Labsheet - 1

# Introduction to R

Machine Learning

BITS F464

2nd Semester, 2019-20

## 1. R:

R is an open source programming language developed for use in the fields of statistics and graphics and is used by data miners and statisticians to analyse and derive insights from data.

R is distributed as a kernel and it's usually used along with RStudio, a popular free IDE for the same. The steps you can follow to set it up on your computer are:

a. Download and Install **R.** Download and Install **R Studio.**
b. Familiarize yourself with the RStudio interface. You can use the following resources:
    i. RStudio Cheatsheets for a variety of purposes and use cases
    ii. List of Keyboard Shortcuts

## 2. Getting Started:

You can follow the following steps to setup and use an elementary script for R within RStudio:

1. Set the current working directory of RStudio as the folder where you would want to store all the code, data files and would act as the root of paths supplied by you. Do this by going into *Session* > *Set working directory* > *Choose directory*. (Verify the directory by running *getwd()* into the terminal within the IDE)
2. Create a script window by selecting *File* > *New file* > *R script.* All the commands hereon would be written into this script file and not directly on the terminal to ensure code usability. Some useful shortcuts are:
    a. *Ctrl* + *R*: Run complete or selected fragment of code

b. *Ctrl + S/O*: Save/Open a file

3. Assign value to a variable x by typing:

```
x <- “Hello World”
```

And then view its value on the console by simply writing x on the script. Your script overall should look something like this

```
x <- "Hello World"
x
```

4. You can now select both these lines to run the code by pressing Ctrl + R and thus, giving the following output:

```
> x <- "Hello World"
> x
[1] "Hello World"
```

This shows that R simply executes the entire script line by line as if you were running it on the terminal one by one. Also, it’s possible to view the value of any variable defined by us in any of the scripts that we have already run both on the GUI and the terminal as the value of each of them are saved in the current workspace.

## 3. Some basic information:

### a. Variable types:

R is a dynamically typed language like Python, i.e. it doesn’t require us to explicitly define the type of any variable that we use. Just like Python, R has the following data types at the most basic level:

i. integer: 0, 1, … stored specifically as an integer
ii. double: 1.2, 1111.3432, etc. stored with a double precision
iii. character: “Machine Learning”, “Regressions”
iv. logical: TRUE, FALSE

It’s possible to view the data type of any variable by using the command, *typeof(var),* where var represents whatever variable we are looking at.

### b. Vectors:

Vectors represent a series of values of the same data type within R and is the equivalent of arrays in other popular programming languages.

To create a vector we often use a simple function such as *c, rep, sample, paste* or an operator such as '':'' for example:

```
 x <- c(0, 1.2, 8/5)
 y <- 1:10
z <- rep(10, times = 2) # Repeat 10 twice
#Random Sample from c with replacement
a <- sample(c(2,5,3), size=4, replace=TRUE)
```

Even single variables are considered vectors of length 1 and this can be verified by looking at *is.vector(var),* for any given variable.

## c. Vector Ops:

Suppose we have 4 vectors, namely:

```
a <- 1:4
b <- 123:126
c <- 12:14
d <- 98:101
```

We can perform the following operations on it:

I. `a + b` and `a - b`
II. `a * b`
III. `a * c` and `c * 1` and `c + 1`
IV. `d[1 : 3] <- c`

Here, I and II will result in element wise operations while III would cause the shorter vector to be repeated to match the dimensions of the longer vector. IV would cause the elements starting from the 1st index until and not including the 3rd element to be replaced with the elements from c.

## d. Matrices:

It's possible to create a matrix using the following command:

```
mat <- matrix(1:12, nrow = 3, ncol = 4)
```

Here, the first parameter defines the list of values to fill the matrix with in column major order (column by column) and the number of rows and columns in the matrix after that. If the number of values supplied is deficient, then R cycles the values to complete the matrix.

You can add new rows and columns by using the *rbind()* and *cbind()* commands.

```
mat <- rbind(mat, c(1,2,3,4))
```

```
          mat <- cbind(mat, c(1,2,3,4))
```

This will give you the following matrix:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10    1
[2,]    2    5    8   11    2
[3,]    3    6    9   12    3
[4,]    1    2    3    4    4
```

e. Matrix Ops:

Let's create 2 matrices to perform the rest of the operations on:

```
          a <- matrix(1:4, nrow = 2, ncol = 2)
          b <- matrix(1:4, nrow = 2, ncol = 2)
```

We can perform the following operations:

```
      I.  a * b     #Element-wise multiplication
     II.  a %*% b   #Matrix Multiplication
    III.  t(a)      #Transpose
```

Some other operations can be found here.

Indexing can be done as (using the matrix defined earlier):

```
      I.  mat[0, ]  #Get the entire row(s)
     II.  mat[ ,0]  #Get the entire column(s)
    III.  mat[1:2,] #Get the first row
```

Basically, the indexing in a matrix is just like that in a vector with individual controllers for row and column. R discards the second dimension from any result if it only contains 1 element and might end up giving a vector like in the 2D matrix case. We can prevent this by using the *drop* flag.

```
          mat[1:2, , drop = FALSE]
          mat[1:2, ]
          #Run both the commands and see the difference
```

f. Tensors

The tensor product of two arrays is notionally an outer product of the arrays collapsed in specific extents by summing along the appropriate diagonals. For example, a matrix product is the tensor product along the second extent of the first matrix and the first extent of the second. Thus ***A %*% B*** could also be evaluated as tensor(A, B, 2, 1), likewise ***A %*% t(B)*** could be tensor(A, B, 2, 2).

Generally, an array with dimension comprising the remaining extents of A concatenated with the remaining extents of B.

If both A and B are completely collapsed, then the result is a scalar without

a dim attribute. This is quite deliberate and consistent with the general rule that the dimension of the result is the sum of the original dimensions less the sum of the collapse dimensions (and so could be zero). A 1D array of length 1 arises in a different set of circumstances, eg if A is a 1 by 5 matrix and B is a 5-vector then tensor (A, B, 2, 1) is a 1D array of length 1.
Some special cases of tensor may be independently useful, and these have got shortcuts as follows.

| | |
|---|---|
| %*t% | Matrix product A %*% t(B) |
| %t*% | Matrix product t(A) %*% B |
| %t*t% | Matrix product t(A) %*% t(B) |

```
A <- matrix(1:6, 2, 3)
dimnames(A) <- list(happy = LETTERS[1:2], sad = NULL)
B <- matrix(1:12, 4, 3)
stopifnot(A %*% t(B) == tensor(A, B, 2, 2))
```

## 4. Flow Control:

Following constructs are available in R: `for, while, repeat, if, else, switch, break` and `continue`. Following *for* loop example is self-explanatory,

```
for( i in 1:10) {
   print(i)
}
```

Another example

```
Vect01 <- c(10, 12, 14, 44, 43, 21, 21)
for( i in Vect01) {
   print(i)
}
```

## 5. Problems:

**Problem 1:**
Populate a square matrix and find out if the sum of elements of ith Row and ith column is same for all i. (sum of elements of the ith row and jth row need not be same)

**Problem 2:**
Write a R program to find inverse of a given non-square matrix.