

# Introduction to R

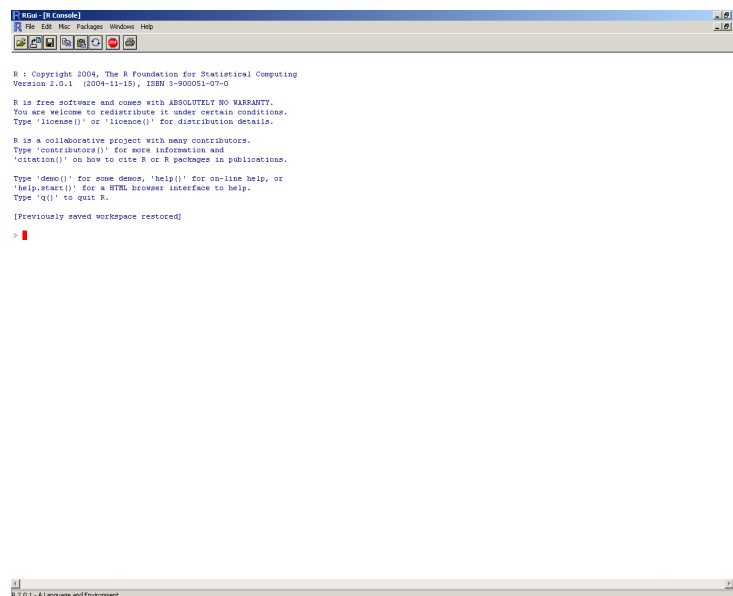
## Master in Statistical Data-Analysis

---

The S language has been developed since the late 1970s by John Chambers and his colleagues at Bell Labs. The language has been through a number of major changes but it has been relatively stable since the mid 1990s. The language combines ideas from a variety of sources and provides an environment for quantitative computations and visualization. S-plus is a commercialization of the Bell Labs code. R is an independent open source version that was originally developed at the University of Auckland but which is now developed by a world wide group of developers. Each version has advantages and problems.

In this outline we will give an introduction to the R program. R is an object-oriented language for statistical programming. It is free-ware and can be downloaded from the web site [www.r-project.org](http://www.r-project.org).

### 1. Getting started in R: some basics



When opening R, you can start working in the R console which is based on a question-and-answer model: when you enter a line with a command and press enter, the program performs the necessary actions and prints results in the same window. When R is ready for input, it prints out its prompt '>'. When working with R, it is handy to type your code in a text-editor and use copy and paste to enter it in the R console. In the newer versions you can make use of a script file ('File - New Script') to type your code in and then select the lines you want to run and execute them with the run-button.

To get a first impression of how R works, we will create a data vector `a = (1, 1, 2, 4, 3, 5, 2, 1, 10, 7)`. This can be done by typing the following command in the R console:

```
a<-c(1,1,2,4,3,5,2,1,10,7)
```

Something useful to know and to make life easier: when pressing the upper arrow key, the previous used command(s) will be recalled.

### Exercises

Press enter and look what you get by entering. Do you understand what each command does?

1. `a[4]`
2. `a[1:5]`
3. `a[a>2]`
4. `a[-1]`
5. `rep(1,3)`
6. `seq(1,3,0.5)`
7. `c(a,1:10)`
8. `length(a)`
9. `cbind(a,1:10)`
10. `rbind(a,a)`
11. `sort(a)`
12. `a[1:3]<-10`
13. `a[c(-1,2)]`
14. `a[NA]`
15. `a<-1:10`  
`names(a)<-LETTERS[1:10]`  
`a[c("A","D","F")]`
16. `a[-11]`
17. How could you choose all elements of a vector which have odd subscripts? Even subscripts?
18. `x<-matrix(1:9,ncol=3)`
19. `x<-matrix(1:9,ncol=3,byrow=T)`
20. `x[1,]`
21. `x[,1]`
22. `x[x>6]`

```

23. x[row(x)>col(x)]<-0
24. diag(x)
25. diag(c(1,2))
26. 1:3+10:12
27. 1+1:5
28. paste(1:5,"A",sep="")
29. Amat<-matrix(c(2,1,4,5),ncol=2)
    Bmat<-matrix(c(3,2,4,7),ncol=2)
    Amat*Bmat
30. Amat%*%Bmat
31. solve(Amat)
32. mean(a)
33. sum(a)
34. cumsum(a)
35. prod(a)

```

Missing values are noted as ‘NA’ in R. R allows vectors to contain a special ‘NA’ value. This value is carried through in computations so that operations on ‘NA’ yield ‘NA’ as result. Sometimes, specifications are needed to handle missing values but you can find these specifications in the R help for the different commands and functions.

Note that R is case-sensitive!

From the exercise we see that R has a rich set of self-describing data structures. These structures describe both the data that can be processed by the language and the language itself. Since the structures are self-describing there is no need for the user to declare the types of the variables. The most basic data type in R is the atomic vector, such vectors contain an indexed set of values that are all of the same type: logical (TRUE or FALSE), numeric, complex and character. The numeric type can be further broken down into integer, single and double types. The function ‘mode’ and ‘storage.mode’ return information about the type of data structure.

```

> mode(1:10)
[1] "numeric"
> storage.mode(1:10)
[1] "integer"
> mode("a string")
[1] "character"
> mode(TRUE)
[1] "logical"

```

R will automatically coerce data to the appropriate type when this is necessary.

```
> 1 + TRUE
[1] 2
```

We can also coerce data structures to an explicit type with the functions ‘as.numeric’, ‘as.character’, ‘as.logical’,...

```
> as.numeric(c("1", "10.5", "text"))
[1] 1.0 10.5 NA .
NA Warning message: NAs introduced by coercion
```

R regards arrays as consisting of a vector containing the array’s elements together with a dimension (or ‘dim’) attribute. A vector can be given dimensions by using the functions ‘array’ or ‘matrix’, or by directly attaching them with the ‘dim’ function.

Examples

- Direct array creation

```
> x<-1:10
> dim(x)<-c(2,5)
> x
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

- Array creation using matrix

```
> x<-matrix(1:10,nrow=2)
> x
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Another data structure available in R is a list. This is a vector which can contain vectors and other lists as elements.

```
> lst<-list(a = 1:3, b = "a list")
> lst
$a [1] 1 2 3
$b [1] "a list"
```

The dollar operator provides a short-hand way of accessing list elements by name.

```
> lst$a
[1] 1 2 3
> lst[["a"]]
[1] 1 2 3 .
```

You can also use the index to get elements of the list and subindexing to get elements of elements of the list.

```
> lst[[1]]
[1] 1 2 3
> lst[[1]][2]
[1] 2
```

In statistics it is common to have categorical variables, indicating some subdivision of data, such as social class, primary diagnosis, tumor stage. Typically, these are input using a numeric code. Such variables should be specified as factors in R. This is a data structure that makes it possible to assign meaningful names to the categories.

The terminology is that a factor has a set of levels—say four for example. Internally, a four-level factor consists of two items: (a) a vector of integers between 1 and 4 and (b) a character vector of length 4 containing strings describing what the four levels are. Let's look at an example:

```
> pain<-c(0,3,2,2,1)
> fpain<-factor(pain,levels=0:3)
> levels(fpain)<-c("none","mild","medium","severe")
```

The first command creates a numerical vector `pain`, encoding the pain level of five patients. We wish to treat this as a categorical variable, so we create a factor 'fpain' from it using the function 'factor'. This is called with one argument in addition to `pain`, namely `levels=0:3`, which indicates that the input coding uses the values 0-3. The latter can in principle be left out, since R by default uses the values in `pain`, suitably sorted.

```
> fpain
[1] none    severe medium medium mild
Levels: none mild medium severe
> as.numeric(fpain)
[1] 1 4 3 3 2
> levels(fpain)
[1] "none"   "mild"   "medium" "severe"
```

The function 'as.numeric' extracts the numerical coding as numbers 1 – 4 and 'levels' extracts the names of the levels. Notice that the original input coding in terms of numbers 0 – 3 has disappeared; the internal representation of a factor always uses numbers starting at 1.

If you don't specify a `levels` argument in 'factor()', the levels will by default be the sorted, unique values represented in the vector. This is not always desirable, when dealing with test variables, since the sorting is alphabetical. Consider for instance,

```
> text.pain<-c("none","severe","medium","medium","mild")
> factor(text.pain)
[1] none    severe medium medium mild
Levels: medium mild none severe
```

## 2. Getting help

When you need help in R, you can consult the R manuals ([www.r-project.org](http://www.r-project.org)). You can also obtain these manuals in HTML format by entering ‘`help.start()`’ in the R console or by help in the menu bar.

When you have questions about the use and possibilities of certain commands, you can type ‘`help(command)`’ or ‘`?command`’ in the R console. For example, if you need more information on the command ‘`lm`’ used for linear models, type one of the following in the R console:

```
help(lm) ?lm
```

‘`apropos`’ is also very useful to search on more options of or alternatives for a certain command. For example, to find other functions or commands that contain ‘`lm`’:

```
[1] ".__C__anova.glm"          ".__C__anova.glm.null" ".__C__glm"
[4] ".__C__glm.null"         ".__C__lm"             ".__C__mlm"
[7] "anova.glm"              "anova.glmmlist"       "anova.lm"
[10] "anova.lmlist"           "anova.mlm"            "contr.helmert"
[13] "glm"                     "glm.control"          "glm.fit"
[16] "hatvalues.lm"           "KalmanForecast"       "KalmanLike"
[19] "KalmanRun"              "KalmanSmooth"         "lm"
[22] "lm.fit"                  "lm.influence"         "lm.wfit"
[25] "model.frame.glm"        "model.frame.lm"       "model.matrix.lm"
[28] "nlm"                     "plot.lm"              "plot.mlm"
[31] "predict.glm"            "predict.lm"           "predict.mlm"
[34] "print.glm"              "print.lm"             "residuals.glm"
[37] "residuals.lm"          "rstandard.glm"        "rstandard.lm"
[40] "rstudent.glm"          "rstudent.lm"          "summary.glm"
[43] "summary.lm"             "summary.mlm"          "anovalist.lm"
[46] "glm.fit.null"           "kappa.lm"             "labels.lm"
[49] "lm.fit.null"            "lm.wfit.null"
```

Hence, new commands are given on which you can find help. If missing values are present in a data set, you can find in the R help how the commands deal with them.

## 3. Importing data in R

The most convenient way of reading data into R is through the function ‘`read.table`’. It requires that the data are available in ASCII format, i.e. they should be created using any plain-text editor (txt or dat files). Each line should correspond to the data of 1 subject and each column should correspond to a variable. The first line of the file can contain a header giving the names of the variables, this should be indicated when reading the data. The path where the data set is stored should also be provided. We will use the data set from problem 1.21 in Neter et al.

Open the data set ‘`CH01PR21.dat`’ on the web site <http://users.ugent.be/~eroelant> or <http://zephyr.ugent.be>. We find that the file contains no header with the names of the variables. To open the data in R, use the following command:

```
> read.table("C:/Temp/CH01PR21.dat",header=FALSE)
```

Note that forward slashes (not backslashes) are used in the filename. R automatically gives the variables the names V1 and V2. The result in R is a data frame and can be stored in an object as follows:

```
airfreight<-read.table("C:/Temp/CH01PR21.dat",header=FALSE)
X<-airfreight$V2 Y<-airfreight$V1
```

The latter 2 commands assign the variables separately to the vectors  $X$  (number of transfers) and  $Y$  (number of broken ampules).

```
> X
[1] 1 0 2 0 3 1 0 1 2 0
> Y
[1] 16 9 17 12 22 13 8 15 19 11
```

Note that choosing  $X$  and  $Y$  as names for the variables in R is completely arbitrary and is not according the conventions where small letters indicate non-random variables and capitals indicate random variables.

You can also change the variable names directly within the data frame by using 'dimnames'.

```
> dimnames(airfreight)
[[1]]
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"

[[2]] [1] "V1" "V2"
```

'dimnames(airfreight)[[2]]' contains the names of the variables and can be changed as follows:

```
> dimnames(airfreight)[[2]]<-c("Y","X")
> airfreight
  Y X
1 16 1 2 9 0 3 17 2 4 12 0 5 22 3 6 13 1 7 8 0 8 15 1 9
19 2 10 11 0
```

However, R won't recognize  $X$  and  $Y$  but only `airfreight$X` and `airfreight$Y` as objects unless we attach the data frame:

```
> attach(airfreight)
> X
[1] 1 0 2 0 3 1 0 1 2 0
> Y
[1] 16 9 17 12 22 13 8 15 19 11
```

Read the data set ‘Birnh.dat’ on the web site <http://users.ugent.be/~eroelant> or <http://zephyr.ugent.be>. We find that the file contains the names of the variables as a header. To open the data in R, we use the command `read.delim` because the columns are now not in fixed width but delimited. `sep=“,”` means that the separator between variables is a comma. Try to open the data set first in a text file to see this.

```
birnhdata<-read.delim("C:/Temp/Birnh.dat",header=TRUE,sep=",")
```

This data will be used later on for the statistics examples. The Birnh-study (Belgian Interuniversity Research on Nutrition and Health) is a large epidemiological follow-up study where nutrition and health data were collected since mid 1980 and where all participants were followed during 10 years (Acta Cardiologica 1984, Volume XXXIX, 4, 258-92). We will look at some variables that were measured in the Flemish regions:

- BMI: Body Mass Index (kg/m<sup>2</sup>)
- SMOKING: 0=no, 1=yes
- TCHOL: Total cholesterol (mg/dl)
- FEMALE: sex, 0=man, 1=woman
- CVD: Cardiovascular death, 0=no, 1=yes, missing if the cause of death wasn't cardiovascular

Note: in newer versions of R it's also possible to use the data editor (‘Edit - Data editor’ and then type the name of the data) to see/change your data.

#### 4. Making your own function in R

It is possible to write your own R functions and/or to modify existing ones. A very simple example is given below. We want to write a function to calculate the standard deviation of a variable as the default functions in R provide the variance only. The function ‘stdev’ is created as follows:

```
stdev<-function(x){sqrt(var(x))}
```

It can now be invoked on a vector ‘a’ like any other function.

```
a<-c(1,1,2,4,3,5,2,1,10,7)
stdev(a)
```

You can also define optional arguments. In the following, the second argument takes on the value 0 if no value is specified for it.

```
> sumsq <- function(x, about = 0) sum((x - about)^2)
```

Because it is not necessary to specify all the arguments to the function, it is important to be clear about which argument corresponds to which formal parameter of the function. In the case of the ‘sumsq’ function the following are equivalent:

```
sumsq(1:10, mean(1:10))
sumsq(1:10, about = mean(1:10))
sumsq(1:10, a = mean(1:10))
```

More details can be found in the R help.

### Exercise

Write a function that computes the geometric mean of any vector ‘x’. The geometric mean of  $n$  values is the  $n$ -th root of the product of the  $n$  values.

### Loops

There are different types of loops in R. ‘for’ statements have the basic form:

```
for (var in vector){
  statements
}
```

The effect of this is to set the value of the variable *var* successively to each of the elements in *vector* and then evaluate all *statements*.

Examples

- ```
x<-1:10
sum <- 0
for( i in 1:length(x) ) {
  sum <- sum+x[i]
}
```
- ```
sum <- 0
for( elt in x) {
  sum <- sum + elt
}
```

There are two statements to control the flow. When a ‘break’ statement is encountered, evaluation halts and transfers to the first statement outside of the enclosing ‘for’, ‘while’ or ‘repeat’ loop. If a ‘next’ statement is encountered then evaluation of the current iteration halts, and the loop is entered with the looping index incremented by one.

```
> for (i in 1:5) if (i == 3) next else print(i)
[1] 1
[1] 2
[1] 4
[1] 5
```

```
> for (i in 1:4) if (i == 3) break else print(i)
[1] 1
[1] 2
```

‘if’ statements have the basic form:

```
if (test) {
  statements
} else {
  statements
}
```

If the first element of *test* is true, the first group of statements is executed, otherwise, the second group of statements is executed. The else clause is optional.

Examples

- `if (any(x < 0) )`  
`stop("negative values encountered")`
- `r <- if( all( x >= 0 ) )`  
`sqrt(x) else`  
`sqrt( x + 0i)`

‘while’ statements have the form `while(condition) expression`, which says that the expression should be evaluated as long as the condition is TRUE. The test occurs at the top of the loop so that the expression might never be evaluated.

Example

```
> y<-12345
> x<-y/2
> while(abs(x*x-y)>1e-10) x<-(x+y/x)/2
> x
[1] 111.1081
> x^2
[1] 12345
```

A variation of the same algorithm with the test at the bottom of the loop can be written with a repeat construction:

```
> x<-y/2
> repeat{
+ x<-(x+y/x)/2 + if(abs(x*x-y)<1e-10) break + }
> x
[1] 111.1081
```

Timing experiments can be a very good way of checking alternative ways of carrying out computations.

```

> sum <- 0
> x <- rnorm(10000)
> system.time({
+ s <- 0
+ for (i in 1:length(x)) s <- s + x[i]
+ })

[1] 0.01 0.00 0.01 NA NA

> system.time({
+ s <- 0 + for (v in x) s <- s + v + })

[1] 0.01 0.00 0.01 NA NA

```

A natural programming construct in R to avoid loops is to ‘apply’ the same function to elements of a list, of a vector, rows of a matrix, or elements of an environment. Some examples are ‘apply’, ‘sapply’, ‘lapply’, ‘mapply’. ‘lapply’ returns a list, ‘sapply’ tries to simplify the result to a vector or a matrix if possible. ‘tapply’ allows you to create tables of the values of a function on subgroups defined by its second argument, which can be a factor or a list of factors.

‘apply’ applies a function over the margins of an array. For example,

```

> x<-matrix(1:9,ncol=3)
> apply(x, 2, mean)

```

computes the column means of the matrix ‘x’, while

```

> apply(x, 1, median)

```

computes the row medians.

```

> BMIwithout<-birnhdata$BMI[!is.na(birnhdata$BMI)]
> tapply(BMIwithout,birnhdata$FEMALE[!is.na(birnhdata$BMI)],median)
      0      1
26.03 26.29

```

### Exercise

Create a function that computes the factorial ( $n! = n \times (n - 1) \cdots 3 \times 2 \times 1$ ). Try to do it one time with and one time without a for-loop.

## 5. Drawing and saving a graph

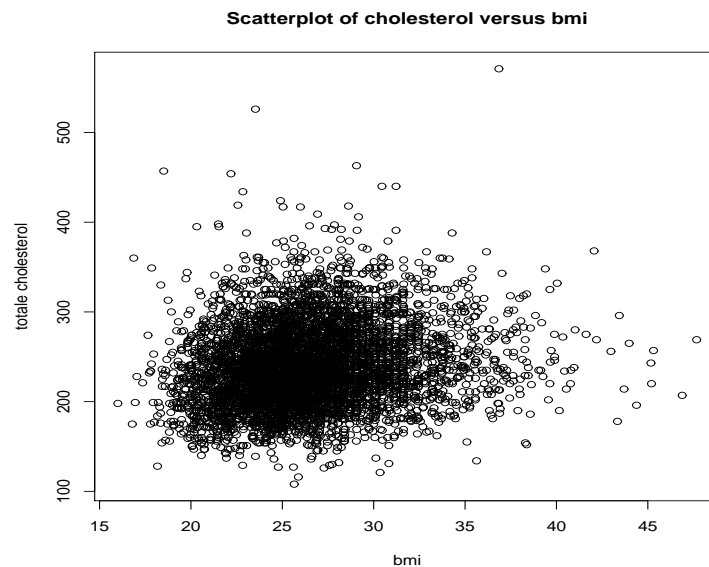
Many functions have plots as part of their standard output. The graph is created automatically in a separate window where it creates a graph sheet that can be saved in many different formats.

Once you created a graph sheet you can save and export using ‘File - Save as’. Different types are possible such as ps, emf, . . . .

There are a number of arguments which may be passed to graphics functions as follows:

- `add=TRUE`: forces the function to act as a low-level graphics function, superimposing the plot on the current plot
- `type=`: controls the type of plot produced as follows: “p” plots individual points, “l” plots lines, “b” both plots points connected by lines, “o” plots points overlaid by lines, “n” no plotting at all.
- `xlab=string`, `ylab=string`: axis labels for the x- and y-axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the plotting function.
- `main=string`: figure title, placed at the top of the plot in a large font
- `sub=string`: sub-title, placed just below the x-axis in a smaller font
- `title(main,sub)`: adds a title `main` to the top of the current plot in a large font and (optionally) a sub-title `sub` at the bottom in a smaller font

```
> plot(birnhdata$BMI,birnhdata$TCHOL,xlab="bmi",ylab="totale cholesterol")
> title("Scatterplot of cholesterol versus bmi")
```



R maintains a list of a large number of graphics parameters which control things such as line style, colors, figure arrangement and text justification among many others. The ‘`par()`’ function is used to access and modify the list of graphics parameters for the current graphics device. You can look in the help which possibilities you have with this function. Interesting to know is that you can create a  $n$  by  $m$  array of figures on a single page. This is done by using `par(mfrow=c(n,m))`.

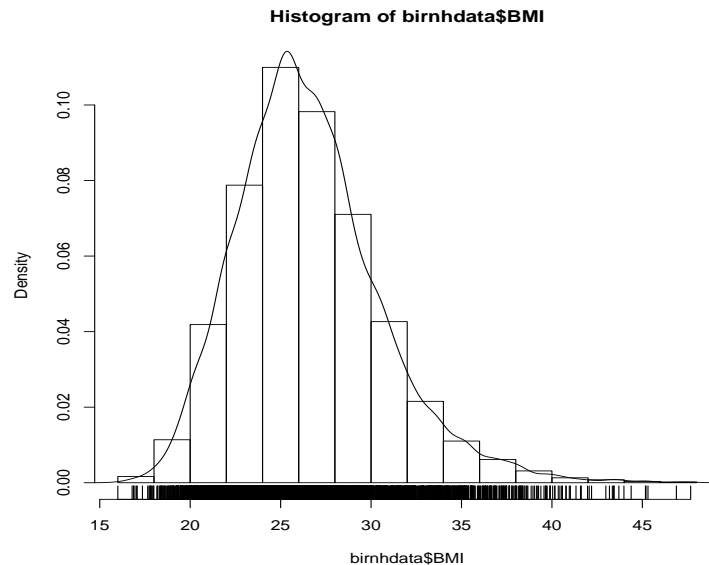
## 6. Other useful functions

To find values of the density function, distribution function, quantile function and to generate random numbers of parametric distributions such as the normal distribution:



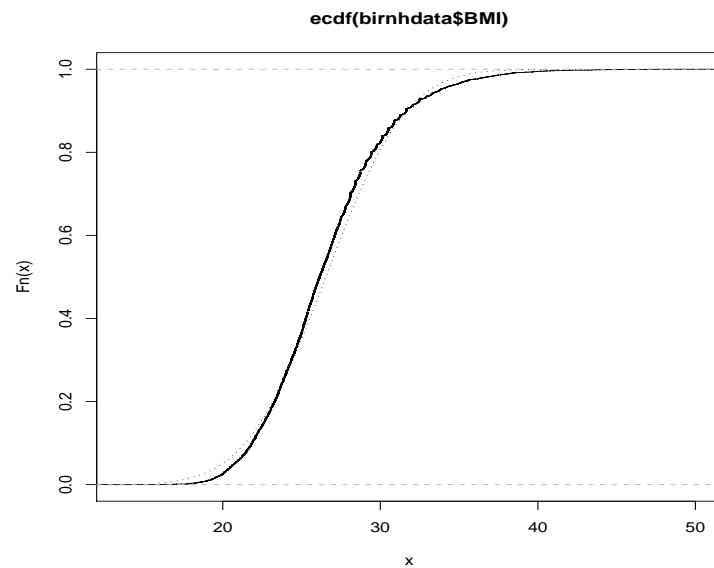
A stem and leaf plot is like a histogram, and R has a function ‘hist’ to plot histograms. You can choose your own bin width instead of using the default bin width. More elegant density plots can be made by ‘density’ and in this example we added a line produced by ‘density’ in this example. The bandwidth ‘bw’ was chosen by trial-and-error.

```
> hist(birnhdata$BMI,prob=TRUE)
> lines(density(birnhdata$BMI,bw=0.5,na.rm=TRUE))
> rug(birnhdata$BMI)
```



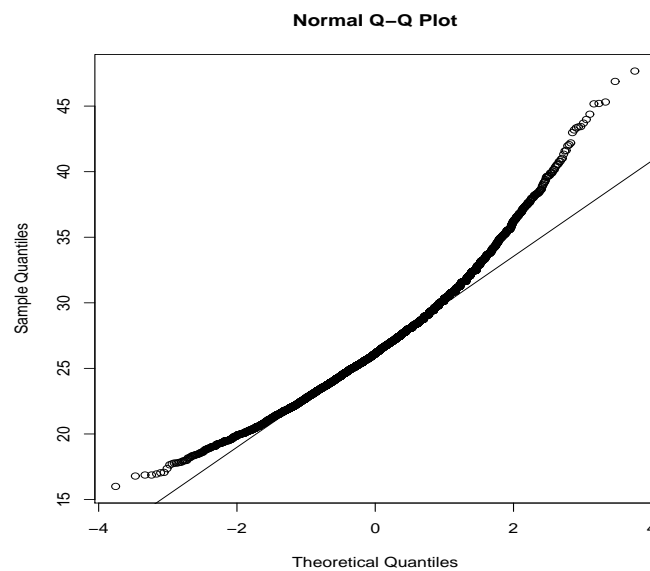
We can plot the empirical cumulative distribution function by using the function ‘ecdf’. To check if the variable is normally distributed, we also fitted a normal distribution and overlaid the fitted CDF.

```
> plot(ecdf(birnhdata$BMI),do.points=FALSE,verticals=TRUE)
> x<-seq(16,52,0.5)
> lines(x,pnorm(x,mean=mean(BMIwithout),sd=sqrt(var(BMIwithout))),lty=3)
```



```
> par(pty="s") #arrange for a square figure region
> qqnorm(birnhdata$BMI)
> qqline(birnhdata$BMI)
```

Quantile-quantile (QQ) plots can help us examine this more carefully. What can you learn from this plot?



Finally, we might want a more formal test of agreement with normality (or not). R provides the Shapiro-Wilk test (`shapiro.test` cannot be done in this case because there are more than 5000 observations) and the Kolmogorov-Smirnov test

```
> ks.test(birnhdata$BMI, "pnorm", mean=mean(BMIwithout), sd=sqrt(var(BMIwithout)))
```

### One-sample Kolmogorov-Smirnov test

```
data:  birnhdata$BMI D = 0.0513, p-value = 1.371e-13 alternative
hypothesis: two.sided
```

```
Warning message: cannot compute correct p-values with ties in:
ks.test(birnhdata$BMI, "pnorm", mean = mean(BMIwithout), sd =
sqrt(var(BMIwithout)))
```

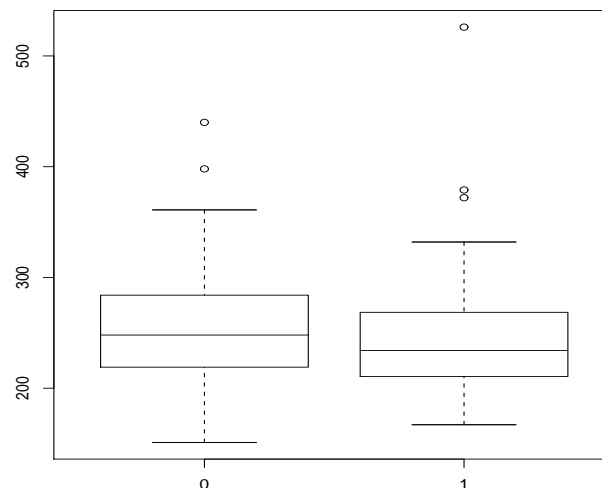
(Note that the distribution theory is not valid here as we have estimated the parameters of the normal distribution from the same sample.)

## 8. One- and two-sample tests

So far we have compared a single sample to a normal distribution. A much more common operation is to compare aspects of two samples. Note that in R, all “classical” tests including the ones used below are in package `stats` which is normally loaded.

We will now compare the total cholesterol (mg/dl) for the patients of the birnh-study who died of a cardiovascular disease between the smokers and non-smokers. A boxplot can give a first insight.

```
> boxplot(birnhdata$TCHOL[birnhdata$CVD==1]~birnhdata$SMOKING[birnhdata$CVD==1])
```



The smokers are the ones and the non-smokers the zeros. What can you learn from this plot?

To test for the equality of the means of the two groups, we can use an unpaired *t*-test by

```
> t.test(birnhdata$TCHOL[birnhdata$CVD==1]~birnhdata$SMOKING[birnhdata$CVD==1])
```

### Welch Two Sample t-test

```
data: birnhdata$TCHOL[birnhdata$CVD == 1] by
birnhdata$SMOKING[birnhdata$CVD == 1] t = 1.3039, df = 159.632,
p-value = 0.1942 alternative hypothesis: true difference in means
is not equal to 0 95 percent confidence interval:
-4.792707 23.417327
sample estimates: mean in group 0 mean in group 1
252.6099 243.2976
```

which doesn't indicate a significant difference on the 5% significance level, assuming normality. By default the R function does not assume equality of variances in the two groups. We can use the F test to test for equality in the variances, provided that the samples come from a normal population.

```
> var.test(birnhdata$TCHOL[birnhdata$CVD==1]~birnhdata$SMOKING[birnhdata$CVD==1])
```

### F test to compare two variances

```
data: birnhdata$TCHOL[birnhdata$CVD == 1] by
birnhdata$SMOKING[birnhdata$CVD == 1] F = 0.8017, num df = 140,
denom df = 83, p-value = 0.2495 alternative hypothesis: true ratio
of variances is not equal to 1 95 percent confidence interval:
0.5396665 1.1683360
sample estimates: ratio of variances
0.8017404
```

which shows no evidence of a significant difference, and so we can use the classical *t*-test that assumes equality of the variances.

```
> t.test(birnhdata$TCHOL[birnhdata$CVD==1]~birnhdata$SMOKING[birnhdata$CVD==1],
var.equal=TRUE)
```

### Two Sample t-test

```
data: birnhdata$TCHOL[birnhdata$CVD == 1] by
birnhdata$SMOKING[birnhdata$CVD == 1] t = 1.3409, df = 223,
p-value = 0.1813 alternative hypothesis: true difference in means
is not equal to 0 95 percent confidence interval:
-4.373374 22.997994
sample estimates: mean in group 0 mean in group 1
252.6099 243.2976
```

All these tests assume normality of the two samples. The two-sample Wilcoxon (or Mann-Whitney) test only assumes a common continuous distribution under the null hypothesis.

```
> wilcox.test(birnhdata$TCHOL[birnhdata$CVD==1]~birnhdata$SMOKING[birnhdata$CVD==1])
```

Wilcoxon rank sum test with continuity correction

```
data: birnhdata$TCHOL[birnhdata$CVD == 1] by
birnhdata$SMOKING[birnhdata$CVD == 1] W = 6820.5, p-value =
0.05724 alternative hypothesis: true mu is not equal to 0
```

## 9. Packages

In R one of the primary mechanisms for distributing software is via packages. CRAN is the major repository for getting packages. You can either download packages manually or use ‘install.packages’ or ‘update.packages’ to install and update packages. In addition, on Windows and in some other GUIs, there are often menu items that facilitate package downloading and updating (see menu ‘Package’). It is important that you use the R package installation facilities. You cannot simply unpack the archive in some directory and expect it to work.

## 10. Session management: saving your workspace

All variables created in R are stored in a common workspace. To see which variables are created in your workspace, type ‘ls()’ in the R console. Variables can be removed using ‘rm(*name variable*)’. When you start R next time, none of these variables will be saved unless you explicitly save your workspace. This can be done using ‘save.image()’ or more conveniently by choosing ‘File - Save workspace’ in the menu bar. You can choose the name and location of your workspace e.g. C:/Temp/Rpart1. In a next R session, you can load a saved workspace again. You can make the location of your workspace also your working directory using ‘File - Change dir’ and specifying the path. This means for example that text files present in this directory can be read without specifying the path or that files created during simulations (see further) will be put in this directory.

## References

Dalgaard P. *Introductory Statistics with R*. Statistics and Computing, Springer, 2002.

Ihaka R. and Gentleman R. *Programming with R, Bioconductor, Education Materials*. 2004.

Neter J., Kutner M.H., Nachtsheim C.J., Wasserman W. *Applied linear statistical models*. McGraw-Hill, 1996.

R manuals ([www.r-project.org](http://www.r-project.org))