Lab 5: Data Analysis

Instructions

Dr Mercedes Torres Torres

Contents

1 Introduction

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

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

2 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. To see a list of in-built data, type:

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

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

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

To use these datasets, I recommend 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.

3 Data Manipulation

For data manipulation, base R offers functions that allow you access different splits of data and perform operations on them, simultaneously. In Lab 2, we covered one of the most powerful functions for this: *apply()*. In this lab, we will introduce another one: *aggregate()*. But first, let's see how to use R to order data.

3.1 Basic Data Manipulation: Ordering Data

If you need to order data, there are two main functions that can help:

- *sort(x, decreasing =)*: sorts a vector into ascending or descending order.
- *order(x, decreasing =)*: returns the ordered indexes of the elements of a vector.

Let's see a very small example illustrating their differences:

```r
v = c(2, 9, 1, 45, -3, 19, -5, 6)

sort(v) # returns ordered v in decreasing order
```

```
## [1] -5 -3  1  2  6  9 19 45
```

```r
order(v) # returns order of the indexes in v
```

```
## [1] 7 5 3 1 8 2 6 4
```

```r
sort(v, decreasing = FALSE) # orders v in increasing order
```

```
## [1] -5 -3  1  2  6  9 19 45
```

```r
order(v, decreasing=FALSE)
```

```
## [1] 7 5 3 1 8 2 6 4
```

The *order()* function in particular is a powerful tool to index relevant instances in your datasets. Let's see a slighly more complex example, using *Iris*. *Iris* is a widely popular dataset and we will be using it in examples throughout this document. It has the measurements in centimeters of the sepal length and width and petal length and width for 150 flowers. It also has the species of the iris of each flower, with 50 flowers from each of the 3 species (*virginica, setosa, versicolor*).

```r
#Order and sort with iris
#Let's load iris
ir = iris

#1. What are the 5 plants with the largest Sepal lentgh?
or_ir=ir[order(ir$Sepal.Length, decreasing = TRUE),]
top5 = or_ir[1:5,]

#R is so powerful that it even lets you nest indexes:
top5a = ir[order(ir$Sepal.Length, decreasing = TRUE),][1:5,] #Pretty neat, right?

#Imagine that you just want to access Sepal Length and Species.
# You can access those values in different ways:
ir[order(ir$Sepal.Length, decreasing = TRUE),c("Sepal.Length", "Species")][1:5,]
```

```
##     Sepal.Length   Species
## 132          7.9 virginica
## 118          7.7 virginica
## 119          7.7 virginica
## 123          7.7 virginica
## 136          7.7 virginica
```

```r
ir[order(ir$Sepal.Length, decreasing = TRUE),][1:5, c("Sepal.Length", "Species")]
```

```
##     Sepal.Length   Species
## 132          7.9 virginica
## 118          7.7 virginica
## 119          7.7 virginica
## 123          7.7 virginica
## 136          7.7 virginica
```

In this example, we used the ordered indexes from sorting according to increasing Sepal Length (*order(ir$Sepal.Length, decreasing = TRUE)*) to index iris as a whole (*ir[order(ir$Sepal.Length, decreasing = TRUE),]*). Then, we accessed the first five values of that sorted data frame.

3.2 Advanced Data Manipulation: *Aggregate()*

The *aggregate()* function is able to split data in a data frame into subsets according to a list of values, and then perform the same operation to each subset. The prototype of the function is *aggregate(X, by, FUN, ..., simplify = TRUE)*, where:

- *X* is an R object (commonly a data frame)
- *by* is a list of the elements by which you will be grouping your data.
- *FUN* is the function that will be applied to each subset.
- *simplify* is a logical value that indicates if results should be simplified into a vector or a matrix.

Let's see an exmaple:

```
ir = iris
#Let's calculate the mean Sepal Length for each Species:
aggregate(ir$Sepal.Length, by= list(ir$Species), FUN=mean)
```

```
##      Group.1     x
## 1     setosa 5.006
## 2 versicolor 5.936
## 3  virginica 6.588
```

```
#Let's calculate the summary (min, max, mean, median, Q1 and Q3)
# of iris according to each Species:
aggregate(ir$Sepal.Width, by= list(ir$Species), summary)
```

```
##      Group.1 x.Min. x.1st Qu. x.Median x.Mean x.3rd Qu. x.Max.
## 1     setosa  2.300     3.200    3.400  3.428     3.675  4.400
## 2 versicolor  2.000     2.525    2.800  2.770     3.000  3.400
## 3  virginica  2.200     2.800    3.000  2.974     3.175  3.800
```

```
#We can also calculate the same operation across several variables
# at the same time by accessing them in the first argument
aggregate(ir[,c("Sepal.Length", "Sepal.Width")], by=list(ir$Species), mean)
```

```
##      Group.1 Sepal.Length Sepal.Width
## 1     setosa        5.006       3.428
## 2 versicolor        5.936       2.770
## 3  virginica        6.588       2.974
```

Aggregate is very powerful and extremely customizable. For example, it lets you split data according to several values.

```
#Let's create a new attribute in iris called _indoor_.
#_indoor_ is a boolean that records if a plant is an indoor or outdoor plant.
#For the purposes of this trask, let's give it random values with _sample()_
ir$indoor = sample(c(TRUE,FALSE), nrow(ir), TRUE)

#Now ir is ready
head(ir)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species indoor
## 1          5.1         3.5          1.4         0.2  setosa   TRUE
## 2          4.9         3.0          1.4         0.2  setosa   TRUE
## 3          4.7         3.2          1.3         0.2  setosa   TRUE
## 4          4.6         3.1          1.5         0.2  setosa  FALSE
## 5          5.0         3.6          1.4         0.2  setosa   TRUE
## 6          5.4         3.9          1.7         0.4  setosa   TRUE
```

```r
#Next, let's calculate the mean of each Sepal.Length
#according to Species also considering if they are indoor or outdoor plants
aggregate(ir$Sepal.Length, by=list(ir$Species, ir$indoor), mean)
```

```
##      Group.1 Group.2       x
## 1     setosa   FALSE 5.025000
## 2 versicolor   FALSE 5.992000
## 3  virginica   FALSE 6.504000
## 4     setosa    TRUE 4.988462
## 5 versicolor    TRUE 5.880000
## 6  virginica    TRUE 6.672000
```

```r
#You can save the results as a data frame, and customise the name of the columns
mean_by_sp_ind = aggregate(ir$Sepal.Length, by=list(ir$Species, ir$indoor), mean)
colnames(mean_by_sp_ind)= c("Species", "Indoor", "Average")
mean_by_sp_ind
```

```
##      Species Indoor  Average
## 1     setosa  FALSE 5.025000
## 2 versicolor  FALSE 5.992000
## 3  virginica  FALSE 6.504000
## 4     setosa   TRUE 4.988462
## 5 versicolor   TRUE 5.880000
## 6  virginica   TRUE 6.672000
```

```r
#Similarly to before, you can also calculate the operation for more than one variable:
aggregate(ir[,1:4], by=list(ir$Species), mean)
```

```
##      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

Alternatively, *aggregate()* has a different prototype that is allows you to use formulas to split the data. This prototype is *aggregate(formula, data, FUN, subset)*, where:

- *formula* details the in the manner $y \sim x$ or *cbind(y1, y2)* $\sim$ *x1+x2*,where y and *cbind(y1,y2)* are the numeric data to be split and x or *x1 + x2* are the grouping variables.
- *data* is the data frame with the variable used in the formula.
- *FUN* is the function that will be applied to each subset.
- *subset* is an optional vector specifying a subset of observations to be used

```r
#Let's reproduce the results from the previous two chunks of code using this prototype:
aggregate(Sepal.Length ~ Species, data = ir, mean)
```

```
##      Species Sepal.Length
## 1     setosa        5.006
## 2 versicolor        5.936
## 3  virginica        6.588
```

```r
aggregate(Sepal.Length ~ Species + indoor, data = ir, mean)
```

```
##      Species indoor Sepal.Length
## 1     setosa  FALSE     5.025000
## 2 versicolor  FALSE     5.992000
## 3  virginica  FALSE     6.504000
```

```
## 4     setosa   TRUE     4.988462
## 5 versicolor   TRUE     5.880000
## 6  virginica   TRUE     6.672000
```

You can perform the operation on several attributes as well:

```
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species, ir, mean)
```

```
##      Species Sepal.Length Sepal.Width
## 1     setosa        5.006       3.428
## 2 versicolor        5.936       2.770
## 3  virginica        6.588       2.974
```

The . (dot) operator allows you to select all of the variables in the formula

```
aggregate(. ~ Species + indoor, data = ir, mean)
```

```
##      Species indoor Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa  FALSE     5.025000    3.479167     1.470833   0.2291667
## 2 versicolor  FALSE     5.992000    2.776000     4.260000   1.3200000
## 3  virginica  FALSE     6.504000    2.936000     5.420000   2.0080000
## 4     setosa   TRUE     4.988462    3.380769     1.453846   0.2615385
## 5 versicolor   TRUE     5.880000    2.764000     4.260000   1.3320000
## 6  virginica   TRUE     6.672000    3.012000     5.684000   2.0440000
```

```
#You can use subset to make sure you only access part of the data frame
#This instruction aggregates according to Species and calculates the mean
#of Sepal.Length and Sepal.Width of those plants whose Petal.Width is >0.6
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species, data = ir,
          subset = Petal.Width>0.6, mean)
```

```
##      Species Sepal.Length Sepal.Width
## 1 versicolor        5.936       2.770
## 2  virginica        6.588       2.974
```

Finally, let's explore how we can combine our own functions with aggregate. If you are going to call your function more than once, you can define it separately, and then call it inside *aggregate()*. For example:

```
#Let's calculate the mean according to Species of only the largest X values
#for each Species
meanX = function(vec, n){mean(head(vec[order(-vec)], n))}

#Now, let's calculate the mean of the top-5 samples according to Sepal.Length
#aggregated by Species
aggregate(ir$Sepal.Length, by = list(ir$Species), FUN= function(x) meanX(x, 5))
```

```
##      Group.1    x
## 1     setosa 5.64
## 2 versicolor 6.82
## 3  virginica 7.74
```

```
#You can also calculate the mean of the top-10 samples according to
#Petal.Width and aagregate it according by Species and Indoor
aggregate(ir$Petal.Width, by = list(ir$Species, ir$indoor), FUN= function(x) meanX(x, 10))
```

```
##      Group.1 Group.2    x
## 1     setosa   FALSE 0.32
## 2 versicolor   FALSE 1.50
## 3  virginica   FALSE 2.28
```

```
## 4     setosa    TRUE 0.36
## 5 versicolor    TRUE 1.53
## 6  virginica    TRUE 2.32
```

As you can see, we defined *meanX()* once, but you can use it as many times as you need.

On the other hand, if you are only going to use your own function once, you may wish to define it inside the call to *aggregate()*. In this case, however, you will need to be careful about additional arguments for your function (such as n in our previous example). Let's see a few options. For example:

```
#Let's imagine that we only want the Top5 values always
# This will allow for hardcoding 5
aggregate(ir$Sepal.Length, by = list(ir$Species),
          FUN= function(x) mean(head(x[order(-x)], 5)))
```

```
##      Group.1    x
## 1     setosa 5.64
## 2 versicolor 6.82
## 3  virginica 7.74
```

```
#If you want to get the TopN, where N is a variable, then you can use external parameters
aggregate(ir$Sepal.Length, by = list(ir$Species),
          FUN= function(x, n=5) mean(head(x[order(-x)], n)))
```

```
##      Group.1    x
## 1     setosa 5.64
## 2 versicolor 6.82
## 3  virginica 7.74
```

```
aggregate(ir$Sepal.Length, n=5, by = list(ir$Species),
          FUN= function(x, n) mean(head(x[order(-x)], n)))
```

```
##      Group.1    x
## 1     setosa 5.64
## 2 versicolor 6.82
## 3  virginica 7.74
```

```
aggregate(ir$Sepal.Length, 5, by = list(ir$Species),
          FUN= function(x, n) mean(head(x[order(-x)], n)))
```

```
##      Group.1    x
## 1     setosa 5.64
## 2 versicolor 6.82
## 3  virginica 7.74
```

```
#You can can more have as many external parameters as you need.
# Let's create a function that calculates the mean within a range (n,m)
aggregate(ir$Sepal.Length, by = list(ir$Species),
          FUN= function(x, n=5, m=6) mean(x[x>n & x<m]))
```

```
##      Group.1        x
## 1     setosa 5.313636
## 2 versicolor 5.604348
## 3  virginica 5.766667
```

```
aggregate(ir$Sepal.Length, n=5, m=6, by = list(ir$Species),
          FUN= function(x, n, m) mean(x[x>n & x<m]))
```

```
##      Group.1        x
## 1     setosa 5.313636
```

```
## 2 versicolor 5.604348
## 3  virginica 5.766667
```

```
aggregate(ir$Sepal.Length, 5, 6, by = list(ir$Species),
          FUN= function(x, n, m) mean(x[x>n & x<m]))
```

```
##      Group.1        x
## 1     setosa 5.313636
## 2 versicolor 5.604348
## 3  virginica 5.766667
```

4 Basic Data Analysis: Base R

Data analysis will be your first step towards understanding your data and any problems it may present. Data analysis and visualisation are used together to diagnose any type of pre-processing that your data will need. Through analysis, visualisation and pre-processing, you will work towards creating a new version of your original dataset so that your data is is clean, concise, and does not have any redundancies or useless information. This will help ensure that your eventual predictions are as accurate as possible.

In Lecture 4, we defined three different types of statistical measurements: centrality measures, dispersion measures, and measures of the relationships between variables. Let's see how to implement them in R, but first, let's see how R can give us an overview of our dataset.

4.1 Exploring Data

The first thing you might want to do after loading a dataset is to take a look at it. R offers several functions that can help you take a peek into the data you will be using and assess the type of each attribute.

- *View(dataset)* will show the whole *dataset* in a new window.
- *tail(dataset,x)* will show the last x intances of *dataset* in the console.
- *head(dataset,x)* will show the first x instances of *dataset* in the console.
- *names(dataset)* will return the name of the attributes in the dataset.
- *str(dataset)* will return a summary of of the type of each attribute and the first few values.

Let's see an example:

```
#CO2: data from experiment on cold tolerance of grass species Echinochloa crus-galli.
co2 = CO2

#View(co2) #commented to avoid dumping long text in this document

#Without a second argument, head() and tail() will show the first/last few instances
head(co2)
```

```
##   Plant   Type  Treatment conc uptake
## 1   Qn1 Quebec nonchilled   95   16.0
## 2   Qn1 Quebec nonchilled  175   30.4
## 3   Qn1 Quebec nonchilled  250   34.8
## 4   Qn1 Quebec nonchilled  350   37.2
## 5   Qn1 Quebec nonchilled  500   35.3
## 6   Qn1 Quebec nonchilled  675   39.2
```

```
tail(co2)
```

```
##    Plant        Type Treatment conc uptake
## 79   Mc3 Mississippi   chilled  175   18.0
```

```
## 80   Mc3 Mississippi   chilled  250   17.9
## 81   Mc3 Mississippi   chilled  350   17.9
## 82   Mc3 Mississippi   chilled  500   17.9
## 83   Mc3 Mississippi   chilled  675   18.9
## 84   Mc3 Mississippi   chilled 1000   19.9
```

```
#With a second argument, they will show that many
head(co2,1)
```

```
##   Plant   Type  Treatment conc uptake
## 1   Qn1 Quebec nonchilled   95     16
```

```
tail(co2,3)
```

```
##    Plant        Type Treatment conc uptake
## 82   Mc3 Mississippi   chilled  500   17.9
## 83   Mc3 Mississippi   chilled  675   18.9
## 84   Mc3 Mississippi   chilled 1000   19.9
```

```
names(co2)
```

```
## [1] "Plant"     "Type"      "Treatment" "conc"      "uptake"
```

```
#str(co2)
```

4.2 Centrality Measures

Centrality measures give information about the central tendency of a distribution.

4.2.1 Mode

Is the element that occurs most frequently in the sample. The mode is not unique, a dataset might have more than one mode. If there are to modes, the dataset is called *bimodal*, while if there are more than two, it is often referred to as *multimodal*.

R does not have a built-in function to calculate the mode of a sample, so you will need to create yours if needed.

4.2.2 Mean

The arithmetic mean of a sample $x_1, x_2, \ldots, x_n$, usually denoted by $\bar{x}$, is the sum of the sampled values divided by the number of items in the sample, denoted by n

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

The *mean()* function in R will return the mean of the values passed. R also offers the functions *colMeans()* and *rowMeans()* to calculate the means attribute-wise or instance-wise, respectively, when needed.

4.2.3 Median

The median is the value that sits right in the middle of your data, if it was arranged in order. If the data elements $X = (x_1, x_2, \ldots, x_n)$ are sorted, the median is:

$$med(X) = \begin{cases} x_{(\frac{n+1}{2})} & \text{if } n \text{ is odd} \\ \frac{1}{2}(x_{\frac{n}{2}} + x_{(\frac{n}{2}+1)}) & \text{if } n \text{ is even} \end{cases} \quad (1)$$

To calculate the median, use the *median()* function in R.

4.3 Dispersion Measures

Dispersion measures give information about the spread of a distribution.

4.3.1 Standard Deviation

The standard deviation (SD or σ) measures variation of a set of values. Sets with low standard deviation have their data close to the mean, while sets with a high standard deviation has data points spread out over a wider range.

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Where $x_1, x_2, \ldots, x_n$ are the observed values of the sample items, $\bar{x}$ is the mean value of these observations, and n is the number of observations in the sample.

The built-in *sd()* function calculates the standard deviation in R.

4.3.2 Range

The range of an attribute X with the values: $X = (x_1, x_2, \ldots, x_n)$ will be the difference between its maximum value and its minimum value. Ranges are useful to compare attributes. Attributes with very different ranges (or distributions) will need to be normalised, so they all move to a similar space (which we will cover in Chapter 8). This will improve your analysis, comparison, and classfication of the data.

$$range(X) = max(X) - min(X)$$

To calculate the range, you can use the built-in *range()* function. However, note that it will return both the minimum and the maximum values of the data, not the actual difference.

4.3.3 Interquartile Range

The interquartile range is equal to the difference between 3th and 1st quartile. The IQR is extremely useful because it gives information about the spread in the middle 50% of the data.

$$IQR = Q3 - Q1$$

4.4 Relationship Measures

Relationship measures help us compare two or more attributes. As we will see in future chapters, this will be extremely useful to assess how much information each attribute is bringing into the dataset.

4.4.1 Pearson Correlation

The Pearson Correlation Coefficient measures the linear correlation between two continuous variables X and Y. It ranges between $[+1, -1]$, where $+1$ indicates total positive linear correlation, 0 indicates no linear correlation, and -1 indicates total negative linear correlation.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

where *cov* is the covariance, σ_X is the standard deviation of X, σ_Y is the standard deviation of Y.

High correlations between attributes, both positive or negative, can indicate that those attributes have redundant or repeated information. Therefore, if you are dealing with datasets with highly-correlated attributes, you may have to transform it and decorrerlate it before you can use it in machine learning.

5 Advanced Data Analysis: *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.

5.1 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")
```

```
## Warning: package 'dplyr' was built under R version 3.6.2
```

For more information and examples on all *dplyr* functions, see: https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

You may also wish to install *tidyverse*, a collection of R several packages designed for data science. *tidyverse* containts *ggplot2*, *dplyr*, *tidyr*, *readr*, *purrr*, *tibble*, *stringr*, and *forcats*. In this course, we will cover *ggplot2* and *dplyr*. For more information, visit their website: https://www.tidyverse.org/

```
install.packages("tidyverse")
```

5.2 Using *dplyr*

In this lab, we will introduce the most useful functions from *dplyr*. These are: *filter()*, *arrange()*, *select()*, *mutate()*, *summarize()*, *sample_n()*, *sample_frac()*, and *group_by()*.

It is important to remember that, while *dplyr* provides a simpler way of transforming and accessing data, it is crucial to know that you can use basic R to carry out the same processes.For this reason, in this section we will show how to use the most popular *dplyr* functions and their equivalent implementations in basic R.

Let's start by loading *Iris* into a variable again.

```
ir = iris
```

5.2.1 Filter

Used to find rows and cases in a data frame where certain conditions are true. The prototype is *filter(data frame, condition)*.

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

You can also combine *filter* with conditions:

```
# using dplyr
filter(iris, Species=="setosa" & Sepal.Length>5.5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.8         4.0          1.2         0.2  setosa
## 2          5.7         4.4          1.5         0.4  setosa
```

```
## 3          5.7         3.8          1.7         0.3  setosa
```

```
# using basic R
iris[iris$Species=="setosa" & iris$Sepal.Length>5.5,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 15          5.8         4.0          1.2         0.2  setosa
## 16          5.7         4.4          1.5         0.4  setosa
## 19          5.7         3.8          1.7         0.3  setosa
```

Note that *filter()* re-numbers all returned rows from 1, while indexing returns their original row number.

5.2.2 Arrange

Arranges (orders) rows in a data frame by variables or attributes. The prototype is *arrange(data frame, attributes)*.

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

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

```
arrange(iris, Sepal.Length,Sepal.Width)[1:5,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          4.3         3.0          1.1         0.1  setosa
## 2          4.4         2.9          1.4         0.2  setosa
## 3          4.4         3.0          1.3         0.2  setosa
## 4          4.4         3.2          1.3         0.2  setosa
## 5          4.5         2.3          1.3         0.3  setosa
```

```
#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
iris[order(iris$Sepal.Length, iris$Sepal.Width,
           decreasing = FALSE),][1:5,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 14          4.3         3.0          1.1         0.1  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 42          4.5         2.3          1.3         0.3  setosa
```

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

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          4.3         3.0          1.1         0.1  setosa
## 2          4.4         3.2          1.3         0.2  setosa
## 3          4.4         3.0          1.3         0.2  setosa
## 4          4.4         2.9          1.4         0.2  setosa
## 5          4.5         2.3          1.3         0.3  setosa
```

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

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 14          4.3         3.0          1.1         0.1  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 42          4.5         2.3          1.3         0.3  setosa
```

5.2.3 Select

The prototype is *select(data frame, var1,. . . ,varX)*. The *select()* function keeps the variables $var_1, ..., var_X$ you input as parameters from your data frame.

```
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"))
```

5.2.4 Mutate

The prototype is *mutate(data frame, expression)*. It adds new columns to a data frame so that they follow the function in *expression*.

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

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

5.2.5 Summarize

The prototype is *summarize(data frame, function(var1,. . . ,varX))*. *Summarize* creates a 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))
#you can calculate several functions at a time
summarise(ir, avg = mean(Sepal.Length),
          std= sd(Sepal.Length), total=n())
summary(ir)
```

5.2.6 Sample_n

The prototype is *sample_n(data frame, n)*. It samples *n* rows from a data frame or a any other data structure.

```
# Returns five random rows in iris.
# Due to the random nature of this, your results will most likely never match!
sample_n(iris,5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          5.7         2.8          4.1         1.3 versicolor
## 2          4.6         3.4          1.4         0.3     setosa
## 3          4.8         3.0          1.4         0.1     setosa
## 4          5.2         2.7          3.9         1.4 versicolor
## 5          6.8         3.0          5.5         2.1  virginica
```

```
iris[sample(1:nrow(iris)),][1:5,]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 135          6.1         2.6          5.6         1.4  virginica
## 94           5.0         2.3          3.3         1.0 versicolor
## 45           5.1         3.8          1.9         0.4     setosa
## 99           5.1         2.5          3.0         1.1 versicolor
## 3            4.7         3.2          1.3         0.2     setosa
```

5.2.7 Sample_frac

Samples fixed fraction from data frame. The prototype is *sample_frac(datafrane,fraction)*.

```
sample_frac(iris,0.01) # samples 1% of the data - 2 instances
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          4.6         3.6          1.0         0.2    setosa
## 2          7.4         2.8          6.1         1.9 virginica
```

```
iris[sample(1:nrow(iris)),][1:ceiling(nrow(iris)*0.01),]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 58          4.9         2.4          3.3         1.0 versicolor
## 56          5.7         2.8          4.5         1.3 versicolor
```

5.2.8 Group_by

The prototype is *group_by(data frame, variable)*. It groups or splits data by one or more variables. On its own, *group_by* is not very useful: all it does is convert an existing data frame into a "grouped data frame" where operations are performed by group.

```
# Not very useful - only a data frame is returned.
group_by(ir, Species)
```

```
## # A tibble: 150 x 8
## # Groups:   Species [3]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows, and 3 more variables: `Sepal.Length * 2` <dbl>,
## #   DoubleSepalL <dbl>, PetalRatio <dbl>
```

```
# You can group by two variables, too.
# But, again, nothing happens when group_by is used alone.
group_by(ir, Species, Petal.Length)
```

```
## # A tibble: 150 x 8
## # Groups:   Species, Petal.Length [48]
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows, and 3 more variables: `Sepal.Length * 2` <dbl>,
## #   DoubleSepalL <dbl>, PetalRatio <dbl>
```

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

```
#Calculates standard deviation of Petal Width by Species
summarize(group_by(ir, Species), sd(Petal.Width))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   Species    `sd(Petal.Width)`
##   <fct>                  <dbl>
## 1 setosa                 0.105
## 2 versicolor             0.198
## 3 virginica              0.275
```

```
aggregate(ir$Petal.Width, by=list(ir$Species), FUN=sd) #Base R
```

```
##      Group.1         x
## 1     setosa 0.1053856
## 2 versicolor 0.1977527
## 3  virginica 0.2746501
```

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

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   Species        r
##   <fct>      <dbl>
## 1 setosa     0.743
## 2 versicolor 0.526
## 3 virginica  0.457
```

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

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 2
##   LargeRatio `n()`
##   <lgl>      <int>
## 1 FALSE         48
## 2 TRUE         102
```

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

```
## `summarise()` regrouping output by 'LargeRatio' (override with `.groups` argument)
```

```
## # A tibble: 6 x 3
## # Groups:   LargeRatio [2]
##   LargeRatio Species    `n()`
##   <lgl>      <fct>      <int>
## 1 FALSE      setosa         1
## 2 FALSE      versicolor    13
## 3 FALSE      virginica     34
## 4 TRUE       setosa        49
## 5 TRUE       versicolor    37
## 6 TRUE       virginica     16
```

5.3 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:

```
#Read this as: take Sepal Length, then calculate the mean.
ir$Sepal.Length %>% mean
```

```
## [1] 5.843333
```

```
#This is the same as writing:
mean(ir$Sepal.Length)
```

```
## [1] 5.843333
```

```
pi %>% trunc
```

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

```
#or
trunc(pi)
```

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

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

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   Species      avg
##   <fct>      <dbl>
## 1 setosa      1.46
## 2 versicolor  4.26
## 3 virginica   5.55
```

```
#or
summarise(group_by(ir,Species), r=mean(Petal.Length))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   Species        r
##   <fct>      <dbl>
## 1 setosa      1.46
## 2 versicolor  4.26
## 3 virginica   5.55
```

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
# You need an auxiliary variable to store the non_virginica plants first
non_virg = ir[ir$Species!="virginica", c("Petal.Length")]
sum(non_virg>3.5)
```

```
## [1] 45
```

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

```
##   n()
## 1  45
```

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

```
## [1] 45
```

Let's break it down:

```
ir %>% # start with ir data frame then...
  filter(Species!="virginica", Petal.Length>3.5) %>%
```

```
  #... return rows of not virginica plants and petal length is over 3.5 then...
  nrow() # 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) %>%
  arrange(desc(petal_w_l)) %>%
  head(3) %>% select(Species, petal_w_l)
```

```
##     Species petal_w_l
## 1 virginica 0.4705882
## 2 virginica 0.4509804
## 3 virginica 0.4423077
```

If we break it down:

```
    ir %>% #start with ir, then...
  mutate(petal_w_l = Petal.Width/Petal.Length)%>%
  #add a new field that calculates petal width to length ratio then...
    arrange(desc(petal_w_l)) %>%  # order them in descending order, then...
    select(Species, petal_w_l) %>%  #keep the species and  ratio attbs then...
    head(3) #then show the 3 largest ones
```