INGI2262 – Machine Learning

Assignment 1

An Introduction to R

1 Introduction

During this course you will learn how to apply machine learning algorithms, and sometimes to adapt them. To do so, you should get familiar with the software environment provided with the R programming language.

The R input data to machine learning methods is usually stored in several data files, represented, for instance as numerical matrices, or, more generally, *data frames*. Those can be seen as a generalization of matrices, which are able to store different types of data simultaneously. Such data often require to be (pre-)processed before or when the actual learning algorithm is applied. This data processing can be done directly within the R environment. The purpose of this assignment is to introduce you the basics of the R programming language and to give you some practice of work in the R software environment.

At the end of this assignment, you should be able to write a small R program, load it and execute it to perform a simple machine learning task. Through this assignment you will be led iteratively to the creation of a R program. The incremental steps proposed hereunder are just a suggested flow. At each step, it is recommended to reuse the code of previous steps as much as needed.

2 Your first R session

R is a free software environment for statistical computing and graphics[1]. It compiles and runs on a wide variety of Unix platforms, Windows and MacOS. It is available on the INGI student machines and you are welcome to install it as well on your own computer. R offers a powerful *programming language*, with loads of pre-existing *packages*. R also comes with an interpreter to execute interactively R *functions* from a R *terminal*. R also includes lots of *graphical tools* to visualize data, models and results.

The first chapter of the R tutorial[2] contains only 6 pages that **must** be read first. Delighted and brave as you are, we strongly recommend to pursue your reading till ending chapter 3 on page 15. The rest of this tutorial could be used as a quick reference guide.

For those liking to put things in practice immediately and learn from it, we also recommend the book *Statistics and Data with R, an applied approach through examples* by Y. Cohen and J. Cohen[3]. At times, we will refer to this book using the *SDR* acronym.

[1] Check http://www.r-project.org

[2] Check http://cran.r-project.org/doc/manuals/R-intro.pdf

[3] The first part of this book is available from the course website.

We will guide you below through the use of **R** from a simple *terminal*, that is a simple window where you type in **R** commands and get results.

An alternative approach would be to install and to use **RStudio** which is a powerful IDE[4] offering many additional functionalities: an R source editor, integrated help, package management tools, and many more... You can get a sense of the RStudio environment by watching the short R video tutorial available from Moodle (look at the links included in the first week schedule). The **R** terminal mentioned below is just one specific window of RStudio, known as the *Console*. We let you discover the roles of the other RStudio windows by yourself, if you decide to use this IDE.

We stick now to the minimal view of a unique window. You should check first that R is installed in your computing environment. In a Unix/Linux context, launching it amounts to type **R** in a terminal. You should then get something like the following lines. The last line (starting with **>**) is the R *prompt* used to enter commands[5].

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (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.

  Natural language support but running in an English locale

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.

>
```

Let us first define a vector **x** made of the 10 digits:

```
> x <- 0:9
```

We can check what **x** has been assigned to[6],

```
> x
 [1] 0 1 2 3 4 5 6 7 8 9
```

Note that we could have used **=** instead of **<-** as an assignment operator[7].

```
> x = 0:9
> x
 [1] 0 1 2 3 4 5 6 7 8 9
```

We can now multiply each element of **x** by **2**, and check the result:

[4] Integrated Development Environment.

[5] We let you translate our discourse to a MacOS or Windows context if needed.

[6] Read the documentation to figure out what **[1]** refers to.

[7] The subtle difference between both operators is beyond the scope of this introduction.

```
> x <- x*2
> x
 [1]  0  2  4  6  8 10 12 14 16 18
```

An alternative way of defining a vector is through the **seq** function.

```
> y <- seq(from=1,to=20,by=2)
> y
[1]  1  3  5  7  9 11 13 15 17 19
```

From your careful reading of the R documentation, you surely recall that typing **?** in a R terminal, followed by the name of a function, returns the usage of this function. We suggest the following example.

```
> ?seq
```

We can now multiply **x** and **y** component-wise and see the result

```
> x * y
[1]   0   6  20  42  72 110 156 210 272 342
```

Beyond what you define yourself, such as **x** and **y** above or your own R functions (as you will learn soon), your R environment comes with several *installed packages* already containing various functions or datasets. To get the list of those packages, type the following.

```
> library()
```

Some of those packages are loaded by default in your current working environment, also known as your R session. The following command returns the list of currently *attached packages*.

```
> sessionInfo()
```

To get a sense of what a specific package, say **stats**, is doing, you simply pass the package name as an argument to the help command:

```
> help(stats)
```

If you need a function from an installed package not yet attached to your session, you should load it first. Here is how to do it for the **lattice** package:

```
> library(lattice)
```

This package will be available till the end of your R session or till you detach it.

```
> detach(package:lattice)
```

Before ending your first R session, you can check the list of objects currently available in your environment.

```
> ls()
[1] "x" "y"
```

Whenever you finish your R session through the **q()** command, you are offered the possibility to save your workspace.

```
> q()
Save workspace image? [y/n/c]: y
```

If you answer **y** (yes) , the **x** and **y** objects will be already defined, and assigned to the same values, the next time you launch a R session in the same directory[8]. Whenever you start a new R session, you should then see:

[8]The environment is saved in the `.RData` file in the current directory.

```
...

[Previously saved workspace restored]

>
```

Note however that the additional packages you loaded explicitly during a R session need to be reloaded whenever you start a new session. Writing your own R programs will help you to automate this step. Note also that saving your environment from one R session to the next may be convenient but at times confusing if you forgot that you had already defined some objects in a previous session. Listing the available objects through **`ls()`** is useful in this regard. In contrast, you can also make sure that you clean things up. Here is a convenient function that you can write and call to do so.

```
# Clean up all symbols/functions defined in the Global Environment
> clean <- function() {
    env = .GlobalEnv
    rm(list=ls(envir=env),envir=env)
}
> clean()
```

3 A simple linear regression example

A standard machine learning problem is the estimation of a regression function, as presented in the introductory lecture. The simplest case, called *linear regression*, aims at modeling the relationship between an output variable y and one or several explanatory variables x_j's by a linear function

$$f(\boldsymbol{x}) = w_0 + \sum_{j=1}^{p} w_j \cdot x_j,$$

where p is the number of explanatory variables, collectively denoted as the vector $\boldsymbol{x}$. The parameter w_0 is called the intercept which influences where the linear function intersects the axes. The additional model parameters, or weights, are $w_1, \ldots, w_p$.

The quality of such a function to accurately model the dependence between $\boldsymbol{x}$ and y, that is, to which extent y can be accurately predicted from the explanatory variables $\boldsymbol{x}$, is usually measured according to a *squared loss*:

$$\sum_{i=1}^{m} (f(\boldsymbol{x}_i) - y_i)^2,$$

where m is the number of data points on which the model f is assessed. The loss or error function reaches its minimal value (0) whenever each y_i is perfectly predicted by its corresponding $f(\boldsymbol{x}_i)$.

We will first consider the simplest example where y depends only on a single explanatory variable x (that is $p = 1$) and we will further assume that the intercept w_0 is 0. In other words, we restrict the set of possible models to lines passing through the origin. Such a simple linear model can be defined in a simple R function, which you are invited to encode yourself in a R session.

```
> linearmodel1 <- function (w, x) {
      w * x
  }
```

The error function is here the squared loss:

```
> sqloss <- function (w, x, y) {
      sum ((linearmodel1(w,x) - y)^2)
  }
```

The single parameter of our model is represented by **w**, here the slope w_1 of a line, while **x** and **y** represent the input/output data. Changing the **w** value will affect the model, which will be reflected in the loss or error function. Let us now generate and plot some data:

```
> x <- 1:10
> y <- 3*x
> plot (x,y)
```

Here, by construction, y has a linear dependence on x with a specific slope of 3. A simple linear model should thus perfectly explain such a dependence. Let us represent this model with $w_0 = 0$ and $w_1 = 3$.

```
> w0 <- 0
> w1 <- 3
> abline (w0, w1, col = "red")
```

We can check that the loss is indeed null with such a model.

```
> sqloss (w1, x, y)
```

In general, optimizing the loss function with respect to the parameters provides the best model. To illustrate this, let us plot the values of the loss function for different values of the parameter w_1. We first generate a range of parameter values in the interval $[-2, 5]$.

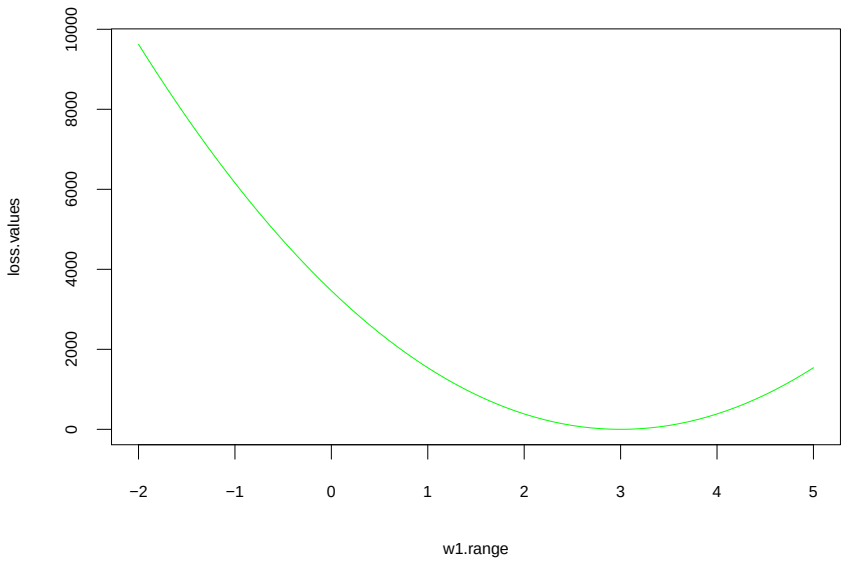
```
> w1.range <- seq (-2,5, by=0.1)
```

We compute the loss values for all parameter values in the specified range[9]:

```
> loss.values <- sapply (w1.range, sqloss, x, y)
```

Let us represent graphically the value of the loss as a function of the parameter w_1.

```
> plot(w1.range, loss.values, col="green", type = "l")
```



Such a bowl-shaped function is *convex* which has the nice property of having a single minimum. To find the optimal parameter value, we can here simply optimize along the single dimension w_1

[9] You are invited to consult the R documentation to fully understand what this instruction is actually performing.

```
> optimize (f=sqloss, w1.range, x, y)
```

If everything has gone smoothly so far, you should have rediscovered, with little surprise, that the optimal value of the loss is 0 and is obtained for $w_1 = 3$.

You are invited to play with the above ideas and repeat those experiments while

1. considering a non-zero intercept,
2. adding a small and distinct random fluctuation[a] around each y_i,

As you are unlikely to want to type again and again similar instructions, we recommend you to use a text editor[b] to store your functions (or more generally any R code) in a R source file. Call it, for example, **`linear.R`** and load it through your R session as follows.

```
> source("linear.R")
```

[a] Hint: **`?runif`**

[b] Some text editors, like *emacs*, have a convenient statistical mode to edit R programs and to execute them jointly. An interesting alternative is RStudio available at http://www.rstudio.com. This IDE includes a specific window where you can directly edit some R code and, for instance, execute it interactively line by line.

4 A multi-dimensional linear model

We are now ready to work with real data and to generalize the linear regression model to several input dimensions. In other words, we will try to predict a response variable y from several input variables x_j's, also known as *covariates*, *explanatory* or *independent variables*.

The *bodyfat* dataset includes 15 variables (such as **`Age`**, **`Height`**, **`Weight`**, ...), including a body fat index (**`BFI`**) for a human being. The dataset is stored in a comma separated file, `bodyfat.csv` available on Moodle for this assignment. The first step is to load this data into your R session.

```
> bodyfat <- read.csv("bodyfat.csv", row.names=1)
```

The data is now stored in **`bodyfat`** in the form of a very common R object known as a *data frame*. Such a data structure generalizes the notion of matrix by allowing elements of different types. You are invited to consult the R documentation to learn more about data frames (see for example *SDR*, section 2.2.3). The first way to look at this (small amount of) data is to type in the name of this data frame in your R session and check the output.

```
> bodyfat
```

You can check the dimensions of this data frame and get their respective names:

```
> dim(bodyfat)
> dimnames(bodyfat)
```

R also offers very convenient graphical tools to *look at the data*[10]. A scatter plot is easily produced by looking at the data in pairs

[10] Did we already insist on this aspect?

```
> pairs(bodyfat)
```

You can restrict such a visualization to specific variables, such as **BFI**, a bofy fat index, **Weight**, and **Height**.

```
> pairs(bodyfat[,c("BFI", "Weight", "Height")])
```

For instance, the upper-right corner of the scatter plot shows you how **BFI** varies as function of **Height**. An alternative view relies on a lattice plot (see *SDR*, section 3.6) through the **xyplot** command of the **lattice** package.

```
> library (lattice)
> xyplot(BFI ~ Weight | Height, data = bodyfat)
```

Anything on the left of **~** is a dependent variable and on the right an independent variable. The vertical bracket **|** indicates conditioning. So, we plot **BFI** as a function of **Weight** conditioned on the **Height**.

We will now try to predict the body fat index **BFI** as a linear function of the **Height** and the **Weight** of the person. The coefficients (or weights) of such a linear model can be conveniently estimated through the **lm** function.

```
> linearmodel <- lm(BFI ~ Height + Weight, data = bodyfat)
> w <- linearmodel$coefficients
```

The computation of predicted values from such a linear model can be encoded as follows:

```
> predict <- function (x, w) {
    intercept <- w[1]
    intercept + as.matrix(x) %*% w[2:length(w)]
}
```

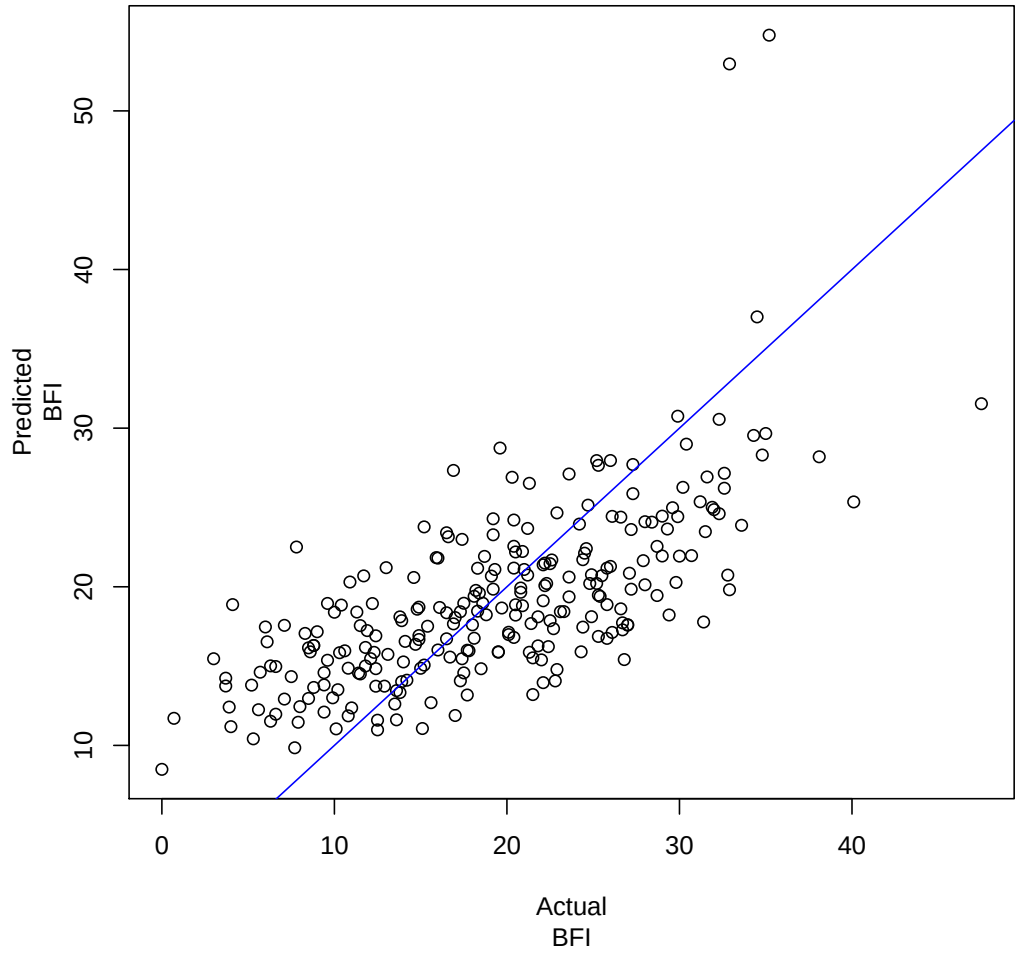
Recall that the first component **w[1]** of the weight (= coefficient) vector is the intercept. The remaining components are accessed through **w[2:length(w)]**. We also generalized the form of the input data since **x** is assumed to store several dimensions for each observation, and consequently we use the matrix product operator **%*%**. The predicted values for those obervations are easily available.

```
> x <- bodyfat[,c("Height","Weight")]
> predicted <- predict(x,w)
```

We can plot the predicted values as compared to the actual **BFI** values and check visually the discrepancies

```
> plot (bodyfat[,"BFI"],predicted, xlab="Actual BFI", ylab="Predicted BFI")
> abline (0,1, col="blue")
```

Note that the blue line is **not** a graphical representation of the linear model (which is no longer a line but a plane since we are working in 2D now) but a way to look at the differences between the actual response values (on the horizontal axis) and the predicted ones from the model (on the vertical axis).



You are asked to perform the following tasks.

1. write a simple function to compute the associated squared loss and call it, for instance, as follows

```
> y <- bodyfat[,"BFI"]
> squareloss(y, predicted)
```

2. propose a model estimating the linear dependence between the body fat index (the **BFI** variable) and the 3 following covariates: **Weight, Abdomen, Biceps** denoting respectively the weight, abdomen circumference and the biceps circumference of the person. Report the model coefficients (= the linear model weights) and argue whether the respective signs of these coefficients make sense. Report a plot representing the relationship between the predicted **BFI** by your model and the actual **BFI** across all samples in this dataset.

3. So far we have estimated a model on a whole dataset and assessed its performance through the squared loss computed on the **same** data. Such a procedure includes an optimistic bias. Indeed, what machine learning is really about is to estimate a model on some data and use it to make predictions on **new and previously unseen** observations. Write a function to split the *bodyfat* dataset in 50 % (= 126 observations) **training** and 50 % (= 126 other observations) **test**. The split should be conducted as a random sampling without replacement (each observation is exactly selected once, either in the train or the test)[a]. By repeating (say 100 times) this experiment for several random partitioning of the dataset into train and test, you should get an idea of the average loss you can expect on the train versus the test. Do you observe the optimistic bias mentioned above[b]?

   Report plots to sustain your claim. Note that the actual plots that you produce in a R session can be easily saved in a file to be included in another document. A convenient way to do so is to call the **pdf** function with a specific filename as argument. For example:

```
> pdf(file="LinearFit.pdf")
```

   Next, you call whatever graphical functions you want to use

```
> plot (...)
> abline (...)
```

   Finally, you finish the writing to the file and close it

```
> dev.off()
```

[a] Hint: **?sample**.

[b] For a sound comparison of the situations before and after splitting the data in 2 parts, it might be relevant to normalize the computed loss (*e.g.* by the number of samples on which it is computed).

5 Your TODO List

- **Join a group** of 2 students on Moodle.
- Check on Moodle the statement about your **active participation to the course assignments**. Each individual student must agree about this policy by logging on Moodle and by checking the "Agreed" option [a] (look at the first week activities).
- **Submit** for your group on Moodle **a PDF report** with answers to the questions of this assignment and **the R code** you have produced in a separate `.R` file. Include a brief description of your use of R functions.
- You are also welcome to use the Moodle student forum for this course in order to discuss any difficulty you would find in using the R environment.

[a] Failure to do so will imply a 0 grade for all course projects.