Data Analytics and Mining (CSMDM21)

Data Mining in R: a Tutorial

Prof Giuseppe Di Fatta

Department of Computer Science

University of Reading

Contents

Introduction

R (http://www.r-project.org) is a free, open source, interpreted language and environment for **statistical computing and graphics**, developed by volunteers as a service to the community. R is a fully featured programming language dedicated to data. R is an implementation of the S programming language and was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R is distributed under the terms of the Free Software Foundation's GNU General Public License.

R uses a command line interface. Several graphical user interfaces have been developed and are freely available. For example, RGUI is distributed with the pre-compiled version of R for Microsoft Windows and StatET is a plugin for Eclipse. According to a survey over 600 R users the top three R interfaces used more frequently are: the R console, RStudio and Eclipse-StatET.

The growing popularity of R is consistent with independent polls of data miners conducted by KDnuggets and Rexer Analytics’s Annual Data Miner Survey. KDnuggets.com is community website of resources for Data Mining and Analytics. Rexer Analytics’s Annual Data Miner Survey is the largest survey of data mining, data science, and analytics professionals in the industry. In the last few years, R has risen in popularity among data miners. In 2010 it overtook IBM SPSS Statistics and SAS (the commercial leader for advanced analytics) to become the most used tool. In the 2011 Rexer Analytics’s survey showed that R is now being used by close to half of all data miners (47%).

R provides a wide variety of statistical (classical statistical tests, time-series analysis, linear and nonlinear modelling, classification, clustering, etc.) and graphical techniques for data visualisation. The main strength of R lies in the community developed extensions, the R packages. The capabilities of R are extended through user-created packages, which allow specialized techniques, graphical devices, import/export capabilities, reporting tools, etc. A core set of packages is included with the installation of R, with thousands additional packages available at the Comprehensive R Archive Network (CRAN) (http://cran.r-project.org). For a comprehensive list of contributed packages see: http://cran.r-project.org/web/packages/available_packages_by_name.html.

In the Web, the community provides plenty of tutorials and examples. A number of R manuals have also been edited by the R Development Core Team (http://cran.r-project.org/manuals.html). The official R manuals are also available as PDF files in the R distribution. Some parts of this tutorial have been created from the manual "An Introduction to R" to describe the basic syntax of the R language (http://cran.r-project.org/doc/manuals/r-release/R-intro.html).

The R Console

R is an interpreted language which is typically used through a command-line interpreter, the R Console. In Windows the R Console is started with the executables R.exe or RGUI.exe. In the console, you can type your commands (R statements) after the prompt symbol '>' and see the result of their interpretation. When a graph is generated, it will appear in a separate window or in another graphical device (e.g., a file). To quit the R console, type the function *quit*() or its alias *q*().

My First R Program

In the R command line interface type:

```
> x <- 'hello'
> y <- 'world'
> x
[1] "hello"
> y
[1] "world"
> paste(x,y)
[1] "hello world"
>help(paste)
```

In this code the variables x and y are assigned to the string "hello" and "world". Typing a variable prints its value. The function *paste* is used to concatenate the two strings. The function *help*(fun1) opens an help Web page with information on any specific named function fun1. Alternatively, you can also type:

```
>?paste
```

Programming Paradigm

R supports procedural programming with functions and object-oriented programming with generic functions. A generic function dispatches the function (method) specific to that type of object. For example, the generic function *print*(objectName) can print almost every object type. R can also operate as a general matrix calculation toolbox, e.g. similarly to MATLAB.

If you need to verify which objects are currently stored in memory, the function *objects* (or *ls*) can be used. The function *remove* (or simply *rm*) can be used to remove objects from memory. Anything typed after a '#' symbol is ignored buy the interpreter.

```
> ls()        #alt.: objects()
[1] "x" "y"
> rm(x,y)     #to remove all objects: rm(list=ls())
```

Check the syntax of the function *rm* using the help page.

```
>?rm
```

All objects created during an R session can be stored permanently in a file for use in future R sessions. At the end of each R session you are given the opportunity to save all the currently available objects. If you indicate that you want to do this, the objects are written to a file called *.RData* in the current directory, and the command lines used in the session are saved to a file called *.Rhistory*. When R is started at later time from the same directory it reloads the **workspace** from these files.

Expressions and Assignments

Expression and assignment can be type and executed in the R console.

```
> 1+2        #an expression
[1] 3
> 2^3
[1] 8
> x=4        #an assignment
> x          #prints the value of the variable x
[1] 4
>
```

The following tables show the binary and logical operators. Binary operators work on vectors and matrices as well as scalars.

Binary Operators

Operator	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^ or **`	exponentiation
`x %% y`	modulus (x mod y) 5%%2 is 1
`x %/% y`	integer division 5%/%2 is 2

Logical Operators

Operator	Description
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	exactly equal to
`!=`	not equal to
`!x`	Not x
`x | y`	x OR y
`x & y`	x AND y
`isTRUE(x)`	test if x is TRUE

R Data Structures

R operates on named data structures. The simplest such structure is the numeric **vector**, which is a single entity consisting of an ordered collection of numbers. To set up a vector named x consisting of five numbers, use the R function *c* to combine its arguments, as in the following command:

```
> x <- c(1.1, 2.3, 3.14, 6.28, 69.96)
> x
[1]  1.10  2.30  3.14  6.28 69.96
```

The assignment operator is "<-". Assignments can also be made using the function *assign*.

```
> assign("x", c(1.1, 2.3, 3.14, 6.28, 69.96))
```

And assignments can also be made in the other direction.

```
> c(1.1, 2.3, 3.14, 6.28, 69.96) -> x
```

To find out more about R numbers and vectors, see
http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Simple-manipulations-numbers-and-vectors

Vectors are collections of homogeneous data components, i.e. of the same data type. In R, the data type of the components of a vector is known as **mode**. Primitive data types, pardon modes, are: **numeric**, **complex**, **logical**, **character** and **raw**. R also operates on ordered sequences of objects, which are of mode **list**. Each object in a list can be of any mode: lists are collections of objects with heterogeneous types. Contrary to vectors, lists are "recursive" as their components can themselves be lists.

To find out more about R objects, see
http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Objects

A **factor** is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length. Given a list of values, a factor can be generated from the list with the function *factor*. The set of unique values in the list is known as the **levels** of the factor.

For example:

```
> values <- c('C','A','D','B','C','A','B','B','B','A')
> factors <- factor(values)
> factors
 [1] C A D B C A B B B A
Levels: A B C D
> levels(factors)
[1] "A" "B" "C" "D"
>
```

The levels of factors are stored in alphabetical order, or in the order they were specified to factor if they were specified explicitly.

To find out more about R factors, see
http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Factors

By the mode of an object (e.g., vector) we mean the basic type of its fundamental constituents. This is a special case of a "property" of an object. Another property of every object is its length. The functions *mode*(object) and *length*(object) can be used to find out the mode and length of any defined structure. The mode and length are also called **intrinsic attributes** of an object. Further properties of an object are usually provided by *attributes*(object).

Arrays and Matrices

An array in R can have one, two or more dimensions. It is simply a vector which is stored with additional attributes giving the dimensions (attribute "dim") and optionally names for those dimensions (attribute "dimnames"). A two-dimensional array is equivalent to a matrix. One-dimensional arrays often look like vectors, but may be handled differently by some functions. There are two ways to create matrices or arrays from vectors: using the creator functions *matrix*() and *array*(), or simply changing the dimensions using the *dim*() function.

```
Usage: array(data = NA, dim = length(data), dimnames = NULL)
data: a vector giving data to fill the array.
dim: the dim attribute for the array to be created, that is an integer
vector of length one or more giving the maximal indices in each dimension.
dimnames: either NULL or the names for the dimensions.
```

Example for array:

```
> myArray <- array(c(1,2,3,4,5,6), dim=c(3,2))  #2D array
> myArray
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> myVect <- c(1,2,3,4,5,6)
> dim(myVect) <- c(3,2)
> myVect
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> myArray <- array(1:8, dim=c(2,2,2))    #3D array
> myArray
, , 1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
, , 2
     [,1] [,2]
[1,]    5    7
[2,]    6    8
>
```

Example for matrix (compare it to the first array example above):

```
> A <- matrix(c(1,2,3,4,5,6),
+ nrow=3, ncol=2,
+ byrow=TRUE)      #fill matrix by rows (default is FALSE)
> A
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
>
> A[1,2]           #access an element
[1] 2
> A[1,]            #access a row
[1] 1 2
>
```

To find out more about R arrays and matrices, see
http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Arrays-and-matrices

Although a matrix is just a 2D array (with two subscripts), it is given importance with a set of dedicated operators and functions. For example, t(A) is the matrix transpose function; nrow(A) and ncol(A) give the number of rows and columns in the matrix A.

To find out more about R matrix facilities, see
http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Matrix-facilities

Lists and Data Frames

A **list** is an object consisting of an ordered collection of objects known as its components. Contrary to vectors, the components of a list do not need to be of the same mode or type.

Example:

```
> Lst <- list(name="Bob", wife="Mary", no.children=3, child.ages=c(4,7,9))
> Lst
$name
[1] "Fred"

$wife
[1] "Mary"

$no.children
[1] 3

$child.ages
[1] 4 7 9
>
> Lst[[2]]       #access to 2nd component of the list
[1] "Mary"
> Lst$wife       #alt. access to 2nd component of the list
[1] "Mary"
> Lst[["wife"]] #alt. access to 2nd component of the list
[1] "Mary"
```

```
> Lst[[4]]       #access to 4th component of the list
[1] 4 7 9
> Lst[[4]][2]    #access to 2nd el. of 4th component of the list
[1] 7
>
```

Lists can be combined together.

```
> mylist.A <- list("Bob", 32, TRUE)
> mylist.B <- list("Jack", 35, FALSE)
> mylist.C <- list("Alex", 28, FALSE)
> mylist.ABC <- list(mylist.A, mylist.B, mylist.C)
> mylist.ABC
>
```

A **data frame** is a list with class "data.frame". There are some restrictions on lists that may be made into data frames:

- The components must be vectors (numeric, character, or logical), factors, numeric matrices, lists, or other data frames.
- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively.
- Numeric vectors, logicals and factors are included as is, and by default character vectors are coerced to be factors, whose levels are the unique values appearing in the vector.
- Vector structures appearing as variables of the data frame must all have the same length, and matrix structures must all have the same row size.
- A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

Typically a data frame is used for storing data tables. It is a list of vectors of equal length.

Example:

```
> nums <- c(3, 5, 1)
> str <- c("hello", "bye", "ciao")
> bools <- c(TRUE, FALSE, FALSE)
> df <- data.frame(nums, str, bools)
> df
  nums   str bools
1    3 hello  TRUE
2    5   bye FALSE
3    1  ciao FALSE
>
> class(df)
[1] "data.frame"
>
>
> class(iris)      # built-in data frame
[1] "data.frame"
> head(iris)       #for a preview: it shows only a few rows
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
>
```

Around 100 datasets are supplied with R (in package datasets), and others are available in packages (including the recommended packages supplied with R). To see the list of datasets currently available use the function *data*().

File I/O

Data can be loaded from files using the appropriate function from libraries. The following examples show how to load data from files in, respectively, CSV and Excel formats.

```
> myData = read.csv("mydata.csv")  # load data into a data.frame
>
> library(gdata)                    # load the gdata package
> myData = read.xls("myData.xls")  # read data from first sheet
>
```

Data can be loaded directly from the Web. The function *read.table* reads a file in table format (i.e., values separated by blank characters) and creates a data frame from it.

```
> phddata<-read.table("http://data.princeton.edu/wws509/datasets/phd.dat")
> phddata
```

Similarly it is possible to write a matrix or data frame, for example, to a text file with the function write.table.

```
> nums = c(1, 2, 3)
> strs = c("aa", "bb", "cc")
> bools = c(TRUE, FALSE, TRUE)
> df <- data.frame(nums, strs, bools)
> write.table(df, file = "C:/myRprojects/myTable.txt")
```

Control Structures

The typical control structures are summarised in the following table.

Control structure	***R syntax***

`if-else`	`if (cond) expr`
	`if (cond) expr1 else expr2`
	`ifelse(test,yes,no)`

`for`	`for (var in seq) expr`
`while`	`while (cond) expr`
`switch`	`switch(expr, ...)`

Functions

Many functions are available in (built-in functions, built-in libraries) and many more are available from contributed packages. The following tables provide a non-exhaustive list of useful built-in functions. These can be applied to variables, many to vectors and matrices as well.

Numeric Function	***Description***
`abs(x)`	`absolute value`
`sqrt(x)`	`square root`
`ceiling(x)`	`ceiling(4.366) is 5`
`floor(x)`	`floor(4.366) is 4`
`trunc(x)`	`trunc(4.366) is 4`
`round(x,digits=n)`	`round(4.366, digits=2) is 4.37`
`signif(x,digits=n)`	`signif(4.366, digits=2) is 4.4`

cos(x), sin(x),tan(x)	also acos(x), cosh(x), acosh(x), etc.
log(x)	natural logarithm (base e)
log10(x)	logarithm in base 10
exp(x)	e^x (e= 2.718282...)
pretty(range,n)	sequence of about n equally spaced round valuesin the range. pretty(c(-2,2), 4) returns -2 -1 0 1 2In this case equiv. to: -2:2

String Function	***Description***
substr(x, start=n1, stop=n2)	Extract or replace substrings in a character vector. x <- "abcdef" substr(x, 2, 4) is "bcd" substr(x, 2, 4) <- "22222" is "a222ef"
grep(pattern, x, ignore.case=FALSE, fixed=FALSE)	Search for pattern in x. If fixed =FALSE then pattern is a regular expression. If fixed=TRUE then pattern is a text string. Returns matching indices. grep("A", c("b","A","c"), fixed=TRUE) returns 2
sub(pattern, replacement, x, ignore.case =FALSE, fixed=FALSE)	Find pattern in x and replace with replacement text. If fixed=FALSE then pattern is a regular expression. If fixed = T then pattern is a text string. sub("\\s",".","Hello There") returns "Hello.There"
strsplit(x, split)	Split the elements of character vector x at split. strsplit("abc", "") returns 3 element vector "a","b","c"

paste(..., sep="")	Concatenate strings after using sep string to separate them.
	paste("x",1:3,sep="") returns c("x1","x2" "x3")
	paste("x",1:3,sep="M") returns c("xM1","xM2" "xM3")
	paste("Today is", date())

toupper(x)
tolower(x)
	Uppercase/ Lowercase conversion

Statistical Function	***Description***
mean(*x*, **trim=**0, **na.rm=**FALSE)	mean of object x # trimmed mean, removing any missing values and # 5 percent of highest and lowest scores mx <- mean(x,trim=.05,na.rm=TRUE)
sd(*x*)	standard deviation of object(x). also look at var(x) for variance and mad(x) for median absolute deviation.
median(*x*)	median
quantile(*x*, *probs*)	The function quantile produces sample quantiles corresponding to the given probabilities, where x is the input numeric vector. # 30th and 84th percentiles of x y <- quantile(x, c(.3,.84))
range(*x***)**	Range of values in x
sum(*x***)**	Sum of values in x
diff(*x*, **lag=***1***)**	lagged differences, with lag indicating which lag to use
min(*x***)**	minimum of values in x
max(*x***)**	maximum of values in x
scale(*x*, **center=**TRUE, **scale=**TRUE**)**	The function scale centers and/or scales the columns of a numeric matrix.

dnorm(x)
pnorm(q)
	Normal distribution: dnorm gives the density, pnorm the distribution function, qnorm the

`qnorm(p)`
`rnorm(n, m=0,sd=1)`
	`quantile function and rnorm random deviates.`
	`# plot standard normal curve`
	`x <- pretty(c(-3,3), 30)`
	`y <- dnorm(x)`
	`plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i")`

`dbinom(x, size, prob)`	`binomial distribution`
`dpois(x, lamda)`	`poisson distribution`
`dunif(x, min=0, max=1)`	`uniform distribution`
`xtabs`	`The function xtabs creates a contingency table from cross-classifying factors: cross tabulation or crosstabs. (See also table in the base package)`

Functions in R can be treated like any other object: they are first class objects. Functions can be passed as arguments to other functions and can be nested so that a function can be defined inside of another function. The return value of a function is the value of the last expression evaluated in the function body.

Functions have named arguments, the formal arguments included in the function definition. Function calls can make use of all or only some of the formal arguments: arguments can be missing or might have default values. R functions arguments can be matched positionally or by name. For example, consider the built-in function *mean*:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

The following calls to the function *mean* are all equivalent.

```
> mydata <- c(1.2, 2.3, 4.5, 8.4)
> mean(mydata)
> mean(x=mydata)
> mean(trim=0.0, x=mydata)
> mean(na.rm=FALSE, trim=0.0, x=mydata)
> mean(mydata, 0.0, FALSE)
```

Nevertheless, it is a good practice for the sake of code readability to adopt the order of arguments of the function definition.

Moreover, function arguments can be partially matched. R establishes the match between formal and actual parameters following this order of operations: (1) exact match, (2) partial match, (3) positional match. For example, the following calls to the function *plot* are all valid.

```
> xvals <- c(1:100)
> yvals <- xvals^2
> plot(xvals,yvals)                # 2 positional matches
> plot(x=xvals, yvals, xlab="x")   # 2 exact and 1 positional matches
> plot(xvals, yvals, xla="x axis")  # partial match of arg xlab
```

A variable number of formal arguments is indicated by "...". It can be used in generic function to pass the additional argument on to other functions and it can be used when the number of arguments is not known in advance. An example of the second case is the function *paste* to merge a list of any number of string arguments.

```
> paste("a", "b", sep="-")
> paste("a", "b", "c", sep="-")
```

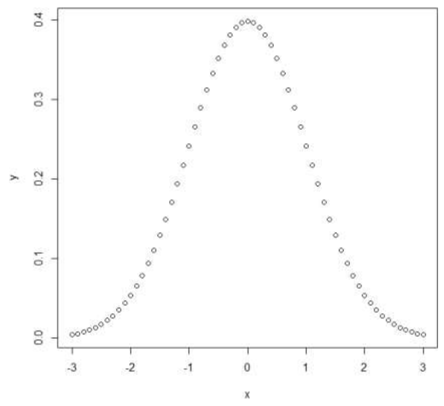
In R user-defined functions follow this syntax:

```
myfunction <- function(arg1, arg2, ... ){
        statements
        return(object)    #optional
}
```

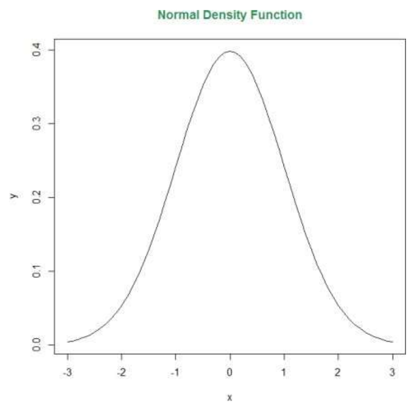
Plots and Graphs

R has very extensive and powerful graphic facilities. The generic function for plotting is *plot*(). For more details about its parameter, type ?plot in the r console.

```
> x <- seq(-3, 3, .1)
> y <- dnorm(x)
> plot(x, y)
>
```

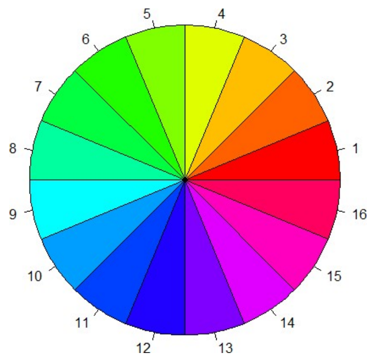


```
> x <- seq(-3, 3, .1)
> y <- dnorm(x)
> plot(x, y, type="l")  #type "l": data points joined into a line
> title("Normal Density Function", col.main="seagreen")
> colors()              #to print the list of color names
>
```



A pie chart can be generated with the function pie().

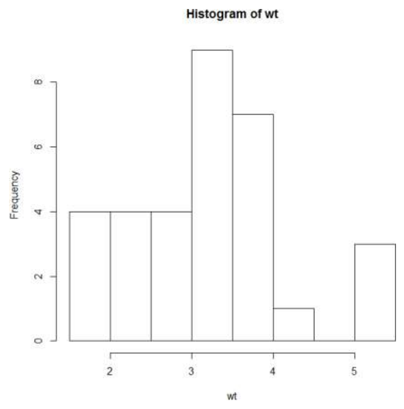
```
> pie(rep(1,16),col=rainbow(16))
>
```



Histograms (function *hist*) show frequency distribution for a numeric variable. Bar graphs (function *barplot*) show frequency distribution for a categorical variable.

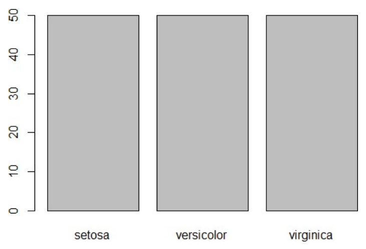
The dataset mtcars is used in the next example. The data comprises fuel consumption and other aspects of automobile design and performance for 32 cars. If you type the help command ?mtcars, you can find out about the dataset and its attributes.

```
> attach(mtcars)
> hist(wt, main="Histogram of wt")
>
```



In order to plot the histogram of categorical data, we first need to extract the frequency table (function *table*) of the values and then use the *barplot* function.

```
> barplot(table(iris$Species))
>
```



Scatter plots show the association between two numeric variables. Here is an example to plot two user-defined functions.

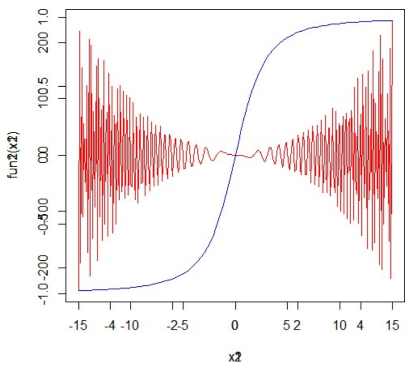
```
> fun1 <- function(x){(1-4*x-x^3/17)*sin(x^2)}
> x1 <- seq(-15,15,0.1)
> plot(x1,fun1(x1), type="l", col="red")
>
> fun2 <- function(x) { x/sqrt(1+x^2)}     # sigmoid function
> x2 <- seq(-5,5,0.1)
> plot(x2,fun2(x2), type="l", col="blue")
>
```

In the above example the same plot area is used to plot the two functions. However, the first curve is lost when the second is drawn.

Graphical objects (e.g. curves) can added to the current plot area by using functions such as points(), lines(), text(), etc.

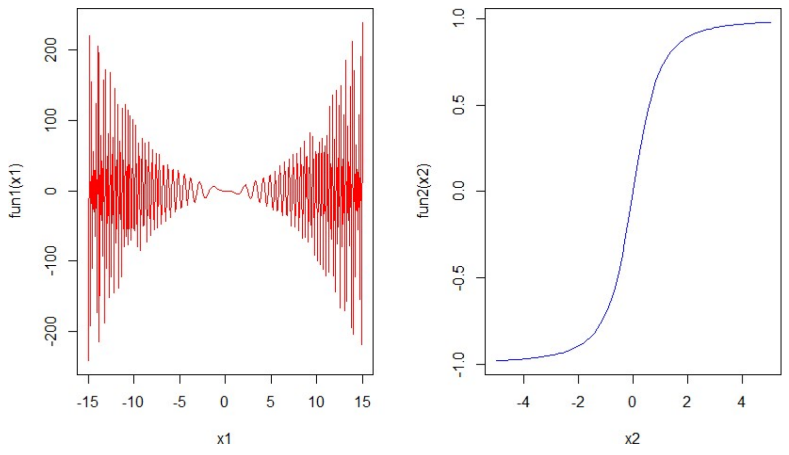
Multiple plots can also be combined into one overall graph, using either the par( ) or layout( ) function.

```
> plot(x1,fun1(x1), type="l", col="red")
> par(new=TRUE)
> plot(x2,fun2(x2), type="l", col="blue")
>
```



A matrix of nrows x ncols plots can be created with the par() function with the option mfrow=c(nrows, ncols) that are filled in by row. The option mfcol=c(nrows, ncols) allows to fill in the plot matrix by column.

```
> par(mfrow=c(1,2))
> plot(x1,fun1(x1), type="l", col="red")
> plot(x2,fun2(x2), type="l", col="blue")
>
```



Graphical Devices

Graphs can be displayed and can be saved in different file formats by using a graphical device. During an R session, using a high-level graphics function will cause a device to be opened if no device is currently open. Which device type is given by the option "device" which is by default set as a screen device. Only one device is the 'active' device: this is the device in which all graphics operations occur. Devices are associated with a type name (see list below) and a number in the range 1 to 63.
There is a "null device" (device 1) which is always open but is just a placeholder: any attempt to use it will open a new device.

The functions in the package dev provide control over multiple graphics devices.

- dev.cur(): gives the current (active) device
- dev.list(): gives the circular list of open devices
- dev.next(which = dev.cur()):select the next open device
- dev.prev(which = dev.cur()):select the previous open device
- dev.off():shuts down the specified device (by default the current).
- dev.set():makes the specified device the active device
- dev.new():opens a new device of specified type (by default the one for the platform display)
- graphics.off():shuts down all open graphics devices

R will open a new device automatically, but the function dev.new() allows to open further devices when needed.

The following graphics devices are always available:

- pdf: it allows to write PDF graphics commands to a file
- postscript: it allows to write PostScript graphics commands to a file
- xfig: the device for XFIG graphics file format
- bitmap: bitmap pseudo-device via Ghostscript
- pictex: it allows to write TeX/PicTeX graphics commands to a file

The following devices are also available when R is compiled accordingly:

- X11: the graphics device for the X11 windowing system
- cairo_pdf, cairo_ps PDF and PostScript devices based on cairo graphics.
- svg: SVG device based on cairo graphics.
- png: PNG bitmap device
- jpeg: JPEG bitmap device
- bmp: BMP bitmap device
- tiff: TIFF bitmap device
- quartz: the graphics device for the OS X native Quartz 2d graphics system

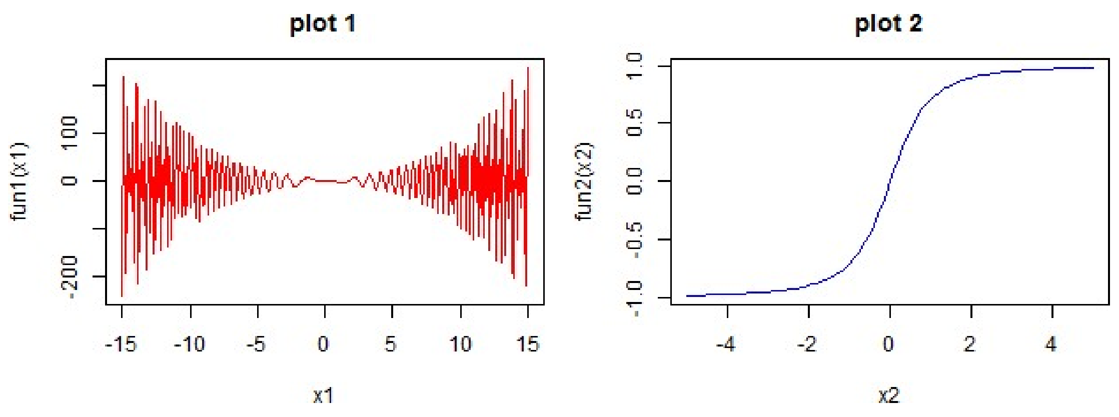
Plots displayed in the screen device can always be saved in different file formats from the contextual menu or the menu File in the RGUI interface.

Plots can be directly (without screen display) saved into a file by using the graphical device of the specific format. For example, the function postscript starts the graphics device driver for producing PostScript graphics.

```
> setwd("C:\\myRprojects\\")    #set the current working dir
> #open a postscript file
> postscript("testfigure.eps", paper="special",
+ height=4.8, width=10, horizontal=FALSE)
> #plotting parameters: matrix of 2 plots
> par(mfrow=c(1,2), mar = c(4,4,3,1))
> #draw plot 1
> plot(x1,fun1(x1), type="l", col="red", main="plot 1")
> #draw plot 2
> plot(x2,fun2(x2), type="l", col="blue", main="plot 2")
> dev.off()
>
```

Similarly, the functions *bmp*, *jpeg*, *png* and start the graphics devices, respectively, for BMP, JPEG, PNG and TIFF format bitmap files.

```
> setwd("C:\\myRprojects\\")    #set the current working dir
> png("testfigure.png", height=240, width=640)
> par(mfrow=c(1,2), mar = c(4,4,3,1))
> plot(x1,fun1(x1), type="l", col="red", main="plot 1")
> plot(x2,fun2(x2), type="l", col="blue", main="plot 2")
> dev.off()
>
```



R has a very rich set of data visualisation functions. Many manuals and tutorials freely available online and there are specific books dedicate to this topic. There are also contributed packages for graphical support, such as **lattice** and **ggplot2**. The package lattice (http://lattice.r-forge.r-project.org/) is a powerful and elegant high-level data visualization system with an emphasis on multivariate data. The package ggplot2 is an advanced plotting system for R (https://ggplot2.tidyverse.org/), which provides a rich set of components with user-friendly wrappers and is considered one of the most popular R packages.

R Console I/O (R scripts)

The execution of the R console starts an interactive session with input from the keyboard and output to the screen. However, the input can be taken from a file containing R commands (a script) and the output can be directed to an output file.

For the following example, we assume we have written a text file ("myRscript.txt") with some commands. Commands can be terminated either by a ";" or by a new line.

The function *source*("myRscript.txt") is used to execute the commands listed in the script file. The *sink*("myOutputFile.txt") can be used to redirect the output to a file. The function *sink* does not redirect graphic output, for which you need to use a graphical device.

The current working directory is the directory where input and output file are located. The function *getwd* returns an absolute file path representing the current working directory of the R process; *setwd*(dir) is used to set the working directory to the specified directory.

Packages

An R package is a bundling of R code, documentation, data, demos, vignettes (long-form documentation and guides), compiled code (C, Fortran, Java, etc.) and automated tests. R packages are typically distributed through the Common R Archive Network (CRAN, http://cran.r-project.org). A new package is tested over many quality checks before it is published on CRAN.

For simplicity, we can consider R packages as collections of R functions and data. The directory where packages are stored is called the 'library'. R comes with a standard (core, built-in) set of packages, with thousands additional packages available at CRAN. A new package needs to be (1) installed only once and (2) loaded into the session before its functions can to be used in the session. The command *install.packages*(package) can be used to install a new package. The command *library*(package) can be used to load a package into the current session. In the RGUI interface there is a useful menu "Packages" dedicated to the management of packages.

The following commands can be used to find out about the packages in your R installation.

```
> .libPaths()     # to get the library path
> library()       # to see all installed packages
> search()        # to see packages which are currently loaded
>
```

Given a package, for information and a complete list of functions, one can use the command library(help = "package-name"). For example, for the package "stats":

```
> library(help = "stats")
```

Core packages:

- base: the R Base Package contains the basic functions which let R function as a language: arithmetic, input/output, basic programming support, etc.
- graphics: the R Graphics Package
- stats: functions for statistical calculations and random number generation.
- utils: the R Utils Package
- foreign: read data stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, etc.
- datasets: the R Datasets Package
- parallel: support for Parallel computation in R

A few examples of contributed packages:

- ggplot2: a plotting system based on the grammar of graphics
- lattice: high-level data visualization system with emphasis on multivariate data
- cluster: functions for cluster analysis
- class: functions for classification (kNN, LVQ, SOM)
- rpart: recursive partitioning and regression trees (decision trees)
- tree: classification and regression trees
- party: recursive partitioning

An updated list of contributed packages is available at http://cran.r-project.org/web/packages/.

R users (developers) can create their own packages. This is outside the scope of this course. For more information refer to one of the many manuals and tutorials available on the topic. For example:

- the R manual "Writing R Extensions" (http://cran.r-project.org/doc/manuals/R-exts.html),
- "Creating R Packages: A Tutorial" (http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf) and
- "Step by Step Tutorial to creating R Packages " (https://www.stt.msu.edu/users/hengwang/R%20package%20tutorial.pdf).

Packages can installed from the RGUI menu or by typing the following commands in the R console.

```
> install.packages("ggplot2")  # download and install the library
> library(ggplot2)             # makes it available in the current session
```

You are typically asked to select a CRAN mirror, i.e. which server should you use to download the package.

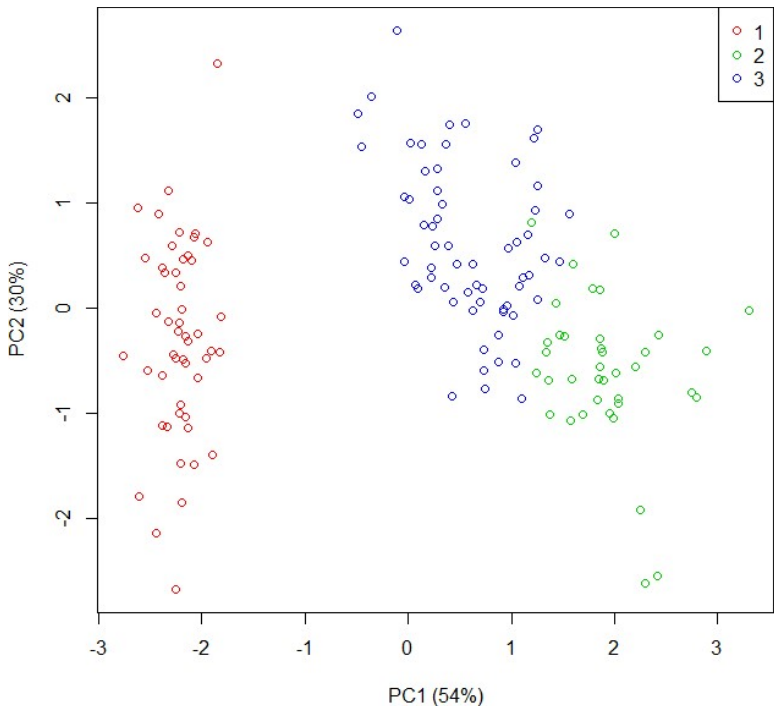
If in your system you have no write access to the installation directory of R, the above command may not work. In this case, you need to indicate a directory where to store the downloaded package.

```
> install.packages("ggplot2", lib="/mydir/myRpackages/")
> library(ggplot2, lib.loc="/mydir/myRpackages/")
```

Cluster Analysis - K-Means and Hierarchical Clustering

There are many built-in and package functions for performing Cluster Analysis (Clustering). For an extensive list of functions see https://cran.r-project.org/web/views/Cluster.html. For example, just for the K-Means algorithm there are many implementations: kmeans (package stats) and Kmeans (package amap). In the following example, we use the *kmeans* functionof the built-in package "stats" to perform Clustering in the *iris* dataset.

```
> set.seed(123)
> iris.km <- kmeans(iris[, -5], 3, iter.max = 100)
> clusterTable <- table(iris[, 5], iris.km$cluster)
> clusterTable
              1  2  3
  setosa     50  0  0
  versicolor  0  2 48
  virginica   0 36 14
>
> #using PCA to plot the clusters in 2D
> dat <- as.matrix(iris[,-5])
> pca <- prcomp(dat, retx=TRUE, center=TRUE, scale=TRUE)
> CID <- iris.km$cluster
> COLOR <- c(2:4)
> plot(pca$x[,1], pca$x[,2], col=COLOR[CID], cex=1,
    xlab=paste0("PC1 (", round(pca$sdev[1]/sum(pca$sdev)*100,0), "%)"),
    ylab=paste0("PC2 (", round(pca$sdev[2]/sum(pca$sdev)*100,0), "%)"))
> legend("topright", legend=levels(factor(iris.km$cluster)), col=COLOR, pch=1)
```

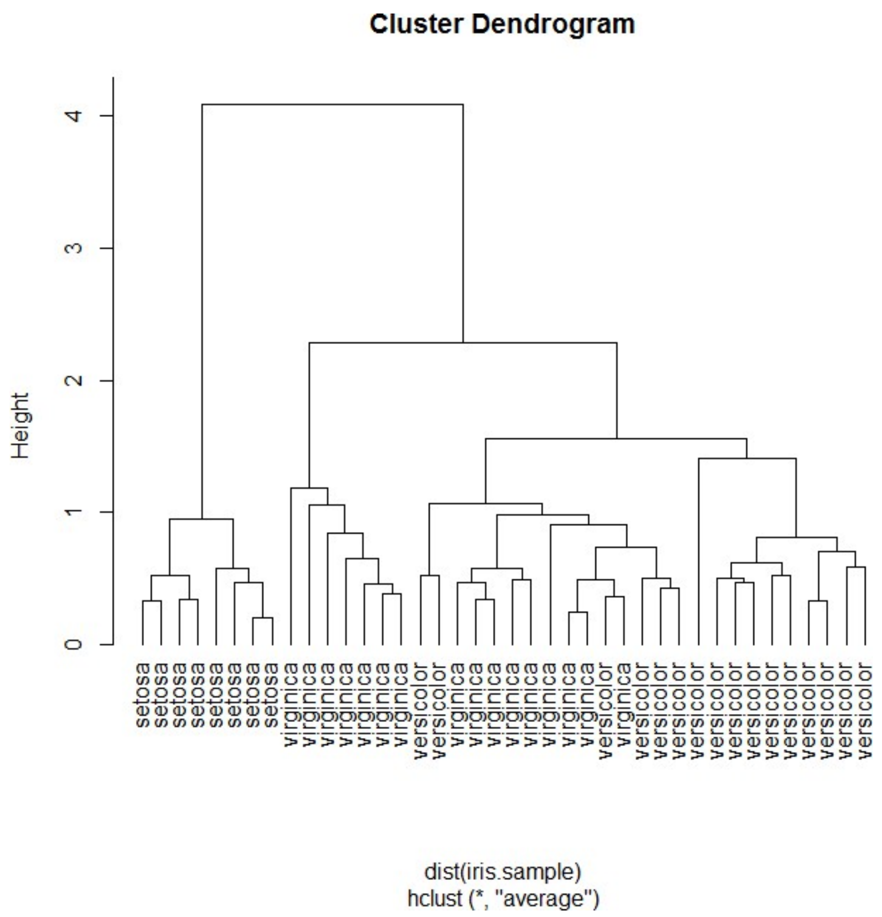


More about the functions:

- prcomp (PCA): https://stat.ethz.ch/R-manual/R-devel/library/stats/html/prcomp.html
- plot (Graphical Parameters): https://www.statmethods.net/advgraphs/parameters.html

The following example shows how to generate a hierarchical clustering for a sample of the iris data.

```
> idx <- sample(1:dim(iris)[1], 40)
> iris.sample <- iris[idx,]
> iris.sample$Species <- NULL
> hc <- hclust(dist(iris.sample), method="ave")
> plot(hc, hang=-1, labels=iris$Species[idx])
>
```



More about the functions:

- hclust: https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/hclust

Classification - Decision Trees

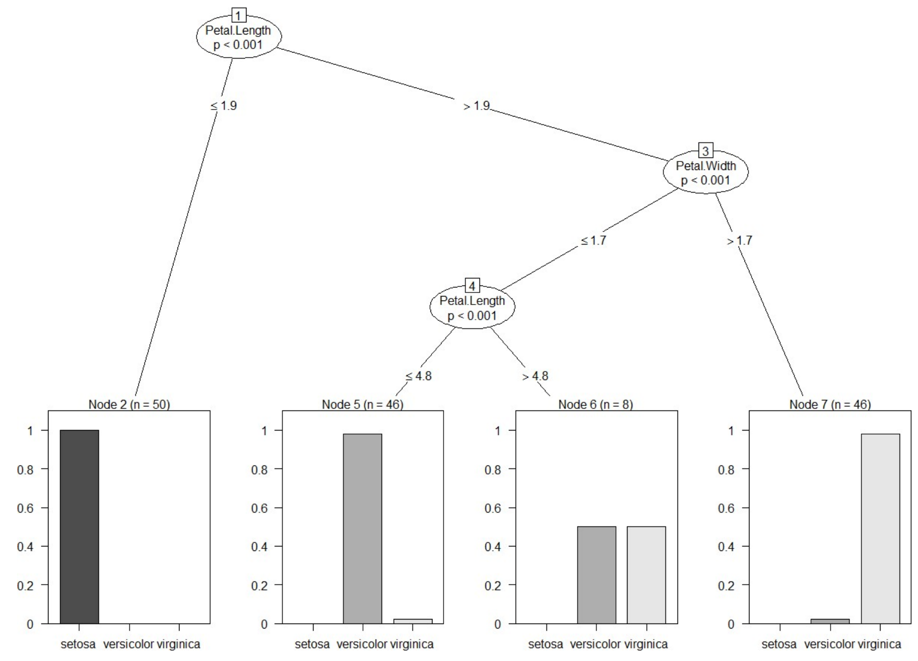
Decision Trees can be used for Classification and Regression. In Classification the predicted attribute, the class, is a nominal attribute; while in Regression the predicted outcome is a numerical value. The term Classification And Regression Tree (CART) analysis is used to refer to both and often the two approaches are combined in a single R package.

The first example is based on the package **party** for recursive partitioning. Information about the package are available at its CRAN page (http://cran.r-project.org/web/packages/party/index.html). Before using the functions of the package, it needs to be installed.

```
> install.packages("party")
> library("party")
```

A decision tree is generated for the iris dataset with the function *ctree*. Information on the tree is extracted with the function print and a graphical representation of the tree is generated with *plot*.

```
> iris.ctree <- ctree(Species ~ Sepal.Length + Sepal.Width + Petal.Length +
+ Petal.Width, data=iris)
> print(iris.ctree)
> plot(iris.ctree)
```



The above decision tree can be tested on some test data. For the sake of the code example we will just use the same data used for training, although we know this should not be done in real applications.

```
> iris_test <- iris         # Warning: never test on the training data!
> predictions <- predict(iris.ctree, new_data=iris_test)
> table(predictions, iris_test$Species)
predictions  setosa versicolor virginica
  setosa         50          0         0
  versicolor      0         49         5
  virginica       0          1        45
>
```

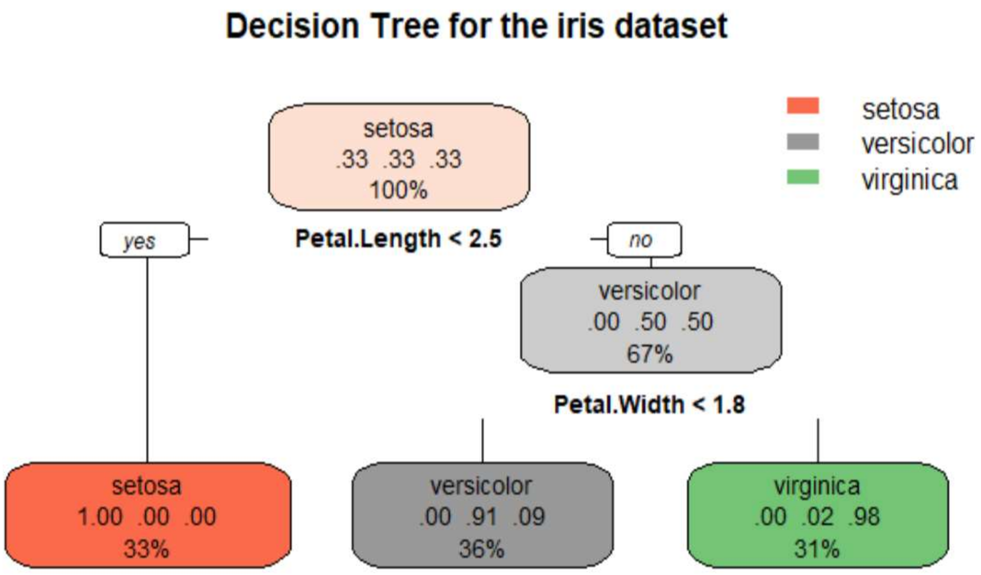
(Note: proper model evaluation will be carried out in exercises during the practical sessions.)

The next example performs the same task with the package **rpart** for recursive partitioning. Information about the package are available at its CRAN page (http://cran.r-project.org/web/packages/rpart/index.html). The package needs to be installed:

```
> install.packages("rpart")
> install.packages("rpart.plot")
> library("rpart")
> library("rpart.plot")
```

In the code example below, a decision tree is generated for the iris dataset with the function *rpart*. Information on the tree is extracted and visualised (*summary*, *printcp*, *plotcp* and *rpart.plot*).

```
> # generate a decision tree
> iris.dt <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length +
+ Petal.Width, data=iris, method="class")
> # extract info on the tree
> summary(iris.dt) # detailed summary of splits
> printcp(iris.dt) # display the results
> plotcp(iris.dt)  # visualize cross-validation results
> # plot tree
> rpart.plot(iris.dt, main="Decision Tree for the iris dataset")
>
```



More about the functions:

- ctree: https://www.rdocumentation.org/packages/partykit/versions/1.2-13/topics/ctree
- rpart: https://www.rdocumentation.org/packages/rpart/versions/4.1-15/topics/rpart

Understanding the Outputs of the Decision Tree Tool:
https://community.alteryx.com/t5/Alteryx-Designer-Knowledge-Base/Understanding-the-Outputs-of-the-Decision-Tree-Tool/ta-p/144773

How does Complexity Parameter (CP) work in decision tree:
https://discuss.analyticsvidhya.com/t/how-does-complexity-parameter-cp-work-in-decision-tree/6589