Introduction to R

Installation

Installation of the basic R package is fairly simple. You need to complete the following steps:

1. Visit the website of one of the mirrors of the R project (e.g. http://cran.gis-lab.info/)
2. Follow the link corresponding to your operating system (*Download R for Linux/Mac/Windows*)
3. Download the installer (using the instructions on the website, the download starts when you click on "*install R for the first time*")
4. Launch the installer (for example, in Windows that is *R-3.2.0-win.exe*)

Along with the basic R package we suggest installing the *R Studio*, software that completes the core R with a convenient integrated development environment (IDE) that often makes life much easier. The website of this project where you can find the installation instructions is http://www.rstudio.com/.

In all the examples below we will work in *R console*. It appears automatically when you launch R Studio in the upper left part of the screen and invites the user to start typing R commands with the following message:

```
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> Start typing your R code here
```

Data types

Like any other programming language, R supports different **data types** to work with different kinds of values: integer, numeric, logic etc. The basic data types in R are described in the table below.

Data types	Value examples
integer	`0L, 1L`
numeric	`0, 1, 2.3, Inf, NaN`
complex	`3+4i`
logical	`TRUE, FALSE, T, F`
character	`"hi"`

The R language manages the types **dynamically**: this means that, when seeing an expression like one of those in the right column of the table, the R interpreter automatically determines its type. For example, if you want assign a logical value to some variable x, you don't need to define the type of x explicitly, you

have only to actually assign the value to it with no declarations (the assignment operator in R is an arrow "<-"):

```
> x <- TRUE
```

Having an already existing variable x, you can perform a range of operations (functions) to determine or change its type:

- Getting the type: `class(x)`
- Type check: `is.[type](x)`
- Type cast: `as.[type](x)`

For example:

```
> x <- 3.14
> class(x)
[1] "numeric"
> is.numeric(x)
[1] TRUE
> is.integer(x)
[1] FALSE
> as.integer(x)
[1] 3
```

It is often the case that the data about certain parameter values for certain observations is missing. R provides a mechanism for handling that case. Missing (unknown) observation values have a special NA type in R ("Not available"). The corresponding operations are:

- Checking that a value of a variable is of type NA: `is.na(x)`
- Filtering out the NA values in an array of observations: `na.omit(data)`

The `na.omit()` function is very useful when you want to calculate some statistics based on the data that possibly has some missing values (NA). If that is the case, then the `mean()` function (that computes the mean) also returns NA, which is not the desired behavior in most cases. Here `na.omit()` solves the problem:

```
> data <- c(1, 2, 3, NA, 4)
> mean(data)
[1] NA
> mean(na.omit(data))
[1] 2.5
```

Compare with MATLAB: is has a similar mechanism for solving this problem, namely the functions like `nanmean()`.

Basic data structures

Values of basic data types (numbers, strings etc.) are usually grouped into **data structures** that make it possible to work with arrays of data as with a single object. The basic data structures in R are listed in the table below. In the right column, there are examples of expressions that can be used to construct the corresponding data structures.

Data structure	Definition examples
vector	`c(1L, 2L, 3L), 1:3, vector("integer", 3)`
list	`list(1L, 2.3, "hi", F)`
factor	`factor(c("Male", "Female", "Male", "Male"))`

matrix	`matrix(1:6, nrow = 2, ncol = 3)`

data.frame	`data.frame(`
	`  age = 18:23,`
	`  height = c(170, 171, NA, 176, 173, 180),`
	`  sex = factor(c("m", "f", "m", "m", "f", "m"))`
	`)`

The **vector** data structure is in fact a list of values of the same type: in the examples above, the first two vectors hold the values `[1, 2, 3]`, and in the last example the vector of three zeros gets created. A **list** differs from a vector in that it can hold values of different types at the same time, including other lists. **Factor** is a sort of vector used to encode categorical (nominal) data (is the example above it is used to encode the observations of a variable that takes two values: "Male"/"Female"). A **matrix** in R can be created by specifying the vector of the values in its cells and its dimensions (nrow/ncol parameters). Finally, **data.frame** objects hold data tables (observations of several attributes). Data frames are thus the basic data structure for many real-world data analysis tasks. An example of data.frame creation can be seen below. This data frame contains data about six people (with age/heigh/sex attributes). After the code you can see the resulting data table:

```
data <- data.frame(
 age = 18:23,
 height = c(170, 171, NA,
            176, 173, 180),
 sex = factor(c("m", "f", "m",
                "m", "f", "m"))
)
```

	age	height	sex
1	18	170	m
2	19	171	f
3	20	NA	m
4	21	176	m
5	22	173	f
6	23	180	m

Functions in R

In programming, a **function** is a named section of a program that performs a specific task. Functions are often also called methods or procedures. A function usually takes some parameter(s) as its input and produces some value as its output. This return value can be a number, a logical value, or some complex object like a plot.

In R, a user can both use a set of predefined functions incorporated into the language (like the `is.number()` function that we've already seen before) and define his/her own functions that perform some specific computations he or she needs. Here is the syntax of function definition in R:

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

To call a function, just write its name and pass the arguments to this function in parentheses:

```
> myfunction(<value1>, <value2>, …)
[1] <some return value>
```

Note that the functions in R are just like objects: they are assigned to variables (`myfunction` in the example above) and can be passed to other functions as parameters. This is what makes R different from popular non-functional languages like C or Java.

Let us define a simple function that computes the square of a number, and test it:

```
> square <- function(x) { return (x * x) }
> square(3)
[1] 9
```

It is also possible that a function calls itself in its body (main part). This is what is called **recursion**. A classic example of a function that can be defined recursively is the factorial, defined for a non-negative integer n as *the product of all positive integers less than or equal to* n. Note also the usage of the `if ... else` expression in its definition. It allows us to specify what the function should do when the expression is true and when it's not:

```
> factorial <- function(x) {
      if (x == 1) {
          return (1)
      } else {
          return (x * factorial (x - 1))
      }
  }
> factorial(1)
[1] 1
> factorial(5)
[1] 120
```

Finally, let us re-implement the same `factorial` function without recursion. We can do that with a **for-loop**, which is yet another basic construct in most programming languages. A for-loop makes it possible to iterate through a collection of values and perform some steps at each iteration. The implementation of the factorial function follows its definition: we iterate through all positive integers less than or equal to `x` (the range of these values is denoted as `1:x`) and add it to the overall product accumulated in the variable called `result`:

```
factorial <- function(x) {
    result <- 1
    for (i in 1:x) {
        result <- result * i
    }
    return (result)
}
> factorial(1)
[1] 1
> factorial(5)
[1] 120
```

Working with data (data frames)

As mentioned above, **data.frame** objects provide us with the basic "container" for data in R. Having a data frame object (let's call it `data`), you can pass it is a parameter to different built-in R functions to get its dimensions, attribute names or to extract only certain parts of the data:

```
> nrows(data)  # number of rows (records)
[1] 6
> ncol(data)  # number of columns (attributes)
[1] 3
> names(data)  # attribute names
[1] "age" "height" "sex"
> head(data, 3)  # overview of the data (only the first three rows)
     age   height     sex
   1  18      170       m
   2  19      171       f
   3  20       NA       m
```

Another useful function is called `summary()`. It outputs the statistics for all the attributes in the data frame (minimum/maximum, quantiles, the number of NA values):

```
> summary(data)
          age           height         sex
 Min. :18.00      Min. :170       f:2
1st Qu.:19.25    1st Qu.:171      m:4
Median :20.50    Median :173
 Mean :20.50      Mean :174
3rd Qu.:21.75    3rd Qu.:176
 Max. :23.00      Max. :180
                   NA's :1
```

Data slicing in a data frame can be done both by rows and by columns. For both, there is the syntax of form `data[<row filter>, <column filter>]`. For example, to retrieve the specified rows from the data table, run:

```
> data[1,]  # only the first row
  age height sex
1  18    170   m

> data[1:3,]  # rows from the first to the third
  age height sex
1  18    170   m
2  19    171   f
3  20     NA   m

> data[c(1,3),]  # only the first and the third rows (indices vector)
  age height sex
1  18    170   m
3  20     NA   m
```

To get certain attributes only, you can use the `data[<row filter>, <column filter>]` syntax again:

```
> data[, "height"]
[1] 170 171 NA 176 173 180
```

R also provides you with an alternative way to refer to attributes by name in a data frame separating the data frame's name from the name of the variable with a dollar sign ($):

```
> data$height
[1] 170 171 NA 176 173 180
```

This "$" syntax also works for named elements extraction out of an R list, as well as for R objects.

More complicated **queries** to a data frame can combine filters by rows and by columns. In the example below, we retrieve the age of all men older than 20:

```
> data[data$age > 20 & data$sex == "m", "height"]
[1] 176 180
```

Other useful functions include `all()` and `any()`. These functions answer the question whether the specified condition is true for all or any record in the data, correspondingly:

```
> all(data$age > 16)
[1] TRUE

> any(data$age > 60)
[1] FALSE
```

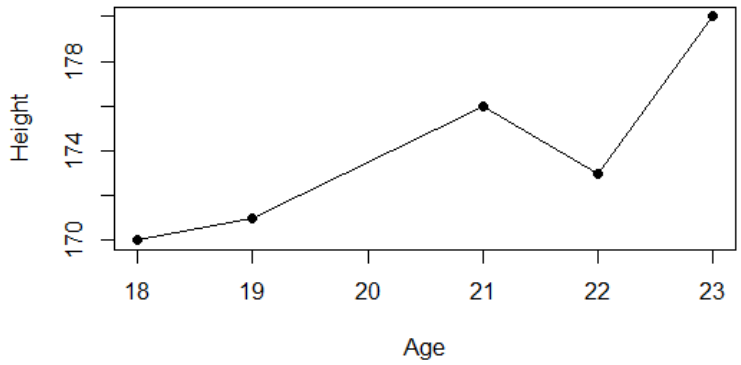
Finally, you can add **new attributes** to a data frame by passing the vector of values of this new attribute for all records present in the data frame:

```
> data$foo <- c(6, 5, 4, 3, 2, 1)
> head(data, 3)
  age height sex foo
1  18    170   m   6
2  19    171   f   5
3  20     NA   m   4
```

Data visualization

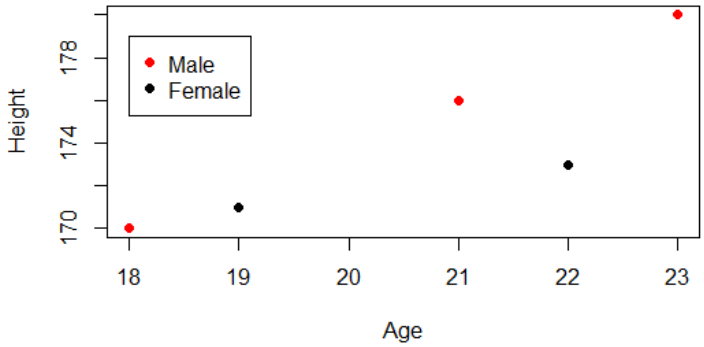
In the simplest case, you can try to visualize the dependencies between attribute pairs using plots. For this purpose, R has the `plot()` function that builds a **dot diagram** (**scatter plot**) on the plane. There is also the `lines()` function for connecting the dots on the plot with lines. Let's construct plot to estimate the dependency between height and age for people in the dataset we have been using in the examples above:

```
> plot(data$age, data$height, pch=19, xlab="Age", ylab="Height")
> lines(data[!is.na(data$height),]$age, na.omit(data$height), pch=19)
```



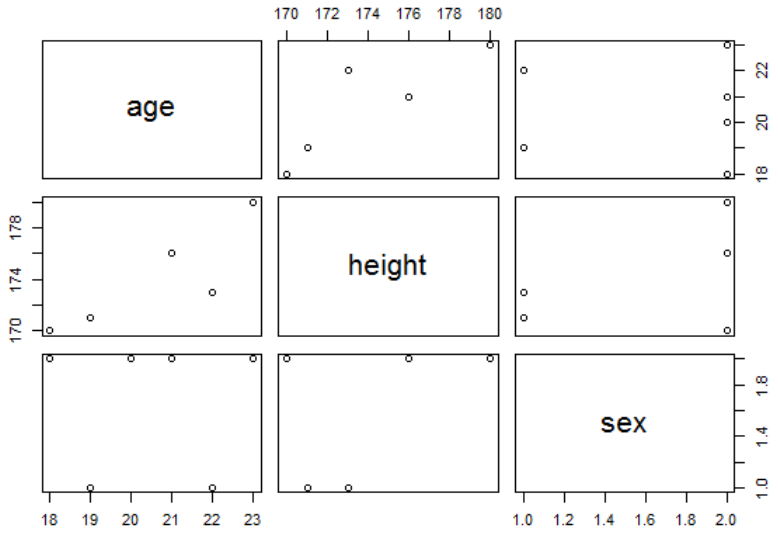
In fact, it makes little sense to connect dots in a scatterplot since in a larger data set there definitely would be several observations with the same age that would make this plot too messy. But what is worth trying to improve the visualization is the separation of points on our scatter plot by sex. In the example below we do this by coloring the dots in red for men and in black for women. Note the alternative syntax for specifying the attributes used here while constructing the scatter plot (`data$height ~ data$age`), and also how the `col` parameter is used to set the dot colors:

```
> plot(data$height ~ data$age, pch=19, col=data$sex, xlab="Age", ylab="Height")
> legend(18, 179, legend=c("Male", "Female"), col=c("red", "black"), pch=19)
```



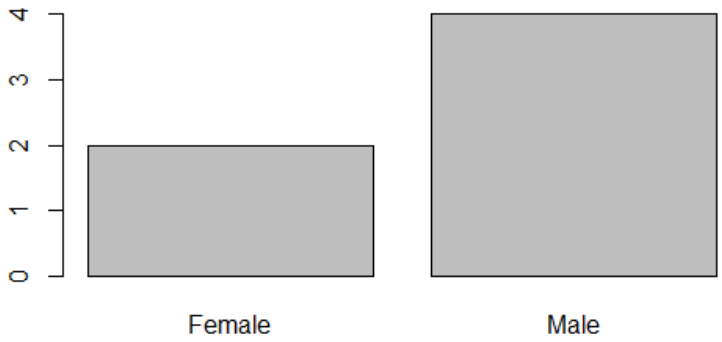
The `plot()` function can also build the **scatterplot matrix** for all attribute pairs if you pass the whole `data.frame` object as its argument. Such a matrix can be useful as one of the first steps in data analysis: it gives a clear idea of which attribute pairs expose a certain dependency and which seem to be uncorrelated:

```
> plot(data)
```



Finally, to build **histograms** in R, one can use the `barplot()` function. This function should be provided with not just the source data, but with the information about frequency of appearance of different values of the attribute being investigated. These frequency values can be computed with the `table()` function. In the example below we also pass as the second argument to `barplot()` the names of the bars on the resulting histogram:

```
> barplot(table(data$sex), names.arg=c("Female", "Male"))
```



Importing data

The most simple way to load some existing data set to R is to read it from a file. The R language has a range of functions for reading the data from different formats: `read.csv()` for CSV tables, `read.xlsx()` (from package xlsx) for Excel tables, `fromJSON()` (package RJSONIO) for reading the JSON data. All these functions produce a `data.frame` object. In the example below, we first download a data set called «Iris flower» from the web to a temporary CSV file and then read this file in R:

```
> fileUrl <- "http://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"
> download.file(fileUrl, destfile="iris.csv")
> iris.data <- read.csv("iris.csv")
> head(iris.data, 3) # iris.data is a data frame
  X5.1 X3.5 X1.4 X0.2 Iris.setosa
1  4.9  3.0  1.4  0.2 Iris-setosa
2  4.7  3.2  1.3  0.2 Iris-setosa
3  4.6  3.1  1.5  0.2 Iris-setosa
> colnames(iris.data) <- c("Sepal.Length", "Sepal.Width", "Petal.Length",
"Petal.Width", "Species") # Renaming attributes
> head(iris.data, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width     Species
1          4.9         3.0          1.4         0.2 Iris-setosa
2          4.7         3.2          1.3         0.2 Iris-setosa
3          4.6         3.1          1.5         0.2 Iris-setosa
```

Reading matrices in R can be done with the combination of `read.csv` and `as.matrix()` functions. Let's assume you have the following contents in the *matrix.txt* file residing in your current working directory (you can figure out what your current working directory is by calling `getwd()` in R and change it using the `setwd(<path>)` function):

```
0,.11,.22,.4
.11,0,.5,.3
.22,.5,0,.7
```

Then you can read this matrix as follows:

```
> m <- as.matrix(read.csv("matrix.txt", header=FALSE))
> m
      V1   V2   V3  V4
```

```
[1,] 0.00 0.11 0.22 0.4
[2,] 0.11 0.00 0.50 0.3
[3,] 0.22 0.50 0.00 0.7
```

Saving the matrix back to a file is simple as well with the `write.table()` function. If you don't set the `row.names` and `col.names` parameters to FALSE, row/column names will be written to the output file along with the raw data:

```
> write.table(m, "matrix2.txt", sep=",", row.names=FALSE,
col.names=FALSE)
```