A recap on R

STAT30270 – Statistical Machine Learning

Contents

1 R and RStudio

R is an open source statistical software and programming language describing itself as *A Programming Environment for Data Analysis and Graphics*. The latest release is freely available at `http://www.r-project.org`. R is a free software environment for statistical computing and graphics. It is open-source and is constantly being updated as people from across the world add new packages, which implement a multitude of statistical and machine learning techniques. It is used by a large number of academic researchers and companies across the world: Google, Meta, Novartis, Lloyds of London, Deloitte… and the list goes on!

Given the open source nature of R, a number of alternatives (and/or interfaces) to the basic version of R have become available. The most popular is probably **RStudio**. RStudio is an integrated development environment for R, which provides many features and a clean and professional interface to R, all these together make much nicer and efficient to work with the software. The RStudio interface is available through the **Posit** corporation website here: `https://posit.co/`.

The layout of the console, graphics and script windows in RStudio are well organized and easy to read and to work with. You can store a copy of all of your code in the script in order to keep a record of all the data analysis steps and re-run code as required. You can also create projects which make very easy to work in a structured way on different data analysis tasks. For more information about projects see here: `https://r4ds.had.co.nz/workflow-projects.html`.

2 Resources

In this module, we will use R mostly at a high level for implementing the machine learning methods presented in the material, hence the lab notes are designed to get us up to speed with the various methods quite quickly, sometimes at the expense of skipping some of the programming aspects. You will get familiar with some of these programming aspects of R as we use it throughout the course, while the programming basics of R will be given as already acquired.

If you are not fully familiar with R or want to refresh programming aspects of R, then you will need to cover some of the fundamentals in more detail as extra study material. Some preliminary material is given below. Bear also in mind that multiple references and manuals are available online to learn R.

- A set of notes called *An Introduction To R* is available at:
  http://cran.r-project.org/doc/manuals/R-intro.pdf
- The book *An Introduction to Statistical Learning* contains a lot of R code examples and is downloadable here:
  https://www.statlearning.com/
- The online book *The R Cookbook* is an excellent guide to learn R from the basics:
  https://rc2e.com/

If you want to refresh some concepts or R programming, **it is strongly suggested that you take some time to go through Chapters 1-6, 8, 10-11 of the R Cookbook**.

3 Some fundamentals of R needed for this module

Below we refresh some basic aspects of R programming. This document offers a recap of some functions that will be useful throughout the course. Check the documentation of these functions for further details about their usage. We will also pick up some of the fundamentals of R as we learn how to implement the different machine learning methods in R.

3.1 Packages

There are many packages that have been written for R. Just look at `http://cran.r-project.org` to see how many are now available. When developing a new statistical model/method, many researchers will produce an R package to implement the method and make it available to the community. This is one of the reasons why R has proven to be so popular.

We will need to use the `kernlab` package to implement support vector machine classifiers (you will learn about them later). To install the package then you can just type:

```
install.packages("kernlab")
```

The package can then simply be loaded via the `library` function:

```
library(kernlab)
```

3.2 Help!

Routinely when you are using R, you will need information and help on a particular topic or function. To do this, you just type `?function-name`, which returns info on the function. Try for example

```
?mean
```

Similarly, sometimes you will not know the command/function/package that you want to use in R in relation to a particular topic. A very useful tool to help here is the `help.search()` or `??` command. For example, to find out about about quantiles you may type

```
help.search("quantiles")
# same as
??quantiles
```

This will provide a number of options, some of which are useful to compute quantiles.

In general **it is strongly recommended that you take time to read the documentation of the functions and packages** you are using during data analysis.

3.3 Objects

R is an *object-oriented* programming environment, which means that when programming in R, you can create objects to store data, results, function, and everything you may need for analysis. There are many different types of objects in R and you can even create your own objects. The main types of objects that we will use are: `vector`, `matrix`, `array`, `function`, `list`, and `data.frame`.

The definition of these objects in R is below:

- `vector` – A vector of objects of the same type.
- `matrix` – A two-way array that stores items of the same type.
- `array` – A multi-way object that stores items of the same type.
- `function` – Code that takes R objects as an input and outputs other R objects as output.
- `list` – An object that can be used to store objects of different types.
- `data.frame` – A matrix-like object but where the columns can have different types and correspond to entries of a list (in fact a `data.frame` is a particular type of list).

Objects have the great advantage that can be manipulated easily, as well as it easy to access elements of interest within objects.

3.4 Simple manipulations on vectors

We can create vectors simply by using different alternatives according to the type of vector. Try the following commands:

```
x1 <- c(2,3,4,5,6)
x2 <- 1:5
x3 <- c("red", "blue", "green", "black")
x4 <- seq(from = 2, to = 5, by = 0.1)
x5 <- seq(from = 0, to = 10, length = 21)
x6 <- sample(1:100, 20)
x7 <- rnorm(100, mean = 0, sd = 3)
```

Notice that vectors include elements of the same type. To print any object, you just type its name on the console.

```
x1
x3
x7
```

To access components of a vector you can use the indexing operator `[]`. For example:

```
x1[3]                  # 3rd element of vector x1
x5[c(2, 4, 6, 8)]      # 2nd, 4th, 6th and 8th elements of the vector
x7[c(19:27, 99:88)]
```

Elements can be dropped from a vector combining the indexing with the minus sign:

```
x3[-3]         # remove element 3
x1[-c(1,2)]    # remove elements 1 and 2
```

Elements of a vector can be accessed also using logical expressions:

```
x5[ x5 < 7 ]    # selects elements of x5 less than 7
x3[ x3 == "red" | x3 == "green" ]    # selects only red or green
x7[ x7 > 1 ]
```

3.5 Matrices

A matrix is simply a 2-dimensional array. A matrix can be created using the command `matrix` and accessed via the `[]` operator, this time providing two entries. Take some time to go through these lines of code:

```
mat1 <- matrix(sample(1:50, 12), 3, 4)   # create some matrices
mat2 <- matrix(runif(9), 3, 3)
mat3 <- matrix(rnorm(9), 3, 3)

# common operations
t(mat1) %*% mat2
crossprod(mat1, mat2)  # same but faster
mat1 %o% mat2          # outer product
```

```
# dimension-wise operations
colSums(mat1)
rowSums(mat1)

# manipulate content
mat1[1,3]      # access elemnt in row 1, column 3
mat1[2,]       # access row 2
mat1[3:1,2:3]  # access a subset
mat2[,3]       # access column 3
mat2[-3,]      # remove row 3
```

Remember that the algebra operations in R are vectorized. Run these examples and try also other operations.

```
mat2 + mat3
mat2 * mat3
mat1[,1] + mat1[,2]
mat1[1,] + mat2[1,]   # be careful!
```

```
## Warning in mat1[1, ] + mat2[1, ]: longer object length is not a multiple of
## shorter object length
```

3.6 Lists

A list is an object that can contain different types of objects. Elements of a list can be easily accessed using the `$` operator. Try to run the following example.

```
# create a list
y <- list(a = 1:5, b = matrix(1:16, 4, 4), c = c("A", "B", "C"))
y

y$a
y$b
y$b[1:2,1:2]
```

3.7 Functions

Functions are the building block of R, in fact all the operators are functions - try to run `"+"(2, 5)` from the console. There are loads of base R functions available for various tasks. You can also create your own functions. Arguments of R functions can be assigned default values, and argument matching is either by name or by position. Explore the different examples below, and create and play with other functions.

```
fun <- function(x, y = 10) {
  z <- x * y
  out <- z^2
  return(out)
}

fun(x = 4)
fun(4)
fun(4, 2)
fun(y = 5)
```

3.8 Choices

"If" statements are control structures allowing to test a condition and perform an operation depending whether the condition is true or false. The condition evaluates to a single `TRUE` or `FALSE` and the output of `if` is a value which can be assigned to an object. Try this example.

```
x <- sample(1:100, 1)
```

```r
y <- if ( x > 50 ) {
  # some instructions here
  z <- x^2
} else {
  # other instructions here
  z <- log(x)
}
x
y
z  # also available in the workspace
```

Sometimes you will need to modify the entries of a vector or a matrix according to certain logical conditions. The **vectorized** if function `ifelse` can handle a vector of logical conditions and is useful for in-place modifications to a vector or matrix. Check the documentation for `?ifelse` and run the examples below.

```r
x <- sample(1:100, 5)
x
ifelse(x > 50, "big", "small")

# if condition met, change, otherwise leave it as it is
ifelse(x > 50, x/100, x)

# change entries of a matrix according to condition
mat <- matrix(runif(10), 2, 5)
ifelse(mat > 0.5, 1, mat)
```

3.9 `for` loops

Standard `for` loops take an iterator variable and assign it successive values from a sequence or vector. Loops are mostly used for iterating over the elements of an object and perform repeated operations. Any iterator variable can be used, and looping can be performed over any set of elements. Check the examples below.

```r
for ( i in 1:3 ) {
  # simply print iterator
  print(i)
}

x <- 1:10
for ( i in 2:10 ) {
  # compute strange rolling sum
  x[i] <- x[i] + x[i-1]
}
x

w <- letters[1:4]
for ( letter in w ) {
  print(letter)
}
```

Loops can be nested in order to perform multiple operations and operations on multivariate objects. We will make use of nested loops in the following labs, but sometimes loops should be avoided. Check the example below - **NOTE:** if unsure about some of the commands, check the help documentation!

```r
# compute matrix product C = AB
A <- matrix(1:12, 4, 3)
B <- matrix(sample(1:30, 9), 3, 3)
C <- matrix(NA, nrow(A), ncol(B))

for ( i in 1:nrow(A) ) {
```

```
  for ( j in 1:ncol(B) ) {
    C[i,j] = sum( A[i,] * B[,j] )
  }
}
C

# waaaaay better
A %*% B
```

3.10 `while` loops

A `while` loop repeat operations until a condition is no longer true. This type of construct is useful for implementing numerical algorithms, where convergence is assessed iteratively and the algorithm stops when there is no longer (significant) change in the objective function.

Let's compute the square root of a number numerically using the Hero method.

```
x <- 7

condition <- TRUE         # initialization
root <- prev_root <- 4    # initial guess for sqrt value
while ( condition ) {
  # compute sqrt(x) using Hero method
  root <- (root + x/root) / 2
  condition <- abs(root - prev_root) > 1e-5  # check convergence
  prev_root <- root
}

root
sqrt(x)
```

4 Data exploration

A `data.frame` is a particular type of list arranged in the form of a matrix. Dataframes are used to store data in R, and several commands are used to import data from files into R. Take a look at `read.csv` and `read.table`. Many datasets are also already loaded in R. Run `data()` and have a look. Try for example `data("iris")` and `?iris`.

In general, there are multiple options to load data in R from a file, according to the format of the file. We will mostly work with data files having the following formats:

- `rda` (or `RData`) - This is the standard data format of R and it usually contains copy of objects present in a workspace. Data files of this type are loaded in the workspace using functon `load`.
- `csv` and `txt` - Comma separated values (.csv) and text documents that contains plain text (.txt) are standard formats for storing data. These can be open using standard spreadsheet softwares and can be imported in R using functions `read.csv` and `read.table`.

As an example, we consider data from the World Happiness Report concerning numerical indicators employed to quantify the happiness of a nation; see https://worldhappiness.report/ for more information. The data contain measurements on indicators for a number of countries in the world over 3 different years, and the variable "Happiness" is a proxy measure for the level of happiness of a country. The other numerical variables measure the contribution of each indicator to the calculation of happiness.

```
load("data_happiness.rda")
```

The data are now stored in the dataframe `happiness` available in the workspace. To extract certain useful information from the data, you can use the following functions.

```
head(happiness)
View(happiness)    # not recommended for high-dimensional data
```

```
str(happiness)
dim(happiness)

nrow(happiness)
ncol(happiness)
length(happiness)  # don't forget that a dataframe is a list

colnames(happiness)

summary(happiness)
```

Categorical and discrete variable can be easily explored using the command `table`

```
table(happiness$Macro)
table(happiness$Macro, happiness$Region)
```

In many data analysis tasks we may need to perform an operation to each column of a dataset, or apply a function according to a grouping variable. This could be done by using a `for` loop, iterating over the columns or over a logical variable. The `apply` and `tapply` functions offer a faster and easier solution.

For example, we can use `apply` to compute the mean and standard deviations of each of our numerical variables in the data. The function takes in input the dimension over which applying the function (`1` for rows, `2` for columns) and the function `FUN` that we want to employ, which can be user defined. See `?apply`.

```
apply(happiness[,c(5,7:13)], 2, mean)
colMeans(happiness[,c(5,7:13)])   # very common operation

# compute column standard deviation
apply(happiness[,c(5,7:13)], 2, sd)

# compute column log(mean)
apply( happiness[,c(5,7:13)], 2, function(x) log(mean(x)) )
```

If instead we want to compute the average mean GDP by region, we use the function `tapply`, providing in input the grouping vector in the argument `INDEX` and the function `FUN` (which can be user defined).

```
tapply(happiness$GDP, happiness$Region, mean)
```

Other functions similar to `apply` and `tapply` which may come useful later on are `lapply` and `sapply`, which allow to apply a function over the entries of a list; see their documentation.

We can visualize the data through scatterplots and boxplots using the functions `pairs` and `boxplot`.

```
happiness_num <- happiness[,c(5,7:13)]
pairs(happiness_num)
#
# fancier
pairs(happiness_num, gap = 0,
      col = factor(happiness$Macro),
      pch = 19)

boxplot(Happiness ~ Region, data = happiness)
boxplot(Happiness ~ Year, data = happiness)
```

We can fit a linear regression model of Happiness as a function of GDP, Family and Health. From the scatterplot, the relation between Health and Happiness could be quadratic, so we add a quadratic term.

```
reg <- lm(Happiness ~ GDP + Family + Health + I(Health^2), data = happiness)
summary(reg)

# predict the value of happiness from the model for a country with given values of the covariates
new <- data.frame(GDP = 1.5, Family = 0.9, Health = 0.5)
```

```
predict(reg, new)
```

4.1 Task

Refresh some on the main R commands you are already familiar with by exploring the data. To be up to speed with the module, you should be able to perform the following tasks:

- Create a categorical variable `happy_cat`, indicating if the overall happiness of a country is "high" (greater than 6), "medium" (between 6 and 4) or "low" (less than 4).
- Produce a plot of the numerical data, relating the variables to the different levels of the variable `happy_cat`.
- Identify the top 10 countries by happiness in the year 2017.
- Identify the overall top 3 countries by happiness across all years and regions.
- Compute the median Happiness and Freedom indexes by Region and produce a visual representation to explore their relation.
- Split the data into two datasets: a dataset with the 2015 and 2016 observations, and a dataset with the 2017 observations. Fit a regression model of Happiness as a function of all the other numerical variables (except Rank) using the 2015-2016 data. Use this regression model to predict the values of happiness in 2017 and check the quality of the predictions using a visual representation.

Note that other functions not presented in this document could be used to implement the tasks above.