# G54DMA - Lab 4: Data Analysis in R

## Instructions

In this lab session, we will learn how to use R to analyse data. You will also learn how to use the popular package *dplyr* to manipulate and transform data.

For a comprehensive reference manual on all things R, please refer to *An Introduction to R* by Venables et al., which can be found on Moodle and here:

https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

For detailed information about the package *dplyr*, refer to its main document here:

https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

Thorough the labs, we will be using *An Introduction to R* as a reference manual. Chapters or sections of interest will be signalled with "***Reading:***" in the Instruction sheets.

## Learning outcomes

**After this lab session, you will be able to:**

- **Use R's built-in datasets**
- **Use R to describe data, including statistical summaries and cross-tabulations**
- **Use the package *dplyr* to manipulate and transform data**

## 1. Datasets and Repositories

During your work, you might want to test methods or develop different algorithms without having to worry about data collection and data management. For these purposes, you can use Data Repositories or Built-in Datasets in R. Data repositories and built-in datasets present a wide catalogue of different datasets that can be used for different tasks, such as analysis, visualisation, clustering or classification.

R has several built-in data sets available, including the famous ***Iris*** data. Type:

```
data() #gives repository of built-in datasets in R
```

to see a list of in-built data.

To access a dataset, just type their name. For example:

```
Orange #accesses the Orange dataset
```

To use these datasets, it is recommended that you save them in a variable and then use that variable in your exercises.

```
ir = iris
```

There are many more repositories on the internet that you can use. One of the most popular ones is the UCI Machine Learning Repository. You can download data files to experiment with from the UCI Machine Learning Repository:

http://archive.ics.uci.edu/ml

## 2. Analysing data

In this lab, we are going to use the ***Iris*** dataset for all our examples. First of all, copy it into your own variable. This will allow us to change names, modify values, etc.

The type of the ***Iris*** dataset variable, or 'class' as it is known in R terminology, is not a matrix, but a data.frame. Then, each field in the data.frame has its own class.

```
class(ir) #class of ir is data.frame
class(ir$Sepal.Length) #class of field Sepal.Length in ir is
numeric
```

As we saw in the previous labs, data frames broadly behave in a similar way to a 2D matrix in which each column can be of a different type.

Once ***Iris*** is in such a variable, we can set, view and use the names of each of the columns, rather than using the numeric column indices. For this, use the names in the original dataset:

```
ir$Sepal.Length #equivalent to ir[,1]
ir$Species #equivalent to ir[,5]
```

You can also change these names to whatever you want assigning it to `names(`*`dataset`*`)`.

Let's change the name of the numeric attributes so they are all written in lower case and also change the Species attribute to *class*, since it represents the classification of each different plant.

```
names(ir) = c("sepal.length", "sepal.width",  "petal.length",
"petal.width", "class")
names(ir)
ir$Species # Returns null because there is no field with that
name any more
ir$class
```

Of course, the numeric indices can still be used.

```
ir[1,]
ir[1:3,]
ir[c(1,51,101),]
```

Have a play with other indexing to understand the capabilities of R.

Now that you know how to load and modify data to access it easily, let's learn how to analyse it.

There are various functions to calculate statistical properties of data, such as:

```
mean(ir$sepal.length)
median(ir$petal.width)
```

You can also obtain a basic summary of the most important statistical properties of your data with:

```
summary(ir) # Are there any statistical measures that are
missing here?
```

R also has functions to perform tabulations and cross tabulations:

```
table(ir$class)
ir$sepal.length.class = ifelse(ir$sepal.length <6,  "setosa",
"other") # What is this instruction doing?
table(ir$sepal.length.class) # And this one?
table(ir$sepal.length.class, ir$class)
```

***Reading:*** For more information about analysing your data, read point 8.2 of *An Introduction to R*.

# 3. Data manipulation with dplyr

*dplyr* is one of the most popular pakages in R. Developed by Hadley Wickham, it is used for data selection, transformation, and summarisation. It is a very powerful tool, with over 50 functions.

In this lab, we will introduce the *dplyr* package and explain its most useful functions for data analysis and transformation. We will also introduce the concept of "*piping*" instructions in *dplyr*, which will come in handy when we are required to carry out many transformations over the same subset of data.

For more information and examples on all other functions here, see:

https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

## 3.1. Installing *dplyr*

Installing *dplyr* is very easy. Just use the *install.packages* function to install it. If necessary, you may also have to install the *DBI* package due to some dependencies.

```
install.packages("DBI")
install.packages("dplyr")
```

Remember to load the *dplyr* package before calling any of its functions:

```
library("dplyr")
```

## 3.2. Using dplyr

In this module, we will work with the most useful functions from *dplyr.* These are: *filter()*, *arrange()*, *select(), mutate(), summarize(), sample_n(), sample_frac(),group_by()*. We will also introduce the concept of piping.

**IMPORTANT!** While *dplyr* provides a simpler way of transforming and accessing data, it is very important to know that you can use basic R to carry out the same processes. You are expected to know how to do all of these operations in basic R.

### *3.2.1. Main Functions*

Here, we will show how to use the most popular *dplyr* functions and their equivalent implementations in basic R.

**Start by loading *iris* into a variable again. Do not change the names of the attributes.**

- *filter(dataframe, condition)*: Used to find rows and cases where certain conditions are true.

```
ir = iris
filter(iris, Species=="setosa") # using dplyr
iris[iris$Species=="setosa",] # using basic R

filter(iris, Species=="setosa" & Sepal.Length>5) #
using dplyr
iris[iris$Species=="setosa" & iris$Sepal.Length>5,] #
using basic R

# note how filter re-numbers all returned rows from 1,
while indexing returns their original row number.
```

- *arrange(dataframe, data_fields)*: arranges (orders) rows by variables.

```
arrange(iris, Sepal.Length) #dplyr
iris[order(iris$Sepal.Length, decreasing = FALSE),] #
basic R
iris[order(iris$Sepal.Length),] # basic R

#by default, order will sort values in ASCENDING
manner.

arrange(iris, desc(Sepal.Length)) # the desc function
allows you to arrange that data  field in descending
order
```

```
iris[order(iris$Sepal.Length, decreasing = TRUE),]
iris[order(-iris$Sepal.Length),]

arrange(iris, Sepal.Length,Sepal.Width)[1:5,]
iris[order(iris$Sepal.Length, iris$Sepal.Width,
decreasing = FALSE),][1:5,] #by indexing after calling
each function, we can look at the top N  rows (or
results). Here, we are looking at the top 5 results

arrange(iris, Sepal.Length, desc(Sepal.Width))[1:5,]
iris[order(iris$Sepal.Length, -
iris$Sepal.Width),][1:5,]
```

- *select(dataframe, var1,…,varX*): keeps the variables you input as parameters

```
select(ir,Petal.Width, Species)
ir[,c("Petal.Length","Species")]

select(ir,1:3) #you can use indexes, too!
ir[,1:3]

#combine select with functions “starts_with” and
“ends_with” to select groups of variables with
commonalities in their names
select(ir,starts_with("Petal")) #Petal.Length and
Petal.Width
select(ir, ends_with("Length")) #Sepal.Length and
Petal.Length

#combine select with – (dash) to deselect fields
select(ir, -Species)
ir[, -c(5)]

select(ir, -stars_with("Petal"))
```

- *mutate(dataframe, expression)*: adds new columns to the data frame so that they follow the function in *expression*

```
ir = mutate(ir, Sepal.Length*2)
```

```
ir = mutate(ir, DoubleSepalL = Sepal.Length*2)
ir$DoubleSepalL = ir$Sepal.Length*2 #this option
allows you to name the field in the dataframe

ir = mutate(ir, DoubleSepalL = Sepal.Length*2,
PetalRatio = Petal.Length/Petal.Width) #you can add
several columns at a time
ir$DoubleSepalL = ir$Sepal.Length*2
ir$PetalRatio = Petal.Length/Petal.Width
```

- *summarize(dataframe, function(variable))*: creates summary statistics for a column in the data frame. There are many statistics functions you can call: *sd()*, *min()*, *max()*, *median()*, *sum()*, *cor()* (correlation), *n()* (length of vector), *first()* (first value), *last()* (last value) and *n_distinct()* (number of distinct values in vector).

```
summarise(ir, mean(Sepal.Length))
summarise(ir, n_distinct(Species))
summarise(ir, avg = mean(Sepal.Length), std=
sd(Sepal.Length), total=n()) #you can calculate several
functions at a time

summary(ir)
```

- *sample_n(data, n)*: samples *n* rows from a table

```
sample_n(iris,5) # Returns five random rows in iris
iris[sample(1:nrow(iris)),][1:5,]
# due to the random nature of this operation, your
results will most likely never match!
```

- *sample_frac(datafrane,fraction)*: samples fixed fraction from data

```
sample_frac(iris,0.01) # samples 1% of the data – 2
instances
iris[sample(1:nrow(iris)),][1:ceiling(nrow(iris)*0.01)
,]
```

- *group_by(dataframe, variable):* groups or splits data by one or more variables. On its own, *group_by* is not very useful: all it does is converting an existing

dataframe into a "grouped dataframe" where operations are performed by group.

```
group_by(ir, Species)# Not very useful – only a dataframe is returned.
group_by(ir, Species, Petal.Length)# You can group by two variables, too. But, again, nothing happens when group_by is used alone.
```

However, once *group_by()* is combined with *summarize()*, you can calculate very sophisticated statistics about your data quickly and clearly:

```
summarize(group_by(ir, Species), sd(Petal.Width))#Calculates standard deviation of Petal Width by Species
aggregate(ir$Petal.Width, by=list(ir$Species), FUN=sd) #Base R
```

```
summarize(group_by(ir, Species), r=cor(Sepal.Length, Sepal.Width))#Calculates the correlation between Sepal Length and Width by Species.
```

```
ir$LargeRatio = ir$PetalRatio>3 #Creates a new bolean variable to flag large petals.
summarize(group_by(ir, LargeRatio),n())#Calculates how many large and small petals there are

summarize(group_by(ir, LargeRatio,Species),n())#Calculates number of samples by Species and by PetalRatio
```

### *3.3. Piping with dplyr*

One of the most powerful characteristics of *dplyr* is that it lets you pipe the output of one function into another. That means that you take the result from one function and feed it to the first argument of the next function.

You may have encountered the Unix pipe | before. In R, we use *%>%* to pipe data, and we read it as "*then*".

You can use *%>%* with most R functions. For example:

```
      ir$Sepal.Length %>% mean #Read this as: "take iris' Sepal Length,
then calculate the mean". This is the same as writing:
      mean(ir$Sepal.Length)

      pi %>% trunc #or:
      trunc(pi)

      #It can also be used with other dplyr functions:
      group_by(ir, Species) %>% summarise(avg= mean(Petal.Length))
#or
      summarise(group_by(ir,Species), r=mean(Petal.Length))
```

However, applying it this way, it is not very useful. Piping becomes extremely helpful when you need to carry out sequential operations on the data previously obtained.

Let see a small example: imagine that you want to count how many non-virginica plants have a petal width over 3.5 in *iris*. Let's see how to solve this three ways:

```
#A. Using basic R – One of the many possible implementations
      non_virg = ir[ir$Species!="virginica", c("Petal.Length")]
#You need an auxiliary variable to store the non_virginica plants
first
      sum(non_virg>3.5)

#B. Using dplyr with no piping
      summarise(filter(ir,Species!="virginica",Petal.Length>3.5),P
etal.Length), n())

#C. Using dplyr with piping – follows a more intuitive process:
      ir %>% filter(Species!="virginica", Petal.Length>3.5) %>%
nrow()

      # Let's break it down:
      ir # start with ir dataframe
      %>% filter(Species!="virginica", Petal.Length>3.5) #then,
      return those rows such as their species is not virginica and
      their petal length is over 3.5
      %>% nrow() #then, count how many rows there are
```

And a more complex example: using *iris*, imagine that you want to find out the species of the three samples with the largest petal width to petal length ratio. You can do this using piping in a very straightforward way:

```
       ir %>% mutate(petal_w_l = Petal.Width/Petal.Length) %>% ar
range(desc(petal_w_l))%>% select(Species, petal_w_l) %>% head(3)

       #Breaking it down:
       ir %>% mutate(petal_w_l = Petal.Width/Petal.Length) #start
with ir, then, add a new field that calculates petal width to len
gth ratio
          %>% arrange(desc(petal_w_l)) #then, order them in desce
nding order
          %>% select(Species, petal_w_l) #then, keep the species
and the ratio attributes
          %>% head(3) #then show the 3 largest ones
```

Now, you are thinking with pipes.