a vector by referring to the index number:

> age[2] <- 30

> age

[1] 18 30 22 43 23 54 60 20 33 72

You can get part of the vector by using the “:” operator:

> age[1:3]

[1] 18 30 22

There are many handy functions that can help you create or manipulate vectors:

c(3:6) Creates a range of numbers: (3,4,5,6)

c(10:1) Creates a range of numbers in reversed order

seq(0,10,0.5) Generates a range of numbers between 1 and 10, with steps of 0.5

rep(5,10) Repetition of 10 times the number 5

sort(c(3,6,7,1)) Sorts a vector

You can also make calculations directly on all items in a vector:

x <- c(2,5,3,6,9)

x\*2

[1] 4 10 6 12 18

log(x, base=2)

[1] 1.000000 2.321928 1.584963 2.584963 3.169925

You can use the logical operators within a vector to obtain the elements meet a specific condition or not:

age[age>30]

[1] 43 54 60 33 72

age[age>20 & age<50]

[1] 21 30 43 23 33

With the function which() you can extract the indices of the elements that yield TRUE when extracting elements using a logical vector or operation.

which(age>50)

You can add two vectors, as long as they are of the same length:

x <- c(2,5,3,6,9)

y <- c(10,8,4,5,6)

x+y

[1] 12 13 7 11 15

Two vectors can be joined by the concatenate command:

c(x,y)

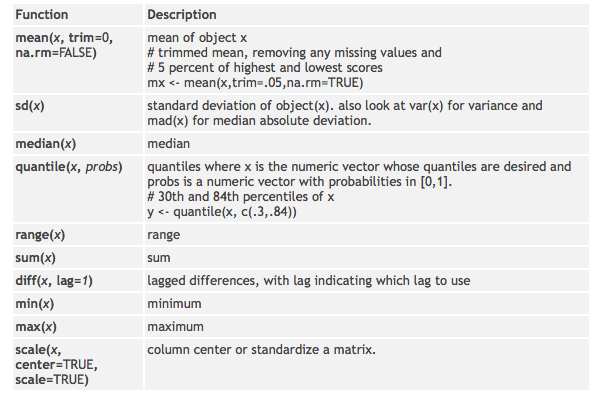
[1] 2 5 3 6 9 10 8 4 5 6

Useful functions for logical vectors are any() and all(). With the function any() you can test if at least one of the elements of a logical vector is TRUE. With the function all() you can test if all elements of a logical vector are TRUE.

all(age>18)

any(age<20)

**Descriptive statistics**



## Factors

A factor is a vector object used to specify a discrete classification (grouping) of variables. For instance, for a sample of individuals we can create the following vector

s <- c("male", "female", "male", "male", "female", "female", "female")

By default this is a vector of character strings. It can be converted into a factor with two levels (male or female) with the command

gender <- factor(s)

*Try typing*

*print(s)*

*print(gender)*

*and check out the differences.*

Sometimes the levels of a factor will have a natural ordering that we want to record. For instance, for the variable

obeseGroup <- factor(c("lean", "fat", "normal", "fat", "normal", "lean", "fat"))

'lean' means less weight than 'normal', while 'fat' means more weight than 'normal'. This can be done by specifying the order of levels in the factor() function.

obeseGroup <- factor(obeseGroup, levels=c("lean", "normal", "fat"))

It may be convenient to define a variable as an ordered factor, because statistical analyses can make use of it.

Numeric factors can be converted to factors using the function cut(). If for instance body mass index (BMI) is measured of a sample of individuals

bmi <- c(18.7, 23.4, 21.5, 28.9, 24.6, 19.9, 26.5)

we can convert it into categories BMI<=20, 20<BMI<=25, and BMI>25 with the command

bmiGroup <- cut(bmi, breaks=c(0,20,25,Inf[[1]](#footnote-1)))

bmiGroup <- cut(bmi, breaks=c(0,20,25,Inf1), labels=c(‘underweight’, ‘normal’, ‘overweight’))

**>>>>>>>>>>>>>>>>>> Exercise 2: >>>>>>>>>>>**

**Matrix:**

R can also deal with multi-dimensional objects. For instance, you can create two-dimensional matrices in R with the command matrix(). With the options ncol= and/or nrow= you can specify the number of columns or rows, respectively.

A matrix can be constructed in many ways. For instance like this:

values <- matrix(c(2,1,6,1,5,7), nrow =3, ncol=2)

values

>   [,1] [,2]   
[1,]    2    1   
[2,]    1    5   
[3,]    6    7

As you can see, a matrix is created from a vector of numbers, containing 3 rows (nrow) and 2 columns (ncol). You can use dim() to retrieve the dimensions of an R object, e.g., matrix or data frame.

To extract a single value from a matrix, we need to specify the column and the row number: matrix[row,col]

values[3,1]

[1] 6

If you leave out the column, you can get the entire row and vice versa:

values[3,]

[1] 6 7

values[,1]

[1] 2 1 6

Other handy functions to apply on matrices are: transpose(),to switch rows and columns, and rownames() and colnames() to set and retrieve names for your rows and columns, respectively.

Another way to create a matrix is by first creating separate vectors and then combining them either by columns or rows using the functions cbind() or rbind(), respectively. You need to check the length of the vectors carefully, because if one vector is shorter than the other R will repeat the values of the shorter one till it has the same length as the longer one.

Example of cbind():

x<-c(1:5)

y<-c(6:10)

z<- cbind(x,y)

## Data frame

For matrices all elements of the object need to have the same format. However sometimes it is convenient to have columns of different formats. Data frames can deal with this issue.

Data frames can be created from scratch by combining vectors:

age <- c(25, 30, 56)

gender <- c("male", "female", "male")

weight <- c(80, 55, 70)

mydata <- data.frame(age,gender,weight)

*Try: You can use class() to find out the structure type of mydata.*

*Try: You want to add a new data entry for a female of age 31, who weighs 65 kg. You could do this by binding the vector c(31, "female", 65) as a row to the data set mydata using the function rbind(). What is wrong with this approach?*

*Hint: check the classes of the variables in the data frame.*

A correct way to add rows to a data.frame is to create a new data frame for which the column names are the same to those of the original data frame. To do this you have to specify the column names when creating the data frame with the command.

extrarow <- data.frame(age=31, gender="female", weight=65)

*Try: Now try again to add the extra row to the data frame using rbind(). Does this give the correct result?*

The column names of a data frame can be changed with the function colnames() or names(). Similarly row names can be adapted with the function row.names().

*Change the column names of mydata to ("AGE", "GENDER", "WEIGHT") and the row names to ("Bob", "Mary", "Jason", "Susan").*

Like with matrices and arrays certain elements can be extracted from the data frame using either

mydata[3,1]

or

mydata["Susan", "GENDER"].

*Try: What are the data values for Jason? And what are the weights of all individuals?*

*Try: What command would you use to select the female individuals from the data frame mydata?*

The columns of a data frame can also be extracted by the following command

mydata$WEIGHT

This is shorter variant of mydata[, "WEIGHT"]. It should however be noted that this shorter notation is less error proof. E.g. the command mydata$Weight does not give an error – simply returns an empty vector NULL, while mydata[, "Weight"] does. An advantage of the $ notation is however that it is easy to add new columns to the data frame. Assigning values to e.g. mydata$newcolumn will append a column with the name 'newcolumn' to the data frame mydata.

## Lists

Another class of data is a list. This is an object consisting of an ordered collection of objects known as its *components*. There is no particular need for the components to be of the same mode or type. For example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. An example of a list is

mylist <- list(name="Fred", wife=c("Catherine"), parents=c("Patrick", "Corney"), no.children=3, child.ages=c(4,7,9))

This list consists of five components, two of them are character vectors of length 1, one character vector of length 2, one numeric vector of length 1, and one numeric vector of length 3. The various components of list can be extracted from the list using double square brackets. Like before, either indices or names can be used or the $ notation for extracting a component.

*Try: Try the following and see what happens:*

*mylist[[1]]*

*mylist[["wife"]]*

*mylist$no.children*

*mylist[["child.ages"]][1]*

*Try: Calculate the sum of the ages of Fred's children.*

## >>>>>>>>>>>>>> Exercises 3 >>>>>>>>>

## Use of scripts

So far we have typed commands in the console window. In practice it is most useful to use *scripts*, which are text files (usually with the extension .R) in which the commands are stored. These text files can be saved to hard disk and hence the use of scripts makes it easier to repeat an analysis that you did before or to adapt it. Also for quality control issues, it is advised to use scripts since you can then quickly show what analyses you have done. Finally, use of scripts will make life easier for successors or other colleagues, if you move jobs.

So, as mentioned a script in a text file with commands. You can open a new script in R-Studio using the menu. Click on 'File' > 'New File' > 'R Script'. A new pane will then be opened in the upper left side of the window.

In this window you can type in commands like you did in the console window. You can also import commands from the 'History' tab from the right upper pane.

A convenient tool in scripting is to add comments to the script to explain the command(s). This can be done by putting a # in front of the comment. This does not necessarily have to be at the beginning of a line, it can also be put after the command.

It is advised to add comments. While you are working on a script, it often clear to you what each command means. However when you have to repeat or adapt the analyses after e.g. half a year because you need to revise it for your paper, or when a colleague wants to use the same script, it will often turn out to be rather difficult to figure out what you have done. Therefore we recommend to make a habit out of it to add comments while you create a script. It may not be necessary to comment on each line.

Once you have created a script, you can rerun your analysis and import variables, data, and functions (which we will discuss on day 6) from this script into your current workspace using the function source()[[2]](#footnote-2).

# Day 2 Data handling and basic statistic analysis

Yesterday we manually created data sets (data frames) in R, but it is of course also possible to import data from files.

## Working directory

When you want to import data, it is convenient to change the working directory to the directory where the data are stored. To see your current working directory, type getwd(). If you want to change the working directory to –say- "C:\Rcourse", you type

setwd("C:\Rcourse")

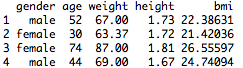
Note: R uses a / instead of \ (as is the common practice on MS-Windows systems) when describing the location of the data!

Within R-Studio, changing the working directory can also be achieved by using the menu "Session/Set Working Directory/" on the top of the screen or by scrolling to the right directory in the tab **Files** in the right lower pane and click on 'More' > 'Set as working directory' in the menu of this pane. With the command dir() in the console window the content of the current working directory is shown.

*Try: Find out your current working directory, and change it when necessary. Inspect if you are in the right directory by checking if the files "mydata.txt" is present.*

## Importing datasets from text files

Reading an entire data frame from an existing external file can be done by using the function read.table(), which will create a data frame*.* The input file can be a text file have the following format:



The data from the file "mydata" can be read in R using the following command:

mydata=read.table("mydata.txt", sep='\t', header=T)

By specifying header=TRUE R assumes the first line to be a line of headings.

When you read in data from a file, it is convenient to first inspect what kind of data you have read in. For a compact display of a data set you can use the function str()*.* Functions head()and tail()can be used to print only a part of a (large) dataset.

You can explicitly assigning row names to a data.frame by adding the argument row.names to the read.table() command*.*

When R reads in character data from a file, it will automatically convert them to factors. If you don't want that, you must add the option as.is=TRUE or stringsAsFactors=FALSE to the read.table() function.

Exporting data sets can be done by the using the write.table()command:

write.table(mydata, "mydata\_txt.txt", sep='\t')

This command will write the current data frame *mydata* as a text-file (here called "mydata\_txt.txt") in your current working directory. As a default, it uses white space as a separator between the various elements. As with read.table(), this function has various arguments which can be used to alter these settings. Note that the first row contains one less element than the other rows, if you also export row.names (row.names=T in write.table()). The first column with the row labels is however not required. R can read in data files with either the row label column present or absent.

## Importing datasets from other formats

Data from other formats can also be imported into R. The package **foreign** provides this option for SPSS, Stata, and SAS. You can read and write spss file using read.spss() and write.foreign().

# Statistical analyses

For the exercises of today we presume some familiarity with statistical methodology. We will not go into detail on the statistical theory, but focus on how various statistical analyses can be done in R. You will learn some basic statistical analysis in R, including t-test, anova, linear regression, etc. You will also learn how to apply linear regression to correct the effect of covariant.

## T-test

For situations in which we want to test whether an observed mean value from an experiment (assuming the underlying distribution is normal) equals a specific value (the null-hypothesis) we perform a one-sample t-test with n-1 degrees of freedom (n representing the sample size). This can be done by using the t.test() function in R, which offers a variety of t-tests.

Let's consider our data (mydata). If we want to test whether the mean BMI in the sample is significantly different from 25 (the cutoff o, we type:

t.test(mu=25, x=mydata$bmi)

In doing this, we are of course assuming a underlying distribution that is normal. This is the output:

One Sample t-test

data: mydata$bmi

t = 0.085842, df = 79, p-value = 0.9318

alternative hypothesis: true mean is not equal to 25

95 percent confidence interval:

24.21624 25.85441

sample estimates:

mean of x

25.03532

The output provides the test statistic (0.085), the degrees of freedom (79=n-1) and the p-value of performing a two-sided t-test (0.9318). In this case, we would not reject the null hypothesis ("mean BMI is 25") using significance level 0.05. As you can see, you have to deduce this conclusion yourself from the given output. R also provides the 95%-percent confident interval [24.21624; 25.85441] and the sample mean estimate for girth (25.03532).

You can also test whether mean height is significantly different between male and female.

female.height = mydata$height[mydata$gender=="female"]

male.height = mydata$height[mydata$gender=="male"]

t.test(female.height, male.height)

This is the output:

Welch Two Sample t-test

data: female.height and male.height

t = -6.7705, df = 75.188, p-value = 2.469e-09

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-0.14473432 -0.07892783

sample estimates:

mean of x mean of y

1.695745 1.807576

Try: Is the height significant different between males and females?

You can assign the statistics result to an object, say t.result

t.result <- t.test(female.height, male.height)

You can use mode(t.result) function to find out t.result is a “List. Using names(t.result), you can retrieve the names of elements. Then you are retrieve specific value to export. For instance, t.result$p.value, you will export the p value of the t-test.

***ANOVA test***

If you want to compare the difference of mean values among multiple groups, you can use ANOVA test. We want to examine whether the mean value of height differs among the different weight groups by ANOVA. The resulting model is called "mod1":

mod1 <- aov(height~group, data=mydata)

The tilde ~ must be read as the phrase "is modeled as a function of''. In general, the basic format for function notation in R is

response variable ~ explanatory variables

We will encounter this function notation (and various formula options for the right hand side) later on when introducing functions lm(): linear model

Summarizing this model with summary(mod1) results in:

Untitled:Users:FuJ:Desktop:Screen Shot 2017-03-09 at 3.31.03 PM.png

According to the analysis, there is not significant (p-value=0.974) difference between the group means at a 5% significance level. The command

The distribution-free equivalent of the ANOVA is the Kruskal Wallis rank sum test: kruskal.test().

*Try: You can try Kruskal.test to compare the difference of height between different weight groups. Do you see the similar result as ANOVA test?*

## Statistical tests for groups (discrete data)

For discrete data, we already saw that summarizing the table-function presents us with the outcome of a chi-square test of independence of the row and column variables of the table. The Chi-Square test of Independence is used to determine if there is a significant relationship between two nominal (categorical) variables. The frequency of one nominal variable is compared with different values of the second nominal variable. Try the following:

x <- c(0,1,1,0,1,1,0,0,0,1)

y <- c(1,1,0,0,1,1,0,0,0,1)

table(x,y)

summary(table(x,y))

Another way to achieve the same result, is by typing

chisq.test(x,y, correct=FALSE)

This will give a chi-square test without continuity correction[[3]](#footnote-3) (due to the argument correct=FALSE). An exact test of independence (which may be required in case of a small sample size) is provided by fisher.test().

Now you can test the distribution of weight group is different between males and females.

chisq.test(mydata$gender, mydata$group)

Note you may see a warning message. It gave the warning because many of the expected values are very small and therefore the approximations of p may not be right.

## Correlation, linear models

You can use cor() to calculate Pearson's correlations between two continuous variables and the function cov() to determine the covariance. The function cor.test() produces the hypothesis test and p-value for testing whether the correlation is different from 0 . Try the following:

cor(mydata$age,mydata$bmi)

cor.test(mydata$age,mydata$bmi)

*Type ?cor.test to figure out how to perform a Spearman-correlation test for the non-parametric version of the above correlation test.*

**Linear Model**

A linear model is fitted by using the function lm() as follows:

modelname <- lm(formula, data=mydata)

As we saw before, the formula part depicts the way that we let the outcome variable (dependent variable, *y*) on the left-hand side of the tilde ~ depend on the predicting or explanatory variables (*x1, x2, x3*,…) on the right-hand side of the tilde ~.

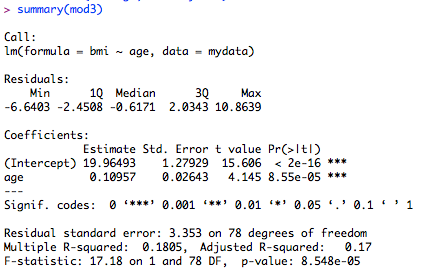
Some examples:

|  |  |  |
| --- | --- | --- |
| **R-Formula** | **Corresponding statistical model:** | |
| y ~ x1  y ~ x1 + 1 | *y = b0 + b1∙x1 + ε* | simple linear regression model of *y* on *x1,* with intercept |
| y ~ 0 + x1  y ~ x1 -1  y ~ -1 + x1 | *y = b1∙x1 + ε* | simple linear regression model of *y* on *x1,* intercept fixed at 0 |
| y ~ x1 + x2 + x3 | *y = b0 + b1∙x1 + b2∙x2 + b3∙x3 + ε* | multiple linear regression model of *y* on *x1, x2 and x3,* with intercept |

The following command fits a simple linear regression model (called "mod2") for the trees data with *volume* as outcome variable and *height* as predictor variable:

mod3 <- lm(bmi ~ age, data=mydata)

The fitted model can be examined by typing summary(mod3), which yields the following output:



From the output, we conclude that the regression coefficient of height is positive, and estimated to be 0.10957. It is significantly different from 0 at 5% level, with p-value equal to 8.55e-05(NB: these numbers must look familiar, as we have encountered this test earlier when examining the correlation between these variables!). We further conclude that the explained variance in *BMI* by variable *age* is about 17%.

# Linear model is also used to other related variables or confounding effects. For instance, the large-scale study of genetic study on BMI has described “*After regression on age and sex and inverse normal transformation of the residuals, we carried out association analyses with genotypes or imputed genotype dosages.*” (Locke et al. Nature 2015). The aim of the study is to investigate the effect of genetics. Thus the effect of unrelated factors (age, sex) on BMI was firstly regressed out and the analysis power of genetic analysis is increased. The linear model is often used in this case and the residuals can be retrieved.

mod4<-lm(bmi~age+gender, data=mydata)

bmi.res <- resid(mod4)

The object bmi.res contains the residuals values after correcting the effect of age and gender.

# >>>>>> Exercise 4 >>>>>>>>>>

## Loops

In big data analysis in life science, we normally need to assess hundreds of thousands parameters simultaneously. For instance, human transcriptome contains over 45,000 transcripts. A transcriptome-wide analysis means that we need to repeat the same analysis for 45,000 genes. Such a repetitive execution can be handled in a loop.

This is easily done with a for() or a while() loop. As the name implies, these loops are written to repeat code (‘loop’) for a certain number of times. The for loop repeats an action **for** every instance.

Usage:

for (i in range){

action

}

For example:

for (i in 1:5){

print(i)

}

Output:

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

# The value of i starts from 1 and increases by 1 at each run. The loop stops when the value of 1 reaches to 5.

Another type of loop is the 'while loop', which will execute a command as long as a certain condition is met.

Usage:

while (condition){

action

}

For example:

x <- 0

while(x < 5) {

print(x)

x <- x+1

}

Output:

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

If this case, the initiative value of x is 1. At the each run, the value of x is changed, increasing by 1. The loop is continuing until the condition (x<5) is not true.

At a loop, you can use c(), rbind() or cbind() to save the result of each analysis. For instance, using mydata, we want to perform t-test and assess the difference of age, weight, height and bmi between male and females. We need to record and write out the summary statistics and P values for each analysis. Try to understand the following code and run it:

all.result = NULL ## create an empty object called result

female.n = which(mydata[,1]=='female')

male.n=which(mydata[,1]=='male')

for (i in 2:ncol(mydata)) {

temp = t.test(mydata[female.n,i], mydata[male.n, i])

t.result = data.frame(factorName=colnames(mydata)[i], Mean\_female=temp$estimate[1], Mean\_male=temp$estimate[2], p.value = temp$p.value)

all.result = rbind(all.result, t.result)

}

Now check the content of all.result. What do you have? Then you can export the all.result to a file using write.table().

## Writing basic functions

In addition to using the predefined functions from the R base distribution, the R language also allows you to create your own functions. It is very handy when you need to repeatedly run many analyses. You can include these analysis steps into a function. This is achieved by using the *function* keyword followed by an optional list of function arguments. For example, consider the function *area* that calculates the area of a rectangle with length *l* and width *w*:

area <- **function**(l,w) {

l\*w

}

Here, the variables *l* and *w* are called the function arguments, and the code between the curly brackets is called the function body. The function body may consist of a single expression, as in the above example, or of a group of expressions, in which case the value of the last expression is the value returned for the function. To be more explicit, one could also write this last expression as

return(l\*w)

User-defined functions are called in the same way as any other function. For example, to use our newly defined function to calculate the area of a rectangle with length 6 and width 5, we simply type

area(6,5)

or

x <- area(6,5)

to assign the results to the object x, so that it can be used for later reference.

**Apply functions**

With functions, you can using apply() to apply functions to each row or each columns of matrix. In this way, the functions will loop through the rows or columns. It is very similar to a *for* loop.

For instance, we can re-run the t-test above to assess the difference of age, weight, height and bmi between male and females.

gender = mydata$gender

data = as.matrix(mydata[,-1]) ## make a matrix. Apply() works better for a matrix

myfunction = function(x, y=gender) {

temp = t.test(x[y=='male'], x[y=='female'])

return(data.frame(Mean\_female=temp$estimate[1], Mean\_male=temp$estimate[2], p.value=temp$p.value))

}

result2 = apply(data, 2, myfunction, y=gender)

Would you check the data structure of result2? Is it a matrix, data frame or a list?

Yes, result2 is a list. You can call rbind() function to bind all elements of a list into a data.frame.

result3 = do.call(rbind, result2)

## Conditional execution

A control statement is a statement that determines that analysis or action will be executed only at a certain condition. For example, suppose that a person with age *a* intends to buy a bottle of wine at the supermarket. The following code returns a different message depending on whether or not he/she is allowed to buy any alcohol:

**if** (a>=18) {

cat("Thank you for your business")

} **else** {

cat("Really? You should know better")

}

The expression between parentheses after the **if** keyword must be a logical expression that is either TRUE of FALSE. If the value of this expression is TRUE, the expressions between the curly brackets after the **if** keyword are executed. Otherwise, the expressions between the curly brackets after the **else** keyword are executed. The else clause is optional and may thus be omitted.

**>>>>>>>>>>>>>>>> Exercise 5 >>>>>>>>>>>**

**Day 3: Data Visualization**

One of the main reasons data analysts turn to R is for its strong graphic capabilities. Today you will learn about different graphs that can be created with R, about adjusting the graphs, and about how to export them.

## Scatterplots

One way to make a scatterplot for comparison of two continuous variable is plot(). As mentioned on Thursday, what kind of graph is created with this function depends on the input. If the input consists of one or two vectors, a matrix, an array, a data frame, or a list, a simple scatterplot will be created.

*Try: Create the following variables*

*x <- rnorm(10)*

*y <- runif(10)*

*z <- matrix(rnorm(20), ncol=2)*

*and try out the following commands*

*plot(y)*

*plot(x,y)*

*plot(z)*

As you may have noticed, the plots appear in the output pane on the lower right side of the window. You can scroll back and forth through all the plots that you created during the session with the arrow buttons in the quick menu of the **Plots** tab (see Figure below). With the red cross button a single plot can be removed and with the 'broom' button all plots can be deleted (also see Figure).



*Figure 4: quick menu of the Plots tab in the output pane of R-Studio*.

The function plot() is very extensive. It allows you to change the titles of the figure and the axes, the type of the plot, the type of the dots or lines, the color of the dots or lines or the background color, the boundaries of the axes, the font or the size of the titles or labels, or the size of the dots or the line width, or print the values on a log-scale. Check ?plotto look at all possibilities.

*Try: load the mydata.Rdata and make a scatterplot for age and BMI.*

load('mydata.Rdata')

plot(mydata$age, mydata$bmi)

plot(mydata$age, mydata$bmi, xlab='Age', ylab='BMI')

You can change the color and the size of dots.

plot(mydata$age, mydata$bmi, xlab='Age', ylab='BMI', col='red', cex=2)

You can also use different shapes. Try ?pch to find different dot patterns. For instance, we plot the dots with a fill blue diamond.

plot(mydata$age, mydata$bmi, xlab='Age', ylab='BMI', pch=18, col='blue')

The colors and type of dots can also be changed per dot. In that case you have to add a vector as input for the option, e.g. col=c("red", "blue") will make dots 1,3,5, etc. red and dots 2,4,6, etc. blue.

Now, we want to specify blue color for male samples and red color for female samples

genderColor = rep('red', length=nrow(mydata))

genderColor[which(mydata$gender=='male')] = 'blue'

plot(mydata$age, mydata$bmi, xlab='Age', ylab='BMI', pch=16, col=genderColor)

The plot() function by default uses dots, but other types of graphs can also be created. For instance a plot with lines. Try to run the following scripts.

par(mfrow=c(3,1))

plot(mydata$age, type='l')

plot(mydata$age, type='p')

plot(mydata$age, type='b')

## You probably have noticed the function par() above. It is used for adjusting graphical parameters. Check the help page (?par) to see the long list of graphical parameters that can be set or queried. Settings that I sometimes change when creating graphs are the margins around the plot area, the distance between axis labels and titles, the number of graphs in one plot, the font and dot size. In the example above, we use par(mfrow=c(3,1)) to display 3 graphs in one plot and place them in 3 rows, one plot per row.

When a graph is created, other features can be added upon this graph, like

* text (with the function text());
* a legend (with the function legend());
* a straight line (with the function abline());
* points (with the function points());
* lines connecting multiple points (with the function lines());
* multiple straight lines (with the function segments());
* polygons (with the function polygon());
* a (sub)title if not already provided with the plot() function (with the function title());
* axes if switched off in the plot() function (with the function axis()).

Now, we can add a legend in the upper left corner of the plot for the color of gender.

plot(mydata$age, mydata$bmi, xlab='Age', ylab='BMI', pch=16, col=genderColor)

legend('topleft',legend=c('male', 'female'), pch=16,col=c('blue', 'red'), bty="n")

It is very often that we need to add the regression line. You can use the following command:

plot(mydata$age, mydata$bmi, xlab='Age', ylab='BMI', pch=16, col=genderColor)

abline(lm(bmi~age, data=mydata))

Scatterplot matrices can also be produced with the function pairs(), which has more options than the plot() function, for instance to show multiple pair-wise scatter plots. Let’s try to make pair-wise plots for age, weight and BMI.

pairs(mydata[,c('age', 'weight', 'bmi')], col='green4', pch=16)

**Histogram and density plots**

A histogram is a graphical representation of the distribution of numerical data. A histogram is a graphical representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable (quantitative variable). To construct a histogram, the first step is to "bin" the range of values—that is, divide the entire range of values into a series of intervals—and then count how many values fall into each interval.

hist(mydata$age, main='Histogram of Age')

You can specify the color and number of breaks.

hist(mydata$age, breaks=12, col="red")

The distribution of a numerical data can also be presented as a density plot.

d <- density(mydata$age) # calculate the density

plot(d) # plot the density

polygon(d, col='lightblue') ## color the area under curve

**Boxplot**

A boxplot is a graphical summary of the distribution of a sample that shows its shape, central tendency, and variability. It is particularly handy to compare the distribution of data between different groups.

For instance, we now compare the distribution of body weight between different gender groups.

boxplot(weight~gender, data=mydata)

**Bar plot**

A bar chart or bar graph is a chart or graph that presents grouped data with rectangular bars with lengths proportional to the values that they represent. Say we can plot the mean of male and females.

man.avg = mean(mydata$weight[mydata$gender=='male'])

woman.avg = mean(mydata$weight[mydata$gender=='female'])

barplot(c(man.avg, woman.avg), names.arg=c('Man', "Woman"), ylab='Weight (kg)')

you can also make multiple barplots. For instance, here we make a barplot to compare the mean value of age, weight, height and BMI between man and woman.

man.data = as.matrix(mydata[which(mydata[,1]=='male'),-1])

female.data = as.matrix(mydata[which(mydata[,1]=='female'),-1])

man.avg = colMeans(man.data)

female.avg = colMeans(female.data)

data = rbind(man.avg, female.avg)

rownames(data) = c('Man', "Woman")

barplot(data, beside=T, col=c('green4', 'gray'))

**Pie chart:**

Another useful graph is the pie chart (pie()) to visually present a proportion data. Say the proportion of different weight groups. On day 1, you have learned to classify individuals based on value. Say, we now classify individuals based on

bmiGroup = cut(mydata$bmi, breaks=c(0,25, 30, Inf), labels=c('normal', 'overweight', 'obese'))

count = table(bmiGroup)

pie(count)

## Exporting graphs

As mentioned before the plots that you have created during the session are shown in the output pane. From this pane, you can export a plot and save it as an image (various format are possible), save it to pdf, or copy it to the clipboard. It is however also possible to give commands in the console window (or in a script) that directly export the plot to file, without showing it first in R-Studio. Plots can be saved in bmp, jpg, tiff, wmf, ps, or png bitmap format or to a pdf file with the following commands, respectively

bmp(filename = "plot.bmp",…)

jpeg(filename = "plot.jpg",…)

png(filename = "plot.png",…)

tiff(filename = "plot.tif",…)

win.metafile(filename = "plot.wmf", …)

postscript(filename = "plot.ps", …)

pdf(filename = "plot.pdf",…)

*Try to export the pie plot you made*

*jpeg(filename = "piePlot\_bmiGroups.jpg")*

pie(count)

*dev.off()*

*NB. Make sure that you set the working directory to a directory where you have permission to write !*

In this case dev.off() also ensures that the file will be properly closed. If you have performed multiple attempts to create the jpeg-file, R will have created multiple graphical devices and hence you may have to repeat the dev.off() command to make sure that every graphical device is closed. It might be that because of various attempts the graph you are interested in, is not closed properly and you may have to recreate the graph after making sure that all plots have been closed properly and your command lines are correctly creating the graph.

Various additional parameters like width, height, resolution can be set to create a graph to your own (or editors) liking.

**Corrplot()**

A pair-wise correlation plot is a very useful plot too. It requires a package corrplot. Please check whether you have this package installed. If not, you can install that package using the knowledge that you have learned. The detailed tutorial of corrplot package can be found in the website: https://cran.r-project.org/web/packages/corrplot/vignettes/corrplot-intro.html

Try:

Let’s read the expression data from the file expression.txt. Please check what is in the row and what is in the column. Now we try to make a pairwise correlation plot for the first 50 genes and observe the correlation patterns among genes.

expr = read.delim(file='~/AeroFS/Fu\_education/courses/BigData/year2017/R\_tutorial/expression.txt', sep='\t', as.is=T)

expr = as.matrix(expr)

cor.matrix = cor(t(expr[1:50,]))

library(corrplot)

corrplot(cor.matrix, tl.cex=0.4, tl.col=gray(0.5))

You can also cluster the genes together by their correlation structure. You can do so by order the samples based on hierarchical clustering.

corrplot(cor.matrix, order='hclust', tl.cex=0.4, tl.col=gray(0.5))

Now you can see that genes form two big groups. You can specify the value for addrect in corrplot() function to add blocks for a better illustration.

corrplot(cor.matrix, order='hclust', addrect=2, tl.cex=0.4, tl.col=gray(0.5))

**Heatmap**

A heat map (or heatmap) is a graphical representation of data where the individual values contained in a matrix are represented as colors.In R, a heatmap is often combined with hclust() that performs hierarchical clustering based on the distances.

There are many different versions of heatmap functions. Here we just use the basic heatmap(). You can use colorRampPalette() to create a set of colors.

## now I create 50 different colors changing from green to black and to red.

mycolor = colorRampPalette(c('green', 'black', 'red'))(n=50)

heatmap(expr[1:50,], col=mycolor)

Now we can also add the barcode to highlight some features for the samples, e.g, NASH grade.

pheno = read.delim(file='phenotype.txt', sep='\t', as.is=T, header=T)

colside = pheno$NASH\_grade

colside[colside==0]='green'

colside[colside==1]='yellow'

colside[colside==2]='orange'

colside[colside==3]='red'

heatmap(expr[1:50,], col=mycolor, scale='row', ColSideColors=colside)

# >>>>>> Exercise 6 >>>>>>>>>>

## Advanced plots

## If you are interested in learning more about graphs in R. There are many useful packages, e.g., the packages ggplot2.

The package ggplot2 is a popular package for plotting various graphs in R. This package, created by Dr Hadley Wickham, offers a powerful graphics language for creating both simple and complex plots. The ggplot2 homepage can be found here: <http://docs.ggplot2.org/current/>

Moreover, R gallery is an excellent source about graph in R. Please check the website.

http://www.r-graph-gallery.com/all-graphs/

1. Inf stands for infinity [↑](#footnote-ref-1)
2. Between the brackets, give the name of the script. Include its whole path, that is directory plus filename, if you have saved the file to a directory other than the default one. [↑](#footnote-ref-2)
3. Using the chi-squared distribution to interpret Pearson's chi-squared statistic requires one to assume that the discrete probability of observed binomial frequencies in the table can be approximated by the continuous chi-squared distribution. This assumption is not quite correct, and introduces some error. To reduce the error in approximation, Frank Yates, an English statistician, suggested a correction for continuity that adjusts the formula for Pearson's chi-squared test by subtracting 0.5 from the difference between each observed value and its expected value in a 2 × 2 contingency table. This reduces the chi-squared value obtained and thus increases its p-value. The effect of Yates' correction is to prevent overestimation of statistical significance for small data. When you are allowed to use the the chi-square test (i.e. large enough dataset), the continuity correction is not needed and therefore we advise to not correct. [↑](#footnote-ref-3)