# Machine Learning in Biological Sciences

**1st Edition**

Theodoros Diakonidis

2024-02-09

# Table of contents

# Preface

Machine Learning and Deep Learning revolution came as a surprise to science and following to our lives. A lot of new practical implementations e.g. in car industry, medicine or physics and much more, helped in releasing human mind from common tasks and provided solutions to a lot of problems, automatizing tedious tasks.

These Notes are written for the Lectures of the Postgraduate Course, **Bioinformatic tools and applications in -omics technologies**.

Its purpose is to introduce students of biological Sciences and not only, the new concepts of Machine Learning implemented in Biological Sciences using the computing Language R. Starting with very common algorithms and providing case studies these notes explain the different types of algorithms and the situations to apply them in common data science projects.

# 1 Introduction

In recent years, with the exponential growth of computing power, machine learning algorithms-techniques become a very useful, sometimes necessary part of natural sciences as they provide researchers, and not only, very useful accurate predictions of future observations which in some cases could be a matter of life-death, illness-no illness, cancer-no cancer. In other words they have become able to predict a distinct, nominal result (**supervised ML**). This type of prediction is called **classification** in machine learning language.

An alternative case, is using the same algorithm to make a prediction of a quantitative (arithmetic) value e.g the height of a newborn child as a result. In that case we are talking about **regression**.

To make any type of the above predictions, it is necessary to construct a model with known results, nominal or quantitative ones, in order to use it for future (unknown) cases.

Machine Learning could also provide algorithms which search for meaningful connection between observables, by grouping cases together, seeking for physical meaning (**unsupervised ML**). An example could be the grouping of patients in terms of a specific disease, e.g. different diabetes types, so as to treat them differently, according to the idiomas of the people of the group that they belong to.

In those notes, raw biological datasets from experiment would be analysed using various machine learning algorithms, ending up to useful conclusions. The idea behind those notes is to prepare the reader to analyse research datasets from scratch, till the final results of his/hers research, using R.

## 1.1 Structure of the notes

In order to provide the reader with a clearer perception of the field and to get acquainted with the structure of the notes, which follows the exact steps of the analysis, it would be better to categorize our data analysis to the following major categories:

Initial data analysis

- Handling different types of data
- Data cleaning and missing data
- Using graphics to further understand the dataset
- Feature scaling
- Application of supervised machine learning algorithms

1. Supervised machine learning I (Classification)
2. Supervised machine learning II (Regression)

- Dimension Reduction Techniques
- Feature selection
- Handling Imbalanced data
- Unsupervised machine learning algorithms

# 2 Handling different types of data

In general, the majority of the data types that would deal with, would be continuous data. Continuous data are data that can take any value. Simple typical examples of such data is **height**, **weight**, blood measurements like **glucose**, **cholesterol levels** and so on. Those types of data are handled in a common way as the next chapters would reveal to us. Here it is necessary to mention how to deal with discrete types of data in order to use them in machine learning algorithms. But first of all we need to know what are the discrete data and the subcategories that they consist of. District data in general are data that information can only take certain values. Such data could be for example, **age**, **gender**, **education level**, **pain level** and so on. We can divide them in subcategories and treat them in a different way, although this is not a strict rule to follow. It depends of the dataset that we have and the algorithm that we are using too.

## 2.1 Categorical variables

The wikipedia definition of categorical data is:

*A **categorical variable** (also called **qualitative variable**) is a variable that can take on one of a limited, and usually fixed, number of possible values, assigning each individual or other unit of observation to a particular group or nominal category on the basis of some qualitative property.*

Examples of such variables is the gender (mentioned above) or the blood type A, AB, B, O, name of the city the patient lives in, e.g Thessaloniki, Athens, New York, Paris etc. or the level of pain from 1 to 10 and so on. We can divide this group in to two distinctive groups. The ordinal and non-ordinal (nominal) categorical variables and show how to treat such cases using R.

### 2.1.1 Ordinal categorical variables

These ar categorical variables that have a clear ordering. E.g.

- Stage of cancer (stage I, II, III, IV)
- Pain level (mild, moderate, severe)
- Satisfaction level (very dissatisfied, dissatisfied, neutral, satisfied, very satisfied)

In such cases we usually use a method called **label encoding** or **ordinal encoding**. Let's make an assumption that we have a column in a dataframe that states the stage of the cancer of the patients. We can use a vector here in order to emulate the specific column with random choices of 4 cancer stages:

```
set.seed(123)
cstage <- sample(c('I','II','III','IV'), size=20, replace = TRUE)
str(cstage)
```

```
 chr [1:20] "III" "III" "III" "II" "III" "II" "II" "II" "III" "I" "IV" "II" ...
```

Typically it would be a character column. If it is a factor we still need to check the order of the elements. We can pick up the order and change them to numeric with one line of code:

```
cstage<-as.numeric(factor(cstage,order=TRUE, levels = c('I','II','III','IV')))
cstage
```

```
 [1] 3 3 3 2 3 2 2 2 3 1 4 2 2 1 2 3 4 1 3 3
```

The **factor()** function here, factored character vector with the order that **levels** parameter asked and then **as.numeric()** transformed the factored values to numbers starting from 1. This is a simple way to prepare your ordinal data for the machine learning algorithm. It is also typical to start labeling from 0 so:

```
cstage<-cstage-1
cstage
```

```
 [1] 2 2 2 1 2 1 1 1 2 0 3 1 1 0 1 2 3 0 2 2
```

and we accomplished the ordinal labeling. An alternative to the above base R way, is using the cRAN package CatEncoders. (For more info check the help of the package).

> **Tip**
>
> We can make use of this method (label encoding) for non ordinal data, (eg. a column in a dataset of cities an infection took place). If no ordering is applied, as is not needed in this case, R will automatically assign the order alphabetically. In this way there is a high probability that the model captures the relationship between cities in such an alphabetical order. This is not what we want actually.

### 2.1.2 Nominal categorical values

These are categorical variables that have no ordering. e.g:

- blood type: A, AB, B, O

- Cities: Thessaloniki, Athens, New York, Paris
- Gender: Female, Male.

The typical way non-ordinal categorical data are handled is using a method called **one-hot encoding.** The idea behind this method, is the following:

We create a new binary column for each category in the original data.

Here's a step-by-step explanation:

1. Identify all unique categories across the categorical variable (e.g genre has 2, female and male)
2. Create a binary column for each category.
3. For each entry, set the column that corresponds to the category to 1, and all other new columns to 0.

For demonstration purposes only we are going to use the dataset Birthweight which would be thoroughly explained in the Supervised Machine Learning-Regression chapter. This dataset includes a column called gender.

```
#install.packages('readxl')
library(readxl)
BirthWeight <- read_excel("BirthWeight.xlsx")
# Removing the unneeded for the example columns id, education, parity
BirthWeight<-BirthWeight[-c(1,6,7)]
```

The caret package will be used to apply the algorithm described above and most specifically its function **dummyvars()**.

```r
library(caret)


#Apply the dummy variable method to all the columns seen appropriate
dummy <- dummyVars(" ~ .", data = BirthWeight)


#Create a new dataset replacing these
newBirthWeight <- data.frame(predict(dummy, newdata = BirthWeight))


head(newBirthWeight)
```

```
  weight height headc genderFemale genderMale
1   3.95   55.5  37.5            1          0
2   4.63   57.0  38.5            1          0
3   4.75   56.0  38.5            0          1
4   3.92   56.0  39.0            0          1
5   4.56   55.0  39.5            0          1
6   3.64   51.5  34.5            1          0
```

So what changed in the newBirthweight dataset, are two new columns, as the number of distinct cases (Female and Male) in the place of the initial gender column. Each time Female appear in the initial column it inserts 1 in the genderFemale column and 0 in the genderMale column and vice versa if Male appears. So two number columns are created in the place of the initial 'gender' column and now we can input the dataset safely to the machine learning algorithm functions. The whole idea could be easily extended in the same way for more than two distinct cases.

These are the commonest cases on how to deal with categorical data. There are many more which are outside of the scope of the specific notes.

# 3 Missing values and imputation

When dealing with real data, directly from research, it is unavoidable not to come across with missing values. The subject is too extensive to cover it in a paragraph of these notes. There are a lot of packages in R that help imputing numbers to your datasets, some of them are MICE, Amelia, missForest, Hmisc, mi and others. A small demonstration of the **MICE** package, the most powerful in terms of imputing missing numbers on datasets in our opinion so far, would be introduced. **Caret** package also includes a couple of methods too, as we are going to use it extensively in the next chapters we will provide with some examples from this package too. Let's start with the dataset that we are going to use. It's name is BirthWeight.

## 3.1 The Birthweight dataset

It includes data of 550 infants of 1 month age []. The recorded variables are the following:

- Body weight of the infant in gr (weight)
- Body height of the infant in cm (height),
- Gender of the infant (gender: Female, Male)
- Birth order in their family (parity: Singleton, One sibling, 2 or more siblings)
- Education of the mother (education: year10, year12, tertiary)

It has no missing values.

```r
library(readxl)

BirthWeight <- read_excel("BirthWeight.xlsx")

summary(BirthWeight)
```

```
      id                weight          height          headc
 Length:550         Min.   :2.920   Min.   :48.00   Min.   :34.0
 Class :character   1st Qu.:3.950   1st Qu.:53.00   1st Qu.:37.0
 Mode  :character   Median :4.330   Median :55.00   Median :38.0
                    Mean   :4.366   Mean   :54.84   Mean   :37.9
                    3rd Qu.:4.770   3rd Qu.:56.50   3rd Qu.:39.0
                    Max.   :6.490   Max.   :62.00   Max.   :41.2
    gender            education            parity
 Length:550         Length:550         Length:550
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character
```

## 3.2 The MICE package

The package is an implementation of the MICE method. Initials MICE stands for **Multiple Imputation by Chained Equations (Azur et al. 2011). It consists of more than 20 different imputation methods (see also the help page of the package) for all types of data. E.g:

- logreg (Logistic Regression) - for binary (2 level) categorical variables

- polyreg (Bayesian polytomous regression) - for categorical variables with more than or equal to two levels
- polr (Proportional odds model) - for **ordered** categorical variables with more than or equal to two levels
- PMM (Predictive Mean Matching) - for any type of variables

  It covers all types of data. Starting from BirthWeight dataset, let's produce some missing values. We are going to use missForest package to randomly delete 10% of the values of the original dataset:

```
#install.packages('missForest')
library(missForest)


BirthWeight.mis <- prodNA(BirthWeight, noNA = 0.1)
```

Applying factor function for gender, education and parity and deleting the index column, we are ready to use MICE.

```
cols <- c("gender", "education", "parity")

# Applying factor function to all chr columns of the dataset
BirthWeight.mis[cols] <- lapply(BirthWeight.mis[cols], factor)

#Deleting the first 'id' column
BirthWeight.mis<-BirthWeight.mis[-1]

summary(BirthWeight.mis)
```

```
     weight          height          headc          gender       education
 Min.   :2.920   Min.   :48.00   Min.   :34.00   Female:248   tertiary:229
```

```
 1st Qu.:3.955   1st Qu.:53.00   1st Qu.:37.00   Male  :239   year10  :168
 Median :4.340   Median :55.00   Median :38.00   NA's  : 63   year12  : 82
 Mean   :4.365   Mean   :54.93   Mean   :37.87                NA's    : 71
 3rd Qu.:4.770   3rd Qu.:56.50   3rd Qu.:39.00
 Max.   :6.490   Max.   :62.00   Max.   :41.20
 NA's   :55      NA's   :48      NA's   :47
                parity
 2 or more siblings:161
 One sibling       :175
 Singleton         :165
 NA's              : 49
```

Applying MICE e.g. using predictive mean matching (pmm) method suitable for any kind of data:

```
#install.packages("mice")
library(mice)

imputed_Data <- mice(BirthWeight.mis, m=1, maxit = 1, method = 'pmm', seed = 500)
```

```
 iter imp variable
  1   1  weight  height  headc  gender  education  parity
```

Some of the MICE package imputation methods

| | | |
|---|---|---|
| `pmm` | any | Predictive mean matching |
| `midastouch` | any | Weighted predictive mean matching |
| `sample` | any | Random sample from observed values |
| `cart` | any | Classification and regression trees |
| `rf` | any | Random forest imputations |
| `mean` | numeric | Unconditional mean imputation |
| `norm.nob` | numeric | Linear regression ignoring model error |
| `norm.boot` | numeric | Linear regression using bootstrap |
| `norm.predict` | numeric | Linear regression, predicted values |
| `quadratic` | numeric | Imputation of quadratic terms |
| `ri` | numeric | Random indicator for nonignorable data |
| `logreg` | binary | Logistic regression |
| `logreg.boot` | binary | Logistic regression with bootstrap |
| `lasso.logreg` | binary | Lasso logistic regression |
| `polr` | ordered | Proportional odds model |
| `polyreg` | unordered | Polytomous logistic regression |
| `lda` | unordered | Linear discriminant analysis |

We succeed in imputing data to the dataframe. Our new dataframe is:

```
imputed_BirthWeight <- complete(imputed_Data,1)


summary(imputed_BirthWeight)
```

```
     weight          height          headc          gender       education
 Min.   :2.920   Min.   :48.00   Min.   :34.00   Female:283   tertiary:264
 1st Qu.:3.960   1st Qu.:53.00   1st Qu.:37.00   Male  :267   year10  :190
```

```
 Median :4.340   Median :55.00   Median :38.00                year12  : 96
 Mean   :4.373   Mean   :54.86   Mean   :37.88
 3rd Qu.:4.780   3rd Qu.:56.50   3rd Qu.:39.00
 Max.   :6.490   Max.   :62.00   Max.   :41.20
                parity
 2 or more siblings:174
 One sibling       :193
 Singleton         :183
```

In the case of inputing m>1, MICE produces more than one imputed datasets and the user could choose a specific dataset by changing the index number of the second argument of the complete function above.

## 3.3 The function preProcess() in caret package

An alternative would be to use caret package and the **preProcess()** function but only for numerical values. There are 3 different algorithms available, knnImput, bagImput nad medianImput. The first two methods are using machine learning algorithms to predict the missing values, using the data of the specific column. we are going to discuss about them in the supervised machine learning chapter, the third one is imputing the median of the column. Let's try one out.

```
library(caret)


# We take only the first 3 numeric columns, weight, height, headc
BirthWeight.mis_caret<-BirthWeight.mis[,1:3]
```

```r
#use the method bagImpute
imputed_Data_caret <- preProcess(BirthWeight.mis_caret, method = "bagImpute")

#the resulted new dataset
imputed_BirthWeight_caret <- predict(imputed_Data_caret,BirthWeight.mis_caret)

summary(imputed_BirthWeight_caret)
```

```
     weight          height          headc
 Min.   :2.920   Min.   :48.00   Min.   :34.00
 1st Qu.:3.982   1st Qu.:53.00   1st Qu.:37.00
 Median :4.340   Median :55.00   Median :38.00
 Mean   :4.372   Mean   :54.91   Mean   :37.89
 3rd Qu.:4.770   3rd Qu.:56.50   3rd Qu.:39.00
 Max.   :6.490   Max.   :62.00   Max.   :41.20
```

We succeeded in filling the missing values. No NA's are reported.

# 4 Data visualization

Coming across a new dataset for inspection and preparation in order to apply a machine learning algorithm, it is always tempting to start directly, dividing the dataset to a train set and a test set applying different types of algorithms in order to find the best model. This practice though should always follow a clean dataset, transformed maybe as we will mention later on, so as the data to expose the structure of the problem in the best way to the machine learning algorithms. The fastest and most useful way to understand your data before proceeding to anything else is data visualization. But what is data visualization?

It is the creation of histograms, scatterplots and other types of plots that we are going to discuss here in order to better understand the dataset e.g:

- histograms of the response answers in order to check an imbalance of the responses and visualization of predictor variables,
- boxplots and violin plots to help discover the distribution or spread of attributes to spot outliers and give you an idea of possible data transformations you could apply,
- scatter plots between the explanatory variables and the dependent to find any type of relation among them,
- correlation plots among the explanatory variables (predictors) to check for possible pair-wise connection,

  and many others too.

  So let's start with an example dataset.

## 4.1 Pima Indians Dataset

The Pima Indians Diabetes Dataset (Larabi-Marie-Sainte et al. 2019), originally came from the National Institute of Diabetes and Digestive and Kidney Diseases, contains information of 768 women from a population near Phoenix, Arizona, USA. The outcome tested was Diabetes, 258 tested positive and 500 tested negative. Therefore, there is one target (dependent) variable and the 8 attributes (TYNECKI, 2018):

1. pregnancies
2. OGTT (Oral Glucose Tolerance Test)
3. blood pressure
4. skin thickness
5. insulin
6. BMI(Body Mass Index)
7. age
8. pedigree diabetes function

The Pima population has been under study by the National Institute of Diabetes and Digestive and Kidney Diseases at intervals of 2 years since 1965. The Pima Indians Diabetes Dataset includes information about attributes that could and should be related to the onset of diabetes and its future complications.

The dataset is part of the mlbench library of CRAN:

```
#install.packages('mlbench')
library(mlbench)
data(PimaIndiansDiabetes)
```

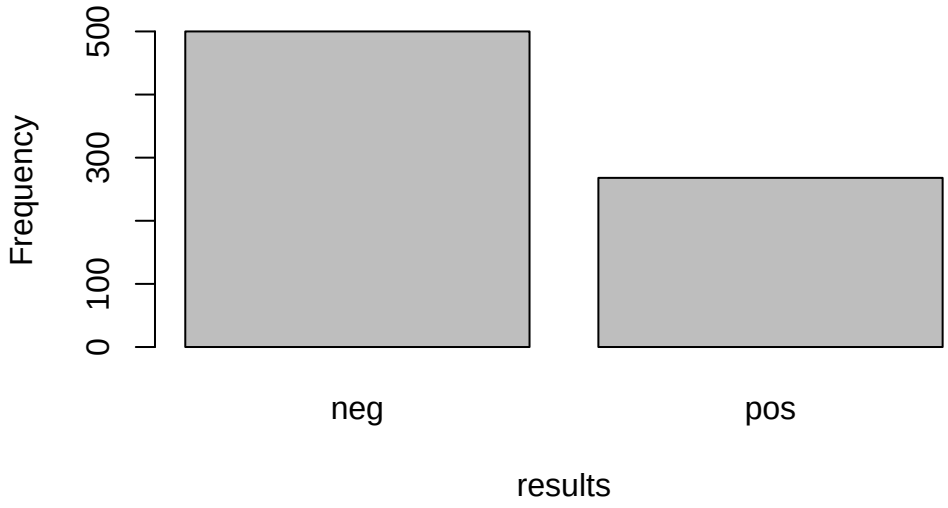
```r
pm <- PimaIndiansDiabetes
str(pm)
```

```
'data.frame':   768 obs. of  9 variables:
 $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
 $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
 $ pressure: num  72 66 64 66 40 74 50 0 70 96 ...
 $ triceps : num  35 29 0 23 35 0 32 0 45 0 ...
 $ insulin : num  0 0 0 94 168 0 88 0 543 0 ...
 $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 0 ...
 $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
 $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
 $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

## 4.2 Visualization of the response answers

A good way to start is by visualizing the responses. It is the last column of the dataset, called diabetes. The simplest way would be using base R and the function barplot():

```r
barplot(table(PimaIndiansDiabetes$diabetes),
        ylab = "Frequency",
        xlab = "results")
```

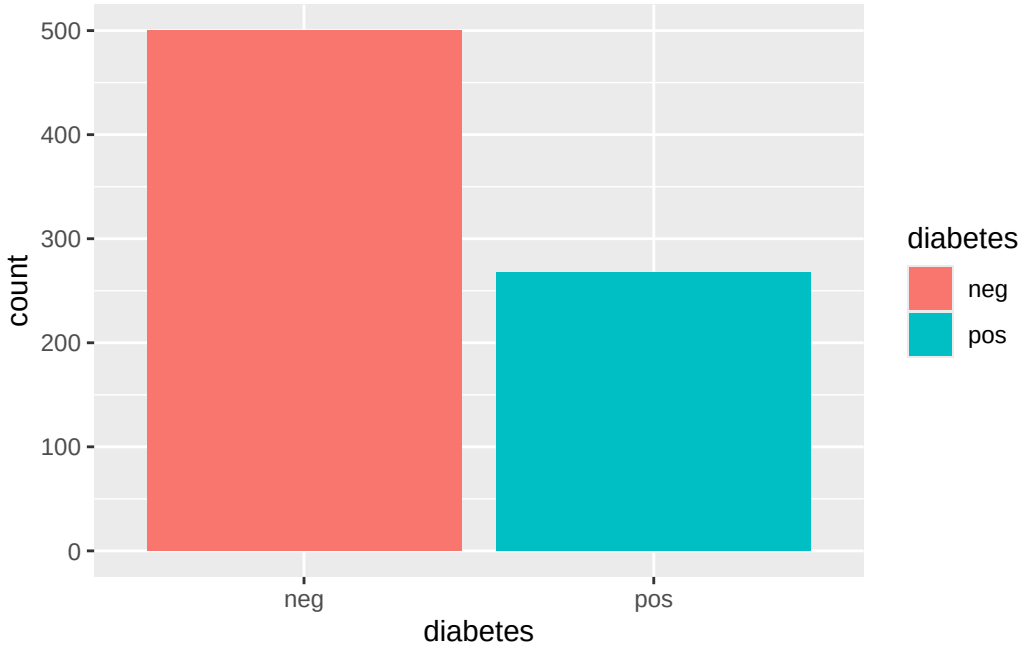



The table() function here, calculates and groups the response answers and the output is used by barplot() to draw the graph.

> **Tip**
>
> It is obvious that there is an imbalance in answers. There are different methods to accomplish a balance in the dataset by downsampling or upsampling or even by using synthetic data, we will mention such techniques in a next chapter.

Now let's make a more beautiful histogram using ggplot library. It is highly recommended to use this instead for presenting results in publications, presentations etc:

```
#install.packages(tidyverse)
library(tidyverse)
pm %>%
  ggplot(aes(x = diabetes, fill = diabetes)) +
  geom_bar()
```




ggplot() is the library used here, for reasons of completeness and in order to use the pipe operator %>% the package tidyverse, superset of ggplot2 is used here. For more info about tidyverse and its use one could check (Wickham et al. 2019).

## 4.3 Visualizations of the dataset

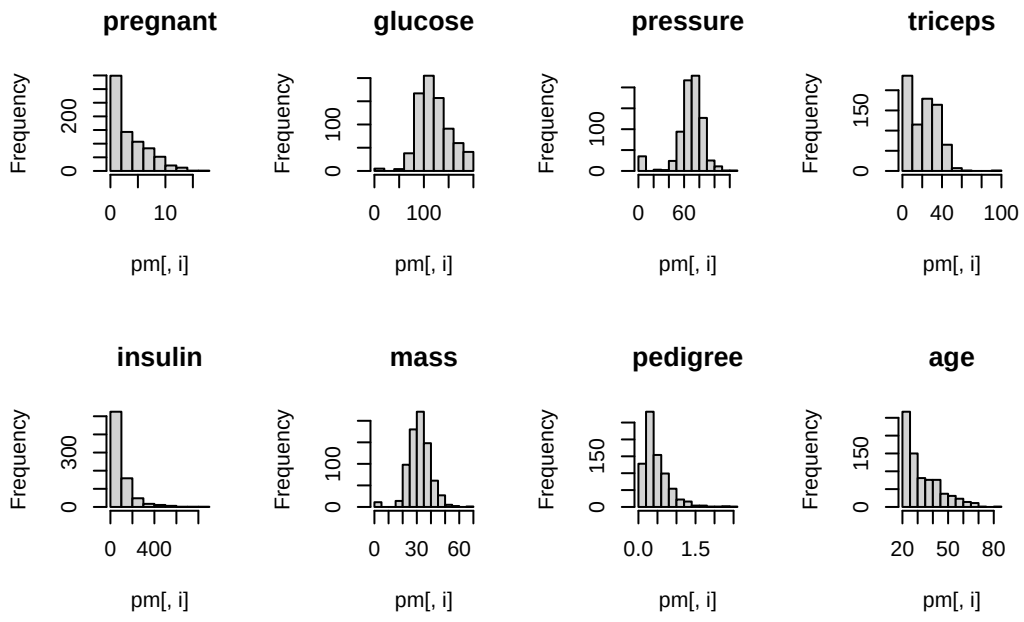
Visualizations are quite important to get a first grip of the dataset. We can spot a lot of abnormalities that would need to be taken care of, before we continue e.g wrong or useless data types, missing values and on the same time the visualizations would give us valuable info about the variables of the dataset and its connections.

### 4.3.1 Visualizing univariate variables

#### 4.3.1.1 Histograms

They are quite valuable in order to understand the 'structure' of our variables. The distribution, central tendency and spread of each of our predictors. We can still use base R (for a quick 'dirty' way to check our data):

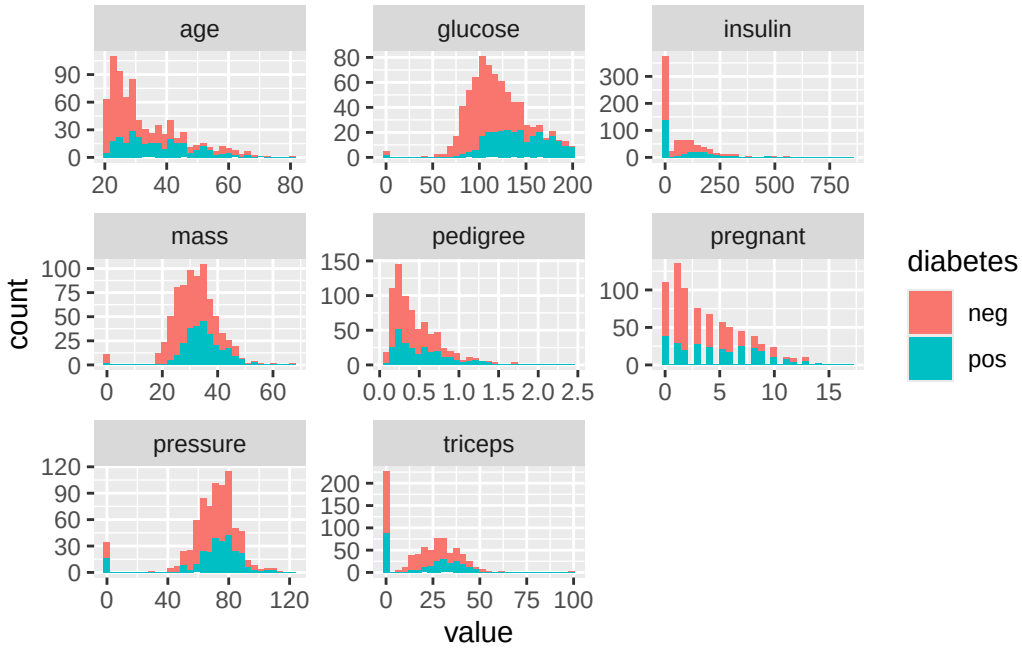
```
# mfrow divides the results in 2 rows and 4 columns
par(mfrow=c(2,4))
# loop the 8 predictors and plot the histograms
for(i in 1:8) {
 hist(pm[,i], main=names(pm)[i])
}
```




At first glance we can say that Pregnant, tricepts, insulin, pedigree and age are skewed while pressure glucose and mass are relatively following a gaussian distribution.

Let's try the ggplot way now for our presentations:

```
pm %>%
  gather("key","value",pregnant:age) %>%
  ggplot(aes(x = value, fill=diabetes)) +
    facet_wrap(vars(key), scales = "free") +
     geom_histogram()
```




To make it more interesting we added the fill=diabetes, parameter in order to connect our predictors with the final response result (positive or negative diabetes).

#### 4.3.1.2 Boxplots

Boxplots are quite useful graphical images in order to fully understand the structure of your variable (distribution, spread, median, outliers). The following image gives a summary of the information that a box plot carries:

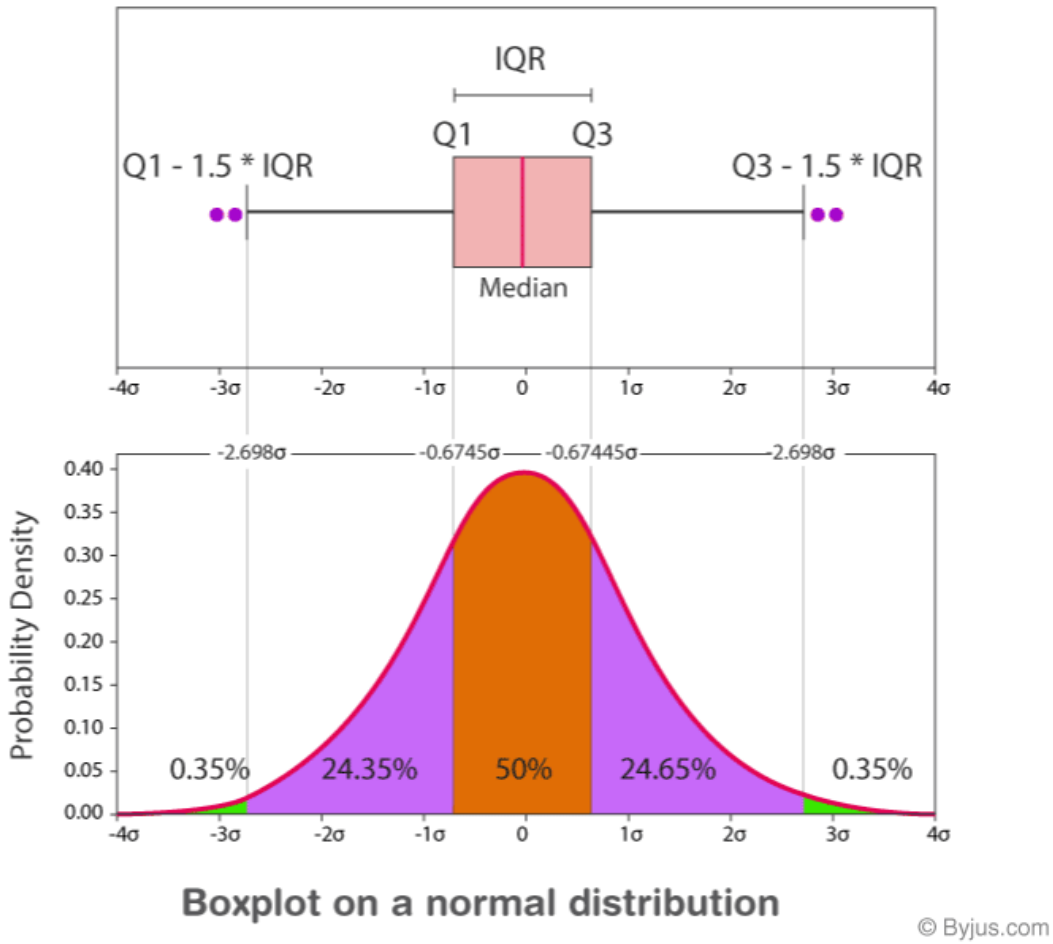
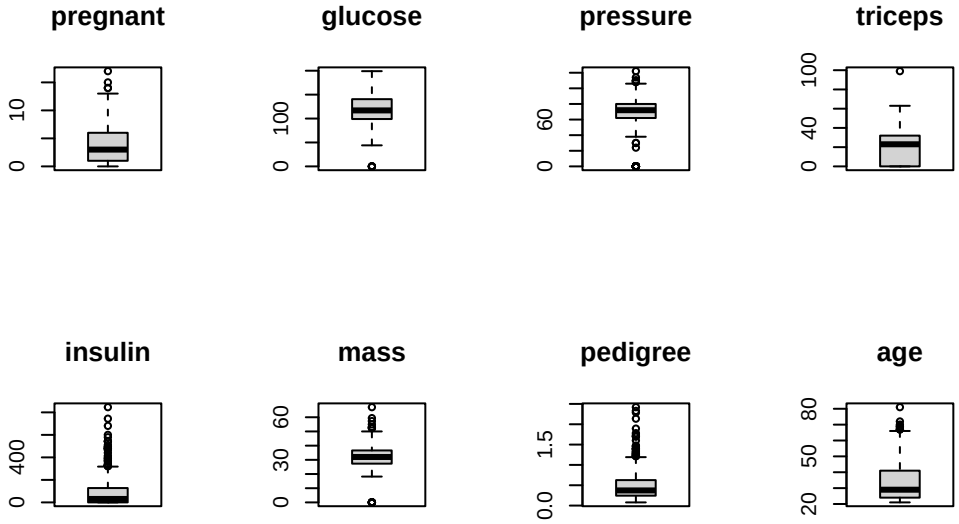



Figure 4.1: A summary of boxplot info on one to one correspondence with a gaussian (The purple dots are the outliers)

R Code for boxplots of all explanatory variables are very similar with that of the histogram:

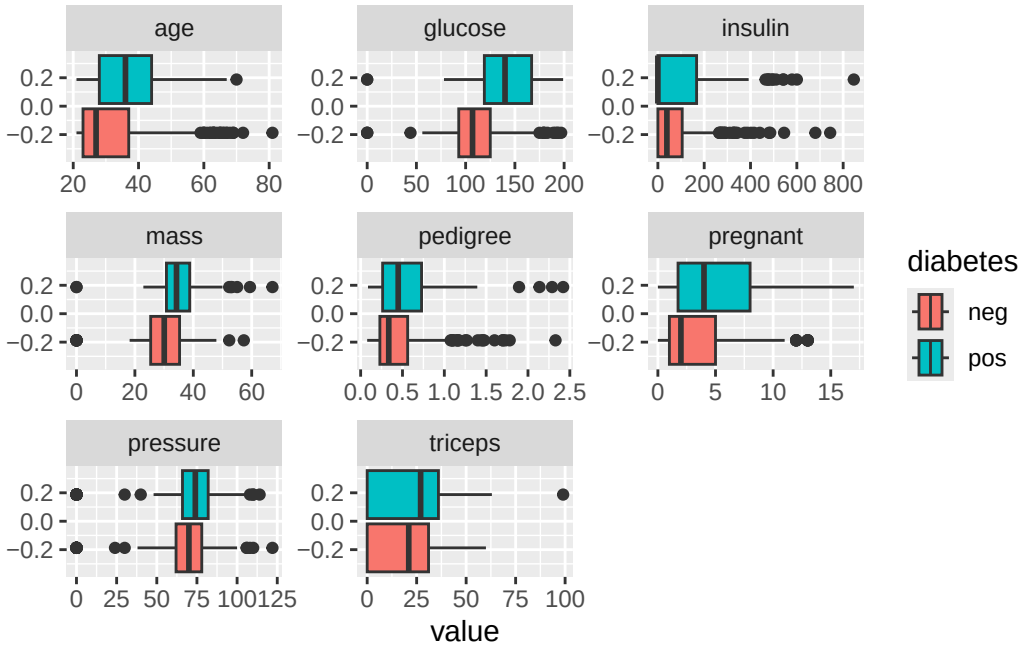
```
# mfrow divides the results in 2 rows and 4 columns
par(mfrow=c(2,4))
# loop the 8 predictors and plot the histograms
for(i in 1:8) {
 boxplot(pm[,i], main=names(pm)[i])
```

```
}
```




only difference is the boxplot() function. To make things more interesting we do the same trick as before in ggplot function, by dividing data according to the diabetes responses:

```
library(tidyverse)
pm %>%
  gather("key","value",pregnant:age) %>%
  ggplot(aes(x = value, fill = diabetes)) +
    facet_wrap(vars(key), scales ="free") +
    geom_boxplot()
```




We create two extra columns in the **pm** dataframe using gather() function (key and value) which include all the info from the columns from pregnant to age, and apply ggplot() to those in order to have all diagrams for the 2 diabetes classes.

#### 4.3.1.3 Violin plots

A violin plot is somehow an extension of the boxplot. It comes with an extra kernel density estimation on variable data distribution, to get a sense of the way your data is distributed.

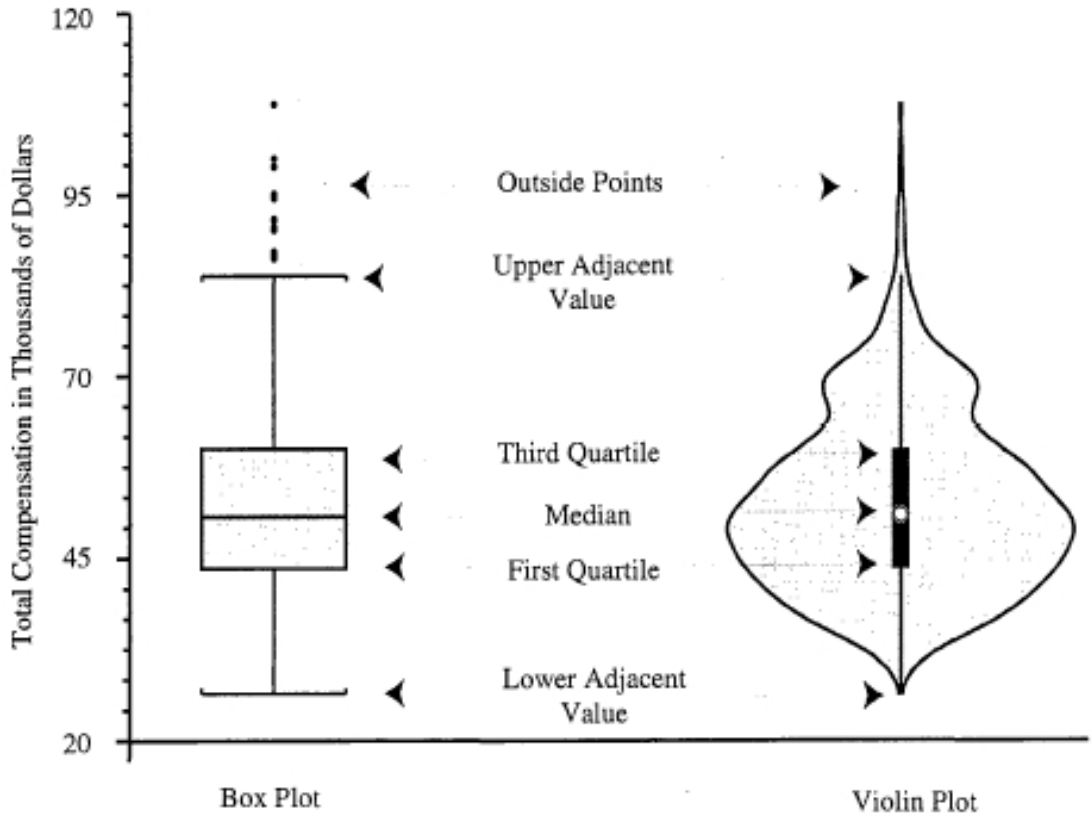
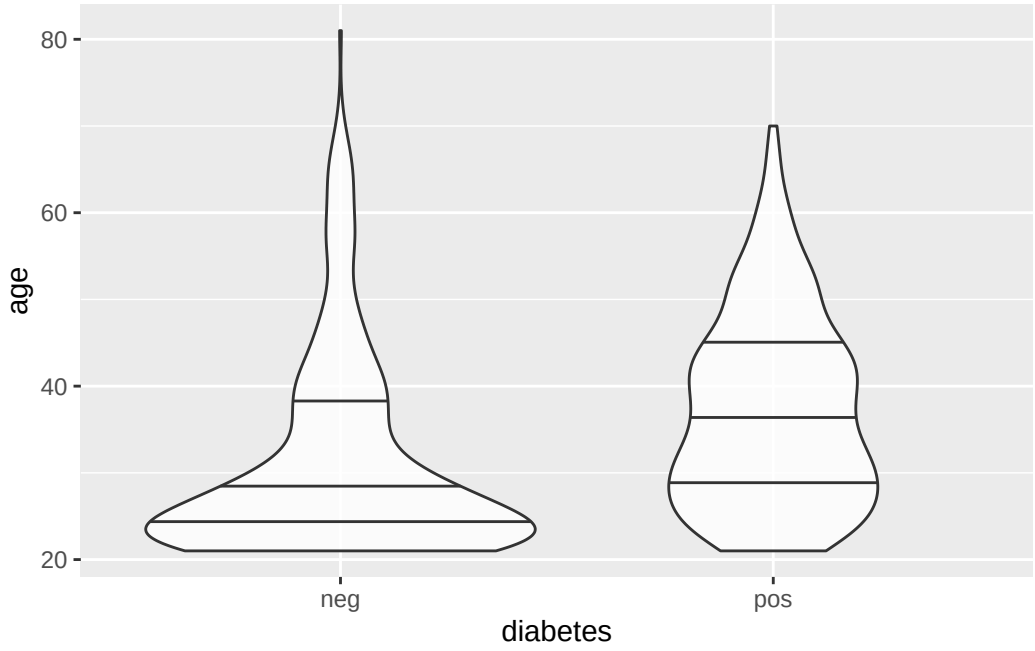



Figure 4.2: Violin plot vs box plot

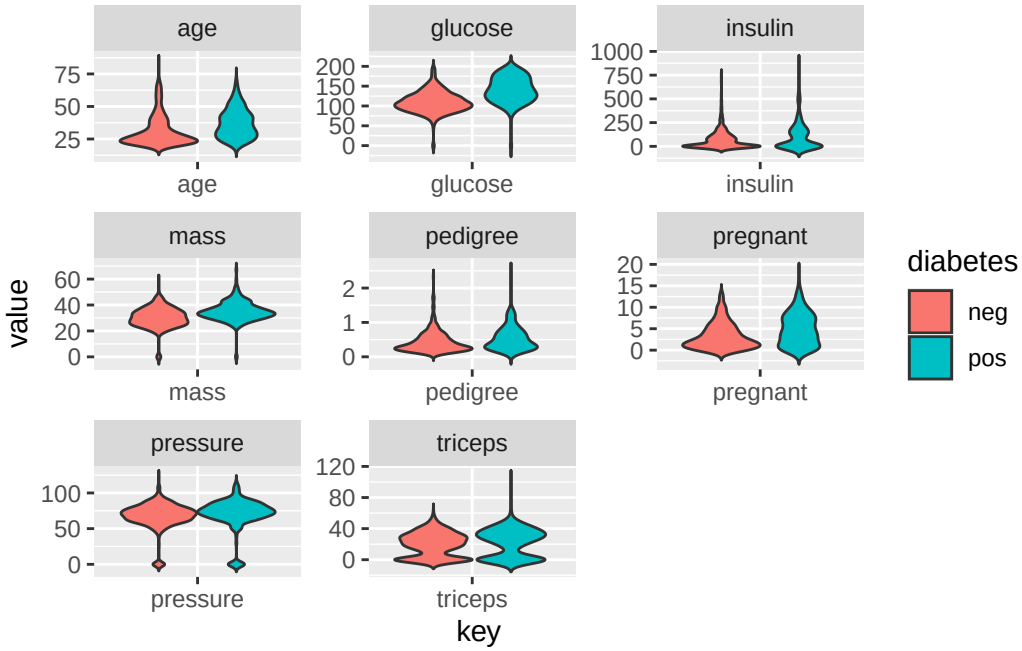
To plot a single violin plot the ggplot package is necessary:

```
pm %>%
  ggplot(aes(x= age, y = diabetes)) +
  geom_violin(alpha = 0.8,
# inserting quartile lines 25%, 50% and 75%
              draw_quantiles = c(0.25, 0.5, 0.75))+
# flipping the coordinate system
  coord_flip()
```




To plot all of them we do the trick that already applied for the boxes case:

```
pm %>%
  gather("key","value",pregnant:age) %>%
  ggplot(aes(x= key, y = value, fill = diabetes)) +
  facet_wrap(vars(key), scales = "free")+
  #trim=FALSE does not trim the tails of the plots
  geom_violin(trim=FALSE)
```




Getting more info about data points, distribution provides us with a better understanding of the dataset.

### 4.3.2 Visualizing the connections between variables

#### 4.3.2.1 Correlation plots

Another interesting point, quite useful, is to check correlation between variables. It will provide useful information about the connection of the predictors to each other and with the dependent variable (in case they are numeric).

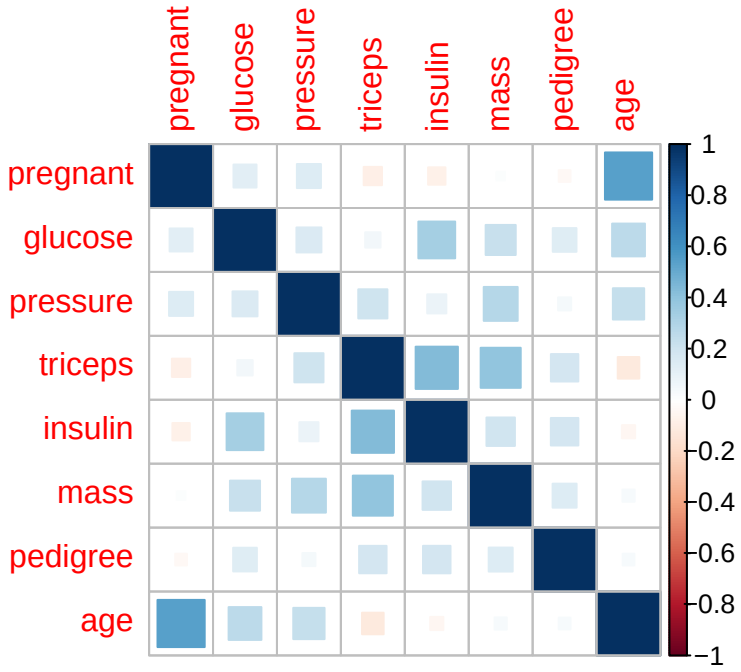
- Some predictors could have a strong correlation with each other. This could be a reason of discarding one of them. As they contribute the same information multiple times could be a reason for overfitting. The phenomenon is called **multicollinearity.**
- Having a look at the correlation could discover a strong relationship of a predictor with the result (dependent variable). This is a hind that the specific predictor has a strong

impact on predicting the result.

> **Tip**
>
> This info would be quite useful in applying regression models as we will see in the regression chapter afterwards

We can use the corrplot library for checking correlations among predictors:

```
#install.packages('corrplot')
library(corrplot)
# load the data
# calculate correlations between all predictors
correlations <- cor(pm[,1:8])
# create correlation plot
corrplot(correlations, method="square")
```

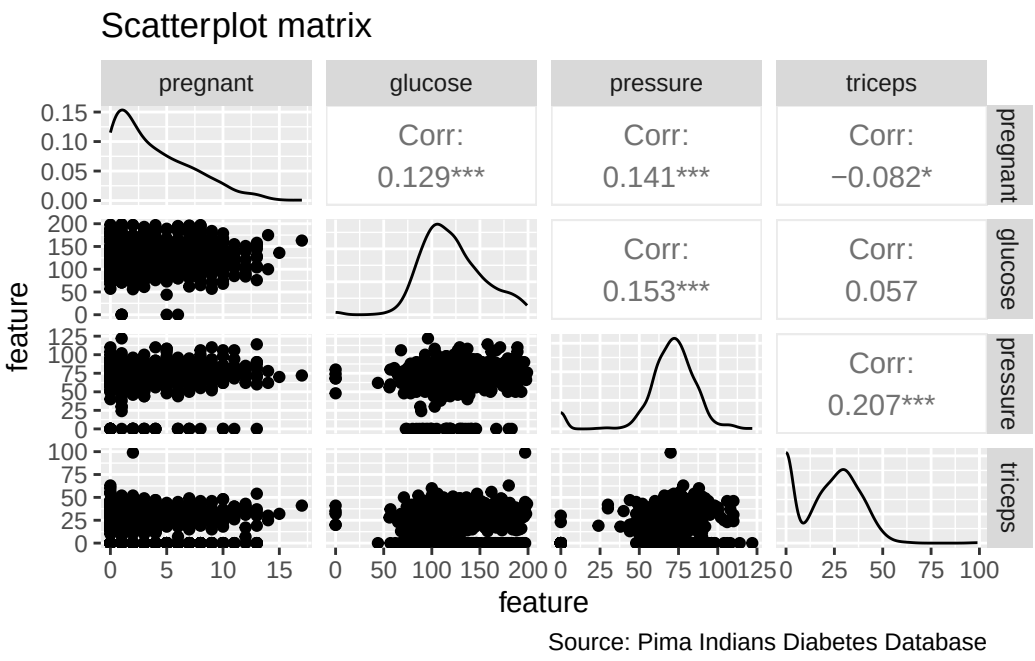



An alternative would be ggcorr() function of the tidyverse package.

#### 4.3.2.2 Scatter plots

Scatterplots could give us a more detailed information about the distributions of data inside dataset and provide us also with trends. A great tool to draw scatterplots fast and with beautiful explanatory results is the GGally package. It provides a scatterplot matrix using ggpairs() function (an extension of pairs() base R function):

```
library(GGally)
# scatterplot matrix
ggpairs(pm,
# Taking first 4 out of 8 predictors
        columns = c(1:4)) +
  labs(x = "feature",
       y = "feature",
       title = "Scatterplot matrix",
       caption = "Source: Pima Indians Diabetes Database")
```



Scatterplots of each pair visualized in left side of the plot and Pearson correlation value and significance displayed on the right side. In the diagonal in the middle a density plot of each variable. A density plot is quite similar of a histogram. It actually shows proportions instead of counts (histogram). You can draw density plots using ggplot and the geom.density() function.

### 4.3.3 Visualizing missing values

Finally, an important thing to know is missing values. If there are any and how they are distributed. A nice package for giving all that information in hand is naniar. Let's try it. We are going to use the Birthweight.mis dataset.

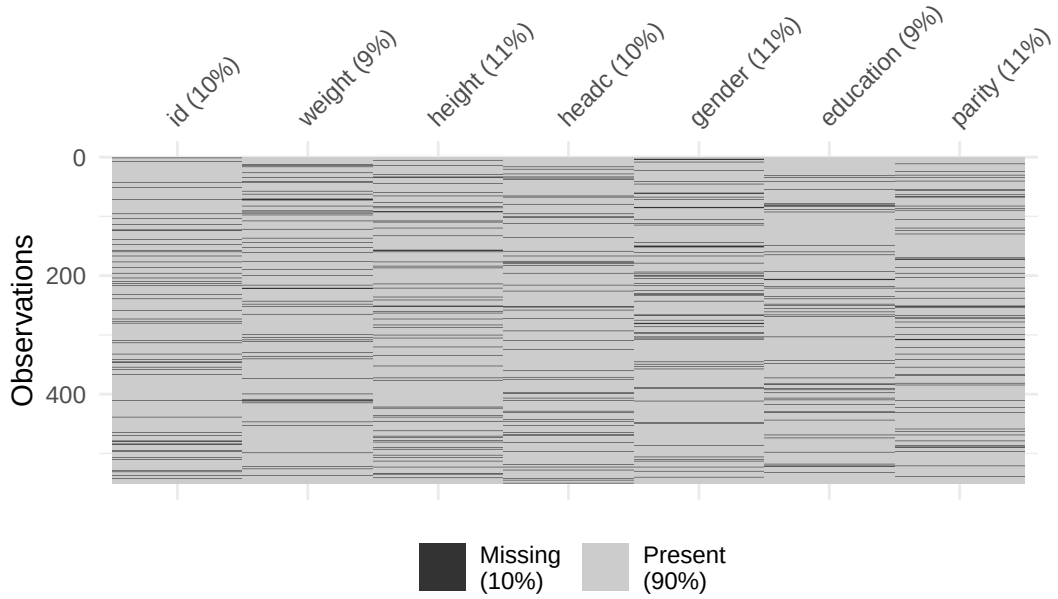
```
library(readxl)
BirthWeight <- read_excel("BirthWeight.xlsx")

library(missForest)
BirthWeight.mis <- prodNA(BirthWeight, noNA = 0.1)

#install.packages("naniar")
library(naniar)
summary(BirthWeight.mis)
```

```
      id                weight          height          headc
 Length:550         Min.   :2.920   Min.   :48.00   Min.   :34.00
 Class :character   1st Qu.:3.950   1st Qu.:53.00   1st Qu.:37.00
 Mode  :character   Median :4.330   Median :55.00   Median :38.00
                    Mean   :4.364   Mean   :54.91   Mean   :37.86
                    3rd Qu.:4.760   3rd Qu.:56.50   3rd Qu.:39.00
                    Max.   :6.490   Max.   :62.00   Max.   :41.00
```

```
                  NA's   :52      NA's   :60      NA's   :53
    gender           education           parity
 Length:550         Length:550         Length:550
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character
```

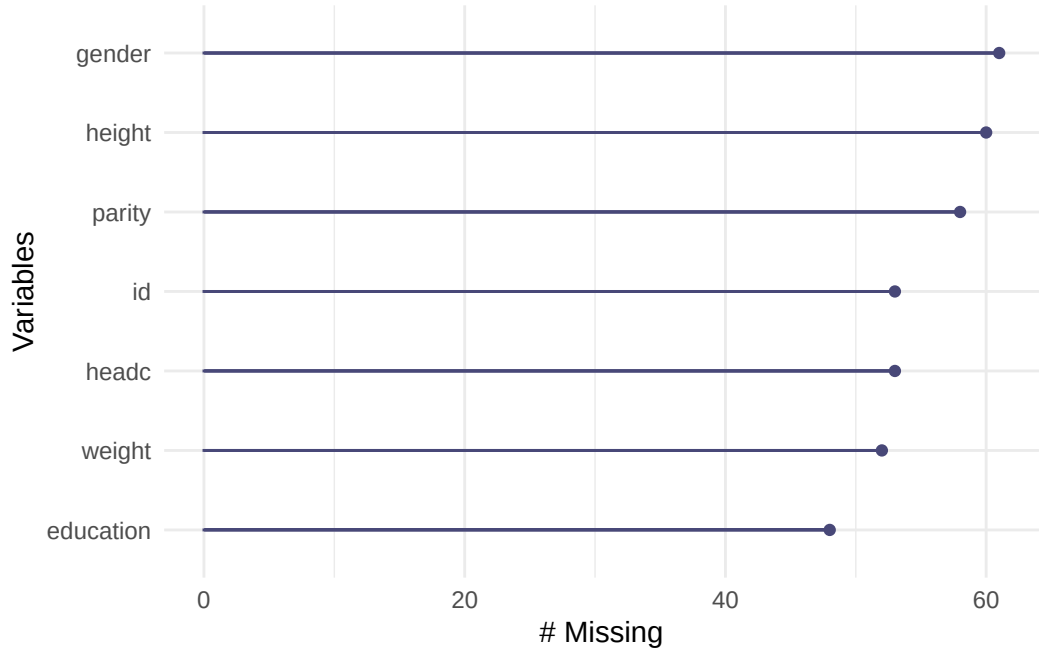
The vis_miss() function provides us with a visualization of the missing values:

```
vis_miss(BirthWeight.mis)
```




We can also have a plot with the number of missing values in each variable in our dataset using the gg_mis_var() function of the package:

```
gg_miss_var(BirthWeight.mis)
```




Finally some useful functions for insight of the dataset:

```
# The number of variables of the dataset that have NA's
n_var_miss(BirthWeight.mis)
```

```
[1] 7
```

```
# A summary of the NA's of the dataset for each variable.
miss_var_summary(BirthWeight.mis)
```

```
# A tibble: 7 x 3
  variable  n_miss pct_miss
  <chr>      <int>    <num>
1 gender        61    11.1
```

```
2 height        60    10.9
3 parity        58    10.5
4 id            53     9.64
5 headc         53     9.64
6 weight        52     9.45
7 education     48     8.73
```

# 5 Feature scaling

## 5.1 The golub dataset

Before applying any prediction algorithms for supervised machine learning or grouping algorithms for the unsupervised case, let's discover one of the datasets we are going to analyse and apply feature scaling so as to prepare it for further analysis. The dataset is called golub (Golub et al. 1999). It includes the calculated gene contributions of 72 patients which have two types of cancer Acute Myeloid Leukemia (AML) and Acute Lymphoblastic Leukemia (ALL) which are cancers of the blood and bone marrow. The rows of the dataframe are the 72 patients and the columns are the 7129 contributing genes. There are 5 more columns, **class, sample, type, FAB, gender.** The class column defines the group of cancer each of the 72 patients belong to. Two categories (ALL, AML).

```
GOLUB<-readRDS(file = "golub")
golub <-as.data.frame(GOLUB)[1:7134]
# Results of the first 4 genes for the first 5 patients:
golub[1:5, 1:4]
```

```
  AFFX.BioB.5_at AFFX.BioB.M_at AFFX.BioB.3_at AFFX.BioC.5_at
1           -214           -153            -58             88
2           -139            -73             -1            283
3            -76            -49           -307            309
```

```
4          -135          -114           265            12
5          -106          -125           -76           168
```

```
# The 2 cancer groups that the 72 patients belong to:
golub$class
```

```
 [1] "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
[13] "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
[25] "ALL" "ALL" "ALL" "AML" "AML" "AML" "AML" "AML" "AML" "AML" "AML" "AML"
[37] "AML" "AML" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
[49] "ALL" "AML" "AML" "AML" "AML" "AML" "ALL" "ALL" "AML" "AML" "ALL" "AML"
[61] "AML" "AML" "AML" "AML" "AML" "AML" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
```

```
library(openintro)  # for data
library(tidyverse)  # for data wrangling and visualization
library(knitr)      # for tables
library(broom)      # for model summary
```

The idea of feature scaling is connected with data distances, as it is going to be explained more analytically afterwards. The greater the distances the greater the contribution. In the specific dataset genes have a priori similar contributions. As there are different quantities, that they are measured in each column, genes, it would be a good idea to make our dataset uniform, so as to discover the main contributors, to avoid numbers with different orders of magnitude co-existing which could lead in false predictions.

Two cases accomplishing the target, which are the following:

1. Min-Max Normalization
2. Standardization.

## 5.2 Min-Max Normalization

In this type of normalization the data are transformed in a [0,1] range. The following equation

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

where x' is the normalized value and x the initial one, min(x) and max(x) the minimum and maximum values of a variable respectively. In this type of normalization the maximum value of the dataset is given the value of 1 the minimum of 0 and the rest lie in the (0,1) range.

To implement this in R, we create and use a function called **normalize** :

```
#First strip off the non-gene columns of the dataset (last 5 columns):
genes_golub <-golub[,-c(7130:7134)]

# This function implements the normalization
normalize <- function(x, na.rm = TRUE)
  {
  return((x- min(x)) /(max(x)-min(x)))
}

# The normalize function is applied in each column of the new dataset
normgenes_golub<-apply(genes_golub,2,normalize)
```

The number 2 in the second position of the **apply** function denotes the second dimension of the dataset. Which are the columns. The first is the rows. The normalization is applied to each gene variable, so in the columns. Having a look of the result of the first column:

```
head(normgenes_golub[,1])
```

```
[1] 0.4661922 0.5996441 0.7117438 0.6067616 0.6583630 0.6014235
```

> **Note**
>
> Disadvantage of the scaler: Due to the fact that it is dependent of the minimum and maximum of the variable it gets biased, when outliers are present.

## 5.3 Standarization

The method of standarization reforms the dataset in such a way so as to have mean zero and standard deviation equal to 1. In order to do so the transformation is taken place by applying the formula below:

$$x' = \frac{x - \mu}{\sigma}$$

where $\mu$ is the mean and $\sigma$ the standard deviation of the dataset. It is also called as z score standarization.

To implement this in the dataset a **base** function called scale can be used:

```
# The scale function belongs to the base R
norm_Z_genes_golub<-scale(genes_golub, center=TRUE,scale=TRUE)
```

The center=TRUE, scale=TRUE above could be ignored, as they are the default in the scale function.

**center=TRUE** (centering the variable (gene contribution), subtracting mean value from variable $x - \mu$.

**scale=TRUE** (dividing the variable with standard deviation, in our case the genes' contributions to all patients)

It is obvious now that all columns are scaled to $\sigma = 1$ standard deviation. To verify this just:

```
sd(norm_Z_genes_golub[,1])
```

```
[1] 1
```

To explain it graphically, the $\mu$ subtraction transfers all data around y axis and the division with $\sigma$ leads to $\sigma = 1$ standard deviation.

> **Tip**
>
> The standarization is quite useful when the feature distribution is Normal or Gaussian and contrary to normalization it is not affected so much from outliers.

# 6 Supervised machine Learning -Classification

## 6.1 Splitting the initial dataset

As mentioned in the introduction, supervised machine learning is the process of finding a future result, using a model created by an algorithm from known results.

Now it is time to apply the machine learning algorithms in order to make a prediction. But what type of prediction could be applied when the results are already known? In the specific dataset (golub) there are distinctive results for every patient. It is already known what type of cancer they all have.

It is time to make an agreement. In order to continue, we should divide our dataset into two groups. The first one, the larger (usually 70%-80% of the initial dataset) should be our **training set**. The dataset that we are going to use the algorithm with the known results in order to create a model. The rest of the initial dataset could be used as a **test set,** the set to apply the model and test it, checking and counting the agreement of predicted results with the observed ones, the **accuracy** of the model.

```
# Splitting the dataframe to train set and test set
GOLUB<-readRDS(file = "golub")
golub <-as.data.frame(GOLUB)[1:7134]
library(caTools)
set.seed(1234)
```

```
split <- sample.split(golub$class, SplitRatio = 0.75)
split
```

```
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
[13]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[25]  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
[37]  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
[49]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[61]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

The sample.split() function of **caTools** package takes as an input a vector of data labels. In our case we can take the class column of golub:

```
golub$class
```

```
 [1] "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
[13] "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
[25] "ALL" "ALL" "ALL" "AML" "AML" "AML" "AML" "AML" "AML" "AML" "AML" "AML"
[37] "AML" "AML" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
[49] "ALL" "AML" "AML" "AML" "AML" "AML" "ALL" "ALL" "AML" "AML" "ALL" "AML"
[61] "AML" "AML" "AML" "AML" "AML" "AML" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
```

and it divides them by pseudo random order, in such a way that e.g. 75% of the above will take the **TRUE** value while the rest the **FALSE**. Then we divide golub into two groups in accordance of the labels we have chosen (TRUE, FALSE):

```
train.data<- subset(golub, split == TRUE)
test.data <- subset(golub, split == FALSE)
```

By following that way we have achieved obtaining the two sets of data which are needed for applying the algorithms. Let's start applying them one by one in R.

## 6.2 The distance factor in ML algorithms

Some basic analytic geometry calculus, quite useful in order to continue. Let's first learn or refresh our memory of how to calculate the distance of two points in a n dimensional space. To make things simpler let's say that we are considering a 2 dimensional space.

In euclidean geometry it is known that if $(x_1, x_2)$ are the coordinates of a point in this space and $(y_1, y_2)$ of another, to find the distance between them we use the Pythagorean formula below:

$$\Delta_\chi = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}$$

In case we have an n dimensional space we can expand the result to the following:

$$\Delta_\chi = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2 + ... + (y_n - x_n)^2}$$

This is the most basic metric that is going to be used in terms of the distance. It is applied by default in all the functions that implement the algorithms later on. There are other metrics like **Minkowski distance**, **cosine distance** and so on. We could use them in special cases, but not here.

What about our golub dataset? Can we say of what consists of, in terms of n-dimensional points? One could say that, as we have 72 observables (72 patients) and 7129 genes to count on, we are dealing with 72 points of 7129 dimensions each. So our algorithms are going to count the distance of 72 points in a 7129 dimensional space as mentioned above in order to find an optimal way to make a prediction. How are we going to achieve this? Let's start with the first algorithm.

## 6.3 The K-nn algorithm (k-nearest neighbors)

The simplest algorithm of all is the k-nn algorithm. The idea behind it, is classification by a plurality vote of neighbors. Imagine, as mentioned above, our points (patients) in the training set and the result of them in terms of a cancer class (AML, ALL).

We create a k-nn model in terms of their positions (patients that belong to the training set) in the 7129 space that we have. Any new object (patient) of the test set would be classified in terms of its distance with a k number of neighbors, this number is defined beforehand.

So in case of k=3, we will measure the distance of the specific new object (patient) with the 3 closest ones (of the training set), and if the majority of them belong to e.g. **ALL,** we will classify it as such. Simple as that.

The application in R is as follows,

```
# create a factor vector class to obtain the classification info
class<-factor(train.data$class)


# we get rid of the last 5 unwanted columns of the dataset:
training_set<-train.data[1:7129]
test_set<-test.data[1:7129]
```

we install the class library in order to implement k-nn and we apply the algorithm with the function k-nn():

```
#install.packages('class')
library(class)
set.seed(123)
# we apply the knn in this order:
y_pred <- knn(training_set, test_set, class , k = 5)
```

❗ The KNN function input

**knn(training_set, test_set, class , k = numofneighbors)**

- **training_set:** data-frame that contains the training data.
- **test_set**: data-frame that contains the test data.
- **class:** Factor, containing the true classifications of the trainset (The observables)
- **k:** Number of neighboring points that are taken into consideration. (k-neighbors).

The predictions for the test_set are the following:

```
y_pred
```

```
 [1] ALL ALL ALL ALL ALL ALL AML AML AML ALL ALL ALL AML AML ALL ALL ALL ALL
Levels: ALL AML
```

while the observed ones from the initial dataset can be retrieved from:

```
test_results<-factor(test.data$class)
test_results
```

```
 [1] ALL ALL ALL ALL ALL ALL AML AML AML ALL ALL ALL AML AML ALL ALL AML ALL
Levels: ALL AML
```

## 6.4 The confusion matrix

Now it is time to introduce a matrix that actually summarizes the performance of our classification algorithm. It is called the confusion matrix and is defined as follows:

The number of rows and columns of the matrix is dependent on the number of the distinct cases that we have. For a binary classification case, as ours (ALL, AML) the confusion matrix is a 2X2. For 3 classes 3X3 and so on.

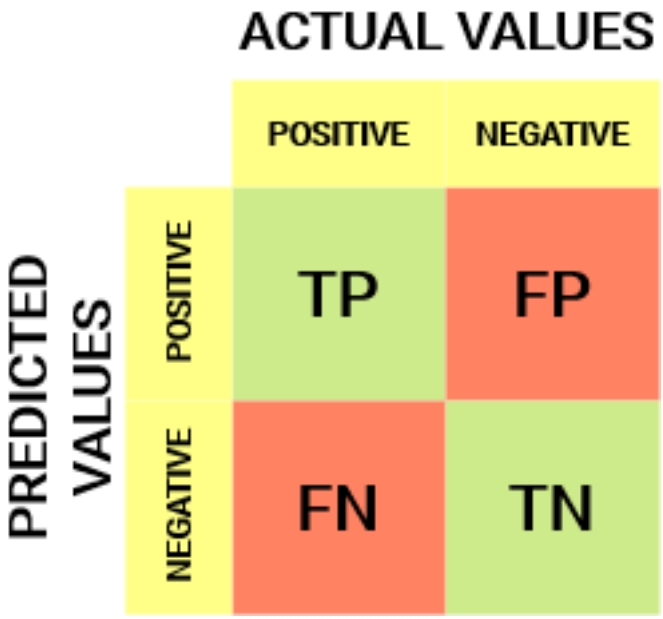



Figure 6.1: A 2X2 matrix for a binary classification case

The matrix provides us with the info of the type of correct-incorrect predictions too. Here let's make an agreement in order to make things more understandable. Let's define positive the **ALL** case and negative the **AML** one.

- **true positive** for correctly predicted the ALL values.
- **false positive** for incorrectly predicted ALL values. (It predicted ALL while the observed case is AML).
- **true negative** for correctly predicted AML values.
- **false negative** for incorrectly predicted AML values. (It predicted AML while the observed case is ALL).

In the same footing with the above:

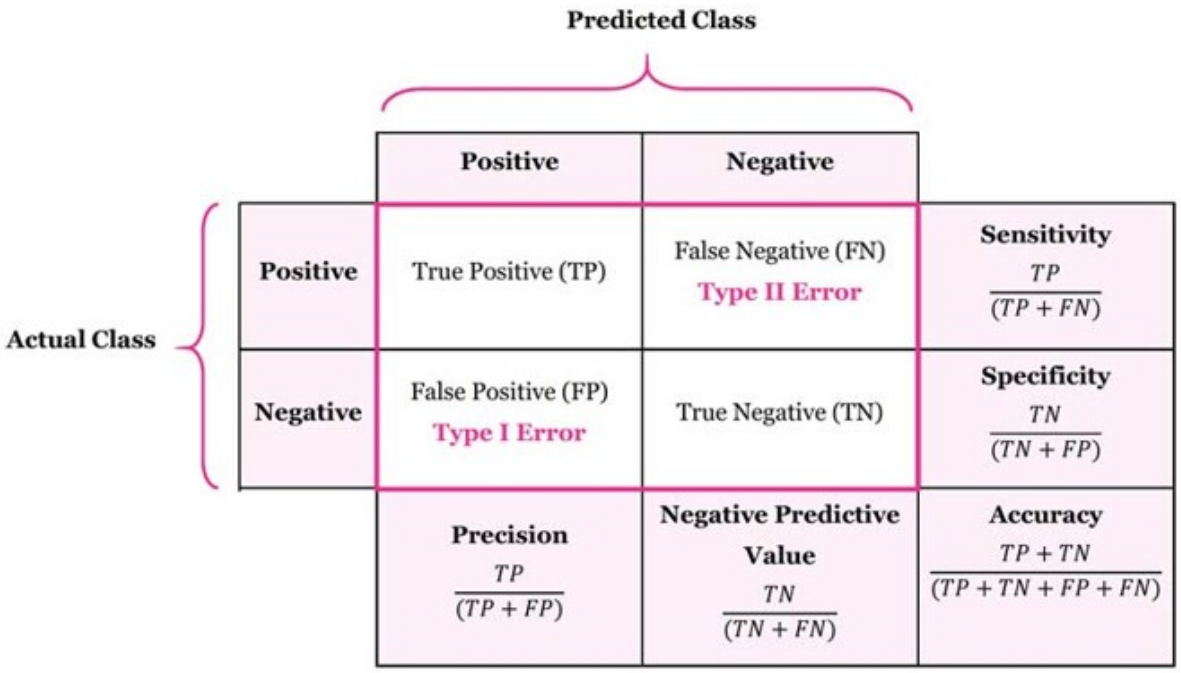



Figure 6.2: Evaluation measures of a CM

Let's check them, one by one:

**Accuracy:** Is the number of correct predictions divided by the total number of dataset cases. It takes values from 0 to 1. 0,5 is the completely random case (no predictability at all).

$$Acc = \frac{TP}{(TP + FP)}$$

**Sensitivity:** The number of correct positives divided by the number of all positives. One can also find it as **recall**:

$$Sensitivity = \frac{TP}{(TP + FN)}$$

**Specificity or True negative rate :** The number of correct negative predictions divided by the total number of negatives.

$$Specificity = \frac{TN}{(TN + FP)}$$

**Precision (Positive predictive value):** The number of correct positive predictions divided by the total number of positive predictions.

$$Precision = \frac{TP}{(TP + FP)}$$

We can calculate the confusion matrix in two ways using R.

Using the base function table():

> ❗ Calculating the Confusion matrix
>
> **cm<-table(testresults,Ypred)**
>
> **testresults**: A factor vector that includes the observed results of the test set.
>
> **Ypred**: A factor vector that includes the results of the predicted model.
>
> In case we need the evaluation measures too, we can use the caret package library:
>
> **library(caret)**
>
> **confusionMatrix(Ypred,testresults)**

#### 6.4.0.1 The K-nn algorithm (k-nearest neighbors) (continued)

Let's apply everything we learned so far:

```
cm <- table(test_results,y_pred)
cm
```

```
            y_pred
test_results ALL AML
         ALL  12   0
```

```
         AML   1   5
```

Using the caret function we get:

```
#confusionMatrix in caret package:
library(caret)
confusionMatrix(y_pred,test_results)
```

```
Confusion Matrix and Statistics

          Reference
Prediction ALL AML
       ALL  12   1
       AML   0   5

               Accuracy : 0.9444
                 95% CI : (0.7271, 0.9986)
    No Information Rate : 0.6667
    P-Value [Acc > NIR] : 0.006766

                  Kappa : 0.8696

 Mcnemar's Test P-Value : 1.000000

            Sensitivity : 1.0000
            Specificity : 0.8333
         Pos Pred Value : 0.9231
         Neg Pred Value : 1.0000
             Prevalence : 0.6667
```

```
       Detection Rate : 0.6667
Detection Prevalence : 0.7222
   Balanced Accuracy : 0.9167

    'Positive' Class : ALL
```

> ❗ When to use the algorithm
>
> - In general, k-nn is more appropriate when the relationships between features and target-classes are numerous, complex, or extremely difficult to understand but elements of a similar class type tend to be fairly homogeneous.
> - On the other hand, if our data has noise or there is no clear separation, things become difficult for k-nn.

## 6.5 The Support Vector Machines (SVM) Algorithm

The support vector machines (svm) algorithm follows the idea of finding the optimal dividing **line** (for two dimensions), **hyperplane** in case of three or more dimensions. The idea behind it is to compute the distance between the line (hyperplane) and the closest points (**support vectors**) to that line. The hyperplane with the biggest distance (**maximised margin**) is called **optimal hyperplane**. Everything works perfectly right when the data are linearly separable. Which means that they can be divided by a line or a plane.

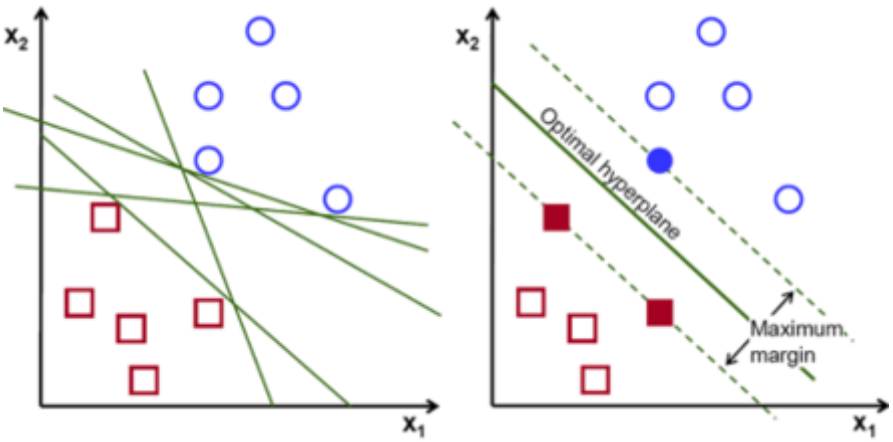



Figure 6.3: SVM in a 2-dimensional plane

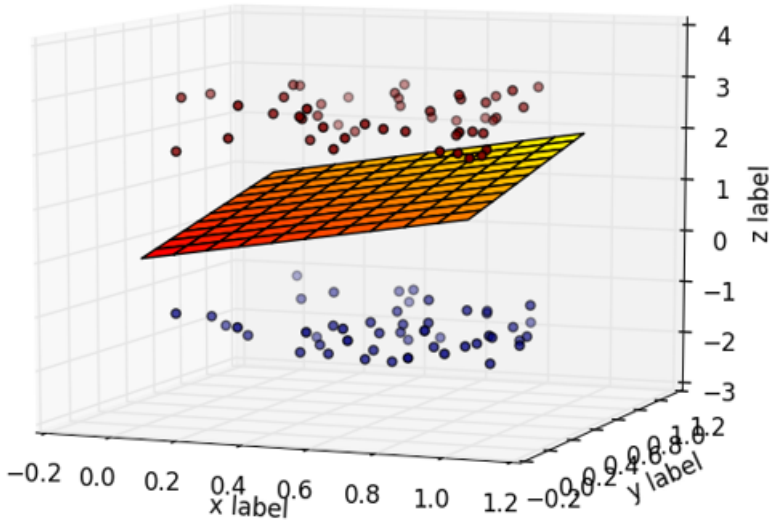



Figure 6.4: SVM in a 3-dimensional plane

What happens when things get more complicated?

The advantage of SVM is that it could transform the data in such a way in order to make them linearly separable. The trick accomplishing that is to apply kernels (functions) that can actually separate our data.

One way of implementing svm in R is by using the svm() function from the **e1071** library. It

actually provides 4 different kernels (linear, polynomial, radial, sigmoid) in order to transform the data in such a way so as to make them linearly separable.

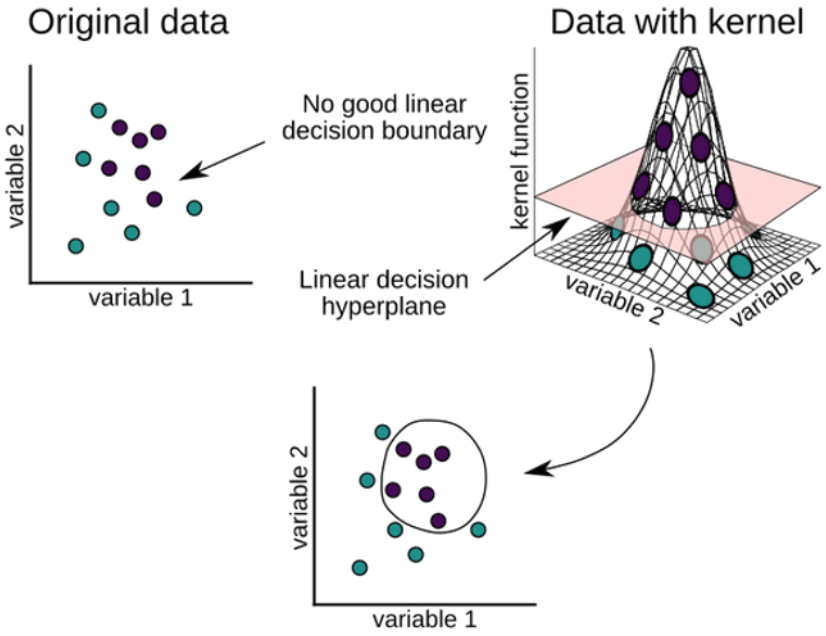



Figure 6.5: SVM using a kernel trick

The coefficients of these functions can be applied by the programmer (see also the help of svm() function).

## 6.6 The Breast Cancer Wisconsin (Diagnostic) Data Set

Now let's apply svm using another dataset. The Breast Cancer Wisconsin (Diagnostic) Data Set.(Blake 1998)

It is actually feature computation of characteristics of the cell nuclei present in digitized images of a **fine needle aspirate** (FNA) of a breast mass.

It includes the followed columns:

1. ID number
2. Diagnosis (M = malignant, B = benign)

   **and Ten real-valued features computed for each cell nucleus:**

3. radius (mean of distances from center to points on the perimeter)
4. texture (standard deviation of gray-scale values)
5. perimeter
6. area
7. smoothness (local variation in radius lengths)
8. compactness (perimeter^2 / area - 1.0)
9. concavity (severity of concave portions of the contour)
10. concave points (number of concave portions of the contour)
11. symmetry
12. fractal dimension ("coastline approximation" - 1)

    The **mean**, **standard error** and **"worst"** or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. Including a last column X with NA's in it, the dataset includes 33 columns in total:

```
cc_data <- read.csv(file = 'data.csv')
str(cc_data)
```

```
'data.frame':   569 obs. of  33 variables:
 $ id                     : int  842302 842517 84300903 84348301 84358402 843786 844359 8445
 $ diagnosis              : chr  "M" "M" "M" "M" ...
 $ radius_mean            : num  18 20.6 19.7 11.4 20.3 ...
```

```
$ texture_mean           : num  10.4 17.8 21.2 20.4 14.3 ...
$ perimeter_mean         : num  122.8 132.9 130 77.6 135.1 ...
$ area_mean              : num  1001 1326 1203 386 1297 ...
$ smoothness_mean        : num  0.1184 0.0847 0.1096 0.1425 0.1003 ...
$ compactness_mean       : num  0.2776 0.0786 0.1599 0.2839 0.1328 ...
$ concavity_mean         : num  0.3001 0.0869 0.1974 0.2414 0.198 ...
$ concave.points_mean    : num  0.1471 0.0702 0.1279 0.1052 0.1043 ...
$ symmetry_mean          : num  0.242 0.181 0.207 0.26 0.181 ...
$ fractal_dimension_mean : num  0.0787 0.0567 0.06 0.0974 0.0588 ...
$ radius_se              : num  1.095 0.543 0.746 0.496 0.757 ...
$ texture_se             : num  0.905 0.734 0.787 1.156 0.781 ...
$ perimeter_se           : num  8.59 3.4 4.58 3.44 5.44 ...
$ area_se                : num  153.4 74.1 94 27.2 94.4 ...
$ smoothness_se          : num  0.0064 0.00522 0.00615 0.00911 0.01149 ...
$ compactness_se         : num  0.049 0.0131 0.0401 0.0746 0.0246 ...
$ concavity_se           : num  0.0537 0.0186 0.0383 0.0566 0.0569 ...
$ concave.points_se      : num  0.0159 0.0134 0.0206 0.0187 0.0188 ...
$ symmetry_se            : num  0.03 0.0139 0.0225 0.0596 0.0176 ...
$ fractal_dimension_se   : num  0.00619 0.00353 0.00457 0.00921 0.00511 ...
$ radius_worst           : num  25.4 25 23.6 14.9 22.5 ...
$ texture_worst          : num  17.3 23.4 25.5 26.5 16.7 ...
$ perimeter_worst        : num  184.6 158.8 152.5 98.9 152.2 ...
$ area_worst             : num  2019 1956 1709 568 1575 ...
$ smoothness_worst       : num  0.162 0.124 0.144 0.21 0.137 ...
$ compactness_worst      : num  0.666 0.187 0.424 0.866 0.205 ...
$ concavity_worst        : num  0.712 0.242 0.45 0.687 0.4 ...
$ concave.points_worst   : num  0.265 0.186 0.243 0.258 0.163 ...
$ symmetry_worst         : num  0.46 0.275 0.361 0.664 0.236 ...
$ fractal_dimension_worst: num  0.1189 0.089 0.0876 0.173 0.0768 ...
```

```
 $ X                       : logi  NA NA NA NA NA NA ...
```

#### 6.6.0.1 The support vector machines algorithm (SVM) (continued)

It is time for a little data cleaning and reformatting:

```
# Eliminating the first and last columns of dataset…
#(No useful info is included) and factorizing the diagnosis column
cc_data$id<-NULL
cc_data$X<-NULL
cc_data$diagnosis<-factor(cc_data$diagnosis)
```

We end up with the first column as a binary classifier and the rest thirty (30) as our new dataset, ready for the implementation of the algorithm process. It is analogous to the k-nn procedure which is mentioned before. Dividing and scaling the dataset:

```
library(caTools)
set.seed(1234)
split <- sample.split(cc_data$diagnosis, SplitRatio = 0.75)
train.data<- subset(cc_data, split == TRUE)
test.data <- subset(cc_data, split == FALSE)

# Scaling all but the first column
train.data[2:31] <-sapply(train.data[2:31],scale)
test.data[2:31] <-sapply(test.data[2:31],scale)
```

In the last two lines of code the base function scale() is applied in all but the first column (it is left out, as an error would occur). Now it is time to apply the algorithm by using the e1071 library as mentioned above and more specifically the linear kernel:

```
library(e1071)
classifier <- svm(formula = diagnosis ~ .,
                  data = train.data,
                  type = "C-classification",
                  kernel = "linear")

# Prediction of the results according to model classifier
ypred <- stats::predict(classifier,test.data)

# The observed results directly from the initial dataframe
test_results<-as.factor(test.data$diagnosis)

#The confusion matrix
cm <- table(test_results,ypred)
cm
```

```
            ypred
test_results  B  M
           B 88  1
           M  1 52
```

```
# Alternative calculation of the confusion matrix with more info
# using the caret package
#library(caret)
#confusionMatrix(ypred,test_results)
```

**!** Calling the svm() function

**library(e1071)**

**classifier = svm(formula = Y ~ ., data = trainset, type = 'C-classification', kernel='nameofkernel')**

**Y ~v1+v2+ ...** : Y is the factor column (response variable) adding up the column (predictor variable) that we are interested in ( We place **~.** In case would like to include all predictor variables to train the model)

**trainset**: A dataframe (training set) where Y is the response variable and v1, v2,… are the predictor variables.

**type:** In case of classification (binary or more than 2 discrete types) we are referring to the **'C-classification'**. The function supports also **'eps-regression'** for regression (mentioned in the next chapter) and others (See also the function help).

**nameofkernel**: The kernel types that the program could use to make the prediction. ('linear', 'polynomial', 'radial', 'sigmoid').

**!** Predicting the result

**ypred = predict(classifier, testset)**

**ypred**: The binary or >2 discrete type prediction for the test set.

**testset:** A dataframe that contains the new data, for prediction.

From the result (the confusion matrix) it is obvious that the prediction model worked quite well. The prediction percentage is more that 90% which means that our model is quite capable of classifying new data. In general SVM is a quite important algorithm (quite accurate).

When to use the algorithm

- The algorithm could be used for both classification and regression. It can predict e.g. the blood class so as the height of a newborn child.
- Not overly influenced by noisy data and not very prone to over-fitting (**Overfitting:**

Results are adjusting very well in the training data but not well in new, test data).

- Finding the best model requires testing of various combinations of kernels and model parameters.

## 6.7 Decision Trees Algorithm

The algorithm's method is based on segmenting the predictor space (the space that is defined by your observables) into a number of simpler, more homogeneous regions, typically using the mean or the mode of the training observables. In other words it is a divide and conquer method where the initial dataset (the **root** of the tree) is divided in smaller and smaller groups of observables called **decision nodes** (the **branches** of the tree) until these subgroups are as homogeneous as possible. The **terminal nodes** (**leaves** of the tree) are presenting the predicting result following the splitting decision tactics starting from the root.

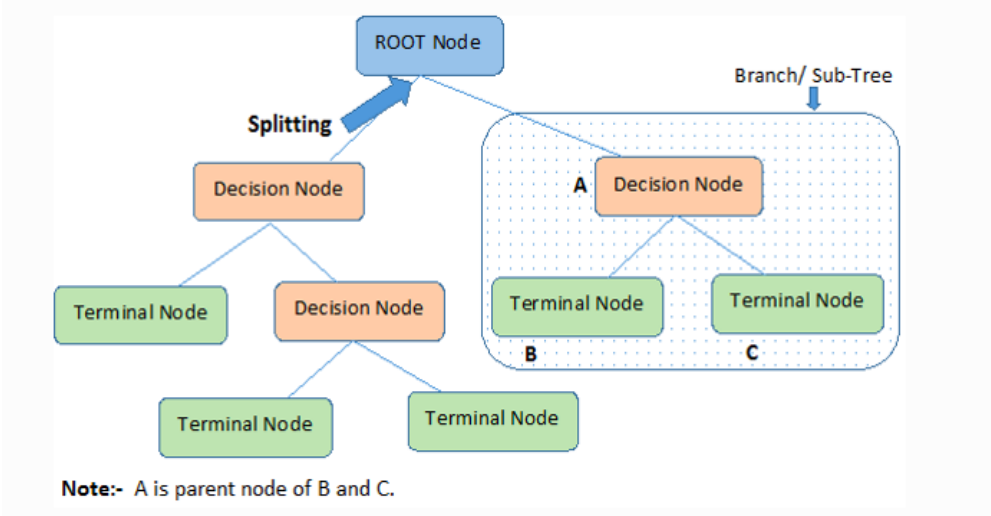



Figure 6.6: The divide and conquer method applied by the decision tree algorithm. The division to Decision nodes (branches) and Terminal nodes (leaves)

The pima indians dataset is going to be used:

```r
#install.packages('mlbench')
library(mlbench)
data(PimaIndiansDiabetes)
pm <- PimaIndiansDiabetes
str(pm)
```

```
'data.frame':   768 obs. of  9 variables:
 $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
 $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
 $ pressure: num  72 66 64 66 40 74 50 0 70 96 ...
 $ triceps : num  35 29 0 23 35 0 32 0 45 0 ...
 $ insulin : num  0 0 0 94 168 0 88 0 543 0 ...
 $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 0 ...
 $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
 $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
 $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

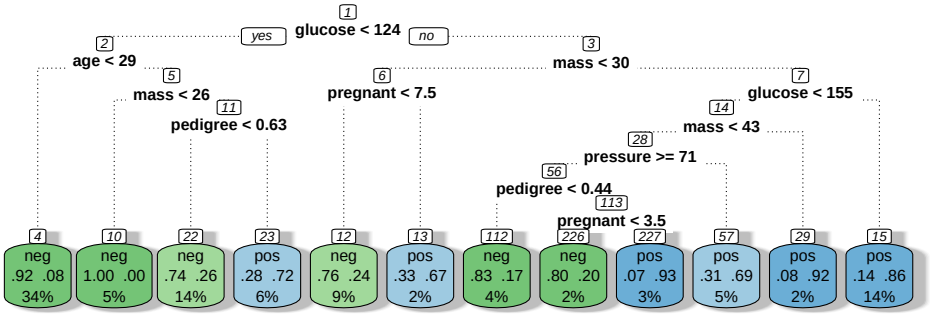
This time no scaling is needed. The decision tree algorithm is based in dividing the data and not in measuring distances between variables as in other algorithms. So there is no need for scaling.

```r
library(caTools)
set.seed(1234)
split <- sample.split(pm$diabetes, SplitRatio = 0.70)
train.data<- subset(pm, split == TRUE)
test.data <- subset(pm, split == FALSE)
```

The function that is used for implementing the algorithm is called rpart() from the so named package, **rpart** package. In order to have a better look of the tree structure, a function is used

here called fancyRpartPlot() of the rattle package. By applying the above to the train.data dataset:

```
#install.packages('rpart')
library(rpart)
classifier = rpart(formula = diabetes ~., method ='class' ,data =train.data)
#install.packages('rattle')
library(rattle)
tree<-fancyRpartPlot(classifier, type=0)
```




Rattle 2024−..a..−10 01:46:34 tdiak_000

Let's explain what is shown …

Now it is time for the prediction. Following the same footing as in the k-nn case:

```
test_predictions =predict(classifier,test.data,type='class')


test_observables<-test.data$diabetes
```

```
cm<-table(test_observables,test_predictions)


cm
```

```
                test_predictions
test_observables neg pos
             neg 125  25
             pos  42  38
```

**!** Calling the rpart() function …

**library(rpart)**

**classifier = rpart(formula = Y ~v1+v2+…, method =‘method_type’ ,data =trainset)**

**Y ~v1+v2+ …** : Y is the factor (response variable) column adding up the columns (predictor variables) that we are interested in ( We place **~.** In case would like to include all the predictor columns in the train of the model)

**data**: A dataframe (The training set) where Y is the factor (response variable) and v1, v2,… are the predictor variables.

**method:** As we are using classification (binary or more than 2 discrete types) we are referring to the ‘class’. The function supports also ‘poisson’,‘anova’ and others (See also the function help).

**!** Predicting the result

**test_predictions =predict(classifier,testset,type=‘class’)**

**type:** The type of prediction, ‘class’ from classification

**testset:** A dataframe that contains the new data (predictor variables), for prediction.

Tip

The decision tree algorithm can deal with nominal-ordinal data too, it is quite easy for someone to grasp the method, and the algorithm is quite fast. The disadvantages is that is not as predictive as others. The tree structure is quite sensitive to data changes. Meaning that even a small change in data could cause a relatively big difference in the tree structure.

## 6.8 Random Forests Algorithm

The random forest algorithm is based on a idea which is called **ensemble learning**. It is as follows: Multiple algorithms are combined together in order to make a stronger predictor. One of the most common ensemble methods is called **bagging**.

Different models are trained together. Each model makes a prediction. The one that belongs to the majority vote is taken as the final decision.

In the case of random forests the algorithm that is used is the same. The decision trees algorithm. The trick that is used is that instead of training the algorithm for the whole training set, only **k random datapoints** of the initial dataset are chosen to train it. So the model that is created is based on those. By applying the same algorithm for different k datapoints we create new models. The number of the models is what we call the number of **trees** in the forest. It is a parameter of the algorithm that the user is inputting. When a new data point arrives all these models are applied to make a prediction. The final decision (the category to which the new data point is assigned) is the one who wins the majority vote of all these models.

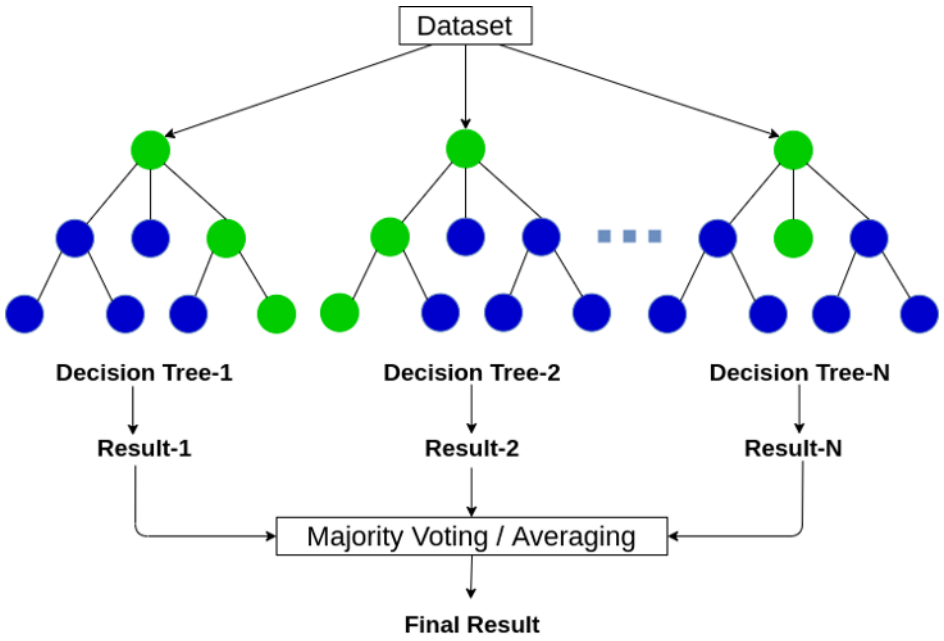



Figure 6.7: Illustration of random forest stages. A set of decision tree algorithms is used in combination, in order to calculate the final prediction. Here based on the majority vote (hard voting).

The function that is used to implement the algorithm is randomforest() of the so called package:

```
#install.packages('randomForest')
library(randomForest)
# separating the class of the data neg or pos
class<-as.factor(train.data$diabetes)
#removing the dataset column from both the training and test set
training_set<-train.data[-9]
test_set<-test.data[-9]
# creating the model using 50 trees
classifier = randomForest(x = training_set, y = class,
                          ntree = 50)
```

Following the same footing as before:

```
test_predictions = predict(classifier,test_set)
# the observed test results
test_observables<-as.factor(test.data$diabetes)


cm <- table(test_observables,test_predictions)
cm
```

```
                test_predictions
test_observables neg pos
             neg 125  25
             pos  36  44
```

**!** Calling the randomForest() function ...

**library(randomForest)**

**classifier = randomForest(x = trainset, y =train_observables ,ntree=numoftrees)**

**test_predictions = predict(classifier,test_set)**

**x**: A dataframe (The training set) with only the predictor variable columns (not the prediction)

**y:** The known results (observed) from the training set.

**test_set:** A dataframe that contains the new data, inspected for prediction.

**test_predictions:** The binary or more than 2 discrete types prediction results for the test_set.

**ntree**: The number of the trees contained in the forest.

When to use the algorithm

The random forest algorithm is quite difficult to interpret, even though that it consists of decision trees that can be easily interpreted. It can handle though quite well different types of datasets with continuous and categorical variables, so as big data.

## 6.9 Artificial neural networks (ANN's)

The artificial neural networks algorithm was built in order to be consistent with the biological neural network structure and operation. The latter consists of nerve cells that are called neurons. Each neuron have inputs synapses called dentrites and outputs (axons) through which links to each other. The electric impulses (info) the neuron receives from its synapses is processed in a certain way into the cell body. If the signal is strong enough it travels through the axon to its terminals (axon buttons) to the next neuron dendrites. It works like a switch.

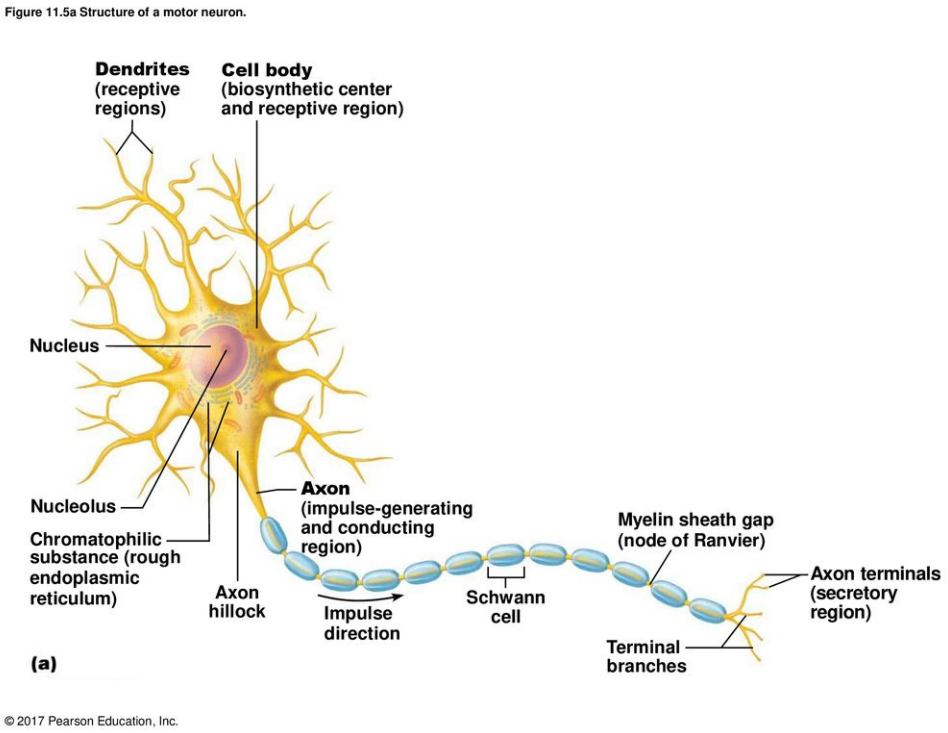



Figure 6.8: Neuron structure.

An Artificial neural network consists of neurons that are connected to each other. They are arranged in layers. From the incoming information, a neuron determines its state by means of a function (summation of the inputs and subsequent scaling with an **activation function**) This function is like a switch if a threshold is reached or not, the output is on or off.

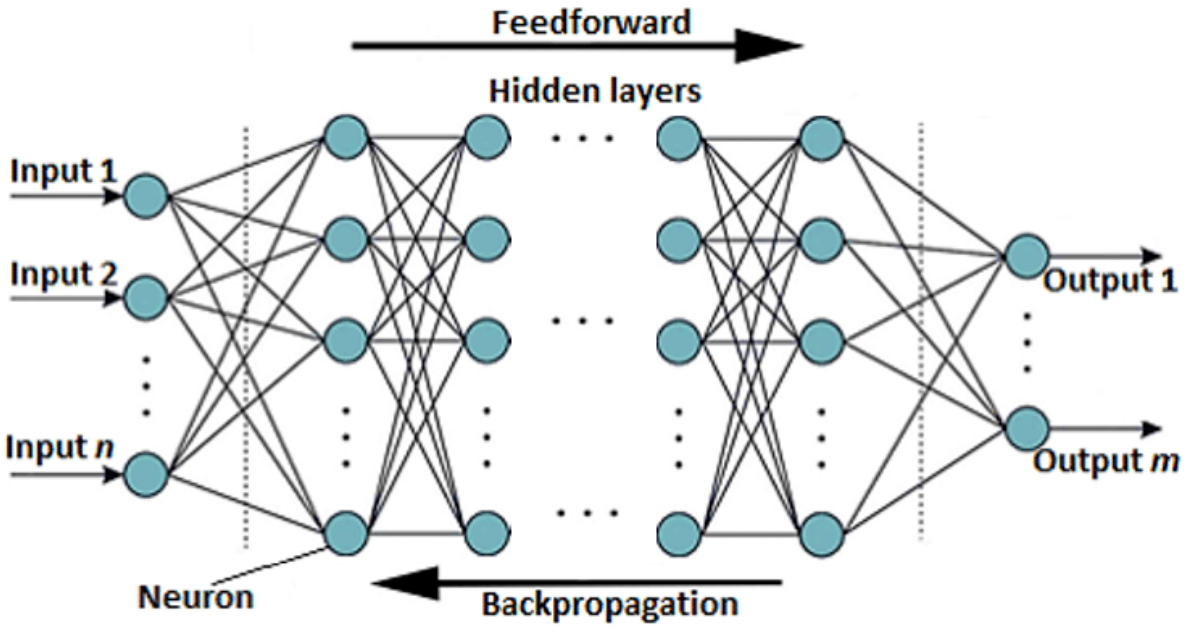



Figure 6.9: Fully connected Neural Network

It is time to apply the algorithm to the pima indians dataset. We initially split the dataset and standarize it as before:

```
library(mlbench)
data(PimaIndiansDiabetes)
pm <- PimaIndiansDiabetes

library(caTools)
set.seed(1234)
split <- sample.split(pm$diabetes, SplitRatio = 0.70)
train.data<- subset(pm, split == TRUE)
test.data <- subset(pm, split == FALSE)

# Scaling all but the last column
train.data[-9] <-sapply(train.data[-9],scale)
```

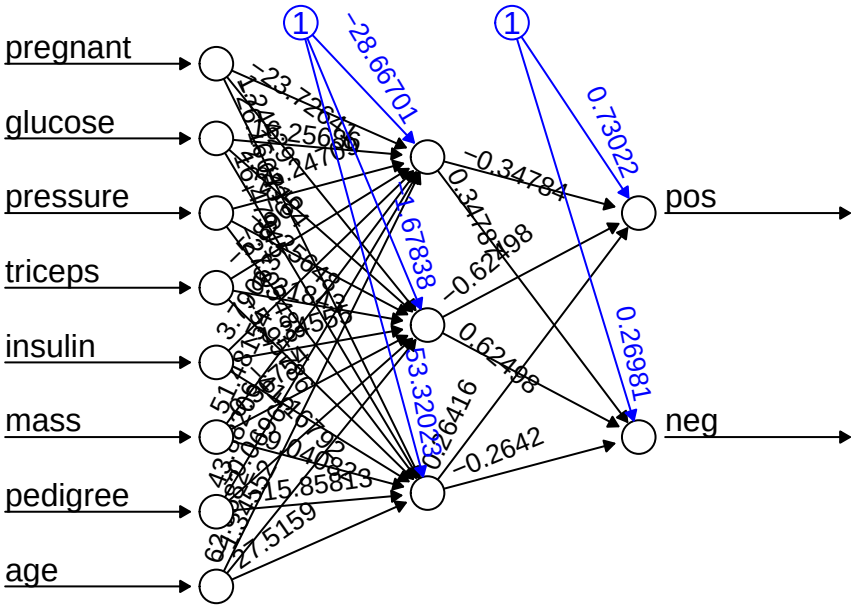
```
test.data[-9] <-sapply(test.data[-9],scale)
```

By installing the neuralnet() function, it is possible to create an ann and apply it to the dataset:

```
#install.packages('neuralnet')
library(neuralnet)
model<-neuralnet(diabetes ~ ., data=train.data, hidden=3)
```

The neuralnet() function provides the user with a graphic representation of the nn and the weights:

```
plot(model,rep = "best")
```




As a last step, it is time to apply the model that was prepared to the test.data subset, to make a prediction. We are using the predict function of the neuralnet package and ask for the net.result:

```
test_predictions = neuralnet::compute(model, test.data[-9])$net.result
```

The result is in terms of probabilities so we are using the max.col() function here to extract the proper result (neg if the prediction of first column is greater than the second or vice versa for the opposite):

```
#
test_predictions=data.frame("result"=ifelse(max.col(test_predictions[,1:2])==1,
                                            "neg","pos"))


# Confusion matrix
cm <- table(test.data$diabetes,test_predictions$result)
cm
```

```
     neg pos
  neg 131  19
  pos  36  44
```

And we end up with a result using the confusion matrix:

> **!** Calling the neuralnet() function
>
> **library(neuralnet)**
>
> **model<-neuralnet(Y ~ v1+v2+ ..., data=trainset, hidden= vectorofhidden-neurons, threshold = numthres)**
>
> **test_predictions = neuralnet::compute(model, test_set)$net.result**
>
> **Y ~v1+v2+ ...** : Y is the factor (response variable) column adding up the columns (predictor variables) that we are interested in ( We place **~.** In case would like to include all the predictor columns in the train of the model)

**trainset**: A dataframe (The training set) with only the observables columns (not the prediction).

**hidden:** A vector of integers specifying the number of hidden neurons (vertices) in each layer.

**threshold**: A numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria

**test_set**: A dataframe that contains the new data, inspected for prediction.

When to use the algorithm?

The greatest advantage of this algorithm is that it can model very complex patterns. On the other side due to its complexity it is quite computationally expensive and prone to over-fitting. It can be used for binary classification and for regression too.

# 7 Supervised Machine Learning-Regression

In order to predict a quantitative result, instead of a binary as the classification methods that were applied before do, it is time to show how these algorithms can do the trick for the latter case too. To begin, it would be constructive in terms of education purposes to start with the simplest and also oldest approach which is the **linear regression**. We will use the BirthWeight dataset:

```
# loading the BirhtWeight dataset
#install.packages('readxl')
library(readxl)
BirthWeight <- read_excel("BirthWeight.xlsx")
```

## 7.1 Linear regression

The simplest case of the linear regression, is a very straightforward approach for predicting a quantitative response Y on the basis of a single predictor variable X. It can be written down in terms of a mathematical equation like this:

$$Y = \alpha + \beta x_i + \epsilon_i$$

Where:

$Y_i$:Response or dependent variable

$X_i$:Independent or explanatory value

: The interception of the line to the Y axis.

: The slope of the regression line.

$\epsilon_i$ : Error or Residual, difference between the observed and predicted value.

And it is called **simple linear regression**. It is actually a linear approach for modelling the relationship between a quantitative response Y and an explanatory variable, as called, X. One can find them as dependent and independent variables too.

It is actually the process of finding the best line that fits the data, using the least squares approach. This approach achieves the above by **minimizing** the **residual sum of squares**:

$$RSS = \sum_{n=1}^{n} {\epsilon_i}^2$$

The sum of squares of all the errors, the sum of differences of the observed values in respect of the predicted values of the model.

Let's start finding a linear model that connects the weight of the infant with the height in the new BirthWeight dataset. The base function lm() is going to be used. In order to show the results the **summ()** function from the package **jtools**, alternatively one can choose the base **summary()** function:

```
#####################  Simple Linear Regression  ####################


## fit regression: weight ~ height
library(caTools)
set.seed(1234)
split <- sample.split(BirthWeight$height, SplitRatio = 0.75)
```

```r
train.data<- subset(BirthWeight, split == TRUE)
test.data <- subset(BirthWeight, split == FALSE)



#install.packages('jtools')
library(jtools)
model_height <- lm(weight ~ height, data = train.data)
summ(model_height, confint = TRUE, digits = 3)
```

```
MODEL INFO:
Observations: 414
Dependent Variable: weight
Type: OLS linear regression

MODEL FIT:
F(1,412) = 398.536, p = 0.000
R² = 0.492
Adj. R² = 0.490

Standard errors: OLS
------------------------------------------------------------
                      Est.     2.5%    97.5%    t val.       p
----------------- -------- -------- -------- --------- -------
(Intercept)         -5.243   -6.191   -4.295   -10.870   0.000
height               0.175    0.158    0.193    19.963   0.000
------------------------------------------------------------
```

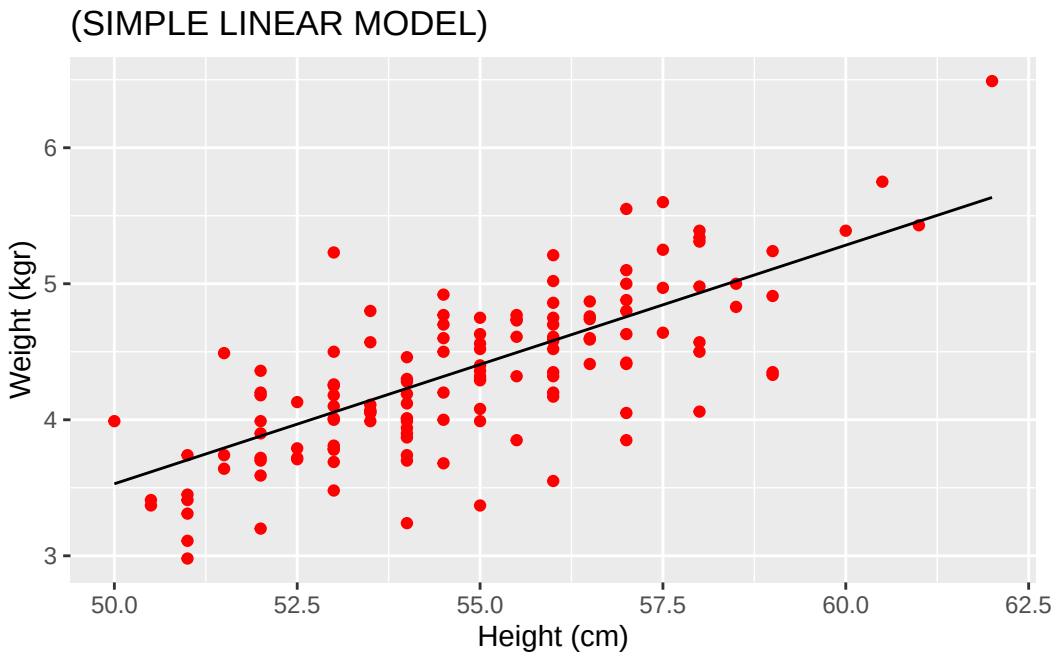
```r
## Alternatively using the summary function:
summary(model_height)
```

```
Call:
lm(formula = weight ~ height, data = train.data)

Residuals:
     Min       1Q   Median       3Q      Max
-1.22649 -0.27724 -0.01605  0.29193  1.35351

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -5.242702   0.482304  -10.87   <2e-16 ***
height       0.175440   0.008788   19.96   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4313 on 412 degrees of freedom
Multiple R-squared:  0.4917,    Adjusted R-squared:  0.4905
F-statistic: 398.5 on 1 and 412 DF,  p-value: < 2.2e-16
```

We can visualize it in two dimensions, the observed data (the one that we take directly from the test.data set) and the line that was drawn by the model that used the train.data set in order to produce it.

```r
# Predicting the test.data results using the model:
y_pred_linear <- predict(model_height, newdata = test.data)
```

```
library(ggplot2)
ggplot() +
    geom_point(aes(x = test.data$height, y = test.data$weight),
               colour = 'red') +
    geom_line(aes(x = test.data$height, y = y_pred_linear),
              colour = 'black')+
    ggtitle('(SIMPLE LINEAR MODEL)') +
    xlab('Height (cm)') +
    ylab('Weight (kgr)')
```




As an extension, new predictor variables can be added to make the prediction more accurate. It is possible to add as many as needed (wanted) in order to accomplish the target. Visually up to 2 can be handed. Here a library called plot3D is used in order to draw the scatterplot with the regression plane this time (extension of the line from two dimensions to plane in 3 dimensions). Keep in mind that here an extra dimension of the headc has been added in order to extend the phasespace to 3 dimensions.

```r
model_height_headc <- lm(weight ~ height+headc, data = train.data)
summ(model_height_headc, confint = TRUE, digits = 3)
```

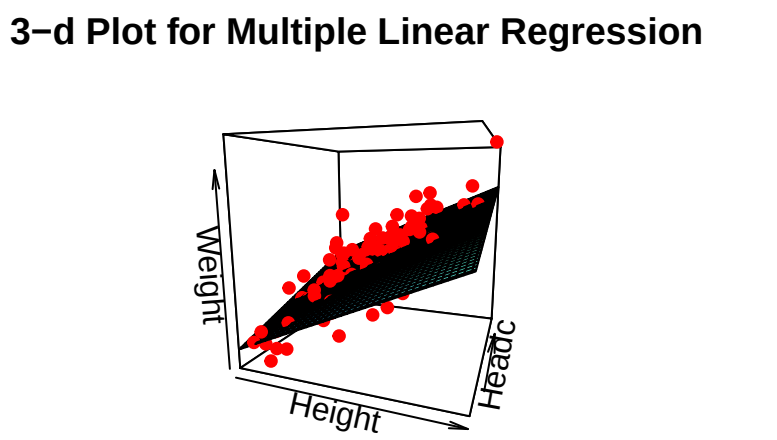
```
MODEL INFO:
Observations: 414
Dependent Variable: weight
Type: OLS linear regression

MODEL FIT:
F(2,411) = 256.607, p = 0.000
R² = 0.555
Adj. R² = 0.553

Standard errors: OLS
------------------------------------------------------------------
                      Est.     2.5%    97.5%    t val.       p
----------------- -------- -------- -------- --------- -------
(Intercept)         -7.878   -8.994   -6.763   -13.880   0.000
height               0.128    0.108    0.148    12.424   0.000
headc                0.138    0.103    0.174     7.667   0.000
------------------------------------------------------------------
```

It can be visualized in 3 dimensions using the function scatter3D() from the package plot3d.

```r
#install.packages('plot3D')
library("plot3D")
```

```
# set the x, y, and z variables from the train.data to create the model
x <- train.data$height
y <- train.data$headc
z <- train.data$weight

# Compute the linear regression
fit <- lm(z ~ x+y)



# set the new test.data x, y, and z variables
x <- test.data$height
y <- test.data$headc
z <- test.data$weight

# create a grid from the x and y values (min to max) and predict
# values for every point this will become the regression plane
grid.lines = 40
x.pred <- seq(min(x), max(x), length.out = grid.lines)
y.pred <- seq(min(y), max(y), length.out = grid.lines)
xy <- expand.grid( x = x.pred, y = y.pred)
z.pred <- matrix(predict(fit, newdata = xy),
                 nrow = grid.lines, ncol = grid.lines)



# scatter plot with regression plane
scatter3D(x, y, z, pch = 19, cex = 0.8,
          #ticktype="detailed",
          col="red",
```

```
          theta = 20, phi = 10, bty="b",
          xlab = "Height", ylab = "Headc", zlab = "Weight",
          surf = list(x = x.pred, y = y.pred, z = z.pred, facets = TRUE,
          col=ramp.col (col = c("dodgerblue3","seagreen2"),
                        n = 300, alpha=0.9),border="black"),
          main = "3-d Plot for Multiple Linear Regression ")
```




**!** Linear Regression Implementation in R

**model<-lm(Y ~ v1+v2+ ..., data=trainset)**

**Y ~v1+v2+ ...** : Y is the factor (response variable) column adding up the columns (predictor variables) that we are interested in ( We place **~.** In case would like to include all the predictor columns in the train of the model)

**data**: A dataframe (The training set) with only the columns of observables (not the prediction)

**summ(model, confint = TRUE, digits =precisiondigits)**

**model**: The model that created from the function

**confint**: include the confidence intervals or not (TRUE or FALSE respectively)

**digits:** The precision in number of digits

Tip

The main advantage of the linear regression is its simplicity and adaptability to any dataset. Although it is a biased model, as it makes the linearity assumption which is not the case for all types of data.

## 7.2 Regression using SVM

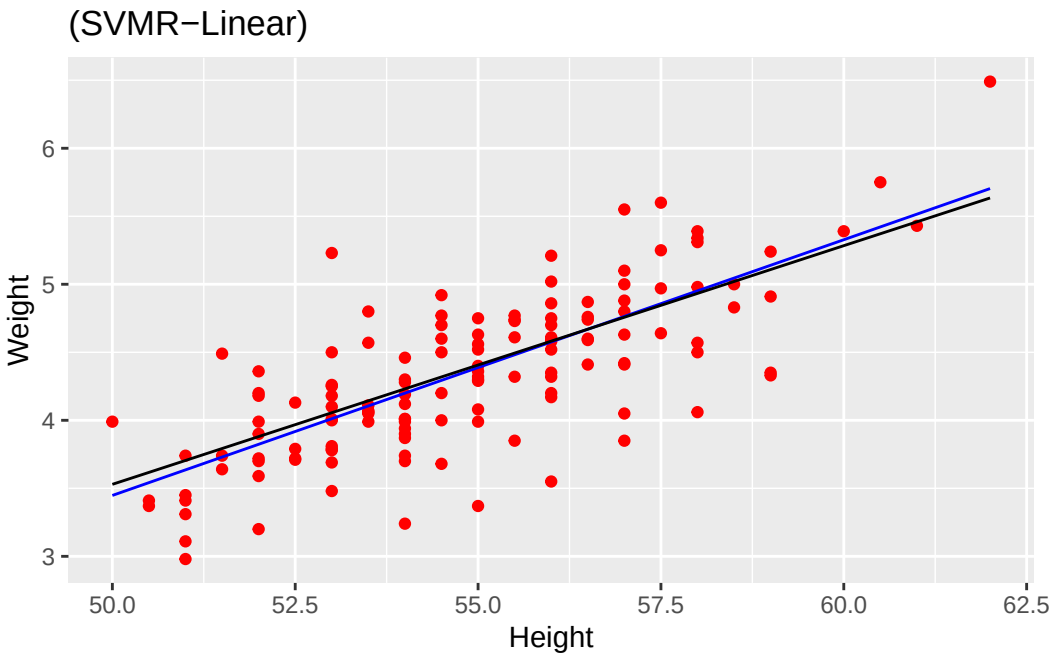
In the same footing as before with the only difference the change in type (eps-regression) the svm algorithm can be applied for regression:

```
library(e1071)
regressor = svm(formula = weight ~ height,
                data = train.data,
                type = 'eps-regression',
                kernel = 'linear')

# Make a prediction for just one result (e.g. for height = 55 cm)

y_pred = predict(regressor, data.frame(height = 55))

# Prediction for all the test.data points, SVR:

y_pred <- stats::predict(regressor, newdata = test.data)
```

```
# Prediction with simple linear regression


#model_height<-lm(weight ~ height, data = train.data)


#y_pred_linear <- predict(model_height, newdata = test.data)
```

In order to juxtapose the results with that of the linear regression:

```
library(ggplot2)
ggplot() +
    geom_point(aes(x = test.data$height, y = test.data$weight),
               colour = 'red') +
    geom_line(aes(x = test.data$height, y = y_pred),
              colour = 'blue') +
    geom_line(aes(x = test.data$height, y = y_pred_linear),
              colour = 'black')+
    ggtitle('(SVMR-Linear)') +
    xlab('Height') +
    ylab('Weight')
```




## 7.3 Regression using decision trees algorithm

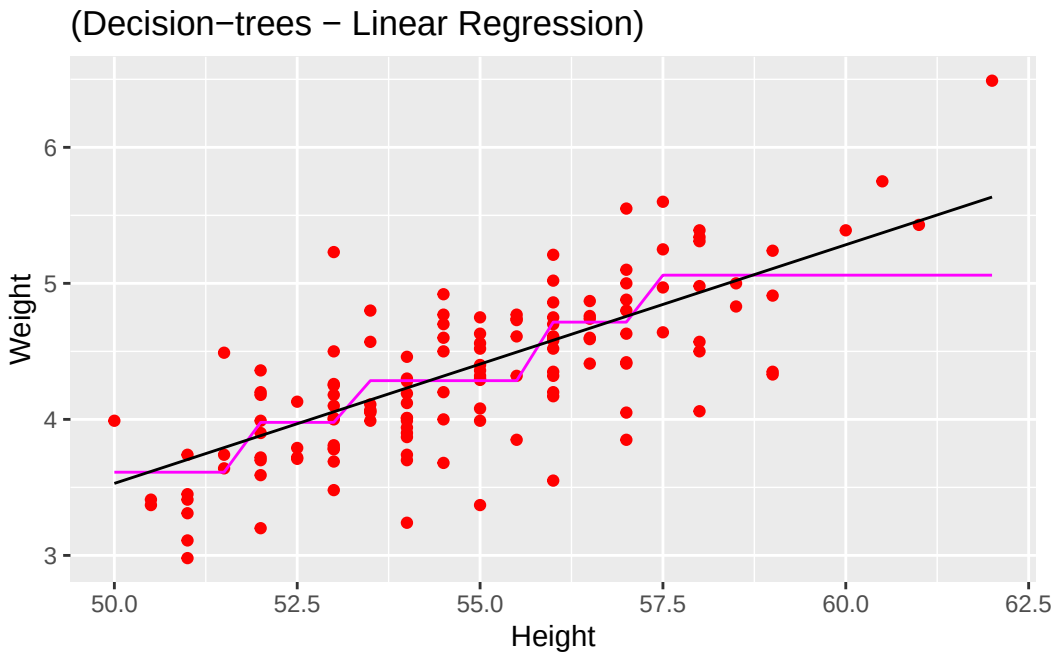
Here things are simpler. The function rpart from the so-called library is used:

```
library(rpart)
regressor = rpart(formula = weight ~ height,
                  data = train.data)

y_pred = predict(regressor, newdata =test.data)
```

In order to juxtapose the results with that of the linear regression:

```
library(ggplot2)
ggplot() +
    geom_point(aes(x = test.data$height, y = test.data$weight),
```

```
            colour = 'red') +
  geom_line(aes(x = test.data$height, y = y_pred),
            colour = 'magenta') +
  geom_line(aes(x = test.data$height, y = y_pred_linear),
            colour = 'black')+
  ggtitle('(Decision-trees - Linear Regression)') +
  xlab('Height') +
  ylab('Weight')
```




## 7.4 Regression using Random Forest

Finally last but not least, the random Forest case:

```
#We choose here ntree=10
library(randomForest)
```

```
randomForest 4.7-1.1
```

```
Type rfNews() to see new features/changes/bug fixes.
```

```
Attaching package: 'randomForest'
```

```
The following object is masked from 'package:ggplot2':
```

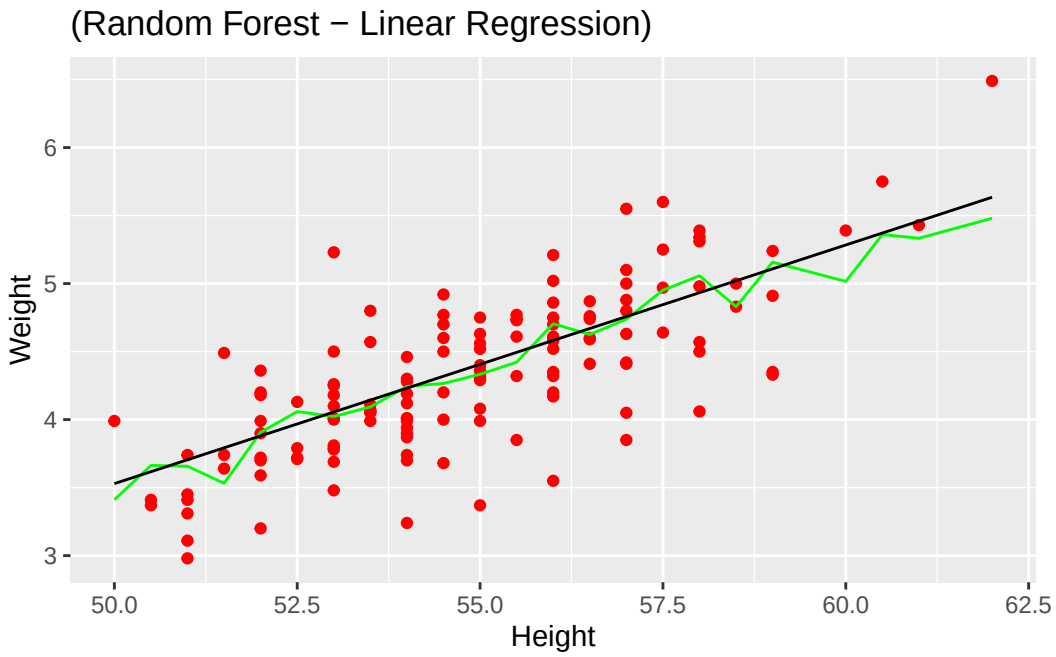
```
    margin
```

```
regressor = randomForest(formula = weight ~ height,
                         data = train.data,
                         ntree = 10)

y_pred = predict(regressor, newdata =test.data)
```

And plotting everything together:

```
library(ggplot2)
ggplot() +
    geom_point(aes(x = test.data$height, y = test.data$weight),
               colour = 'red') +
    geom_line(aes(x = test.data$height, y = y_pred),
              colour = 'green') +
    geom_line(aes(x = test.data$height, y = y_pred_linear),
```

```
            colour = 'black')+
  ggtitle('(Random Forest - Linear Regression)') +
  xlab('Height') +
  ylab('Weight')
```



# 8 Dimension Reduction

The series of methods that turn a set of many variables of a dataset into a smaller one, retaining most of the initial information, is what **dimension reduction** is. There are quite a few methods that could be used to this direction.

## 8.1 Principal Component Analysis (PCA)

The most applicable dimension reduction method is the Principal Component Analysis or PCA in short. The basic idea of the PCA is to construct new variables that carry the variation/information of the data in descending order. This process transforming the data without the need of initial knowledge is the first **unsupervised method** that will be applied in these notes. As the info is in descending order, a small number of new variables could provide us a large amount of the variation, the dataset's info, so by discarding the others we can safely retain much of the information in order to:

1. Use it in a classification prediction case
2. Use it in a regression prediction case
3. Use it to better cluster our data (more in the next chapter)

What can be achieved by all of this fuss? A lot actually.

Big data can be safely reduced to much smaller, so what we can gain from that?

1. Easier to visualize them, so as better to understand them.
2. All the supervised machine learning methods that mentioned before can be applied faster and easier as the calculation becomes less expensive, computationally.
3. Lessen the curse of dimensionality. When the number of observables are far less that the number of variables in a dataset or when we have sparse data (a lot of 0's or NA's as no info is available) or both, it is more difficult for the algorithms to converge due to this sparsity. The algorithm may start to learn from noise instead.
4. Lessen the collinearity effects. Create new variables that are not related to each other and gaining more predicting power.

A good dataset to apply this method, in R, could be the **golub** so as the **breast data Winsconsin**. It would be better to apply it first to the latter, as it is simpler. The first one is left for practice.

As before, the first step would be to split the initial dataset to train set and test set and apply the z-score scaling:

```
cc_data <- read.csv(file = 'data.csv')
# Eliminating the first and last columns of dataset…
# (No useful info is included)
cc_data$id<-NULL
cc_data$X<-NULL
cc_data$diagnosis<-factor(cc_data$diagnosis)
library(caTools)
set.seed(1234)
split <- sample.split(cc_data$diagnosis, SplitRatio = 0.75)
train.data<- subset(cc_data, split == TRUE)
test.data <- subset(cc_data, split == FALSE)
```

It is better to strip out the info about the diagnosis from train set and test set before the pca into two vectors:

```
#Retract the info of the diagnosis results
diagnosis_train<-train.data$diagnosis
diagnosis_test<-test.data$diagnosis
# Strip the datasets from diagnosis column
train.data$diagnosis<-NULL
test.data$diagnosis<-NULL
```

Here we need not to apply scaling, as it is done by the prcomp() function, that applies PCA, by default (see help (?prcomp)):

```
# Pca cc_data
cc_data_pca <- prcomp(train.data, center = TRUE, scale. = TRUE)
```

In order to see and vizualize the eigenvalues and their contribution to the variance the package factoextra needs to be installed. Then the function used is get_eig(), or its alias get_eigenvalue()

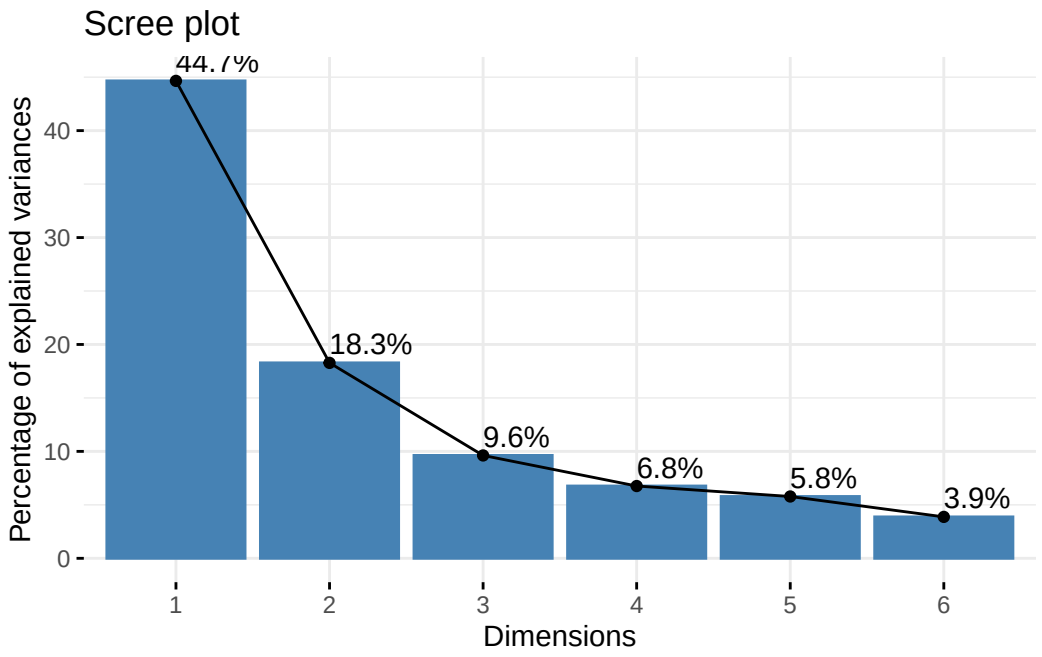
```
#install.packages('factoextra')
library(factoextra)
eig.val<-get_eigenvalue(cc_data_pca)
head(eig.val)
```

```
      eigenvalue variance.percent cumulative.variance.percent
Dim.1  13.396045        44.653485                    44.65348
Dim.2   5.483561        18.278536                    62.93202
Dim.3   2.886459         9.621531                    72.55355
Dim.4   2.028141         6.760471                    79.31402
```

```
Dim.5    1.734329          5.781095                    85.09512
Dim.6    1.162356          3.874520                    88.96964
```

The first six eigenvalues are containing almost 89% of the variability. A rule of thumb is that around 80% of variability is generally enough. So in our case is more that enough. It can be visualized also using the fviz_eig() function of factoextra:

```
numberofterms<-6


# 0
fviz_eig(cc_data_pca, addlabels = TRUE, ncp=numberofterms)
```




Deciding this, it is time to apply it for n=6:

```
cc_data_pca <- prcomp(train.data,rank. = 6)
```

The new datasets would be:

```r
training_set <- stats::predict(cc_data_pca, train.data)


test_set <- stats::predict(cc_data_pca, test.data)
```

using the predict function from the base stats package.

> ⚠ Keeping in mind...
>
> Pay attention to one detail. The new test set is predicted by applying the same transformation with that of train.data. Why is this? Why not to use the prcomp() function in order to produce a test_set with the first six bigger contributions of the variability for that dataset? Because in this way the unknown factor of the test_set is violated. The test_set is supposed to be unknown, in order to make a prediction. By applying pca to the test_set is like knowing the numbers of the unknown new data beforehand. It is like taking compromising the result.

Now it is time for the prediction:

```r
library(e1071)


set.seed(123)
# Bind together the diagnosis_train with the training_set
training_set<-cbind(training_set,diagnosis_train)


str(training_set)
```

```
 num [1:427, 1:7] -1156 -1265 -992 411 215 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:427] "1" "2" "3" "4" ...
  ..$ : chr [1:7] "PC1" "PC2" "PC3" "PC4" ...
```

```r
classifier <- svm(formula = diagnosis_train ~ .,
                  data = training_set,
                  type = "C-classification",
                  kernel = "linear")



y_pred <- stats::predict(classifier, newdata = test_set)



cm <- table(diagnosis_test,y_pred)
cm
```

```
              y_pred
diagnosis_test  1  2
             B 87  2
             M  4 49
```

A very good accuracy is achieved, quite close to the initial one, by using 6/30=20% of the data of the old dataset. Quite remarkable.

## 8.2 t-Distributed Stochastic Neighbor Embedding (t-SNE)

On contrary to PCA, which is a linear dimension-reduction algorithm, t-SNE is a non linear. t-SNE is a nonlinear technique that focuses on preserving the pairwise similarities between data points in a lower-dimensional space. It is concerned with preserving small pairwise distances whereas, PCA focuses on maintaining large pairwise distances to maximize variance.

> **t-SNE explanation**
>
> Stochastic → Not definite but random probability
>
> Neighbor → Concerned only about retaining the variance of neighbor points
>
> Embedding → Plotting data into lower dimensions

The whole idea is based on the following:

**Step 1:** t-SNE constructs a probability distribution on pairs in higher dimensions such that similar objects are assigned a higher probability and dissimilar objects are assigned lower probability.

**Step 2:** Then, t-SNE replicates the same probability distribution on lower dimensions iteratively till the Kullback-Leibler divergence is minimized. (In simple words it moves the randomly projected points around step by step, until the similarities between points in the low-dimensional dataset resemble the similarities between cells in the original dataset.)

*Kullback-Leibler divergence* is a measure of the difference between the probability distributions from Step1 and Step2. KL divergence is mathematically given as the expected value of the logarithm of the difference of these probability distributions.

Let's apply it in R. We are going to use a library called penguins from the palmerpenguins package.

```
#install.packages('palmerpenguins')
library(palmerpenguins)
penguins
```

```
# A tibble: 344 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
```

```
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

We are going to apply t-SNE to the four numeric columns of the dataset bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g in order to end up to 2dimensional representation of the results which can be seen in a graph. Remember that t-SNE purpose is to "visualize high-dimensional data by giving each datapoint a location in a two or three-dimensional map" as was introduced by van der Maaten and Hinton (Maaten and Hinton 2008). We are going to proceed using tidyverse approach as penguins is a tibble and in order to see a different most modern approach in R programming.

As a first step we clean the dataset and remove column that we will have no use of them. We also add an index column.

```
#install.packages('tidyverse')
library(tidyverse)
penguins <- penguins %>%
  drop_na() %>%
  select(-year) %>%
  mutate(ID=row_number())
```

As a next step we apply t-SNE to the 4 numeric columns mentioned above by using the package Rtsne. As the columns that we have are in a different order of magnitude we normalize them using the scale function.

```
set.seed(123)
#install.packages('Rtsne')
library(Rtsne)
penguins_t_SNE <-penguins %>%
  select(where(is.numeric)) %>%
  scale() %>%
  Rtsne(pca=FALSE,perplexity=30,theta=0.0) # Run TSNE)
```

We take the resulting two dimensional dataset and make a tibble in order to plot it.

```
penguins_result<-as_tibble(penguins_t_SNE$Y) %>%
  mutate(ID=row_number())
```

On the same time we draw all the necessary columns from the initial penguins dataset:

```
penguins_metadata <-penguins %>%
  select(c(1,2,7,8))
```

And finally we merge these two datasets by inner joining them in the ID column:

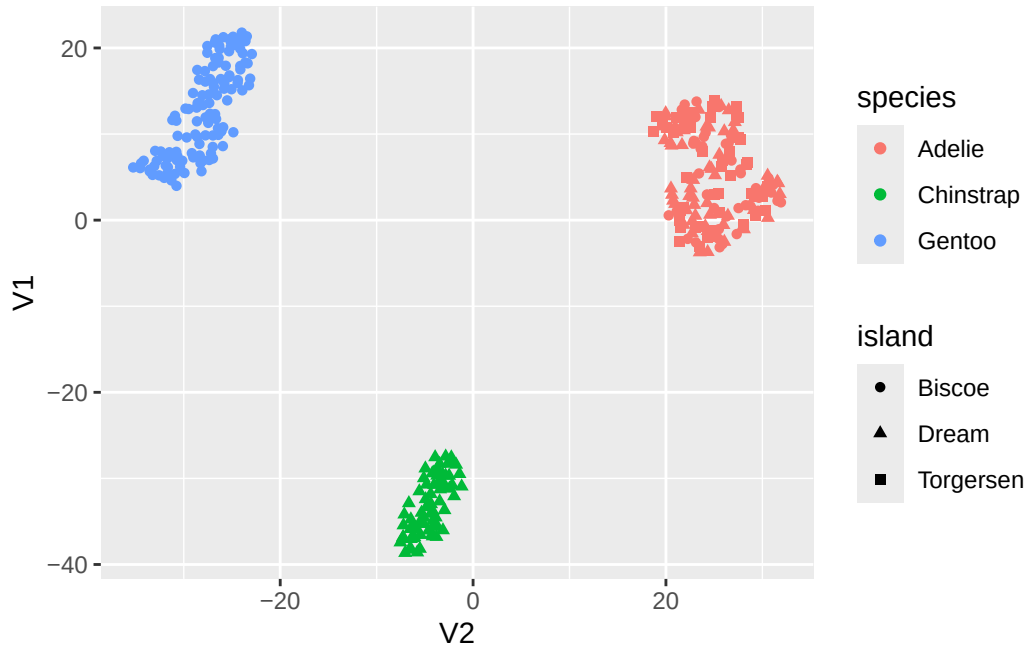
```
penguins_merge<-penguins_result %>%
  inner_join(penguins_metadata, by="ID")
penguins_merge
```

```
# A tibble: 333 x 6
       V1    V2    ID species island    sex
```

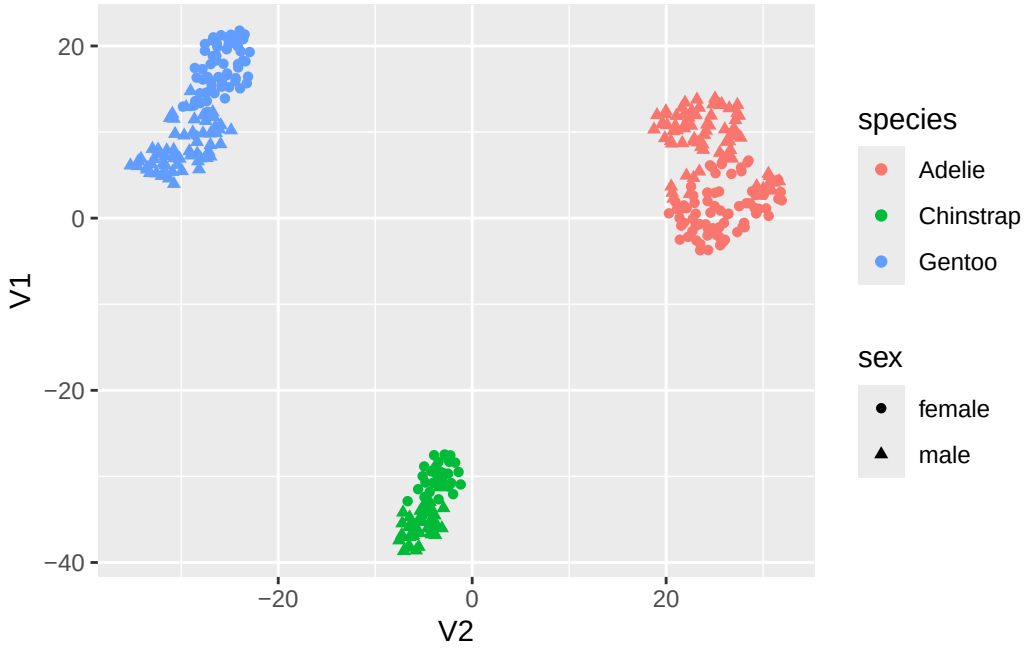
```
    <dbl> <dbl> <int> <fct>   <fct>     <fct>
 1  3.91   30.5     1 Adelie  Torgersen male
 2  2.70   29.2     2 Adelie  Torgersen female
 3  0.532  29.3     3 Adelie  Torgersen female
 4  6.48   28.4     4 Adelie  Torgersen female
 5 11.9    27.5     5 Adelie  Torgersen male
 6  2.71   30.1     6 Adelie  Torgersen female
 7 11.9    24.6     7 Adelie  Torgersen male
 8  1.13   30.3     8 Adelie  Torgersen female
 9 12.2    27.3     9 Adelie  Torgersen male
10 13.2    27.3    10 Adelie  Torgersen male
# i 323 more rows
```

The new dataset is reduced to two-dimentional one V2,V1 and can be plotted in regard of any info that we need (island or sex). Let's try it out using ggplot.

```
penguins_merge %>%
  ggplot(aes(x = V2,
             y = V1,
             color = species,
             shape = island))+
  geom_point()+
  theme(legend.position="right")
```




```
penguins_merge %>%
  ggplot(aes(x = V2,
             y = V1,
             color = species,
             shape = sex))+
  geom_point()+
  theme(legend.position="right")
```




It is obvious that the information that we have is clustered in 3 groups in terms of the species. We can apply clustering on the final dataset using the methods that are given in the Unsupervised machine learning chapter. This is left as a future exercise.

# 9 Finding the best model for your data

Till now a lot of algorithms applied in different datasets with its advantages and disadvantages that noted down. The question that follows logically is what is the best model for my data in order to make a good prediction. There are some extra things that one needs to consider in order to find a good model.

## 9.1 Over-fitting versus under-fitting

It is a common pitfall when applying machine learning algorithms, the program which creates a model to provide a very good prediction for the train set, while the prediction is quite bad for the test set. What the program actually does is to try to memorize the data patterns and the noise of that dataset as good as possible so when the moment of truth comes with a new test set, everything fails. This is what one calls **over-fitting**. One the other hand when we are using a very simplified model to predict our data, the model is unable to capture the relationship between the input and output variables accurately, generating a high error rate on both the training set and unseen data. This is what we call under-fitting. The good fit lies in finding the appropriate model that can predict accurately all the new data that will appear in the future. The good model should pick up the patterns from the training data but not end up memorizing the finer details. This would ensure that the model generalizes and accurately predicts other data samples.

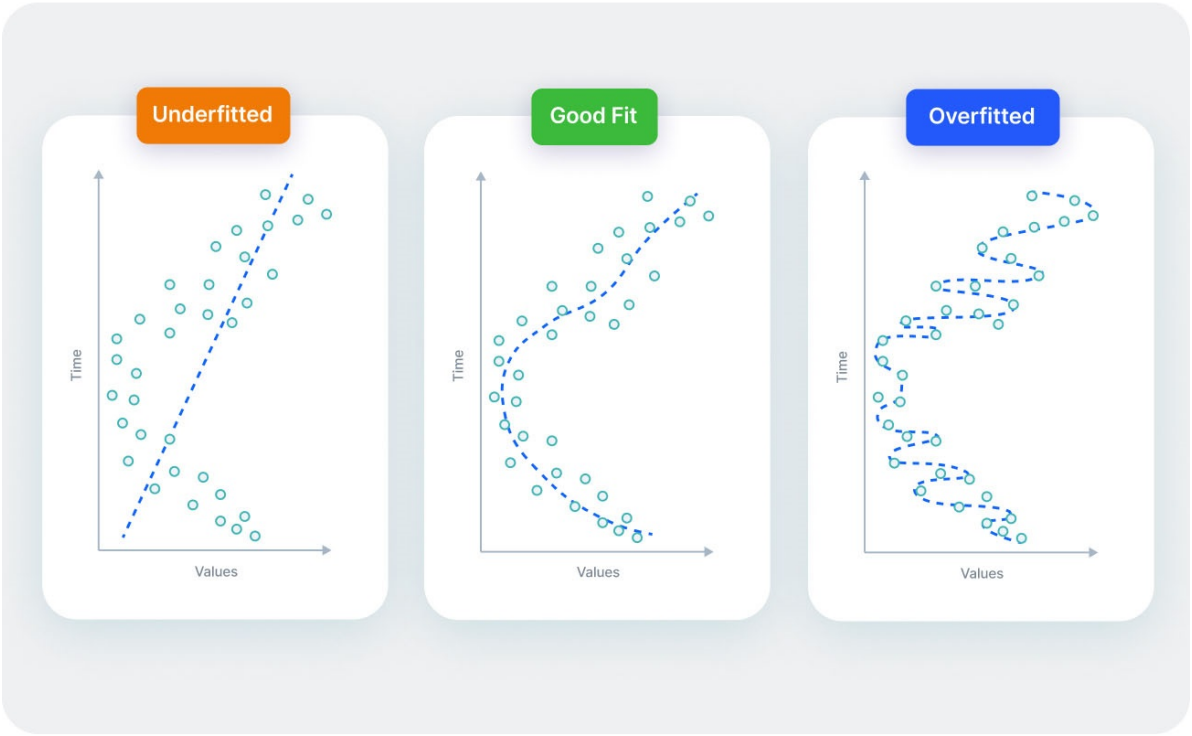



Figure 9.1: The three cases of fit

## 9.2 The bias variance tradeoff

Let's first begin by the definitions of each one of them.

**Bias:** Is the difference between the expected (or average) prediction of the model and the correct value which we are trying to predict. Of course you only have one model, so talking about expected or average prediction values might seem a little strange. However, imagine you could repeat the whole model building process more than once: each time you gather new data and run a new analysis creating a new model. Due to randomness in the underlying data sets, the resulting models will have a range of predictions. Bias measures how far off in general these models' predictions are from the correct value.

**Variance** is the variability of a model prediction for a given data point. Again, imagine you can repeat the entire model building process multiple times. The variance is how much the predictions for a given point vary between different realizations of the model.

To have a conseptual idea bias is reduced and variance is increased in relation to model complexity. As more and more parameters are added to a model, the complexity of the model rises and variance becomes our primary concern while bias steadily falls.

> **!** Important
>
> Complex models (**low-bias high-variance**) are prone to overfitting while simple ones to underfitting (**high-bias low-variance**).

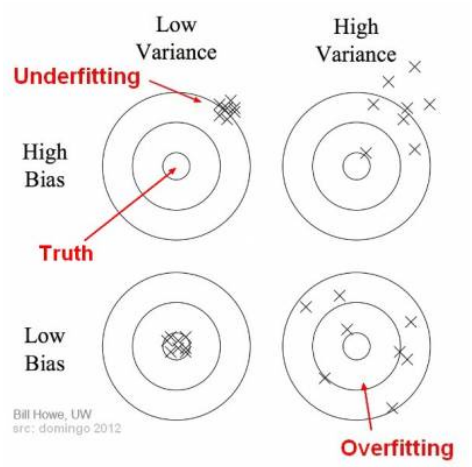



Figure 9.2: Bias Variance Trade-Off

Examples of **low-bias high-variance** machine learning algorithms include: Decision Trees, k-Nearest Neighbors and Support Vector Machines.

Examples of **high-bias low-variance** machine learning algorithms include: Linear Regression, and Logistic Regression.

Tip

- The k-nearest neighbors algorithm has low bias and high variance, but the trade-off can be changed by increasing the value of k which increases the number of neighbors that contribute to the prediction and in turn increases the bias of the model.
- The support vector machine algorithm has low bias and high variance, but the trade-off can be changed by increasing the C parameter that influences the number of violations of the margin allowed in the training data which increases the bias but decreases the variance.

## 9.3 Cross validation

Cross-validation is a technique for validating the model efficiency, by training a model to a subset of input (train) data and validating (testing) it on the unseen data left. This is something different from the general train-test split.

We can divide the methods into two subcategories. The **exhaustive** and **not exhaustive** methods. Here the most common methods are mentioned, among many.

Exhaustive. The idea involves testing the model in all possible ways, it involves splitting the data in all possible ways. The two most common examples are:

1. **Leave one out cross validation**: for each learning set, only one datapoint is reserved, and the remaining dataset is used to train the model. This process repeats for each datapoint. Hence for n samples, we have n different training sets and n test sets.

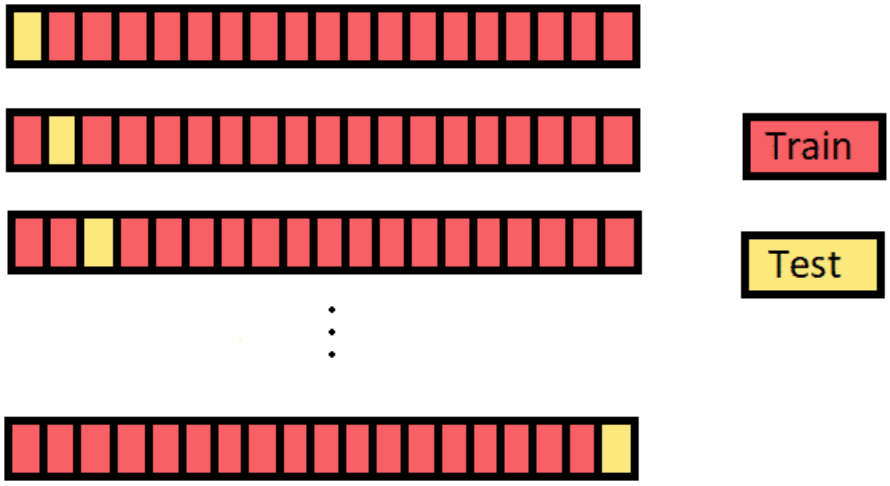



Figure 9.3: Leave One Out Cross Validation (LOOCV)

2. **Leave p-out cross validation**: if there are total n datapoints in the original input dataset, then n-p data points will be used as the training dataset and the p data points as the validation set. The p datapoints are selected randomly from the dataset. This complete process is repeated for all the samples, and the average error is calculated to know the effectiveness of the model.

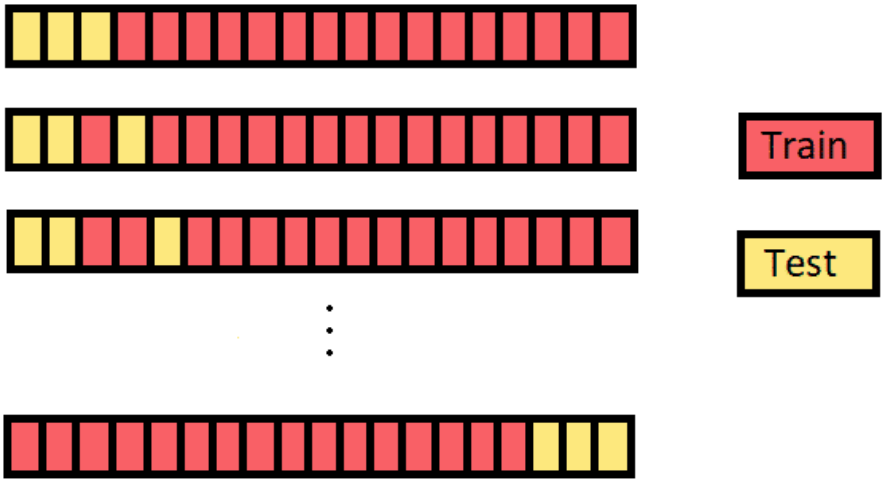



Figure 9.4: Leave p Out Cross Validation

**Non-Exhaustive**: In this method, the original data set is not separated into all the possible permutations and combinations.

1. **The single validation** (hold-out test): if there are total n datapoints in the original input dataset, then n-k data points will be used as the training dataset and the k data points as the validation set. This performance metric from test is reported as estimate of the performance of our model in new data.

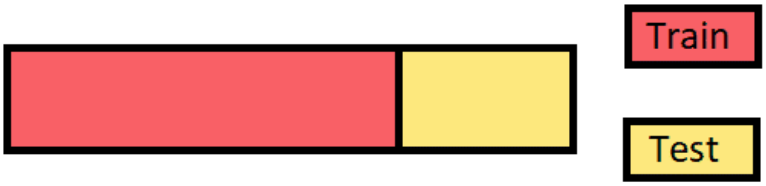



Figure 9.5: Holdout

2. **k-fold cross validation**: divide the input dataset into K groups of samples of equal sizes. These samples are called **folds**. For each learning set, the prediction function uses k-1 folds, and the rest of the folds are used for the test set. The estimate for the performance of the model is the mean of the accuracies of each fold.

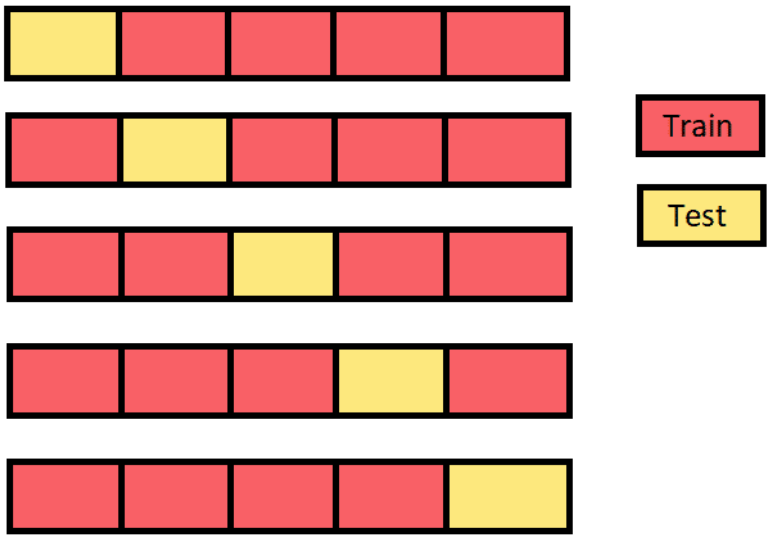



Figure 9.6: k-fold cross validation

## 9.4 The Receiver Operating Characteristic (ROC) curve

So far, in all examples, the classification threshold of equal probabilities is applied. Meaning, that in a question of 2 possible answers for example, if a model predicted that there is a 55% probability that case 1 is the answer to our research question and 45% case 2, the model would choose the first as the final answer. The question that rises is what will happen if these probabilities where not equal. Meaning, if the cutoff threshold for an 1 or 2 result was not 50% but in favor of one of the results and against the other. Will the results would be better in our confusion matrix or worse? A receiver operating characteristic curve, or ROC curve,

is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.

It is created by plotting the true positive rate against the false positive rate. The true positive rate is the sensitivity that was discussed in a previous chapter.The number of correct positives divided by the number of all positives. The false positive rate is defined as follows:

$$FPR = \frac{FP}{(TN + FP)} = 1 - Specificity$$

The points comprising ROC curves indicate the true positive rate vs the false positive at varying thresholds. An interesting illustration is the following, from wikipedia:

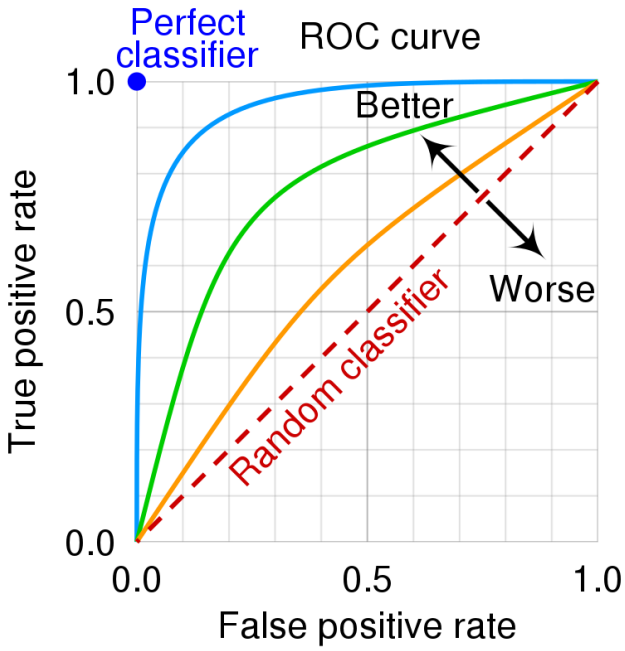



Figure 9.7: Varied classifiers

The better the classifier is, the more area under the curve would be covered. The so called AUC statistic. AUC ranges from 0.5 (Random Classifier) to 1 (Perfect classifier).

# 10 The caret package

The caret package (Kuhn 2008), the initials of which stand for **Classification And REgression Training**, is a complete alternative to all the functions that previously used for classification and regression plus the dimension reduction and cross validation. It includes all the algorithms that have been applied so far and much more.

Everything about caret, all the functions and info could be found in this webpage:

https://topepo.github.io/caret/

All the necessary classification and regression algorithms could be found here:

https://topepo.github.io/caret/available-models.html

An application of the above in the case of the golub dataset is as follows:

## 10.1 Preprocessing the dataset

```
GOLUB<-readRDS(file = "golub")
#####################################################
golub <-as.data.frame(GOLUB)[1:7134]


golub <-golub[1:7130]
```

```
library(caret)
set.seed(1234)
#First step, paritioning the data:
trainIndex <- createDataPartition(golub$class, p = .80,
                                  list = FALSE,
                                  times = 1)
```

**createDataPartition**() function of caret creates the partitions in order of p. **p** will provide the percentage of the training partition and **times** the number of partitions to create. By default everything goes to a list. Here we do not need one, a vector would be perfectly ok to define the training and test set. So **list=FALSE.** The function comes with a lot of arguments. One can check them in help (?createDataPartition).

Next step, the training set and the test set:

```
train.data <- golub[ trainIndex,]
test.data  <- golub[-trainIndex,]
```

To preprocess the data (scale the data using the standarization procedure) the function preProcess() is used which applies that by default:

```
# Data are scaled by default by preProcess() function using
# the standarization method mentioned above:
preProcValues<-preProcess(train.data)

#It is equivalent with the following:
#preProcValues <- preProcess(train.data, method = c("center", "scale"))

# The scaled data:
```

```
train.data_scaled <- predict(preProcValues, train.data)
test.data_scaled <- predict(preProcValues, test.data)
```

In case of a PCA transformation the preProcess is used too. Here for keeping the first 44 of the Principal Components:

```
golub_pca = preProcess(x = train.data, method = "pca", pcaComp = 44)

training_set_pca <- predict(golub_pca, train.data)
test_set_pca <- predict(golub_pca, test.data)
```

> **Caution**
>
> Keep in mind that by using the pca method here, a standarization is also applied.

Next step would be the cross-validation.

## 10.2 Training a model and applying cross validation using Caret

In order to train a supervised model for the golub dataset (classification case) and apply cross validation, the following 2 steps are needed:

1. Define the cv method: (Here the k-fold cross validation will be used)

```
trControl <- trainControl(method  = "cv",
                          number  = 5,
                          savePredictions = TRUE)
```

Using the trainControl() function the method could be defined: (e.g "cv" for k-fold **cross-validation**, "repeatedcv"for **repeated cross validation** (an extension with repeats of cross

validated datasets) , “LOOCV”, leave one out cross validation “LGOCV” leave group out cross validation).

Here the k-fold cross validation is applied using 5 folds. It is provided to the function with the argument **number.** For more info about the function and how to use the other arguments, one could check the help of the function or the manual of caret.

Finally the model is produced using the train function of caret:

```
knnfit <- caret::train(class ~ .,
                       method     = "knn",
                           tuneGrid   = expand.grid(k = 2:10), # Grid search
                           trControl  = trControl,
                       metric     = "Accuracy",
                       data       = training_set_pca)
```

1. The first argument is the nominal data column class (ALL, or AML). As in previous functions We place ~. In case would like to include all the columns in the train of the model. The function automatically detects if it is nominal (classification) or arithmetic (regression). Here class is nominal so it goes for classification.

2. The **method** argument defines the algorithm as one can see it from:

https://topepo.github.io/caret/available-models.html

3. The **trControl** calculates all the cross validation cases with the “Accuracy” as metric. The Accuracy is used in order to find the best in terms of accuracy model. (In Regression the ‘Rsquared’ could be used).

4. And finally the argument **data** is our training set.

   One argument is left. The **tuneGrid** argument. As mentioned before the knn algorithm has low bias and high variance, but the trade-off can be changed by increasing the value of k which increases the number of neighbors that contribute to the prediction and in

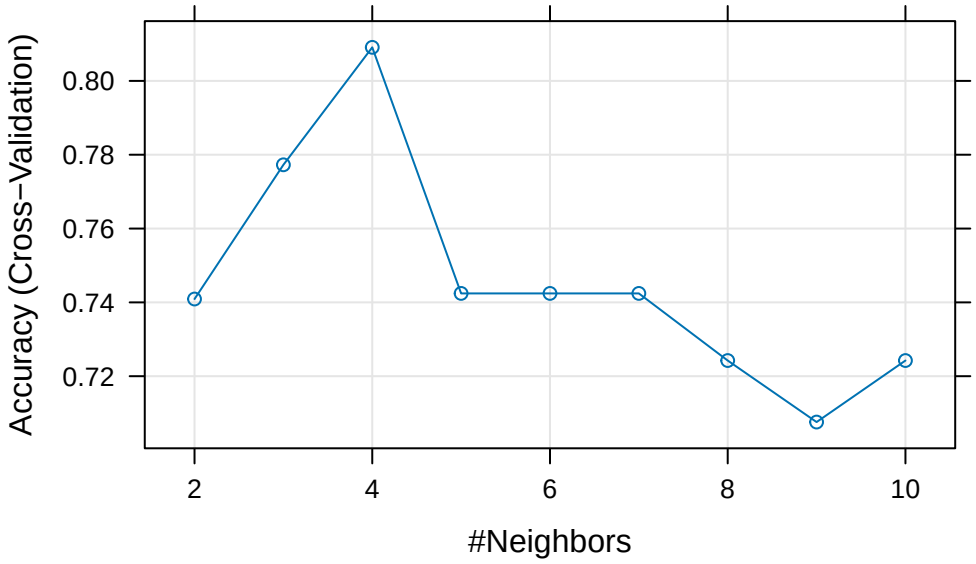
turn increases the bias of the mode. It is time to broaden the results in terms of 10 different k cases and see which k is most appropriate for the best accuracy in prediction:

```
#Checking the model
knnfit
```

```
k-Nearest Neighbors

58 samples
44 predictors
 2 classes: 'ALL', 'AML'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 46, 46, 46, 47, 47
Resampling results across tuning parameters:

  k   Accuracy   Kappa
   2  0.7409091  0.3323254
   3  0.7772727  0.4086154
   4  0.8090909  0.5232959
   5  0.7424242  0.2946514
   6  0.7424242  0.2946514
   7  0.7424242  0.2946514
   8  0.7242424  0.2350769
   9  0.7075758  0.1735385
  10  0.7242424  0.2350769

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 4.
```

The best value is for k=4, in respect of accuracy, which is automatically kept as the best model.

```
# plotting the accuracy against k, the number of terms
plot(knnfit)
```




The best result in terms of accuracy is visually obtained by the plot above.

As the predictions are saved, a look can be taken of the observables-predictions in terms of the folds that created for those models.

```
head(knnfit$pred)
```

```
  pred obs rowIndex k Resample
1  ALL ALL        7 2    Fold1
2  ALL ALL       11 2    Fold1
3  ALL ALL       19 2    Fold1
```

```
4   ALL AML       25 2    Fold1
5   ALL AML       26 2    Fold1
6   AML AML       27 2    Fold1
```

Finally it is time to apply the model to the test_set for the final prediction:

```
#####################prediction knn#################################

treepred = predict(knnfit, newdata=test_set_pca)

class<-as.factor(unlist(test.data[7130]))

table(class,treepred)
```

```
     treepred
class ALL AML
  ALL   9   0
  AML   2   3
```

## 10.3 Calculating ROC curves

There are a lot of packages the reader could use in order to visualize a ROC curve and calculate the corresponding auc. In our case a package called MLeval is going to be used. Applying a small reforming of the previous caret objects for the svm and knn algorithms the ROC curves could appear quite effortless.

```
trControl <- trainControl(method  = "cv",
                          number  = 5,
                          summaryFunction = twoClassSummary,
```

```
                           classProbs = T,
                           savePredictions = TRUE)

knnfit <- caret::train(class ~ .,
                       method     = "knn",
                       trControl  = trControl,
                       metric     = "ROC",
                       data       = training_set_pca)

knnfit
```

```
k-Nearest Neighbors

58 samples
44 predictors
 2 classes: 'ALL', 'AML'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 47, 46, 46, 46, 47
Resampling results across tuning parameters:

  k  ROC        Sens  Spec
  5  0.9294643  1     0.40
  7  0.8928571  1     0.35
  9  0.8915179  1     0.20

ROC was used to select the optimal model using the largest value.
The final value used for the model was k = 5.
```

```r
svmfit <- caret::train(class ~ .,
                       method     = "svmLinear",
                       trControl  = trControl,
                       metric     = "ROC",
                       data       = training_set_pca)

svmfit
```

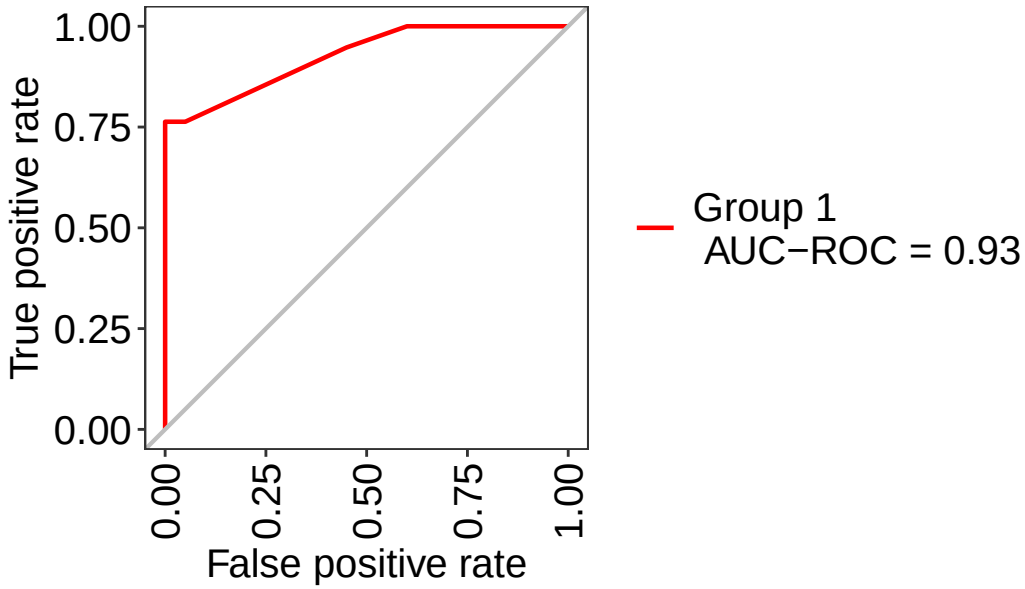
```
Support Vector Machines with Linear Kernel

58 samples
44 predictors
 2 classes: 'ALL', 'AML'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 47, 47, 46, 46, 46
Resampling results:

  ROC        Sens       Spec
  0.8330357  0.8678571  0.45

Tuning parameter 'C' was held constant at a value of 1
```

Applying for the first object (knn), with ROC as a metric, the roc curve and auc will be constructed using the MLeval package:

```r
library(MLeval)


## evaluate knn ROC
# We define ALL as positive as our initial prerequecity
knnROC <- evalm(knnfit,positive='ALL',plots='r',rlinethick=0.8,fsize=15)
```

```
***MLeval: Machine Learning Model Evaluation***

Input: caret train function object

Not averaging probs.

Group 1 type: cv

Observations: 58

Number of groups: 1

Observations per group: 58

Positive: ALL

Negative: AML

Group: Group 1

Positive: 38

Negative: 20
```

```
***Performance Metrics***

Group 1 Optimal Informedness = 0.763157894736842

Group 1 AUC-ROC = 0.93
```

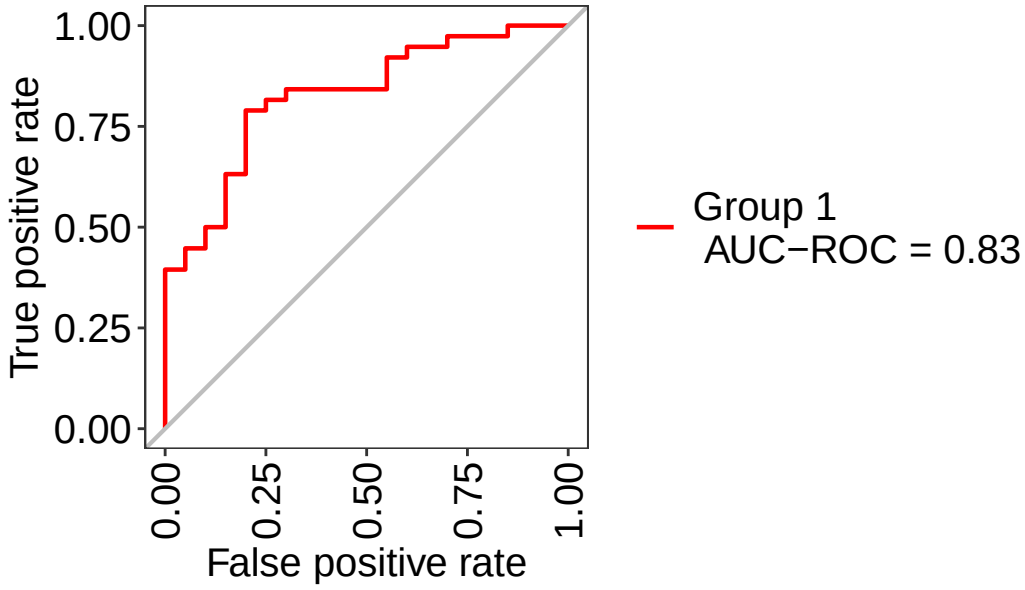



Applying for the second object (svm):

```
svmROC <- evalm(svmfit,positive='ALL',plots='r',rlinethick=0.8,fsize=15)
```

```
***MLeval: Machine Learning Model Evaluation***

Input: caret train function object

Not averaging probs.

Group 1 type: cv
```

```
Observations: 58

Number of groups: 1

Observations per group: 58

Positive: ALL

Negative: AML

Group: Group 1

Positive: 38

Negative: 20

***Performance Metrics***

Group 1 Optimal Informedness = 0.589473684210526

Group 1 AUC-ROC = 0.83
```



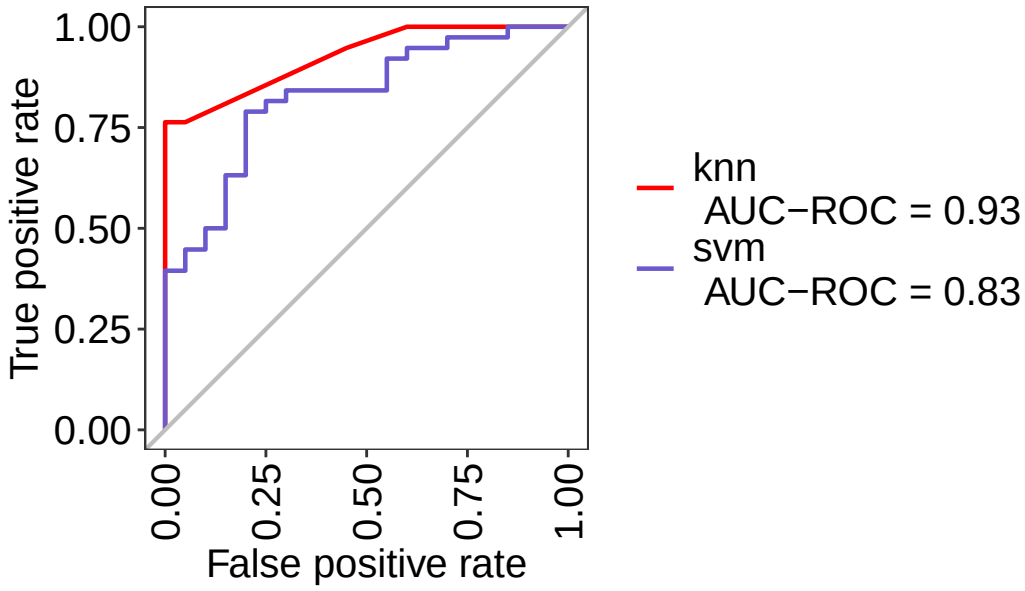

The two graphs could be plotted side by side, in order to graphically decide the best one:

```
# plot rocs
compare_rocs <- evalm(list(knnfit,svmfit),positive = 'ALL',
                      gnames=c('knn','svm'),rlinethick=0.8,
                      fsize=15,plots='r')
```

```
***MLeval: Machine Learning Model Evaluation***

Input: caret train function object

Not averaging probs.

Group 1 type: cv

Group 2 type: cv
```

```
Observations: 116

Number of groups: 2

Observations per group: 58

Positive: ALL

Negative: AML

Group: knn

Positive: 38

Negative: 20

Group: svm

Positive: 38

Negative: 20

***Performance Metrics***

knn Optimal Informedness = 0.763157894736842

svm Optimal Informedness = 0.589473684210526

knn AUC-ROC = 0.93
```

```
svm AUC-ROC = 0.83
```



# 11 Feature selection

In the chapters before, we tried to provide the reader with solutions on:

1. preparing a dataset (imputation, feature scaling) for further data analysis
2. different types of algorithms, applied in different types of problems, in order to find an appropriate way to make accurate predictions
3. minimize the computing time and preserve the accuracy (dimensional reduction)

Now it is time to introduce different techniques in order to filter-transform our dataset so as to reveal the important features which will uncover a model that has the best predictive ability and on the same time is interpretative enough. This is actually the definition of the **feature selection** in machine learning.

There are a lot of different ways to achieve such a goal. Some of them are appropriate for regression others are more appropriate for classification.

## 11.1 Models with Build in Feature Selection

There are models that automatically perform feature selection as part of their training. Common examples of those are linear models with some form of regularization (e.g. lasso, glmnet) and most tree-based models. In the caret package there is a list of all that models:

**ada, AdaBag, AdaBoost.M1, adaboost, bagEarth, bagEarthGCV, bagFDA, bagFDAGCV, bartMachine, blasso, BstLm, bstSm, C5.0, C5.0Cost, C5.0Rules, C5.0Tree, cforest, chaid, ctree, ctree2, cubist, deepboost, earth, enet, evtree, extraTrees, fda, gamboost, gbm_h2o, gbm, gcvEarth, glmnet_h2o, glmnet, glmStepAIC, J48, JRip, lars, lars2, lasso, LMT, LogitBoost, M5, M5Rules, msaenet, nodeHarvest, OneR, ordinalNet, ordinalRF, ORFlog, ORFpls, ORFridge, ORFsvm, pam, parRF, PART, penalized, PenalizedLDA, qrf, ranger, Rborist, relaxo, rf, rFerns, rfRules, rotationForest, rotationForestCp, rpart, rpart1SE, rpart2, rpartCost, rpartScore, rqlasso, rqnc, RRF, RRFglobal, sdwd, smda, sparseLDA, spikeslab, wsrf, xgbDART, xgbLinear, xgbTree**

We are going to apply one of them (rpart) in our PimaIndians dataset.

```
# Train an rpart model and compute variable importance.
set.seed(100)
library(caret)
library(mlbench)
data(PimaIndiansDiabetes)
pm <- PimaIndiansDiabetes
str(pm)
```

```
'data.frame':   768 obs. of  9 variables:
 $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
 $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
 $ pressure: num  72 66 64 66 40 74 50 0 70 96 ...
 $ triceps : num  35 29 0 23 35 0 32 0 45 0 ...
 $ insulin : num  0 0 0 94 168 0 88 0 543 0 ...
 $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 0 ...
```

```
 $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
 $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
 $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

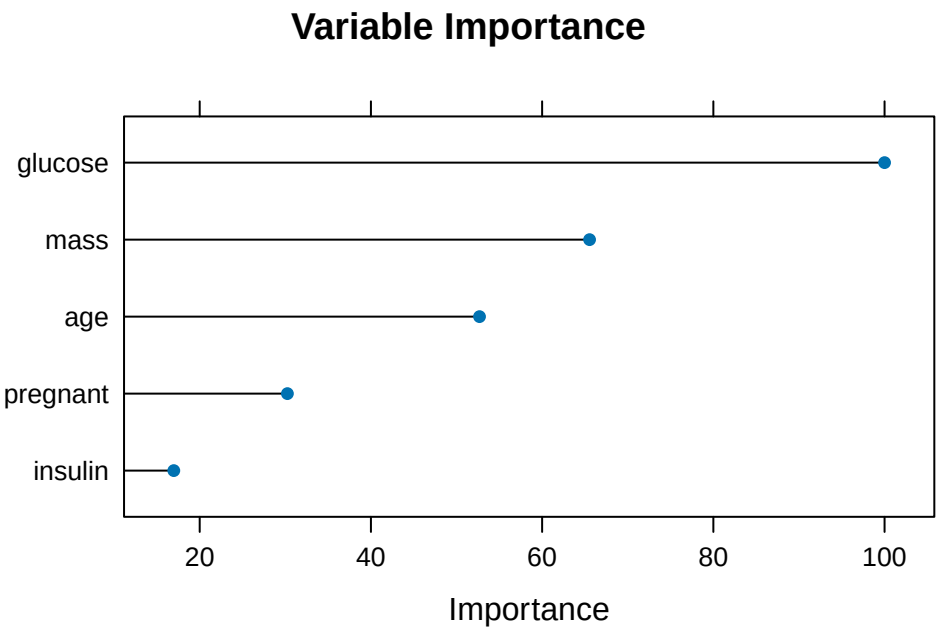
```
rpartMod <- train(diabetes ~ ., data=pm, method="rpart")
# Most important features list
rpartImp <- varImp(rpartMod)
rpartImp
```

```
rpart variable importance

         Overall
glucose  100.000
mass      65.542
age       52.685
pregnant  30.245
insulin   16.973
pedigree   7.522
pressure   0.000
triceps    0.000
```

```
# The performance of the model
getTrainPerf(rpartMod)
```

```
  TrainAccuracy TrainKappa method
1     0.7455293  0.4272746  rpart
```

```r
#Plotting the top 5 most important variables.
plot(rpartImp, top = 5, main='Variable Importance')
```




## 11.2 Recursive Feature Elimination (RFE)

RFE is the most used feature selection method, based on the idea of iteratively removing the least predictive feature from a model until the desired number of features is reached. This feature is determined by the built-in feature importance method of the model. Such models could be linear regression, random forests and many more. Caret has a number of pre-defined sets of functions for several models, here in our example we are going to use random forests (`rfFuncs`). The are also lmFuncs for linear regression and others (Check the caret manual). We are going to use the trControl function here too and apply **rfe()**.

```r
library(caret)
set.seed(123)

# Applying cross validation with 10 folds
ctl <- rfeControl(functions=rfFuncs, method="cv", number=10)

# run the RFE algorithm taking into account all the predictors
results <- rfe(pm[,1:8], pm[,9], sizes=c(1:8), rfeControl=ctl)

# summarize the results
print(results)
```

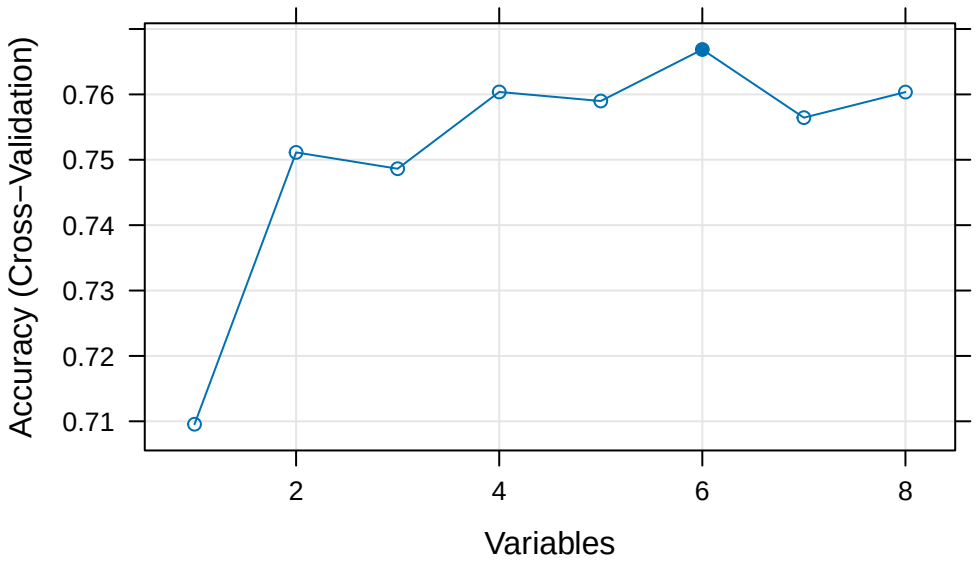
```
Recursive feature selection

Outer resampling method: Cross-Validated (10 fold)

Resampling performance over subset size:

 Variables Accuracy  Kappa AccuracySD KappaSD Selected
         1   0.7096 0.2980    0.04132 0.10848
         2   0.7511 0.4326    0.04230 0.09345
         3   0.7486 0.4391    0.03732 0.07792
         4   0.7604 0.4656    0.02925 0.06917
         5   0.7590 0.4581    0.03692 0.09242
         6   0.7669 0.4699    0.03543 0.08562        *
         7   0.7564 0.4481    0.02678 0.06880
         8   0.7604 0.4542    0.02677 0.07117
```

```
The top 5 variables (out of 6):
   glucose, mass, age, pregnant, insulin
```

```
# Output of the total predictors
predictors(results)
```

```
[1] "glucose"  "mass"     "age"      "pregnant" "insulin"  "pedigree"
```

```
# plotting the results
plot(results, type=c("g", "o"))
```



# 12 Imbalanced data classification

It is typical in life, to come across with data, especially in the case of binary classification problems (two classes of results in the dependent variable), that are imbalanced. Meaning that there is a disparity in the frequencies of the observed classes. Some examples in biological sciences and medicine are:

- Studying Rare genetics mutations. The incidents of such are quite uncommon and their study occurs in a dataset which the vast majority is on common genetic identity.
- Rare species characteristics and properties. The majority of the species belong to the same group. The few examples are quite uncommon contrary to the other group that contains a lot of instances.
- Datasets that include results from people which have a disease caused from rare infections. They are highly imbalanced in favor of the healthy people.

  and lot more.

The problem with such data is that it is quite difficult to train a model that fits well to the new, testing data. The reason is that the model being trained would be dominated by the majority class. So in simple words it would predict this class more effectively than the minority one.

To solve this inequality data problem there are some methods that the researcher could use in order to balance the data. These are the following:

1. **Down-sampling or under-sampling**. Randomly select a subset of appearances from the majority class to match the number of the minority. In such a way we have a balanced dataset for a our machine learning algorithm by sacrificing though a lot of data.
2. **Up-sampling or oversampling.** Randomly sample (with replacement) the minority class to be the same size as the majority class.
3. **Hybrid methods.** Two methods SMOTE and ROSE that create synthetic data in the minority class and downsampling the majority class.
4. **Cost sensitive learning.** It does not create balanced data distribution. Instead, it highlights the imbalanced learning problem by using cost matrices which describes the cost for misclassification in a particular scenario.

> Caution
>
> **Note**: All the above methods should be applied only on the training set , the testing set must be never touched until the final model evaluation step.

We are going to apply all these methods to a new imbalanced dataset called ildb (Indian Liver Patient Dataset).

## 12.1 Applying the methods to an imbalanced data set

### 12.1.1 The Indian Liver Patient data set

It comes directly from UC Irvine Machine Learning Repository (Ramana and Venkateswarlu 2012) and it consists of the following columns:

1. Age (Age of the patient. Any patient whose age exceeded 89 is listed as being of age "90")
2. Gender

3. TB (Total Bilirubin)
4. DB (Direct Bilirubin)
5. Alkphos (Alkaline Phosphotase)
6. Sgpt (Alamine Aminotransferase)
7. Sgot (Aspartate Aminotransferase)
8. Tp (Total Proteins)
9. ALB (Albumin)
10. A/G Ratio (Albumin and Glubolin Ratio)
11. Selector (Selector field used to split the data into two sets (labeled by the experts))

This data set contains records of 416 patients diagnosed with liver disease and 167 patients without liver disease. This information is contained in the class label named 'Selector'. There are 10 variables per patient: age, gender, total Bilirubin, direct Bilirubin, total proteins, albumin, A/G ratio, SGPT, SGOT and Alkphos. Of the 583 patient records, 441 are male, and 142 are female.

```
ilpd <- read.csv(
  paste("https://archive.ics.uci.edu/ml/machine-learning-databases/00225/",
  "Indian%20Liver%20Patient%20Dataset%20(ILPD).csv",sep=""),header = FALSE)

colnames(ilpd) <- c("Age", "Sex", "Tot_Bil", "Dir_Bil", "Alkphos", "Alamine",
     "Aspartate", "Tot_Prot", "Albumin", "A_G_Ratio", "Disease")

str(ilpd)
```

```
'data.frame':   583 obs. of  11 variables:
 $ Age       : int  65 62 62 58 72 46 26 29 17 55 ...
```

```
$ Sex      : chr  "Female" "Male" "Male" "Male" ...
$ Tot_Bil  : num  0.7 10.9 7.3 1 3.9 1.8 0.9 0.9 0.9 0.7 ...
$ Dir_Bil  : num  0.1 5.5 4.1 0.4 2 0.7 0.2 0.3 0.3 0.2 ...
$ Alkphos  : int  187 699 490 182 195 208 154 202 202 290 ...
$ Alamine  : int  16 64 60 14 27 19 16 14 22 53 ...
$ Aspartate: int  18 100 68 20 59 14 12 11 19 58 ...
$ Tot_Prot : num  6.8 7.5 7 6.8 7.3 7.6 7 6.7 7.4 6.8 ...
$ Albumin  : num  3.3 3.2 3.3 3.4 2.4 4.4 3.5 3.6 4.1 3.4 ...
$ A_G_Ratio: num  0.9 0.74 0.89 1 0.4 1.3 1 1.1 1.2 1 ...
$ Disease  : int  1 1 1 1 1 1 1 1 2 1 ...
```

### 12.1.2 Data visualization and imputation

First we need to check the data set for missing values:

```
summary(ilpd)
```

```
      Age            Sex               Tot_Bil          Dir_Bil
 Min.   : 4.00   Length:583         Min.   : 0.400   Min.   : 0.100
 1st Qu.:33.00   Class :character   1st Qu.: 0.800   1st Qu.: 0.200
 Median :45.00   Mode  :character   Median : 1.000   Median : 0.300
 Mean   :44.75                      Mean   : 3.299   Mean   : 1.486
 3rd Qu.:58.00                      3rd Qu.: 2.600   3rd Qu.: 1.300
 Max.   :90.00                      Max.   :75.000   Max.   :19.700

    Alkphos          Alamine          Aspartate         Tot_Prot
 Min.   :  63.0   Min.   :  10.00   Min.   :  10.0   Min.   :2.700
 1st Qu.: 175.5   1st Qu.:  23.00   1st Qu.:  25.0   1st Qu.:5.800
 Median : 208.0   Median :  35.00   Median :  42.0   Median :6.600
```

```
 Mean   : 290.6   Mean   :  80.71   Mean   : 109.9   Mean   :6.483
 3rd Qu.: 298.0   3rd Qu.:  60.50   3rd Qu.:  87.0   3rd Qu.:7.200
 Max.   :2110.0   Max.   :2000.00   Max.   :4929.0   Max.   :9.600

    Albumin        A_G_Ratio         Disease
 Min.   :0.900   Min.   :0.3000   Min.   :1.000
 1st Qu.:2.600   1st Qu.:0.7000   1st Qu.:1.000
 Median :3.100   Median :0.9300   Median :1.000
 Mean   :3.142   Mean   :0.9471   Mean   :1.286
 3rd Qu.:3.800   3rd Qu.:1.1000   3rd Qu.:2.000
 Max.   :5.500   Max.   :2.8000   Max.   :2.000
                 NA's   :4
```

We have some NA's in A_G_Ratio variable. We are going to impute values using the bagimpute method of the *PreProcess()* function of caret as mentioned in the missing values chapter.

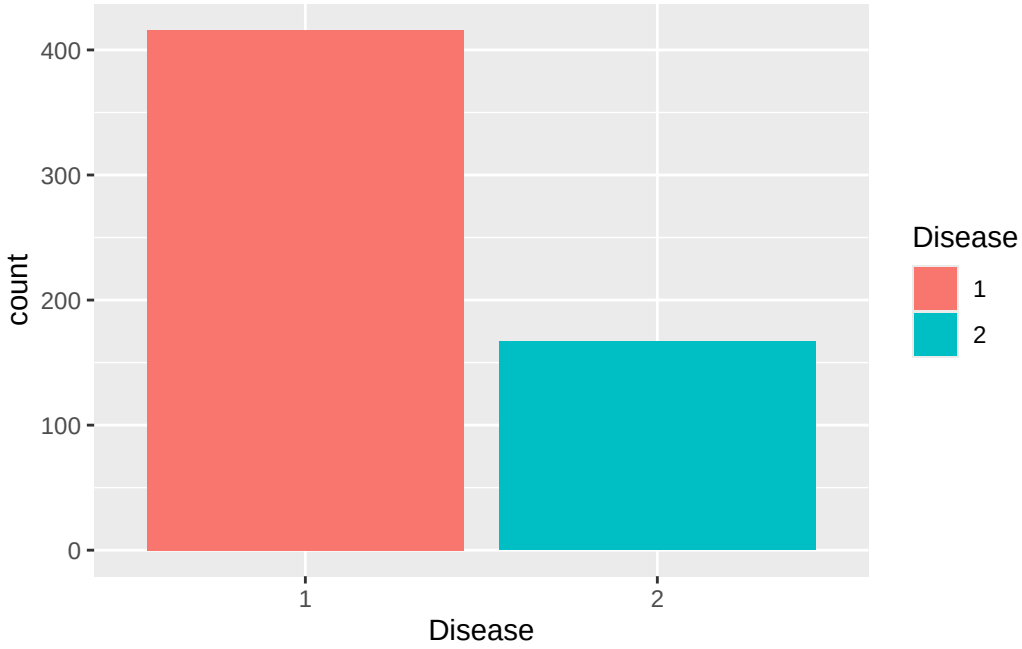
```
library(caret)
ilpd_filled<-preProcess(ilpd,method ="bagImpute")


imputed_ilpd<-predict(ilpd_filled,ilpd)
```

Now we have are data set imputed and ready for inspection. One more thing before starting the visualization of the data. As the summary above shows the Sex variable is of type character and the Disease int. We need to change that in order to proceed checking the imbalance.

```
imputed_ilpd$Sex<-as.factor(ilpd$Sex)
imputed_ilpd$Disease<-as.factor(ilpd$Disease)
```

Now it is time for visualization. Using the package tidyverse as mentioned in the data vizualization chapter:

```
library(tidyverse)


imputed_ilpd %>% ggplot(aes(x = Disease,fill = Disease))+
geom_bar()
```




It is obvious that our data set is imbalanced in respect of Disease. 416 patients diagnosed with liver disease and 167 patients without liver disease. Now it would be appropriate to apply the methods that we mention before.

### 12.1.3 The methods

Before applying any of the methods we split the data into train and test set:

```
library(caTools)
set.seed(1234)
split <- sample.split(imputed_ilpd$Disease, SplitRatio = 0.8)
```

```
train_ilpd<- subset(imputed_ilpd, split == TRUE)
test_ilpd <- subset(imputed_ilpd, split == FALSE)


summary(train_ilpd$Disease)
```

```
  1   2
333 134
```

#### 12.1.3.1 Down-sampling or under-sampling

The caret package is going to be used to downsample the trainset.

```
train_ilpd_down<-downSample(x=train_ilpd[,-ncol(train_ilpd)],
# taking all but the last column
                            y=train_ilpd$Disease
#the last column
)
#The data set created now has a column named Class
#instead of Disease with 50%-50% split
table(train_ilpd_down$Class)
```

```
  1   2
134 134
```

The new trainset is balanced by deleting randomly the extra healthy patients.

#### 12.1.3.2 Up-sampling or oversampling

By the same way, using this time the upSample() function of caret:

```
# taking all but the last column
train_ilpd_up<-upSample(x=train_ilpd[,-ncol(train_ilpd)],
#the last column
                        y=train_ilpd$Disease)
#The data set created now has a column named Class
#instead of Disease with 50%-50% split
table(train_ilpd_up$Class)
```

```
  1   2
333 333
```

The new trainset is balanced by adding using sampling (with replacement) cases from diseased people.

#### 12.1.3.3 ROSE (Random Over-Sampling Examples)

From the manual of the rose package:

*ROSE (Random Over-Sampling Examples) is a bootstrap-based technique which aids the task of binary classification in the presence of rare classes. It handles both continuous and categorical data by generating synthetic examples from a conditional density estimate of the two classes.*

```
#install.packages('ROSE')
library(ROSE)
set.seed(111)
```

```
train_rose<-ROSE(Disease~.,data=train_ilpd)$data


table(train_rose$Disease)
```

```
  1   2
222 245
```

It adds new synthetic data points to the minority class and downsamples the majority class.

#### 12.1.3.4 SMOTE (Synthetic Minority Oversampling TEchnique)

*SMOTE (Chawla et. al. 2002) is a well-known algorithm to fight this problem. The general idea of this method is to artificially generate new examples of the minority class using the nearest neighbors of these cases. Furthermore, the majority class examples are also under-sampled, leading to a more balanced dataset.*

```
#install.packages('smotefamily')
library(smotefamily)



#train_SMOTE = SMOTE(train_ilpd[,-ncol(train_ilpd)],train_ilpd[,ncol(train_ilpd)])
```

# 13 Unsupervised Machine learning, algorithms and examples

Unsupervised learning is the branch of Machine learning where algorithms analyze and cluster unlabeled datasets (Dimension reduction which also belongs to Unsupervised learning). These algorithms discover hidden patterns or data groupings which are not obvious. Its ability to discover similarities and differences in information make it the ideal solution for exploratory data analysis. Two very popular methods (algorithms) in these notes are going to be discussed in detail and applied in r:

1. **Kmeans algorithm**
2. **Hierarchical clustering**

## 13.1 The K-means algorithm

k-means is quite simple and elegant algorithm with a purpose to partition the dataset to k distinct non-overlapping clusters that the user initially defines. This is achieved by partitioning the observations into k clusters such that the total within-cluster variation, summed over all K clusters, is as small as possible. The **cluster variation** is defined as follows:

> The sum of all the pairwise squared Euclidean distances between the observations in a cluster, divided by the total number of observations of the cluster.

The steps of k-means algorithm in order to converge to the final result of k clusters are the following:

1. Choosing the number of K clusters
2. Selecting k-random points in your dataset space (not necessarily belonging to your data) which are called centroids
3. Assign each data point to the closest centroid
4. Compute the new centroid of the cluster
5. Reassign each data to the new centroid
6. Repeat the last 2 steps until no new centroid emerges (The algorithm converges).

The following visual representation of the clustering process would make things more understandable:

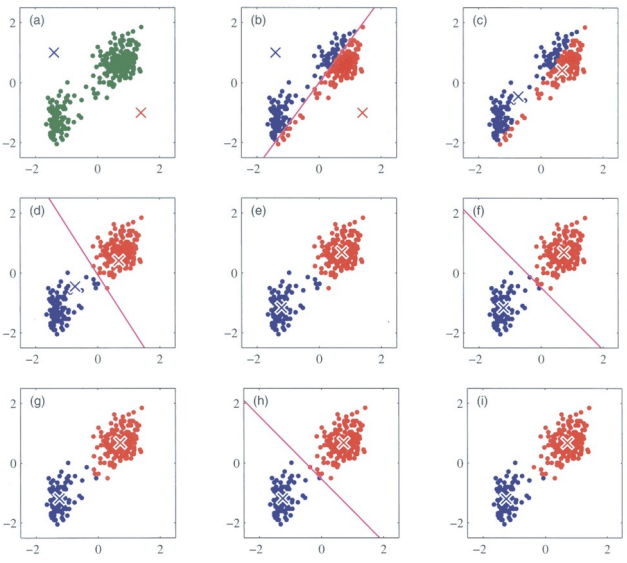



Figure 13.1: Visual representation of the clustering process (k=2, for dataset with 2 columns (2 predictor variables) of data)

The golub dataset is going to be used for the implementation of the algorithm. It would be a good idea to scale the data before the implementation of the clustering, as it is obvious by the definition of the algorithm that it uses distances between the observables (patients in our case) so as to make decisions for the clustering. It would be quite appropriate the distances to be in the same order (the scaling function would be responsible for that) in order to end up with a good clustering.

```
GOLUB<-readRDS(file = "golub")
# We are using only the first 10 genes here just for speeding the process
golub <- data.frame(GOLUB)[1:10]
# Scaling
golub <- scale(golub)
```

Now is time to implement the algorithm. The base function kmeans() will be used on this purpose.

```
fit <- kmeans(x = golub, centers=2, iter.max = 50, nstart =5)


# Info from the results of the algorithm:
fit$cluster  # Which cluster each patient belongs
```

```
 [1] 2 1 2 2 1 1 2 2 1 1 1 1 1 1 1 1 2 1 1 1 1 2 2 2 2 1 2 1 1 1 1 1 2 2 2 2 2 2
[39] 2 2 1 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 2 1 1 2 2 2 2 1 1 1 1 1 1 1 1
```

```
fit$centers  # The centroids of the two groups
```

```
  AFFX.BioB.5_at AFFX.BioB.M_at AFFX.BioB.3_at AFFX.BioC.5_at AFFX.BioC.3_at
1      0.2590249      0.3743804     0.03841863    -0.08731761      0.5624959
2     -0.4317082     -0.6239673    -0.06403105     0.14552935     -0.9374931
```

```
  AFFX.BioDn.5_at AFFX.BioDn.3_at AFFX.CreX.5_at AFFX.CreX.3_at AFFX.BioB.5_st
1       0.5597461       0.08047452      0.4570181     -0.4353877      0.08684915
2      -0.9329102      -0.13412420     -0.7616968      0.7256462     -0.14474858
```

```
fit$size     # The number of patients in each group.
```

```
[1] 45 27
```

> ! The kmeans function input
>
> **km<-kmeans(dtset, nofClusters, iter.max=num, nstart=num)**
>
> • **dtset:** the dataset in the form of dataframe or matrix.
>
> • **nofClusters**: The number of clustering groups.
>
> • **iter.max:** The maximum number of iterations. How many times the algorithm should run to get the result. To converge. (Here usually a 2 digit or 3 digit number should be chosen)
>
> • **nstart:** The Number of random initial sets should be chosen. The number of times the algorithm should run with different initial centers.

Let's give an example so as to explain a little bit more the algorithm implementation. The algorithm sometimes converges into a local minimal instead of a global one. In order to avoid this situation, so as to have the best clustering of all, the function provides the user with the ability to enter more than one time (nstart times), initial cendroids.

e.g. let's suppose that 3 groups are needed and we set nstart=10 and iter.max = 5.

The function would call 10 times 3 different initial centroids, and the minimum result of those 10 would define the 3 final centroids. The algorithm runs 5 times for converging to each final centroid.

## 13.2 Hierarchical clustering

Hierarchical clustering is a method that can provide the user with the a tree of clustering choices, in order her/him to choose the appropriate number of clusters. This is a major advantage compared to the k-means. One can divide the clustering process to two categories, two modes:

**Agglomerative or (Agglomerative Nesting AGNES)**: Supposing that n observations are appearing in our dataset each of them is treated as its own cluster (leaf). The two clusters that are most similar to each-other, in terms of their inter distance, are merged to one. The algorithm proceeds in this way repeating the above similarity process until all observations belong to one single cluster, and that is what is called a dendrogram, is complete.

**Divisive hierarchical clustering:** It's also known as DIANA (Divise Analysis). Works in the opposite way of Agglomerative. It begins with the hypothesis that all the observables belong to one cluster and then partitions the cluster into two least similar clusters. It proceeds recursively to form new clusters until all objects are in their own cluster.

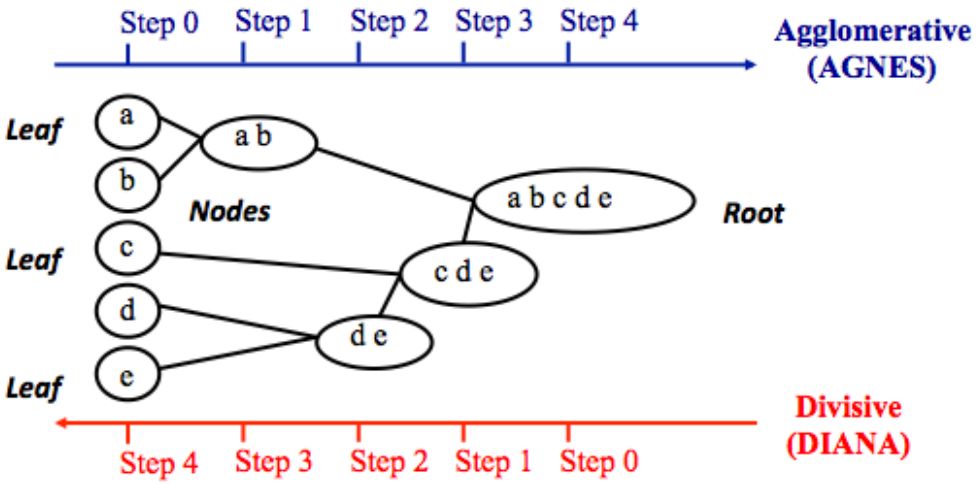



Figure 13.2: Visualization of the steps of the two types of hierarchical clustering.

There are lot of distance choices in hierarchical clustering. A general idea can be given by the

following graph:

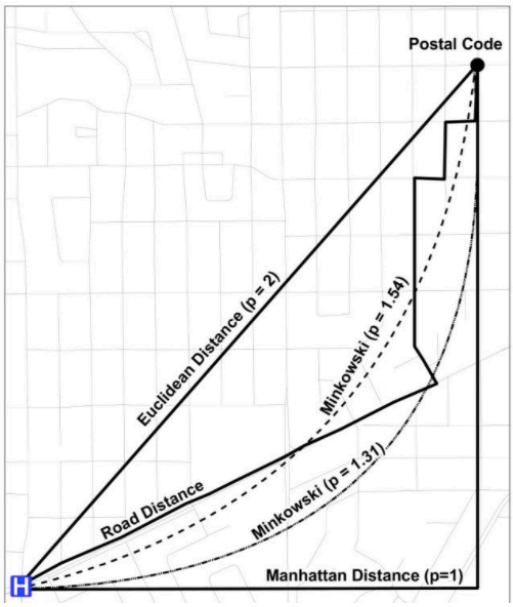



Figure 13.3: Minkowski-Euclidean-and-Manhattan-Distance

There are also different ways of *measuring the dissimilarity between two clusters of observations.* A number of different cluster agglomeration methods (i.e, linkage methods) are available. The most common types, methods are the following:

- **Maximum or complete linkage clustering:** It computes all pairwise dissimilarities between the elements in cluster 1 and the elements in cluster 2, and considers the largest value (i.e., maximum value) of these dissimilarities as the distance between the two clusters. It tends to produce more compact clusters.
- **Minimum or single linkage clustering:** It computes all pairwise dissimilarities between the elements in cluster 1 and the elements in cluster 2, and considers the smallest of these dissimilarities as a linkage criterion. It tends to produce long, "loose" clusters.
- **Mean or average linkage clustering:** It computes all pairwise dissimilarities between the elements in cluster 1 and the elements in cluster 2, and considers the average of these dissimilarities as the distance between the two clusters.

- **Centroid linkage clustering:** It computes the dissimilarity between the centroid for cluster 1 (a mean vector of length p variables) and the centroid for cluster 2.
- **Ward's minimum variance method:** It minimizes the total within-cluster variance. At each step the pair of clusters with minimum between-cluster distance are merged.

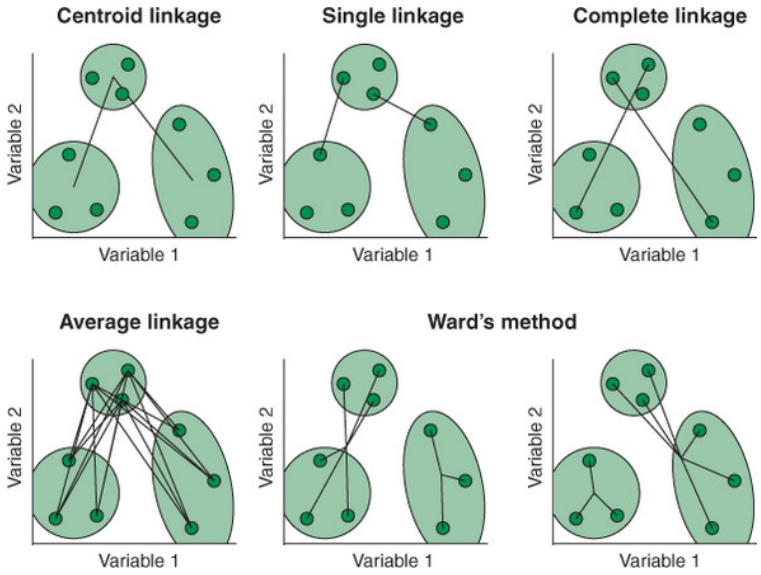
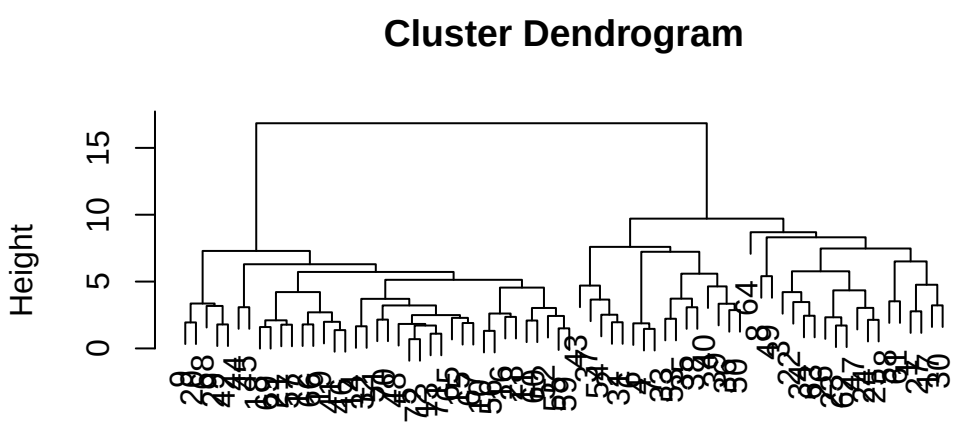



Figure 13.4: Cluster distance measures

More analytic explanation can be found to this video too. No it is time to apply it to the golub dataset.

```
d <- dist(golub, method = "euclidean")  # distance matrix
fit <- hclust(d, method = "ward.D2")


plot(fit)
```




```
groups <- cutree(fit, k = 3)  # cut the tree to 3 different groups
table(groups) # The number patients that belong to a certain group
```

```
groups
 1  2  3
19 16 37
```

**!** Hierarchical Clustering using R

**1st step**: Define the distance for clustering

**d <-dist(dtset,method="thedistancemethod")**

**thedistancemethod:** As mentioned above, a lot of methods could be used in order to calculate the distance between observables. The most common is Euclidean distance. Others included in the dist function are: "maximum", "manhattan", "canberra", "binary" or "minkowski".

**2nd step**: Define the clustering method:

**hc<-hclust(d, method="theclusteringmethod")**

Different types of clusterization are also included "ward.D", "ward.D2", "single", "complete", "average" (= UPGMA), "mcquitty" (= WPGMA), "median" (= WPGMC) or "centroid" (= UPGMC).

**3rd step**: Determine the number of clusters.

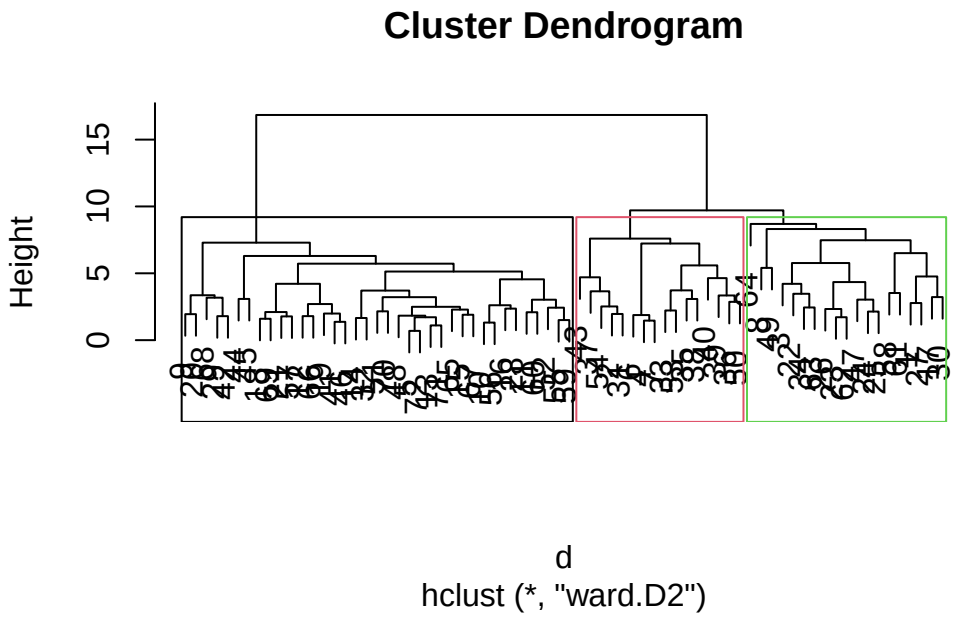
**clustering<-cutree(hc, k="numberofclusters")**

As mentioned above the number could be determined at the end by defining k in the cutree function.

All functions here are base functions, meaning that no libraries need to be loaded or packages to be installed.

To draw the dendogram the rect.hclust() function is used:

```
# drawing the dendogram...
plot(fit)
rect.hclust(fit, k = 3, border = 1:3)
```



Here **fit** is the hierarchical model that was used, at first the fit is plotted and then the rectangulars that define the clusters using the rect..hclust function, where **k** is the number of clusters and color of the **border** is defined by a vector with border colors for the rectagulars.

## 13.3 The DBSCAN algorithm for clustering

The name of the algorithm is actually produced from the initials of the following: (**Density-Based Spatial Clustering of Applications with Noise**). As the full name of the algorithm states is a density-based algorithm. Such algorithms use unsupervised learning methods that identify distinctive clusters in the data, based on the idea that a cluster in data space is a contiguous region of high point density, separated from other such clusters by contiguous regions of low point density. It can also identify Noise and outliers from data. It introduced by Ester at al. 1996

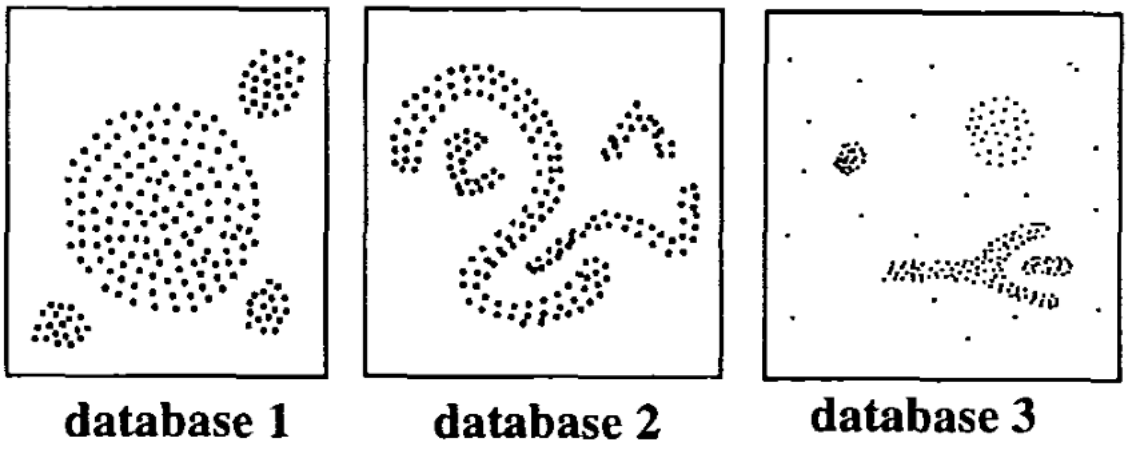



Figure 13.5: Clusters discovered by DBSCAN (from Ester at al. 1996)

Two parameters are needed as an input in order to apply DBSCAN. ("eps") epsilon and ("MinPts") minimum points. Epsilon is the radius of a circle that defines the neighborhood of a point. The MinPts is actually the minimum number of points that must exist in the neighborhood of a point in order to be a **core point**. If the number of its neighbors is less

than MinPts but the point belongs to the neighbourhood of a core point (it is inside its radius epsilon) then the point is called a **border point**. Finally in case none of the above applies the point is called a **noise point**.

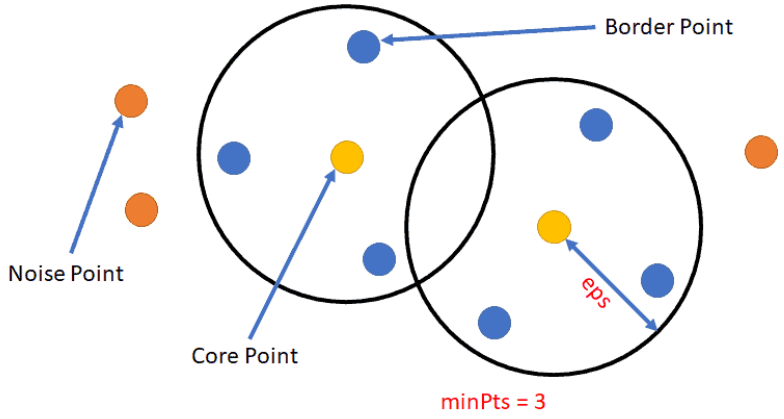



Figure 13.6: The three types of points and the parameters in DBSCAN

More definitions:

- **Direct density reachable**: A point "A" is directly density reachable from another point "B" if: i) "A" is in the neighborhood of "B" and ii) "B" is a core point.
- **Density reachable**: A point "A" is density reachable from "B" if there are a set of core points leading from "B" to "A".
- **Density connected**: Two points "A" and "B" are density connected if there are a core point "C", such that both "A" and "B" are density reachable from "C".

! The steps of DBSCAN algorithm

1. For each point x, compute the distance between x and the other points. Find all neighbor points within distance eps of the starting point x. Each point, with a neighbor count greater than or equal to MinPts, is marked as core point.
2. For each core point, if it's not already assigned to a cluster, create a new cluster.

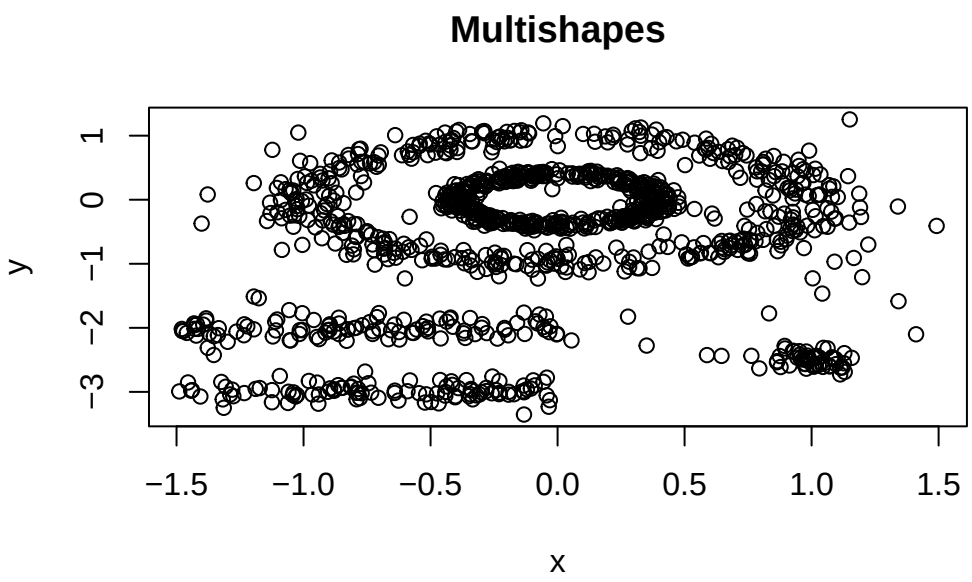
Find recursively all its density connected points and assign them to the same cluster as the core point.

3. Iterate through the remaining not visited points in the data set.

Those points that do not belong to any cluster are treated as outliers or noise.

Enough of theory. Time for practice. To understand how important can DBSCAN algorithm let's use a special dataset, multishapes from factoextra package.

```
# Load the data
data("multishapes", package = "factoextra")
df <- multishapes[, 1:2]


plot(df, main="Multishapes")
```

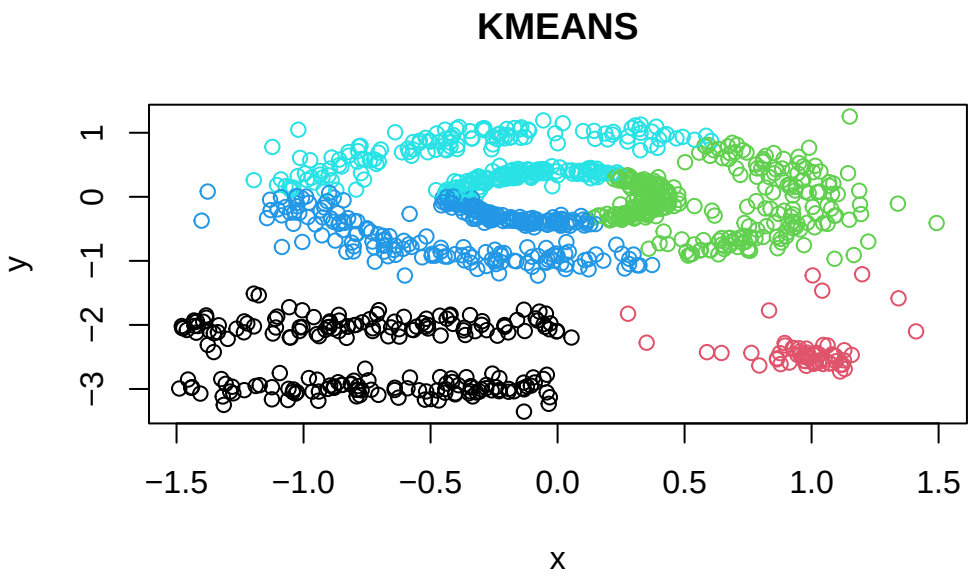



The name of the dataset implies what we see here. It is obvious that finding the clusters would

be a tedious task.

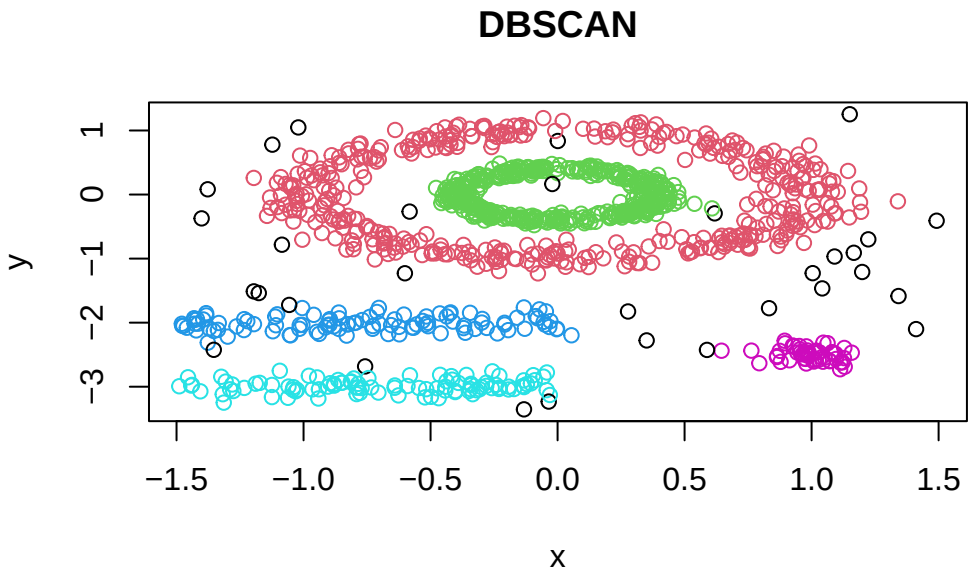
It seems that we have 5 clusters. Can k-means find them? Let's try.

```
kmeans_res <- kmeans(df, centers=5, iter.max = 50, nstart =25)
# Inserting color to clusters
plot(df, col=kmeans_res$cluster, main="KMEANS")
```




K-means did not succeed in finding all the clusters. Remember that k-means tries to produce compact clusters. Let's try with DBSCAN.

```
library("fpc")
set.seed(123)
dbscan_res <- fpc::dbscan(df, eps = 0.15, MinPts = 5)
# Inserting color to clusters
plot(df, col=dbscan_res$cluster+1, main="DBSCAN")
```




It is obvious that DBSCAN did a better job. In order to find the number of points that belong to each cluster we have:

```
print(dbscan_res)
```

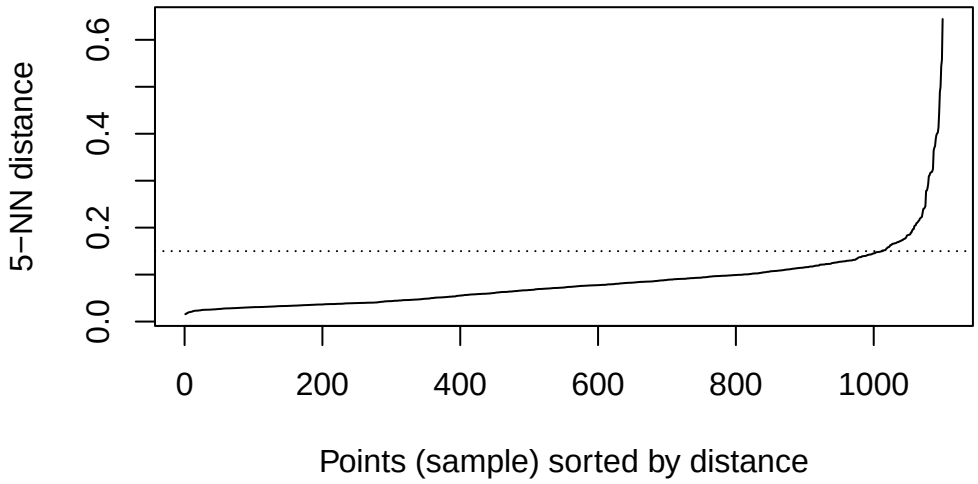
```
dbscan Pts=1100 MinPts=5 eps=0.15
         0   1   2   3  4  5
border 31  24   1   5  7  1
seed    0 386 404  99 92 50
total  31 410 405 104 99 51
```

The algorithm checks the point density and decides that 5 clusters exist. The rest of the points (outliers) are put in the 0th group. The seed line is the total number of the **core** points that belong to the group and **border** the number of the border points.

When a user decides the number of MinPts to use to find out the **epsilon** parameter could become straightforward. A function called **kNNdistplot** from the **dbscan** package ( it a

package that does very similar job as fpc::dbscan function, try it out) could help us finding it out. It helps calculating the distance from a point to its k nearest neighbor (MinPts number here) and plots the number of points in terms of this distance.

```
#install.packages('dbscan')
library(dbscan)
dbscan::kNNdistplot(df, k=5)
abline(h=0.15, lty =3)
```




One can choose the optimal epsilon in the area where the line bends, what we call elbow method (we will discuss it just afterwards) in order to find out the best epsilon to use. Here a line at 0.15 is drawn as this is the optimal epsilon as one could see from the plot.

## 13.4 Finding the optimal number of clusters

One of the most difficult tasks in clustering is finding the appropriate number of clusters. A lot of automatized methods exist in order to accomplish the task. Three of them are going to be explained and applied in R.

1. The elbow method
2. Silhouette method
3. Gap statistic method

### 13.4.1 The elbow method

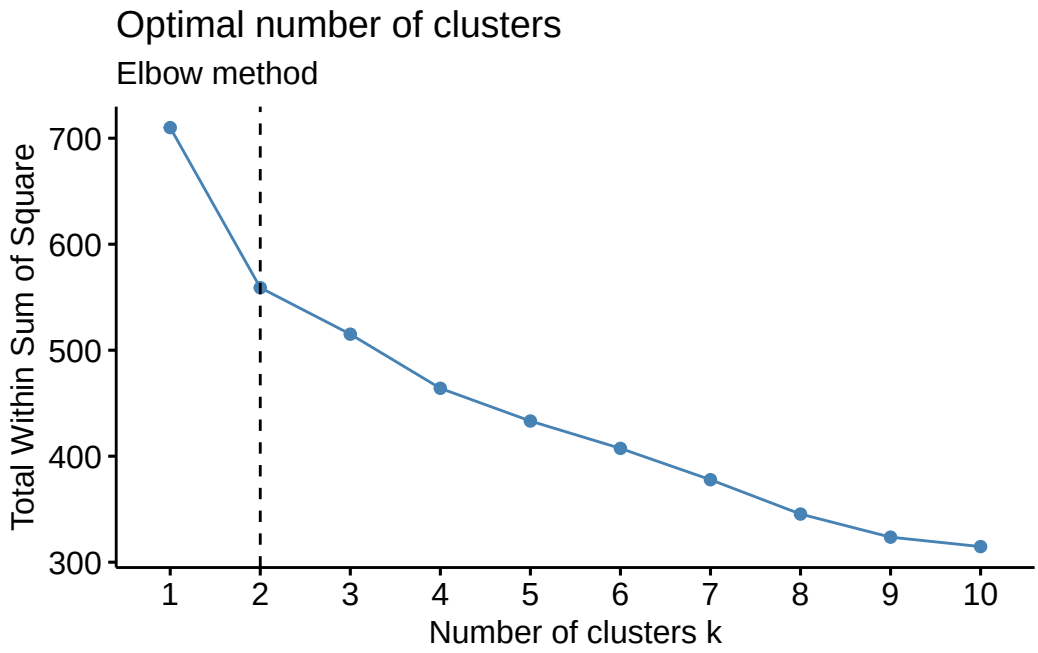
The Within-Cluster Sum of Squares (WSS) or cluster variation as discussed in the kmeans section measures the compactness of the cluster. The elbow method calculates the total wss, for all the clusters for each k. By plotting the curve of wss in respect of the clusters. In order to apply it in R, for visual reasons a part of the golub dataset will be used:

```
GOLUB<-readRDS(file = "golub")


golub <- data.frame(GOLUB)[1:10]


golub <- scale(golub)
```

The graph is actually drawn by the fviz_nbclust() function of factoextra package. for any partitioning clustering methods [K-means, HCUT(hierarchical clustering)].

```
library(factoextra)
fviz_nbclust(golub, kmeans, method = "wss")+
#Adding a line on the optimal number of clusters
```

```
geom_vline(xintercept = 2, linetype = 2)+
# adding caption
labs(subtitle = "Elbow method")
```



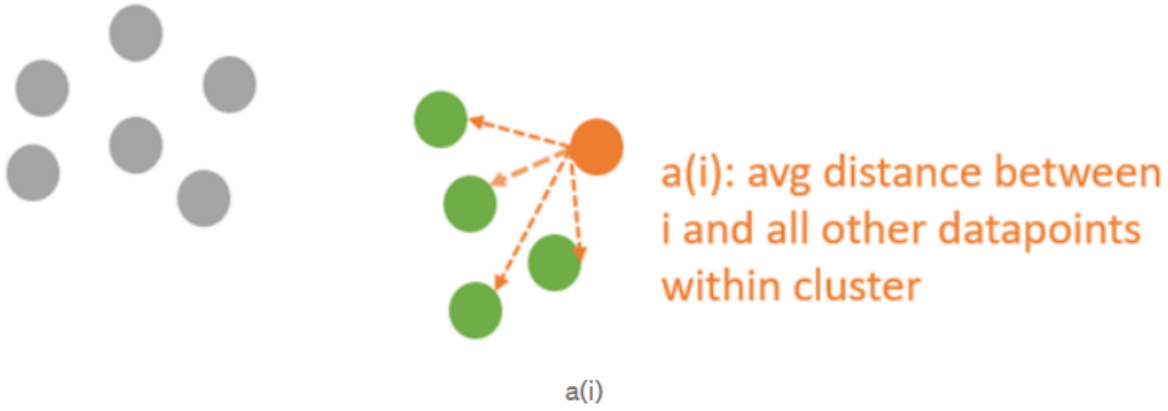

Once the number of clusters exceeds the actual number of groups in the data, the added information will drop sharply, because it is just subdividing the actual groups. Assuming this happens, we check for a sharp elbow in the graph of explained variation versus clusters. So for our case the optimal number of groups is 2.

### 13.4.2 Silhouette method

The method is based on the calculation of the silhouette coefficient.

$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

a(i): Is the average distance between observation i and all the other observations that belong to the specific cluster i belongs.




a(i)

b(i): is the average distance from i to all clusters to which i does not belong.

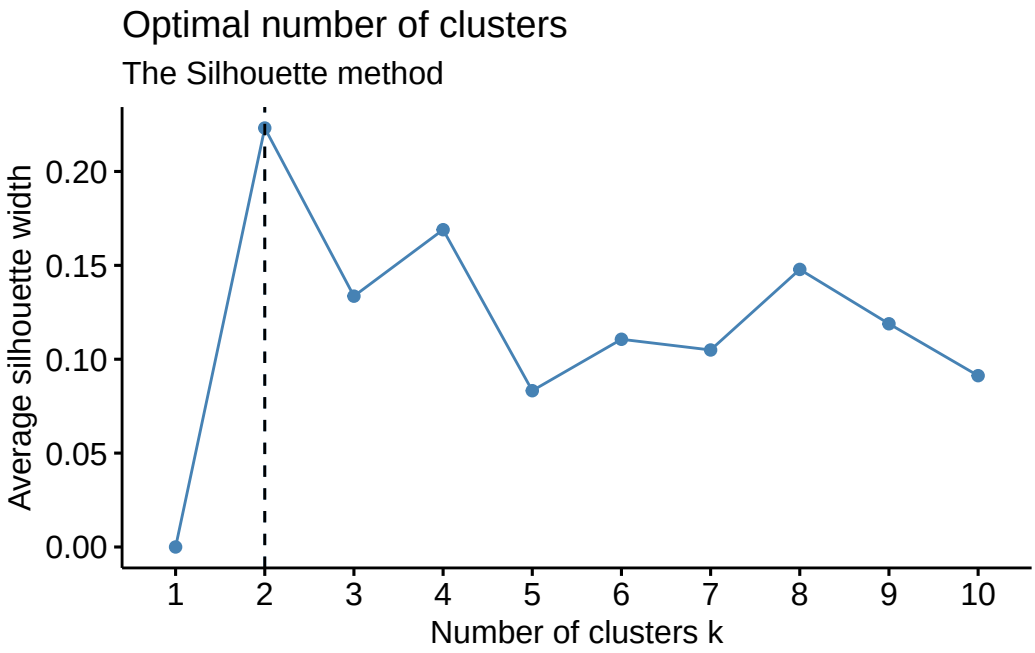

b(i)

The method calculates the average silhouette for every k. Where k is the number of clusters occured from the method of clustering (k-means, hcut etc.).

$$AverageSilhouette = mean\{S(i)\}$$

The maximum of the plot of that curve (avg. silhouette in respect of k) will provide the appropriate number of clusters. Let's apply it in the dataset using the fviz_nbclust() function of factoextra package.

```
fviz_nbclust(golub, kmeans, method = "silhouette")+
#Adding a line on the optimal number of clusters
geom_vline(xintercept = 2, linetype = 2)+
# adding caption
labs(subtitle = "The Silhouette method")
```




### 13.4.3 Gap statistic method

The gap statistic was developed by Stanford researchers Tibshirani, Walther and Hastie in their 2001 paper (Tibshirani, Walther, and Hastie 2001).

The algorithm works as follows:

1. Cluster the observed data, varying the number of clusters from k = 1, ..., kmax, and compute the corresponding total within intra-cluster variation $W_k$.

2. Generate B reference data sets with a random uniform distribution. Cluster each of these reference data sets with varying number of clusters k = 1, …, kmax, and compute the corresponding total within intra-cluster variation $W_{kb}$.

3. Compute the estimated gap statistic as the deviation of the observed $W_k$ value from its expected value $W_{kb}$ under the null hypothesis:

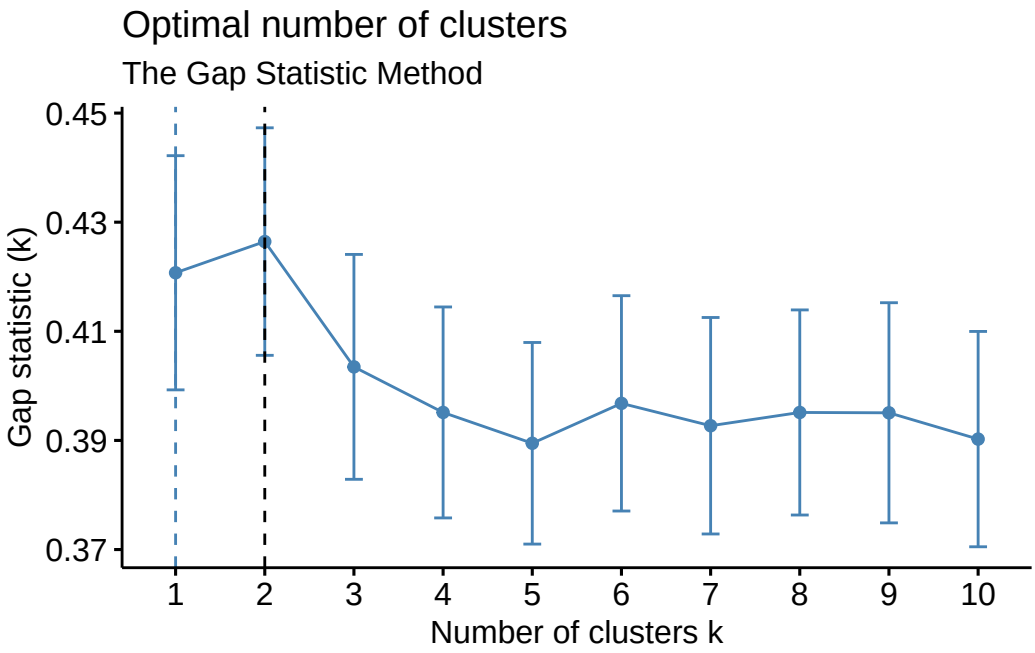
$$Gap(k) = \frac{1}{B}\sum_{b=1}^{B} log(W_{kb}*) - log(W_k)$$

Compute also the standard deviation of the statistics.

4. Choose the number of clusters as the smallest value of k such that the gap statistic is within one standard deviation of the gap at k+1:

$$Gap(k) \geqslant Gap(k+1) - s_{k+1}$$

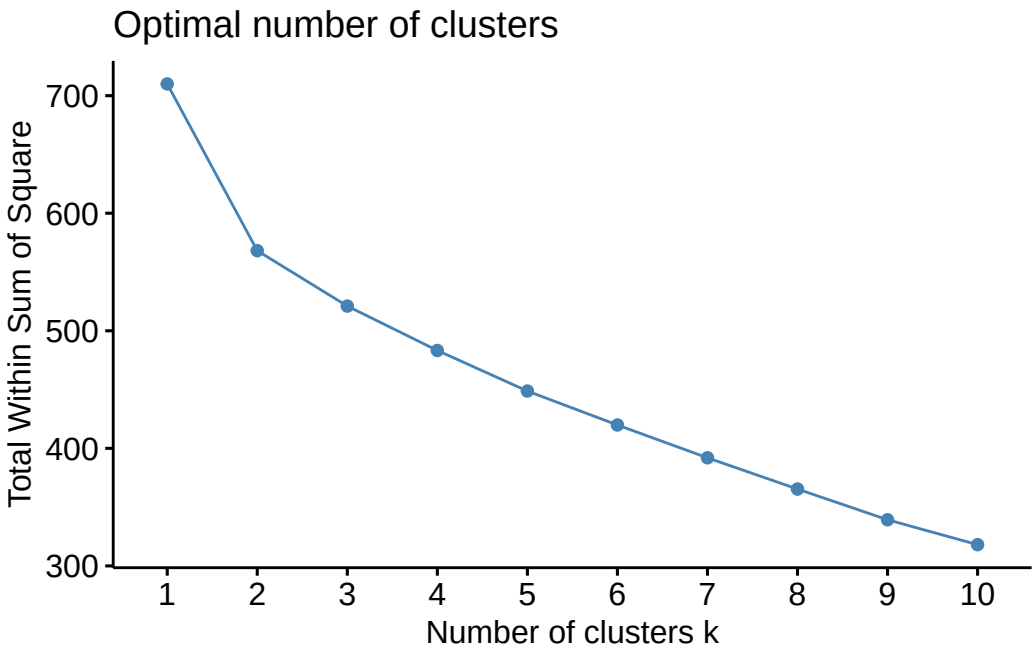
```
fviz_nbclust(golub, kmeans, method = "gap_stat")+
#Adding a line on the optimal number of clusters
geom_vline(xintercept = 2, linetype = 2)+
# adding caption
labs(subtitle = "The Gap Statistic Method")
```




Here in all three cases the verdict coincides, though it is not always the case.

Just an extra info for the case of hierarchical clustering:

```
# for elbow method:

fviz_nbclust(golub, hcut, method = "wss")
```


Optimal number of clusters
Total Within Sum of Square
700
600
500
400
300
1
2
3
4
5
6
7
8
9
10
Number of clusters k

# 14 Summary

These notes provide the reader with the first deep dive to Machine Learning. A start for understanding and implementing the very basic algorithms, to simple datasets, providing answers through correct predictions or clustering and much more to biological questions. It is an introductory course covering a extensive number of the processes that needed in order to complete a complete data analysis.

# 15 References


Azur, Melissa J, Elizabeth A Stuart, Constantine Frangakis, and Philip J Leaf. 2011. "Multiple Imputation by Chained Equations: What Is It and How Does It Work?" *Int. J. Methods Psychiatr. Res.* 20 (1): 40–49.

Blake, Merz, C.L. 1998. "UCI Repository of Machine Learning Databases." http://www.ics.uci.edu/mlearn/MLRepository.html.

Golub, T. R., D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, et al. 1999. "Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression Monitoring." *Science* 286 (5439): 531–37. https://doi.org/10.1126/science.286.5439.531.

Kuhn, Max. 2008. "Building Predictive Models in r Using the Caret Package." *Journal of Statistical Software* 28 (5): 1–26. https://doi.org/10.18637/jss.v028.i05.

Larabi-Marie-Sainte, Souad, Linah Aburahmah, Rana Almohaini, and Tanzila Saba. 2019. "Current Techniques for Diabetes Prediction: Review and Case Study." *Applied Sciences* 9 (21). https://doi.org/10.3390/app9214604.

Maaten, Laurens van der, and Geoffrey Hinton. 2008. "Visualizing Data Using t-SNE." *Journal of Machine Learning Research* 9: 2579–2605. http://www.jmlr.org/papers/v9/vandermaaten08a.html.

Ramana, Bendi, and N. Venkateswarlu. 2012. "ILPD (Indian Liver Patient Dataset)." UCI Machine Learning Repository.

Tibshirani, Robert, Guenther Walther, and Trevor Hastie. 2001. "Estimating the Number of Clusters in a Data Set via the Gap Statistic." *Journal of the Royal Statistical Society: Series*

*B (Statistical Methodology)* 63 (2): 411–23. https://doi.org/10.1111/1467-9868.00293.

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. "Welcome to the tidyverse." *Journal of Open Source Software* 4 (43): 1686. https://doi.org/10.21105/joss.01686.