IE6200: Engineernig Probability and Statistics

LAB 0: Introduction to R

Prof. Mohammad Dehghani

Contents

R History

R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing. The R language is widely used among statisticians and data miners for developing statistical software and data analysis. Polls, data mining surveys, and studies of scholarly literature databases show substantial increases in popularity

RStudio Environment

When you open RStudio, you will see the following four windows (also called panes)

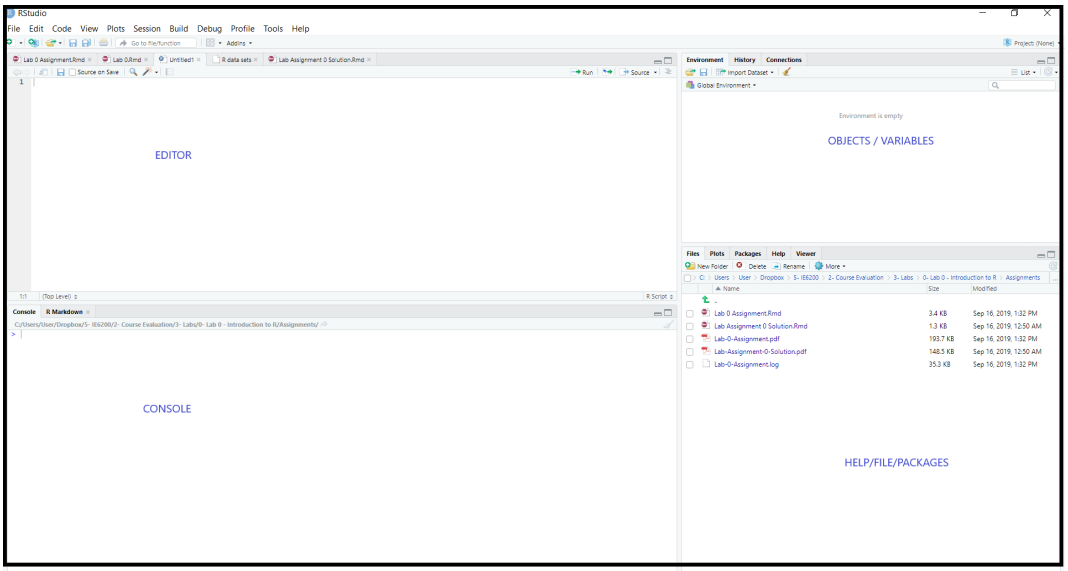

Figure 1: Four Panes of RStudio

- **Editor:** This is where you create and edit 'R Scripts' - your collections of code. R scripts are just text files with the '.R' extension. You will notice that when you are typing code in a script in the Editor panel, R will not actually evaluate the code as you type. To have R actually evaluate your code, click on the 'Run' button on the top right of the Editor. Alternatively, you can use the hot-key "Command + Return" on Mac, or "Control + Enter" on PC to run the seleceted code.
- **Console:** This is where R actually evaluates the code. At the beginning of the console you will see the *greater than* character >. This is a prompt that tells you that R is ready for new code. You can type code directly into the console after the prompt and get an immediate response. For example, if you type 1+1 into the console and press enter, you'll see that R immediately gives an output of 2. Code is usually written in a R Script, while console is used to debug or for quick analysis.
- **Environment:** This panel shows you the names of all the data objects (like vectors, matrices, and dataframes) that you have defined in your current R session. You can also see information like the number of observations and rows in data objects.
- **Files/Plots/Packages/Help:** This panel shows you lots of helpful information including:
  - **Files:** The Files panel gives you access to the file directory on your hard drive.
  - **Plots:** The Plots panel shows all your plots. There are buttons for opening the plot in a separate window and exporting the plot as a $*.pdf$ or $*.jpeg$.
  - **Packages:** Shows a list of all the R Packages installed on your hard drive and indicates whether or not they are currently loaded. Packages that are loaded in the current session are checked, while those that are installed but not yet loaded are unchecked.

- **Help:** Help menu for R functions. You can either type the name of a function in the search window, or use the code to search for a function with the name.

Objects and Functions

Commands in R are generally made up of two parts: **objects** and **functions** . These are separted by **<-**, which can be thought of as 'is created from'. As such, the general form of command is:

```
Object <- Function
```

Which means 'object is created from function'.

Example.

```
x <- 1  #Created a variable x and assigned it the value 1
```

- **Object:** Object is anything created in R. It could be a variable, a collection of variables, a statistical model, etc. Objects can be single values or collection of information.
- **Functions:** Functions are the things you do in **R** to create your objects.

To get information about any function in general, use ?function or help(function).

Data Types

There are several basic R data types that are of frequent occurrence in routine R calculations. These are discussed below:

Character

A *character* object is used to represent string values in R. We convert objects into character values with the as.character() function. We can check the datatype of any variable using the class() function.

```
x <-  as.character(3.14)
x
```

```
## [1] "3.14"
```

```
class(x)
```

```
## [1] "character"
```

Numeric

Decimal values are called *numeric* in R. It is the default computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
x <-  10.5  # assign a decimal value
x
```

```
## [1] 10.5
```

```
class(x)  # get class of x
```

```
## [1] "numeric"
```

Furthermore, even if we assign an integer to a variable k, it is still being saved as a numeric value.

```
k <- 1
k
```

```
## [1] 1
```

```
class(k)
```

```
## [1] "numeric"
```

Integer

In order to create an *integer* variable in R, we invoke the as.integer() function.

```
y <- as.integer(1)
y
```

```
## [1] 1
```

```
class(y)
```

```
## [1] "integer"
```

We can coerce a numeric value into an integer with the as.integer() function.

```
as.integer(3.14)  # coerce a numeric value
```

```
## [1] 3
```

Logical

A *logical* value is often created via comparison between variables.

```
x <- 1
y <- 2
x
```

```
## [1] 1
```

```
y
```

```
## [1] 2
```

```
z <- x > y
z
```

```
## [1] FALSE
```

```
class(z)
```

```
## [1] "logical"
```

Operators

R has many operators to carry out different mathematical and logical operations. Operators in R can mainly be classified into the following categories.

Arithmetic	Relational	Logical
+ addition	< less than	!x Not x
- subtraction	> greater than	x|y x OR y (element-wise)
∗ multiplication	<= less than or equal to	x||y x OR y (only first element)
/ division	>= greater than or equal to	x&y x And y (element-wise)
^ power	== equal	x&&y x And y (only first element)
%% modulus	!= different	
%/% integer division		

Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. You can find some examples below:

```
x <- 5
y <- 16

x
```

```
## [1] 5
```

```
y
```

```
## [1] 16
```

```
x + y
```

```
## [1] 21
```

```
x - y
```

```
## [1] -11
```

```
x * y
```

```
## [1] 80
```

```
y / x
```

```
## [1] 3.2
```

```
y %/% x
```

```
## [1] 3
```

```
y %% x
```

```
## [1] 1
```

```
y ^ x
```

```
## [1] 1048576
```

Relational Operators

Relational operators are used for comparison between values.

```
x <- 5
y <- 16

x
```

```
## [1] 5
```

```
y
```

```
## [1] 16
```

```
x < y
```

```
## [1] TRUE
```

```
x > y
```

```
## [1] FALSE
```

```
x <= 5
```

```
## [1] TRUE
```

```
y >= 20
```

```
## [1] FALSE
```

```
y == 16
```

```
## [1] TRUE
```

```
x != 5
```

```
## [1] FALSE
```

Logical Operators

Logical operators are used to carry out Boolean operations like AND, OR etc. Operators & (single ampersand) and | (single pipe, above Return key in Mac and Enter key in PC) perform element-wise operation producing result having length of the longer operand. But && (double ampersand) and || (double pipe) examine only the first element of the operands resulting into a single length logical vector. **Zero** is considered FALSE and **non-zero** numbers are taken as TRUE.

```
x <- c(TRUE,FALSE,0,6)
y <- c(FALSE,TRUE,FALSE,TRUE)
```

```
!x # NOT x
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

```
x|y # x OR y
```

```
## [1]  TRUE  TRUE FALSE  TRUE
```

```
x||y # x OR y (compare only the first element)
```

```
## [1] TRUE
```

```
x&y # x AND y
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```
x&&y # x AND y (Compare only the first element)
```

```
## [1] FALSE
```

Data Structures

Vectors

The basic data structure in R is the *Vector*. A *Vector* is a sequence of elements of the same data type.

A *numeric* vector:

```
c(10, 5, 3, 6, 21)
```

```
## [1] 10  5  3  6 21
```

A *character* vector:

```
c("aa", "bb", "cc", "dd", "ee")
```

```
## [1] "aa" "bb" "cc" "dd" "ee"
```

Vectors can be combined via the function **c()**. This is a generic function which combines its arguments. For example, the following two vectors n and s are combined into a new vector containing elements from both vectors.

```
n <- c(2, 3, 5)
s <- c("aa", "bb", "cc", "dd", "ee")
c(n, s) # combine n and s
```

```
## [1] "2"  "3"  "5"  "aa" "bb" "cc" "dd" "ee"
```

Vector Arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element.

```
a <- c(1, 3, 5, 7)
b <- c(1, 2, 4, 8)

5 * a
```

```
## [1]  5 15 25 35
```

```
a + b
```

```
## [1]  2  5  9 15
```

If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector.

```
u <- c(10, 20, 30)
v <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
u + v
```

```
## [1] 11 22 33 14 25 36 17 28 39
```

Vector Indexing

We retrieve values in a vector by declaring an index inside a single square bracket "[ ]".

```
s
```

```
## [1] "aa" "bb" "cc" "dd" "ee"
```

```
s[3]
```

```
## [1] "cc"
```

If the index is negative, it specifies the values to be excluded rather than included.

```
s[-3]
```

```
## [1] "aa" "bb" "dd" "ee"
```

If an index is out-of-range, a missing value will be reported via the symbol **NA**.

```
s[10]
```

```
## [1] NA
```

NA stands for 'Not Available' and is an indicator of missing values.

A new vector can be retrieved from a given vector with a numeric index vector, which consists of member positions of the original vector. Here, c creates a vector that has 2 and 3 as its members, as such, 2nd and 3rd elements are retrieved.

```
s[c(2, 3)]
```

```
## [1] "bb" "cc"
```

To produce a vector slice between two indices, we can use the colon operator ':'.

```
s[2:5]
```

```
## [1] "bb" "cc" "dd" "ee"
```

[2:5] returns a vector that has all the integers from 2 to 5, both inclusive. This type of indexing is specially useful when the dataset is large.

Matrix

A *Matrix* is a collection of data elements arranged in a two-dimensional rectangular layout. We produce a matrix in R with the matrix() function.

```
A <- matrix(
  c(2, 4, 3, 1, 5, 7), # the data elements
  nrow=2,              # number of rows
  ncol=3,              # number of columns
  byrow = TRUE)        # fill matrix by rows

A
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    3
## [2,]    1    5    7
```

What would happen if the byrow is set to FALSE? (By defualt it is FALSE).

An element at the m^{th} row, n^{th} column of A can be accessed by the expression A[m, n].

```
A[2, 3]
```

```
## [1] 7
```

The entire m^{th} row A can be extracted as A[m,]. Not mentioning any number after the comma, means get everything from that Hence, it returns all the members of the 2^{nd} row.

```
A[2, ]
```

```
## [1] 1 5 7
```

We can also extract more than one rows or columns at a time. Here, we get all the rows from the 1^{st} and 3^{rd} column.

```
A[ ,c(1,3)]
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    1    7
```

We construct the transpose of a matrix by interchanging its columns and rows with the function t().

```
B <- t(A)

B
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    4    5
## [3,]    3    7
```

Using rbind() and cbind() functions, we can combine two matrices by rows or columns respectively.

```
D <- matrix(
  c(6, 2),
  nrow=1,
  ncol=2) # D has two columns

D
```

```
##      [,1] [,2]
## [1,]    6    2
```

```
rbind(B, D)  # since both B and D have two columns, the rows of B and D are combined
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    4    5
## [3,]    3    7
## [4,]    6    2
```

Lists

A *List* is a generic vector containing other objects, which can be constructed using the list() function. They are great when outputs of a particular function produce different type of objects and need to be stored together.

```
n <- c(2, 3, 5)
s <- c("aa", "bb", "cc", "dd", "ee")
A <- matrix(
  c(2, 4, 3, 1, 5, 7),
  nrow=2,
  ncol=3,
  byrow = TRUE)
x <- list(n, s, A)    # x contains n, s and matrix A

x
```

```
## [[1]]
## [1] 2 3 5
##
## [[2]]
## [1] "aa" "bb" "cc" "dd" "ee"
##
## [[3]]
##      [,1] [,2] [,3]
## [1,]    2    4    3
## [2,]    1    5    7
```

Similar to vector indexing, using single square brackets '[ ]', we can retrieve a particular element or more than one element of a list.

```
x[2]
```

```
## [[1]]
## [1] "aa" "bb" "cc" "dd" "ee"
```

```
x[c(2, 3)]
```

```
## [[1]]
## [1] "aa" "bb" "cc" "dd" "ee"
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    2    4    3
## [2,]    1    5    7
```

The elements 2 of 3 of the list are labelled as [[1]] and [[2]] because the output is a list as well.

In order to reference a list member directly, we have to use the double square brackets "[[ ]]".

```
x[[2]]
```

```
## [1] "aa" "bb" "cc" "dd" "ee"
```

To reference a particular element of a member, we first refer the member using double square brackets "[[ ]]" followed by element in single square brackets '[ ]'.

```
x[[2]][4]
```

```
## [1] "dd"
```

Also, we can modify the element directly:

```
x[[2]][1] <- "bb"

x[[2]]
```

```
## [1] "bb" "bb" "cc" "dd" "ee"
```

Data Frames

A *Data Frame* is used for storing data tables. It is a list of vectors of equal length. For example, the following variable *df* is a data frame containing vectors n and s.

```
n <- c(2, 3, 5)
s <- c("aa", "bb", "cc")
df <- data.frame(n, s)

df
```

```
##   n  s
## 1 2 aa
## 2 3 bb
## 3 5 cc
```

R has some datasets built-in for starting out and practicing. One of them is mtcars(). We will use it to go over data frames. To get a preview of the data frame, we can use either head() or tail() funtions, head() will display the rows from the beginning of the dataset, while tail() will display the last few rows.

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
tail(mtcars)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

In order to see the data types of all the columns in a data frame, we can use the str() function which will compactly display the internal structure of an object.

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
```

```
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

To get the number of rows and columns in a data frame, we use the nrow() and ncol() functions. The current data frame has 32 rows and 11 columns.

```
c(nrow(mtcars), ncol(mtcars))
```

```
## [1] 32 11
```

To get a particular column of a data frame, we use the $ operator. It enables us to get the column of a data frame by its *name (DataFrame$ColumnName)*.

```
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

Here, the numbers [1], [15], [29] are the row index of the column elements, i.e. **21.0** is the *mpg* element in the first row, while **10.4** is the mpg element in the fifteenth row.

These numbers might differ from user to user depending on screen size and resolution.

Working Directory

The code samples above assume the data files are located in the R working directory, which can be found with the function getwd().

```
getwd()
```

```
## [1] "C:/Users/User/Dropbox/5- IE6200/2- Course Evaluation/3- Labs/1- Lab 1 - Introduction to R"
```

Suppose you want to read a file **"abc.csv"** that is in the downloads folder. Since my current working directory is not the downloads folder R will not be able to find the file and will return an error. In such cases we need to use the setwd() function.

You can select a different working directory with the function setwd() and thus avoid entering the full path of the data files.

```
setwd("C:/Users/User/Downloads")
```