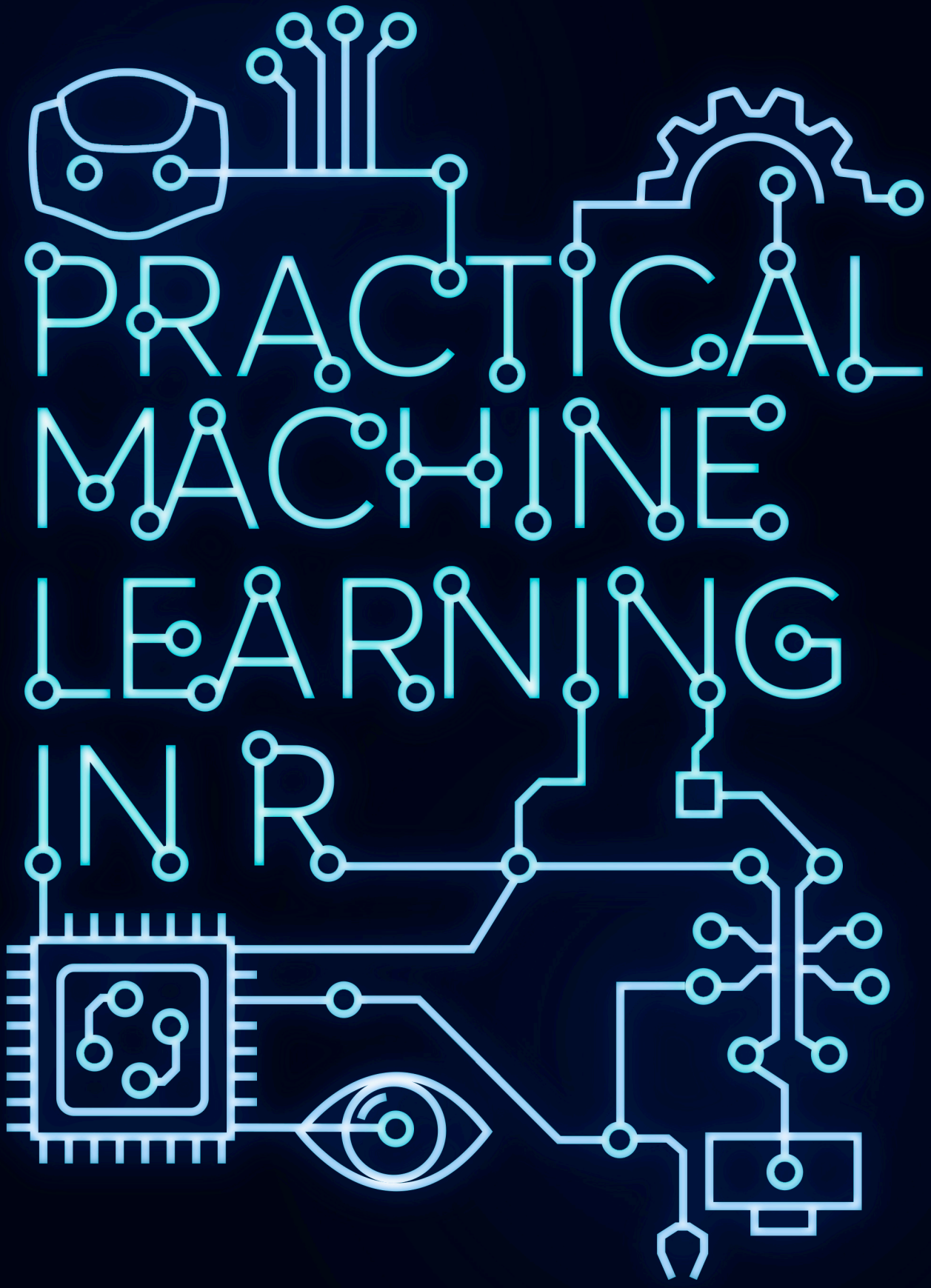
PRACTICAL MACHINE LEARNING IN R



Kyriakos Chatzidimitriou, Themistoklis Diamantopoulos
Michail Papamichail and Andreas Symeonidis

Practical Machine Learning in R

Kyriakos Chatzidimitriou, Themistoklis Diamantopoulos, Michail Papamichail and Andreas Symeonidis

This book is for sale at http://leanpub.com/practical-machine-learning-r

This version was published on 2018-06-17

Contents

Part I - Introduction

Chapter 1 - Introduction to R

1.1 The basics

R is an open source programming language and a free environment, mainly used for statistical computing and graphics. You can find information about R in the official website[1]. By searching with the keyword `R` with other topic-specific words in sites like Google, one can find additional information from sites, blog posts, tutorials, documents etc.

Even through R comes with its own environment: command line and graphical interfaces, one can use the popular RStudio[2], which offers additional graphical functionalities.

When in the R environment (the R prompt is `>`) one can exit by calling the `quit()` function or `q()` for short. When asked if you want to save the workspace, if you reply with a `y` for yes, all the variables that you have during the current R session will be saved into a file named `.Rdata`, in the current working directory. If you later start R in the same directory, the variables and their names will be automatically loaded.

To check which is your current working directory, your can enter:

```
getwd()
```

To set the working directory one can use the `setwd` function. For example setting the working directory to the Desktop directory in a Linux machine:

```
setwd("~/Desktop")
```

What you type at the R prompt is an expression, which R attempts to evaluate and type the result. For example `getwd()` is an expression that is evaluated by calling the function `getwd()` with no arguments. The same for `42`

```
42
```

```
## [1] 42
```

Console Output

Whatever starts with `##` in the book signifies what the reader should see in the console output.

and the same for

[1] https://www.r-project.org/
[2] https://www.rstudio.com/

```
(100 * 2 - 12 ^ 2) / 7 * 5 + 2
```

```
## [1] 42
```

There are also predefined constants like `pi` or `e`

```
sin(pi/2)
```

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

To find out the documentation of a specific function you can enter `?sum` or `help(sum)`. To search for functions, there is the `help.search("sin")` function to help you with that. For certain functions one can see examples of use by typing the expression `example(plot)`. Comments start with `#`, while to assign values to variables you can use `<-` or `=`. For example:

```
a <- 42
b <- (42 + a) / 2
print(a)
```

```
## [1] 42
```

```
print(b)
```

```
## [1] 42
```

With `ls()` one can check all the variables existing in the current R session.

```
ls()
```

```
##  [1] "a"                  "accuracy"           "b"
##  [4] "centers.matrix"     "clustering"         "col"
##  [7] "control"            "ctree"              "ctrl"
## [10] "d"
```

`ls()` output

The output of the `ls` method will differ from the above, based on what are the current variables in your session.

To delete all the variables in the current session you can use the call:

```
rm(list=ls())
```

1.2 Vectors

A vector is a collection of similar types of elements. For example integers. Some things you can do with vectors in R are shown below.

1. Create the vector `a = (10, 5, 3, 100, -2, 5, -50)`:

```
a <- c(10, 5, 3, 100, -2, 5, -50)
a
```

```
## [1]  10   5   3 100  -2   5 -50
```

1. Select the elements of the vector `a` with indices 1, 3, 4, and 5:

```
a[c(1,3:4)]
```

```
## [1]  10   3 100
```

The above expression uses the `c()` function for combining values and the `:` operator that generates sequences `from:to` with step 1. Another easy way of specifying sequences is to use the `seq` function.

```
c(1, 2, 7, 10)
```

```
## [1]  1  2  7 10
```

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1, 6, by=1)
```

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

```
seq(1,6, by=2)
```

```
## [1] 1 3 5
```

```
seq(1,by=2, length=6)
```

```
## [1]  1  3  5  7  9 11
```

Type `?seq` to get to know the function.

1. To check the type of a variable there is the `class` function:

```
class(a)
```

```
## [1] "numeric"
```

1. To check which `a` elements have a value greater than 5:

```
a > 5
```

```
## [1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
```

1. To return the indices of vector `a`, for which the values are TRUE:

```
which(a>5)
```

```
## [1] 1 4
```

1. To get the positive elements of a:

```
b <- a > 0
positives <- a[b]
positives
```

```
## [1]  10   5   3 100   5
```

```
# or more succintly
positives <- a[a>0]
positives
```

```
## [1]  10   5   3 100   5
```

1. To check the length of a vector:

```
length(a)
```

```
## [1] 7
```

1. One can also bind vectors by column (`cbind()`) or by row (`rbind()`)

```
c <- 1:7
rbind(a,c)
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## a   10    5    3  100   -2    5  -50
## c    1    2    3    4    5    6    7
```

```
cbind(a,c)
```

```
##        a c
## [1,]  10 1
## [2,]   5 2
## [3,]   3 3
## [4,] 100 4
## [5,]  -2 5
## [6,]   5 6
## [7,] -50 7
```

1.3 Matrices

To create matrics use the `matrix()` function:

```
matrix(10,3, 2)
```

```
##      [,1] [,2]
## [1,]   10   10
## [2,]   10   10
## [3,]   10   10
```

```
# or
matrix(c(1,2,3,4,5,6), 3, 2)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
# or
matrix(c(1,2,3), 3, 2)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
## [3,]    3    3
```

But let's examine how are we calling the `matrix` function:

```
args(matrix)
```

```
## function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
## NULL
```

So the first argument are the data, then with `nrow` or `ncol` arguments we can declare the number of rows and columns and with the argument `byrow` we declare that we want to fill in the matrix column-by-column if `byrow=FALSE` and row-by-row if `byrow=TRUE`. In the above calls we didn't use the `byrow` argument because the function `matrix` has a default value `byrow=FALSE` as we can also check from the documentation, `?matrix`.

```
m = matrix(1:9, byrow = TRUE, nrow=3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Here we have filled in a matrix with values 1 to 9, by row, with the number of rows equal to 3. This gives us a square 3x3 matrix. R is pretty smart in knowing that the number of columns should be 3 as well!

We can also call `cbind` and `rbind` and other functions like `rowSums`, `colSums`, `mean` and `t` for transpose:

```
m2 <- rbind(m, m)
m2
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]    1    2    3
## [5,]    4    5    6
## [6,]    7    8    9
```

```
rowSums(m2)
```

```
## [1]  6 15 24  6 15 24
```

```
colSums(m2)
```

```
## [1] 24 30 36
```

```
mean(m2)
```

```
## [1] 5
```

For element wise multiplication on can use the * operator while for matrix multiplication you can use the %*% operator.

```
am <- matrix(10:18, byrow = TRUE, nrow = 3)
am
```

```
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
## [3,]   16   17   18
```

```
bm <- matrix(c(3,6,7,10,8,1,2,3,2), byrow = TRUE, nrow = 3)
bm
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    7
## [2,]   10    8    1
## [3,]    2    3    2
```

```
am * bm
```

```
##      [,1] [,2] [,3]
## [1,]   30   66   84
## [2,]  130  112   15
## [3,]   32   51   36
```

```
am %*% bm
```

```
##      [,1] [,2] [,3]
## [1,]  164  184  105
## [2,]  209  235  135
## [3,]  254  286  165
```

```
t(am)
```

```
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
```

1.4 Data frames

Unlike matrices, data frames can store values of different types in their columns. They are used extensively in R for data analysis. As rows usually we have the observations (or samples) and as columns we have the characteristics (or attributes or features). When we read from a file, the result is read as a data frame. Download the zip file r-novice-inflammation.zip[3] and unzip it in the Desktop. Examine the file `inflammation-01.csv` with a text editor to see what we are going to be loading. The read the file:

```
data <- read.csv(file = "data/inflammation-01.csv", header = FALSE)
```

Notice the use of the path including data/ since we previously set the working directory as the Desktop with `getwd()`.

```
dir()
```

[3] http://swcarpentry.github.io/r-novice-inflammation/files/r-novice-inflammation-data.zip

```
##  [1] "_site.yml"                    "ar.html"
##  [3] "ar.pdf"                       "ar.Rmd"
##  [5] "breast-cancer-wisconsin.data" "cfs.html"
##  [7] "cfs.Rmd"                      "classification.html"
##  [9] "classification.Rmd"           "clustering.html"
## [11] "clustering.Rmd"               "CNAME"
## [13] "data"                         "data-eng.html"
## [15] "data-eng.Rmd"                 "dbscan_files"
## [17] "dbscan.html"                  "dbscan.pdf"
## [19] "dbscan.Rmd"                   "docs"
## [21] "dt_files"                     "dt.html"
## [23] "dt.pdf"                       "dt.Rmd"
## [25] "finder.html"                  "finder.Rmd"
## [27] "footer.html"                  "footer.tex"
## [29] "head.tex"                     "hierarchical_files"
## [31] "hierarchical.html"            "hierarchical.pdf"
## [33] "hierarchical.Rmd"             "index.html"
## [35] "index.Rmd"                    "intro.Rmd"
## [37] "kmeans.pdf"                   "kmeans.Rmd"
## [39] "knn.Rmd"                      "ml-tutorials.Rproj"
## [41] "nb.pdf"                       "nb.Rmd"
## [43] "nn.Rmd"                       "other.Rmd"
## [45] "outlier.pdf"                  "outlier.Rmd"
## [47] "pca.Rmd"                      "r.Rmd"
## [49] "README.md"                    "render.sh"
## [51] "retail.data.gz"               "rfe.Rmd"
## [53] "site_libs"                    "style.css"
## [55] "uc.Rmd"                       "wrapper.Rmd"
```

The `dir` function return the files and directories of the file system. The argument `header=FALSE` lets the `read.csv` function know that there is no header row to give the columns names.

With `head(data)` I can check if the data are loaded correctly. It return the first few rows:

```
head(data)
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
## 1  0  0  1  3  1  2  4  7  8   3   3   3  10   5   7   4   7   7  12  18
## 2  0  1  2  1  2  1  3  2  2   6  10  11   5   9   4   4   7  16   8   6
## 3  0  1  1  3  3  2  6  2  5   9   5   7   4   5   4  15   5  11   9  10
## 4  0  0  2  0  4  2  2  1  6   7  10   7   9  13   8   8  15  10  10   7
## 5  0  1  1  3  3  1  3  5  2   4   4   7   6   5   3  10   8  10   6  17
## 6  0  0  1  2  2  4  2  1  6   4   7   6   6   9   9  15   4  16  18  12
##   V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38
## 1   6  13  11  11   7   7   4   6   8   8   4   4   5   7   3   4   2   3
## 2  18   4  12   5  12   7  11   5  11   3   3   5   4   4   5   5   1   1
## 3  19  14  12  17   7  12  11   7   4   2  10   5   4   2   2   3   2   2
## 4  17   4   4   7   6  15   6   4   9  11   3   5   6   3   3   4   2   3
## 5   9  14   9   7  13   9  12   6   7   7   9   6   3   2   2   4   2   0
## 6  12   5  18   9   5   3  10   3  12   7   8   4   7   3   5   4   4   3
##   V39 V40
## 1   0   0
## 2   0   1
## 3   1   1
```

```
## 4   2   1
## 5   1   1
## 6   2   1
```

Other functions I can use are:

```
# type of the variable
class(data)
```

```
## [1] "data.frame"
```

```
# dimensions
dim(data)
```

```
## [1] 60 40
```

which tells me that I have a data frame with 60 observations (instances) and 40 variables.

```
# structure
str(data)
```

```
## 'data.frame':    60 obs. of  40 variables:
##  $ V1 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ V2 : int  0 1 1 0 1 0 0 0 0 1 ...
##  $ V3 : int  1 2 1 2 1 1 2 1 0 1 ...
##  $ V4 : int  3 1 3 0 3 2 2 2 3 2 ...
##  $ V5 : int  1 2 3 4 3 2 4 3 1 1 ...
##  $ V6 : int  2 1 2 2 1 4 2 1 5 3 ...
##  $ V7 : int  4 3 6 2 3 2 2 2 6 5 ...
##  $ V8 : int  7 2 2 1 5 1 5 3 5 3 ...
##  $ V9 : int  8 2 5 6 2 6 5 5 5 5 ...
##  $ V10: int  3 6 9 7 4 4 8 3 8 8 ...
##  $ V11: int  3 10 5 10 4 7 6 7 2 6 ...
##  $ V12: int  3 11 7 7 7 6 5 8 4 8 ...
##  $ V13: int  10 5 4 9 6 6 11 8 11 12 ...
##  $ V14: int  5 9 5 13 5 9 9 5 12 5 ...
##  $ V15: int  7 4 4 8 3 9 4 10 10 13 ...
##  $ V16: int  4 4 15 8 10 15 13 9 11 6 ...
##  $ V17: int  7 7 5 15 8 4 5 15 9 13 ...
##  $ V18: int  7 16 11 10 10 16 12 11 10 8 ...
##  $ V19: int  12 8 9 10 6 18 10 18 17 16 ...
##  $ V20: int  18 6 10 7 17 12 6 19 11 8 ...
##  $ V21: int  6 18 19 17 9 12 9 20 6 18 ...
##  $ V22: int  13 4 14 4 14 5 17 8 16 15 ...
##  $ V23: int  11 12 12 4 9 18 15 5 12 16 ...
##  $ V24: int  11 5 17 7 7 9 8 13 6 14 ...
##  $ V25: int  7 12 7 6 13 5 9 15 8 12 ...
##  $ V26: int  7 7 12 15 9 3 3 10 14 7 ...
##  $ V27: int  4 11 11 6 12 10 13 6 6 3 ...
##  $ V28: int  6 5 7 4 6 3 7 10 13 8 ...
```

```
##  $ V29: int  8 11 4 9 7 12 8 6 10 9 ...
##  $ V30: int  8 3 2 11 7 7 2 7 11 11 ...
##  $ V31: int  4 3 10 3 9 8 8 4 4 2 ...
##  $ V32: int  4 5 5 5 6 4 8 9 6 5 ...
##  $ V33: int  5 4 4 6 3 7 4 3 4 4 ...
##  $ V34: int  7 4 2 3 2 3 2 5 7 5 ...
##  $ V35: int  3 5 2 3 2 5 3 2 6 1 ...
##  $ V36: int  4 5 3 4 4 4 5 5 3 4 ...
##  $ V37: int  2 1 2 2 2 4 4 3 2 1 ...
##  $ V38: int  3 1 2 3 0 3 1 2 1 2 ...
##  $ V39: int  0 0 1 2 1 2 1 2 0 0 ...
##  $ V40: int  0 1 1 1 1 1 1 1 0 0 ...
```

which returns the class and dimensions of the variable `data`, along with a list of the variables with their type and their first values.

```
# statistical summarization of the data frame
summary(data)
```

```
##        V1          V2             V3             V4             V5
##  Min.   :0   Min.   :0.00   Min.   :0.000   Min.   :0.00   Min.   :1.000
##  1st Qu.:0   1st Qu.:0.00   1st Qu.:1.000   1st Qu.:1.00   1st Qu.:1.000
##  Median :0   Median :0.00   Median :1.000   Median :2.00   Median :2.000
##  Mean   :0   Mean   :0.45   Mean   :1.117   Mean   :1.75   Mean   :2.433
##  3rd Qu.:0   3rd Qu.:1.00   3rd Qu.:2.000   3rd Qu.:3.00   3rd Qu.:3.000
##  Max.   :0   Max.   :1.00   Max.   :2.000   Max.   :3.00   Max.   :4.000
##        V6             V7            V8             V9
##  Min.   :1.00   Min.   :1.0   Min.   :1.000   Min.   :2.000
##  1st Qu.:2.00   1st Qu.:2.0   1st Qu.:2.000   1st Qu.:4.000
##  Median :3.00   Median :4.0   Median :4.000   Median :5.000
##  Mean   :3.15   Mean   :3.8   Mean   :3.883   Mean   :5.233
##  3rd Qu.:4.00   3rd Qu.:5.0   3rd Qu.:5.250   3rd Qu.:7.000
##  Max.   :5.00   Max.   :6.0   Max.   :7.000   Max.   :8.000
##       V10             V11             V12             V13
##  Min.   :2.000   Min.   : 2.00   Min.   : 2.00   Min.   : 3.00
##  1st Qu.:3.750   1st Qu.: 4.00   1st Qu.: 3.75   1st Qu.: 5.00
##  Median :6.000   Median : 6.00   Median : 5.50   Median : 9.50
##  Mean   :5.517   Mean   : 5.95   Mean   : 5.90   Mean   : 8.35
##  3rd Qu.:7.000   3rd Qu.: 9.00   3rd Qu.: 8.00   3rd Qu.:11.00
##  Max.   :9.000   Max.   :10.00   Max.   :11.00   Max.   :12.00
##       V14             V15             V16             V17
##  Min.   : 3.000   Min.   : 3.000   Min.   : 3.0   Min.   : 4.000
##  1st Qu.: 5.000   1st Qu.: 5.000   1st Qu.: 6.0   1st Qu.: 6.750
##  Median : 8.000   Median : 8.000   Median :10.0   Median : 8.500
##  Mean   : 7.733   Mean   : 8.367   Mean   : 9.5   Mean   : 9.583
##  3rd Qu.:10.000   3rd Qu.:12.000   3rd Qu.:13.0   3rd Qu.:13.000
##  Max.   :13.000   Max.   :14.000   Max.   :15.0   Max.   :16.000
##       V18             V19             V20             V21
##  Min.   : 5.00   Min.   : 5.00   Min.   : 5.00   Min.   : 5.00
##  1st Qu.: 7.75   1st Qu.: 8.00   1st Qu.: 8.75   1st Qu.: 9.00
##  Median :11.00   Median :11.50   Median :13.00   Median :14.00
##  Mean   :10.63   Mean   :11.57   Mean   :12.35   Mean   :13.25
##  3rd Qu.:13.00   3rd Qu.:15.00   3rd Qu.:16.00   3rd Qu.:16.25
##  Max.   :17.00   Max.   :18.00   Max.   :19.00   Max.   :20.00
```

```
##       V22             V23             V24             V25
##  Min.   : 4.00   Min.   : 4.00   Min.   : 4.00   Min.   : 4.0
##  1st Qu.: 8.00   1st Qu.: 7.75   1st Qu.: 6.75   1st Qu.: 7.0
##  Median :13.00   Median :11.00   Median :10.00   Median :10.5
##  Mean   :11.97   Mean   :11.03   Mean   :10.17   Mean   :10.0
##  3rd Qu.:15.25   3rd Qu.:15.00   3rd Qu.:13.25   3rd Qu.:13.0
##  Max.   :19.00   Max.   :18.00   Max.   :17.00   Max.   :16.0
##       V26             V27             V28             V29
##  Min.   : 3.000   Min.   : 3.00   Min.   : 3.00   Min.   : 3.000
##  1st Qu.: 5.750   1st Qu.: 7.00   1st Qu.: 5.00   1st Qu.: 5.000
##  Median : 9.000   Median :10.00   Median : 7.00   Median : 7.000
##  Mean   : 8.667   Mean   : 9.15   Mean   : 7.25   Mean   : 7.333
##  3rd Qu.:12.000   3rd Qu.:12.00   3rd Qu.: 9.00   3rd Qu.:10.000
##  Max.   :15.000   Max.   :14.00   Max.   :13.00   Max.   :12.000
##       V30             V31             V32             V33
##  Min.   : 2.000   Min.   : 2.000   Min.   :2.00   Min.   :2.000
##  1st Qu.: 3.000   1st Qu.: 4.000   1st Qu.:4.00   1st Qu.:4.000
##  Median : 7.000   Median : 6.000   Median :6.00   Median :5.000
##  Mean   : 6.583   Mean   : 6.067   Mean   :5.95   Mean   :5.117
##  3rd Qu.:10.000   3rd Qu.: 8.000   3rd Qu.:8.00   3rd Qu.:6.000
##  Max.   :11.000   Max.   :10.000   Max.   :9.00   Max.   :8.000
##       V34           V35           V36             V37             V38
##  Min.   :1.0   Min.   :1.0   Min.   :1.000   Min.   :1.000   Min.   :0.0
##  1st Qu.:2.0   1st Qu.:2.0   1st Qu.:2.000   1st Qu.:2.000   1st Qu.:0.0
##  Median :4.0   Median :3.0   Median :4.000   Median :2.000   Median :1.0
##  Mean   :3.6   Mean   :3.3   Mean   :3.567   Mean   :2.483   Mean   :1.5
##  3rd Qu.:5.0   3rd Qu.:5.0   3rd Qu.:5.000   3rd Qu.:4.000   3rd Qu.:3.0
##  Max.   :7.0   Max.   :6.0   Max.   :5.000   Max.   :4.000   Max.   :3.0
##       V39             V40
##  Min.   :0.000   Min.   :0.0000
##  1st Qu.:0.000   1st Qu.:0.0000
##  Median :1.000   Median :1.0000
##  Mean   :1.133   Mean   :0.5667
##  3rd Qu.:2.000   3rd Qu.:1.0000
##  Max.   :2.000   Max.   :1.0000
```

There are also different ways I can select slices of data from the data frame:

```
# first value in data
data[1, 1]
```

```
## [1] 0
```

```
# middle value in data
data[30, 20]
```

```
## [1] 16
```

```
# first four rows and first ten columns
data[1:4, 1:10]
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
## 1  0  0  1  3  1  2  4  7  8   3
## 2  0  1  2  1  2  1  3  2  2   6
## 3  0  1  1  3  3  2  6  2  5   9
## 4  0  0  2  0  4  2  2  1  6   7
```

```
# doesn't have to start from the beginning
data[5:10, 1:10]
```

```
##    V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
## 5   0  1  1  3  3  1  3  5  2   4
## 6   0  0  1  2  2  4  2  1  6   4
## 7   0  0  2  2  4  2  2  5  5   8
## 8   0  0  1  2  3  1  2  3  5   3
## 9   0  0  0  3  1  5  6  5  5   8
## 10  0  1  1  2  1  3  5  3  5   8
```

```
# specific rows and columns
data[c(3, 8, 37, 56), c(10, 14, 29)]
```

```
##    V10 V14 V29
## 3    9   5   4
## 8    3   5   6
## 37   6   9  10
## 56   7  11   9
```

```
# All columns from row 5
data[5, ]
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
## 5  0  1  1  3  3  1  3  5  2   4   4   7   6   5   3  10   8  10   6  17
##   V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38
## 5   9  14   9   7  13   9  12   6   7   7   9   6   3   2   2   4   2   0
##   V39 V40
## 5   1   1
```

```
# All rows from column 16
data[, 16]
```

```
##  [1]  4  4 15  8 10 15 13  9 11  6  3  8 12  3  5 10 11  4 11 13 15  5 14
## [24] 13  4  9 13  6  7  6 14  3 15  4 15 11  7 10 15  6  5  6 15 11 15  6
## [47] 11 15 14  4 10 15 11  6 13  8  4 13 12  9
```

A subtle point is that the last selection returned a vector instead of a data frame. This is because we selected only a single column. If you don't want this behavior do:

```
# All columns from row 5
d1 <- data[5, ]
class(d1)
```

```
## [1] "data.frame"
```

```
# All rows from column 16
d2 <- data[, 16]
class(d2)
```

```
## [1] "integer"
```

```
d3 <- data[, 16, drop=FALSE]
class(d3)
```

```
## [1] "data.frame"
```

Other functions you can call are `min`, `max`, `mean`, `sd` and `median` to get statistical values of interest:

```
# first row, all of the columns
patient_1 <- data[1, ]

# max inflammation for patient 1
max(patient_1)
```

```
## [1] 18
```

```
# max inflammation for patient 2
max(data[2, ])
```

```
## [1] 18
```

```
# minimum inflammation on day 7
min(data[, 7])
```

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

```
# mean inflammation on day 7
mean(data[, 7])
```

```
## [1] 3.8
```

```
# median inflammation on day 7
median(data[, 7])
```

```
## [1] 4
```

```
# standard deviation of inflammation on day 7
sd(data[, 7])
```

```
## [1] 1.725187
```

To do more complex calculations like `the maximum inflammation for all patients, or the average for each day?` we need to apply the function `max` or `mean` per row or column respectivelly. Luckily there is the function `apply` that applies a function for each one of the "margins", 1 for rows and 2 for columns:

```
args(apply) # args return NULL because it prints the information, but every funct\
ion must return something!
```

```
## function (X, MARGIN, FUN, ...)
## NULL
```

```
max_patient_inflammation <- apply(data, 1, max)
max_patient_inflammation
```

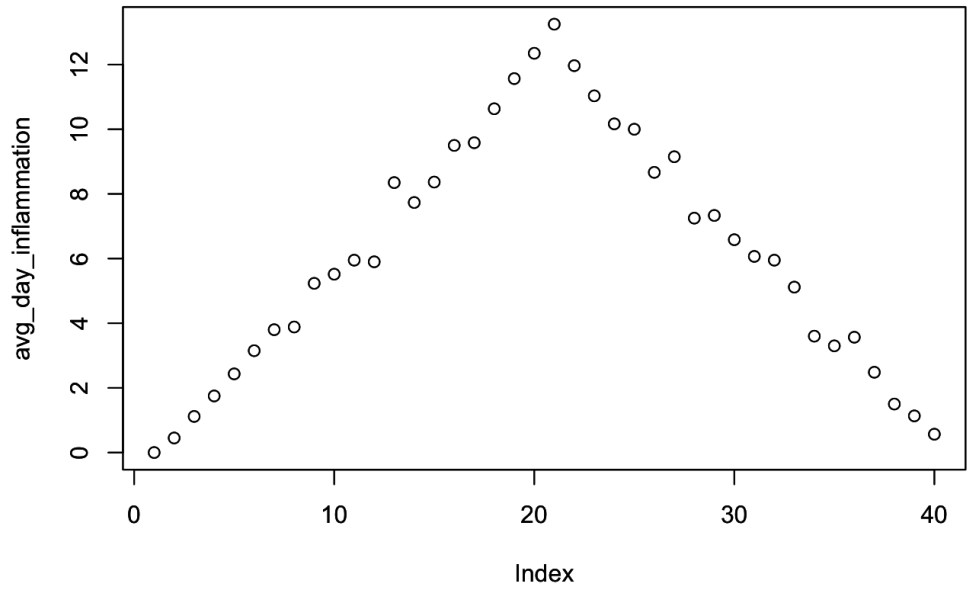
```
##  [1] 18 18 19 17 17 18 17 20 17 18 18 18 17 16 17 18 19 19 17 19 19 16 17
## [24] 15 17 17 18 17 20 17 16 19 15 15 19 17 16 17 19 16 18 19 16 19 18 16
## [47] 19 15 16 18 14 20 17 15 17 16 17 19 18 18
```

```
avg_day_inflammation <- apply(data, 2, mean)
avg_day_inflammation
```

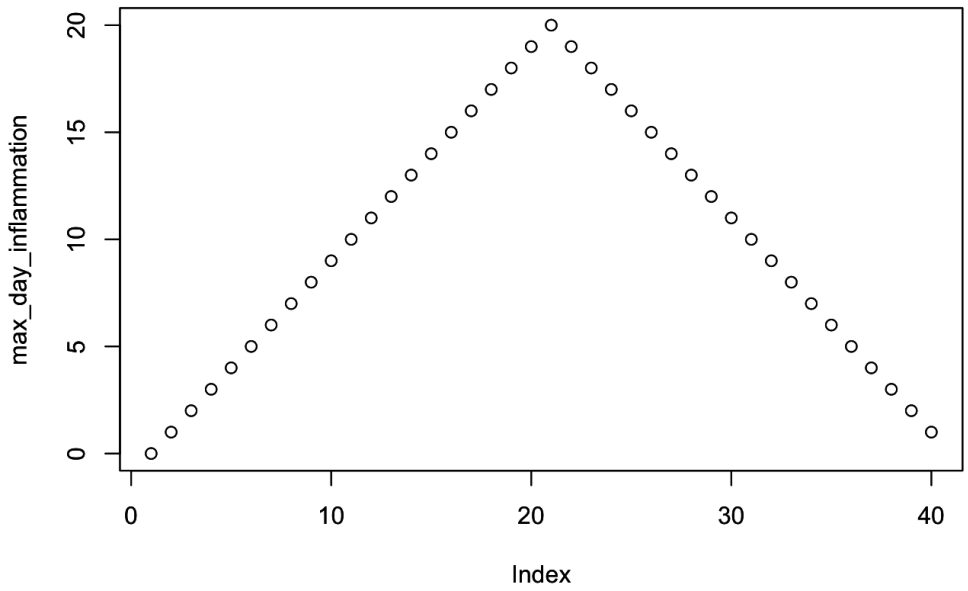
```
##         V1         V2         V3         V4         V5         V6
##  0.0000000  0.4500000  1.1166667  1.7500000  2.4333333  3.1500000
##         V7         V8         V9        V10        V11        V12
##  3.8000000  3.8833333  5.2333333  5.5166667  5.9500000  5.9000000
##        V13        V14        V15        V16        V17        V18
##  8.3500000  7.7333333  8.3666667  9.5000000  9.5833333 10.6333333
##        V19        V20        V21        V22        V23        V24
## 11.5666667 12.3500000 13.2500000 11.9666667 11.0333333 10.1666667
##        V25        V26        V27        V28        V29        V30
## 10.0000000  8.6666667  9.1500000  7.2500000  7.3333333  6.5833333
##        V31        V32        V33        V34        V35        V36
##  6.0666667  5.9500000  5.1166667  3.6000000  3.3000000  3.5666667
##        V37        V38        V39        V40
##  2.4833333  1.5000000  1.1333333  0.5666667
```

Now let's do some plotting

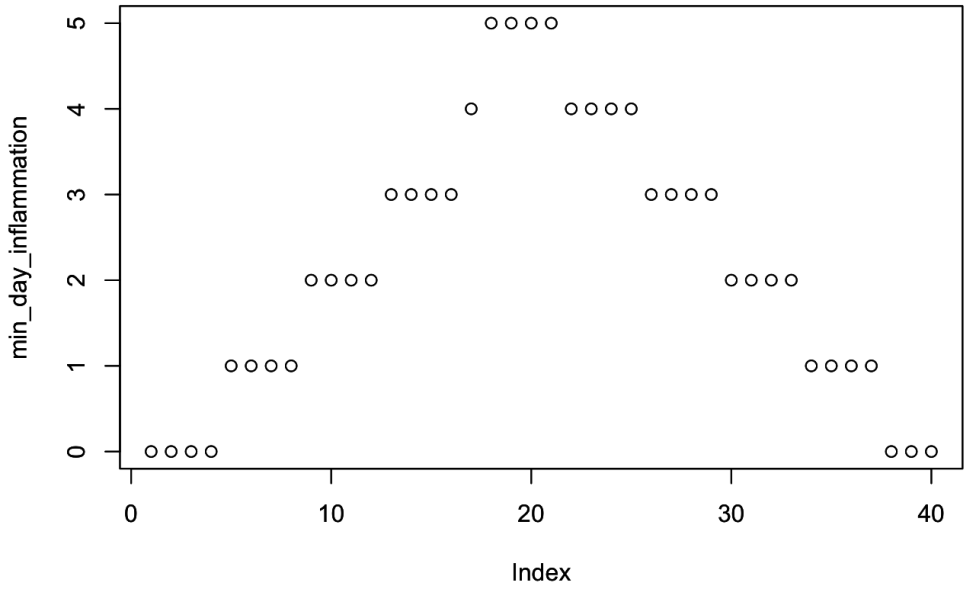
```
plot(avg_day_inflammation)
```




```
max_day_inflammation <- apply(data, 2, max)
plot(max_day_inflammation)
```




```
min_day_inflammation <- apply(data, 2, min)
plot(min_day_inflammation)
```



`plot` is a function with many arguments so you will probably need to study a lot of examples to do what you want (change an axis, name an axis, change the plot points and/or lines, add title, add grids, add legend, color the graph, add arrows and text etc.)

1.5 R Scripts

So far we have been typing directly into the R command line. What we could also do is save a sequence of commands in an R source file to run it at will. The way to do this is to have such a file with an `.R` extension and use the function `source` to run it.

If the source file contains an analysis from the beginning to the end it is a good practice to always clear your session of variables using `rm(list=ls())`. On the other hand if it is used as a library, for example to load some functions you have created, then you probably should not do it. You can also include other source files inside your current source file using the function (you guessed it): `source`

Because it is a good practice to have coding guidelines/conventions for standardizing the way you write your script to make it more readable, Google has some: Google's R Sytel Guide[4]

1.6 Functions

Let's learn how to create functions by creating a fuction `fahr_to_kelvin` that converts temperatures from Fahrenheit to Kelvin:

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

To run a function:

```
# freezing point of water
fahr_to_kelvin(32)
```

```
## [1] 273.15
```

```
# boiling point of water
fahr_to_kelvin(212)
```

```
## [1] 373.15
```

Let's also create a function that converts Kelvin to Celcius:

[4] https://google.github.io/styleguide/Rguide.xml

```
kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}

#absolute zero in Celsius
kelvin_to_celsius(0)
```

```
## [1] -273.15
```

We can also use functions inside functions

```
fahr_to_celsius <- function(temp) {
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}

# freezing point of water in Celsius
fahr_to_celsius(32.0)
```

```
## [1] 0
```

or we can obtain this result by function chaining:

```
# freezing point of water in Celsius
kelvin_to_celsius(fahr_to_kelvin(32.0))
```

```
## [1] 0
```

1.7 `for` loops

Like all the programming language R also has `for` loops to do recurring tasks. In general the syntax is:

```
for (variable in collection) {
  do things with variable
}
```

Let's do an example:

```
best_practice <- c("Let", "the", "computer", "do", "the", "work")

print_words <- function(sentence) {
  for (word in sentence) {
    print(word)
  }
}

print_words(best_practice)
```

```
## [1] "Let"
## [1] "the"
## [1] "computer"
## [1] "do"
## [1] "the"
## [1] "work"
```

or another example:

```
len <- 0
vowels <- c("a", "e", "i", "o", "u")
for (v in vowels) {
  len <- len + 1
}
# Number of vowels
len
```

```
## [1] 5
```

1.8 Making decisions (`if` & `else`)

To make decisions in your R scripts, R provides you with the standard if-else conditional statements

```
num <- 37
if (num > 100) {
  print("greater")
} else {
  print("not greater")
}
```

```
## [1] "not greater"
```

```
# or let's create a function that uses conditionals
sign <- function(num) {
  if (num > 0) {
    return(1)
  } else if (num == 0) {
    return(0)
  } else {
    return(-1)
  }
}

sign(-3)
```

```
## [1] -1
```

You can make decisions using the logical operators

- equal (==)
- greater than or equal to (>=),
- less than or equal to (<=),
- and not equal to (!=).

We can also combine tests. An ampersand, `&`, symbolizes the logical `and`. A vertical bar, `|`, symbolizes the logical `or`.

1.9 Datasets and statistics

By running the function `data()` we can see some datasets that are currently included in the R installation. To check for example the iris dataset, we can for example use the function:

```
head(iris)
```

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

To see the structure of it:

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 \
1 1 ...
```

For a statistical summary

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

For the Species column, one can observe that the summary function did not do the standard statistical calculation like it did with the other variables. From the `str` function we can see that the Species column is a special type of value called *factor*, which is what R uses to declare categorial or ordinal values. More on that in the next section.

We can also get the full attribute of `Sepal.Length` by name through the use of the `$` operator:

```
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

and run the standard statistics:

```
mean(iris$Sepal.Length)
```

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

```
median(iris$Sepal.Length)
```

```
## [1] 5.8
```

```
min(iris$Sepal.Length)
```

```
## [1] 4.3
```

```
max(iris$Sepal.Length)
```

```
## [1] 7.9
```

```
sd(iris$Sepal.Length)
```

```
## [1] 0.8280661
```

```
var(iris$Sepal.Length)
```

```
## [1] 0.6856935
```

```
range(iris$Sepal.Length)
```

```
## [1] 4.3 7.9
```

or other functions like:

```
sort(iris$Sepal.Length)
```

```
##   [1] 4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7 4.7 4.8 4.8 4.8 4.8 4.8 4.9
##  [18] 4.9 4.9 4.9 4.9 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.1 5.1
##  [35] 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.2 5.2 5.2 5.2 5.3 5.4 5.4 5.4 5.4 5.4
##  [52] 5.4 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.6 5.6 5.6 5.6 5.6 5.6 5.7 5.7 5.7
##  [69] 5.7 5.7 5.7 5.7 5.7 5.8 5.8 5.8 5.8 5.8 5.8 5.8 5.9 5.9 5.9 6.0 6.0
##  [86] 6.0 6.0 6.0 6.0 6.1 6.1 6.1 6.1 6.1 6.1 6.2 6.2 6.2 6.2 6.3 6.3 6.3
## [103] 6.3 6.3 6.3 6.3 6.3 6.3 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.5 6.5 6.5 6.5
## [120] 6.5 6.6 6.6 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.8 6.8 6.8 6.9 6.9 6.9
## [137] 6.9 7.0 7.1 7.2 7.2 7.2 7.3 7.4 7.6 7.7 7.7 7.7 7.7 7.7 7.9
```

```
length(iris$Sepal.Length)
```

```
## [1] 150
```

1.10 Factors

The factor() command is used to create and modify factors in R:

```
sex <- factor(c("male", "female", "female", "male"))
```

R will assign 1 to the level `"female"` and 2 to the level `"male"` (because `f` comes before `m`, even though the first element in this vector is `"male"`). You can check this by using the function `levels()`, and check the number of levels using `nlevels()`:

```
levels(sex)
```

```
## [1] "female" "male"
```

```
nlevels(sex)
```

```
## [1] 2
```

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., “low”, “medium”, “high”) or it is required by particular type of analysis. Additionally, specifying the order of the levels allows us to compare levels:

```
food <- factor(c("low", "high", "medium", "high", "low", "medium", "high"))
levels(food)
```

```
## [1] "high"   "low"    "medium"
```

```
food <- factor(food, levels = c("low", "medium", "high"))
levels(food)
```

```
## [1] "low"    "medium" "high"
```

```
min(food) ## doesn't work
```

```
## Error in Summary.factor(structure(c(1L, 3L, 2L, 3L, 1L, 2L, 3L), .Label = c("l\
ow", : 'min' not meaningful for factors
```

```
food <- factor(food, levels = c("low", "medium", "high"), ordered=TRUE)
levels(food)
```

```
## [1] "low"    "medium" "high"
```

```
min(food) ## works!
```

```
## [1] low
## Levels: low < medium < high
```

Acknowledgments

Parts of this chapter were taken and adapted from Software Carpentry Lessons[5] lessons for R. You can check them out to gain a more in depth knowledge of R.

1.11 Challenge

Air Passengers dataset

Use the dataset `AirPassengers` that comes with R and refers to number of passengers traveled every month from 1949 to 1960 in thousands. Because the dataset is of type `Time-Series` or `ts`, you can make it a data.frame through the following commands:

dn = list(paste("Y", as.character(1949:1960), sep = ""), month.abb) airmat = matrix(AirPassengers, 12, byrow = TRUE, dimnames = dn) air = as.data.frame(t(airmat))

Then try to answer the next questions/problems:

- Use the help functionality to try and learn about the functions used above.
- How many passengers traveled in average for the year 1951?
- Which is the maximum number of passengers for the months January and February?
- Calculate the summation per year and assign the result to a vector.
- Plot the vector nicely (names in axes, point and lines for the graph, title the graph, add grid lines)
- Repeat the last two bullets for every month for all the years.

Tip: to transform a row of the data frame to a vector you can use `unlist` (e.g. `unlist(air["Jan",])`)

[5] https://software-carpentry.org/lessons/

Chapter 2 - Introduction to Machine Learning

2.1 Definition

Machine Learning (ML) is a subset of Artificial Intelligence (AI) in the field of computer science that often uses statistical techniques to *give computers the ability to "learn" (i.e., progressively improve performance on a specific task) with data, without being explicitly programmed.*

Machine Learning is often closelly related, if not used as an alternate term, to fields like **Data Mining** (*the process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems*), **Pattern Recognition**, **Statistical Inference** or **Statistical Learning**. All these areas often employ the same methods and perhaps the name changes based on the practitioner's expertise or the application domain.

2.2 Main categories and tasks

The main ML tasks are typically classified into two broad categories, depending on whether there is "feedback" or a "teacher" available to the learning system or not.

- *Supervised Learning*: The system is presented with example inputs and their desired outputs provided by the "teacher" and the goal of the machine learning algorithm is to create a mapping from the inputs to the outputs. The mapping can be thought of as a function that if it is given as an input one of the training samples it should output the desired value.
- *Unsupervised Learning*: In the unsupervised learning case, the machine learning algorithm is not given any examples of desired output, and is left on its own to find structure in its input.

The main machine learning tasks and the ones we will examine in the present textbook, are separated based on what the system tries to accomplish in the end:

- *Classification*: inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more (multi-label classification) of these classes. This is typically tackled in a supervised manner. Spam filtering is an example of classification, where the inputs are email (or other) messages and the classes are "spam" and "not spam".
- *Regression*: also a supervised problem, the outputs are continuous rather than discrete.
- *Clustering*: a set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task.
- *Dimensionality Reduction*: simplifies inputs by mapping them into a lower-dimensional space. Topic modeling is a related problem, where a program is given a list of human language documents and is tasked with finding out which documents cover similar topics.

- *Association Rules learning* (or dependency modelling): Searches for relationships between inputs. For example, a supermarket might gather data on customer purchasing habits. Using association rule learning, the supermarket can determine which products are frequently bought together and use this information for marketing purposes. This is sometimes referred to as market basket analysis.

2.3 Survival guide to machine learning

So you are given a problem. With what methodology are you going to approach it?

First of all you have to understand the problem you are trying to solve with the data you have (or you are given) and ML algorithms. Is the problem defined formally? Unformally? Both? Are there any assumptions that need to be made? How one can go and solve it without ML? What is the motivation behind solving the problem?

The next step is to prepare the data to be used by your favorite ML algorithms since usually data do not come with the format you need or have other problems. One should consider: 1) what data are available, 2) if there are any missing values, 3) if any data need to be removed (for example the student ID, which does not have any predictive value, if we want to predict the average GPA of the student by the end of the semester), 4) to be organized, cleaned and sampled and 5) to be transformed either to other values due to domain knowledge (for example image pixel values to average image intensity or symmetry).

After the data is ready and prepared it is a good practice to select your validation strategy (split, k-fold cross validation, etc.) and test 10 or more basic ML algorithms. This is useful for getting a feeling on which algorithm can produce good results for the specific dataset.

From here on one can do algorithm tuning through hyper-parameter optimization methods, perform more advanced feature selection methods, use ensembles and more…sky is the limit.

Finally, it is good that all the above steps to be logged into an online or offline notepad in order to be disseminated to the stakeholders and be available for future use: what is the problem, what is the solution you derived, what are the results, are there any limitations to your approach and the final conclusions.

So to summarize the main steps are:

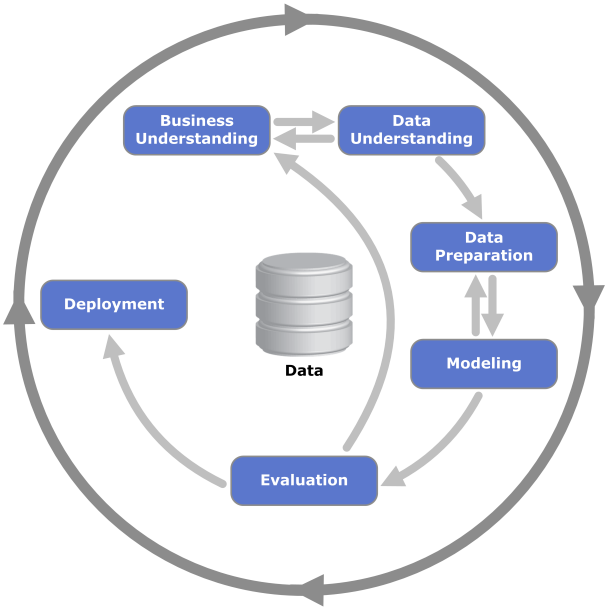
1. Problem definition
2. Data preparation
3. Algorithm application
4. Algorithm optimization
5. Result presentation

If someone wants more formally defined processes for working on data problems there are CRISP-DM and OSEMN. The steps for CRISP-DM[6] (check out the diagram below[7]) are:

[6] https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining

[7] By Kenneth Jensen - Own work based on: ftp://public.dhe.ibm.com/software/analytics/spss/documentation/modeler/18.0/en/ModelerCRISPDM.pdf (Figure 1), CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24930610

- *Business understanding*: understanding the project objectives and requirements from a business perspective, and then converting this knowledge into a data mining problem definition, and a preliminary plan designed to achieve the objectives.
- *Data understanding*: From an initial data collection and proceeds with activities in order to:
  - get familiar with the data,
  - identify data quality problems,
  - discover first insights into the data, or
  - detect interesting subsets to form hypotheses for hidden information.
- *Data preparation*: This phase deals with the construction of the final dataset from the initial raw data.
- *Modeling*: Application of various modeling techniques and parameter calibration to optimal values.
- *Evaluation*: Thorough evaluation of the model before final deployment.
- *Deployment*: The deployment can have many forms, from a generated report to a repeatable data mining process. Of ocurse we should mention that the creation of the model is generally not the end of a project.



CRISP-DM diagram

The steps for OSEMN[8] are:

- Obtain
- Scrub
- Explore
- Models
- Interpret

[8] http://www.dataists.com/2010/09/a-taxonomy-of-data-science/

Part II - Classification

Chapter 3 - Classification with Decision trees

3.1 Introduction

3.1.1 Introduction to Decision Trees

A Decision Tree is a Machine Learning algorithm that is used mainly for classification problems. It can be applied to categorical and continuous data. The main concept of the algorithm is simple; the data are split consecutively according to certain splitting criteria. Decision trees are computationally efficient when training and quite fast for classifying new instances. Furthermore, they are comprehensional when they are not too large, therefore they are frequently used for data exploration. A common pitfall is that they are sensitive to overfitting, which can be confronted by pruning or by adding further criteria when building the tree.

3.1.2 Decision Trees in R

We may use the rpart library to build decision trees in R:

```
library(rpart)
```

Furthermore, we shall use the library rpart.plot to plot the produced trees:

```
library(rpart.plot)
```

Building a new tree requires executing the rpart command:

```
model <- rpart(Target ~ ., method = "class", data = ..., minsplit = ..., minbucke\
t = ..., cp = ...)
```

We can plot the tree using `plot(model)` and `text(model, use.n = TRUE)`, or, alternatively, using the following command:

```
rpart.plot(model, extra = 104, nn = TRUE)
```

In order to check the parameters of rpart, we can run `?rpart`. Additionally, if we execute `?rpart.control`, we can also check the main parameters:

- `minsplit`: the minimum number of instances in a node so that it is split
- `minbucket`: the minimum allowed number of instances in each leaf of the tree
- `maxdepth`: the maximum depth of the tree
- `cp`: parameter that controls the complexity for a split and is set intuitively (the larger its value, the more probable to apply pruning to the tree)

3.2 Splitting Criteria and Decision Tree Construction

To compute the values of different splitting criteria, we will use as an example the training data of the following table, which refer to a problem of binary classification.

Outlook	Temperature	Humidity	Play
Sunny	Hot	High	No
Sunny	Hot	Low	No
Rainy	Hot	Low	Yes
Rainy	Cool	High	Yes
Rainy	Cool	Low	Yes
Rainy	Hot	Low	No
Rainy	Cool	Low	Yes
Sunny	Hot	High	No
Sunny	Cool	Low	Yes
Rainy	Hot	Low	Yes
Sunny	Cool	Low	Yes
Rainy	Hot	High	Yes
Rainy	Cool	Low	Yes
Sunny	Cool	High	No

We are going to apply a decision tree algorithm on the above dataset. We will answer to the following questions:

a) If we split using the Gini index, on which feature (out of Outlook, Temperature, and Humidity) should we perform the first split?

b) If we split using the Information gain, on which feature (out of Outlook, Temperature, and Humidity) should we perform the first split?

c) Build the full decision tree using the Gini index. Also, plot the tree.

3.2.1 Data and Library Imports

Initially, we read the data and import the required librares:

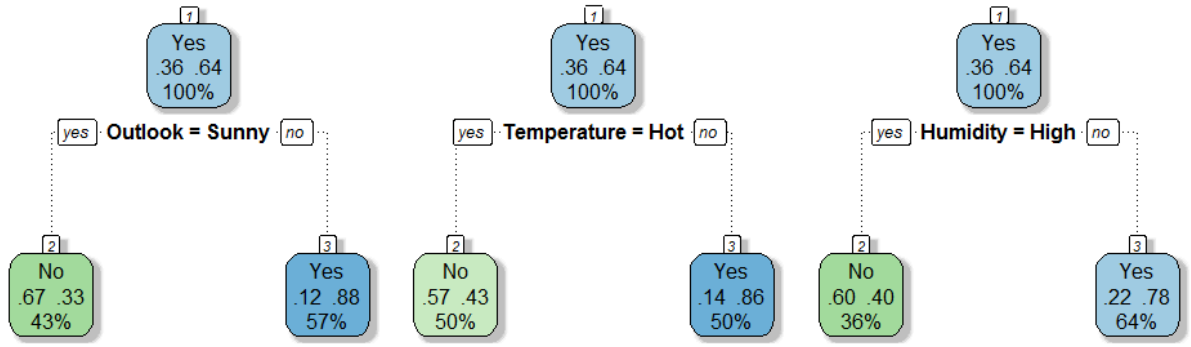
```
weather = read.csv("weather.txt")
library(rpart)
library(rpart.plot)
```

3.2.2 Splitting Criteria

To see the split for the variable Outlook, we execute the following command:

```
model <- rpart(Play ~ Outlook, method = "class", data = weather, minsplit = 1)
```

If we also run this command for Temperature and Humidity, and then plot all three with `rpart.plot(model, extra = 104, nn = TRUE))`, then the following visualizations are provided:



Intuitively, we can already understand which of the three splits will be selected, right?

3.2.2.1 Gini Index

We compute the Gini index for Outlook using the following formulas:

$$GINI(Sunny) = 1 - Freq(Play = No|Outlook = Sunny)^2 - Freq(Play = Yes|Outlook = Sunny)^2$$

$$GINI(Rainy) = 1 - Freq(Play = No|Outlook = Rainy)^2 - Freq(Play = Yes|Outlook = Rainy)^2$$

$$GINI_{Outlook} = Freq(Outlook = Sunny) \cdot GINI(Sunny) + Freq(Outlook = Rainy) \cdot GINI(Rainy)$$

The calculations provide:

$$GINI(Sunny) = 1 - (4/6)^2 - (2/6)^2 = 0.444$$

$$GINI(Rainy) = 1 - (1/8)^2 - (7/8)^2 = 0.219$$

$$GINI_{Outlook} = (6/14) \cdot 0.444 + (8/14) \cdot 0.219 = 0.315$$

Similarly, the GINI for Temperature and Humidity is $GINI_{Temp.} = 0.367$ and $GINI_{Temp.} = 0.394$, respectively. Hence, answering to question (a), the optimal first split using the Gini index is on Outlook.

We can also make these computations using R. For Outlook, we build the following frequency arrays:

```
absfreq = table(weather[, c(1, 4)])
freq = prop.table(absfreq, 1)
freqSum = rowSums(prop.table(absfreq))
```

We calculate the Gini index for Sunny and Rainy:

```
GINI_Sunny = 1 - freq["Sunny", "No"]^2 - freq["Sunny", "Yes"]^2
GINI_Rainy = 1 - freq["Rainy", "No"]^2 - freq["Rainy", "Yes"]^2
```

The total Gini for Outlook is computed by:

```
GINI_Outlook = freqSum["Sunny"] * GINI_Sunny + freqSum["Rainy"] * GINI_Rainy
```

3.2.2.2 Information Gain

We compute the Information Gain for Outlook using the following formulas:

$$Entropy(All) = -Freq(No) \cdot log(Freq(No)) - Freq(Yes) \cdot log(Freq(Yes))$$

$$Entropy(Sunny) = \\ - Freq(No|Sunny) \cdot log(Freq(No|Sunny)) - Freq(Yes|Sunny) \cdot log(Freq(Yes|Sunny))$$

$$Entropy(Rainy) = \\ - Freq(No|Rainy) \cdot log(Freq(No|Rainy)) - Freq(Yes|Rainy) \cdot log(Freq(Yes|Rainy))$$

$$GAIN_{Outlook} = \\ Entropy(All) - Freq(Sunny) \cdot Entropy(Sunny) - Freq(Rainy) \cdot Entropy(Rainy)$$

The calculations provide:

$$Entropy(All) = -(5/14) \cdot log(5/14) - (9/14) \cdot log(9/14) = 0.652$$

$$Entropy(Sunny) = -(4/6) \cdot log(4/6) - (2/6) \cdot log(2/6) = 0.637$$

$$Entropy(Rainy) = -(1/8) \cdot log(1/8) - (7/8) \cdot log(7/8) = 0.377$$

$$GAIN_{Outlook} = 0.652 - (6/14) \cdot 0.637 + (8/14) \cdot 0.377 = 0.164$$

Similarly, the Information Gain for Temperature and Humidity is $GAIN_{Temperature} = 0.105$ and $GAIN_{Temperature} = 0.071$, respectively. Hence, answering to question (a), the optimal first split using the Information Gain is on Outlook.

We can also make these computations using R. Initially, we compute the total entropy of the dataset:

```
freq = prop.table(table(weather[, c(4)]))
Entropy_All = - freq["No"] * log(freq["No"]) - freq["Yes"] * log(freq["Yes"])
```

Then, for Outlook, we build the following frequency arrays:

```
absfreq = table(weather[, c(1, 4)])
freq = prop.table(absfreq, 1)
freqSum = rowSums(prop.table(absfreq))
```

We calculate the entropy for Sunny and Rainy:

```
Entropy_Sunny =
    - freq["Sunny", "No"] * log(freq["Sunny", "No"])
    - freq["Sunny", "Yes"] * log(freq["Sunny", "Yes"])
Entropy_Rainy =
    - freq["Rainy", "No"] * log(freq["Rainy", "No"])
    - freq["Rainy", "Yes"] * log(freq["Rainy", "Yes"])
```

The total information gain for Outlook is computed by:

```
GAIN_Outlook =
    Entropy_All
    - freqSum["Sunny"] * Entropy_Sunny
    - freqSum["Rainy"] * Entropy_Rainy
```

3.2.3 Building the Decision Tree

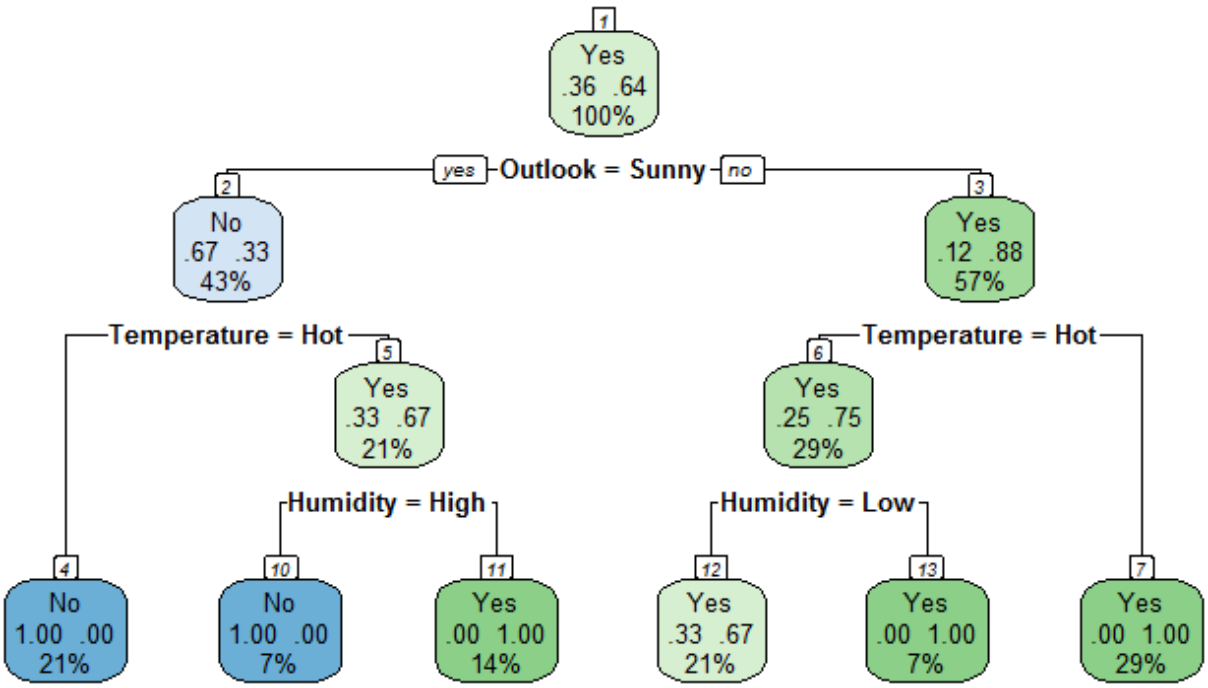
For question (c), we build a decision tree using rpart:

```
model <- rpart(
    Play ~ Outlook + Temperature + Humidity,
    method = "class",
    data = weather,
    minsplit = 1,
    minbucket = 1,
    cp = -1)
```

Furthermore, we can plot the tree with the following command:

```
rpart.plot(model, extra = 104, nn = TRUE)
```

The tree is shown in the following figure:



3.3 Application with Pruning and Evaluation Metrics

A decision tree classifier can also be applied on datasets with continuous predictor variables, such as the iris dataset. From the iris dataset, we shall only keep the first 2 columns of the dataset and drop the last 50 instances, so that we now have a binary classification problem with two predictor features and one target class. We can perform these transformations in R as follows:

```
iris2 = iris[, c(1, 2, 5)]
iris2$Species[c(101:150)] = iris2$Species[c(21:70)]
iris2$Species = factor(iris2$Species)
```

After that, we split the dataset into training and testing data:

```
trainingdata = iris2[c(1:40, 51:90, 101:140),]
testdata = iris2[c(41:50, 91:100, 141:150),]
```

We will answer to the following questions:

a) Build a decision tree using the training data (the minsplit parameter of rpart should be set to 20, which is the default).

b) Apply the model on testdata and compute the precision, the recall, and the f-measure for the two classes.

c) Build two decision trees, one for minsplit equal to 10 and one for minsplit equal to 30, apply them on testdata and compute the precision, the recall, and the f-measure for the two classes.

d) Compare the three models given their f-measure for class versicolor.

3.3.1 Decision Tree Training and Testing

We train the decision tree and plot it using the following commands (questions (a), (c)):

```
model <- rpart(
    Species ~ .,
    method = "class",
    data = trainingdata,
    minsplit = 20)
rpart.plot(model, extra = 104, nn = TRUE)
```

The . in the formula stands for all the remaining variables in the data frame `trainingdata`. We can also execute our model on the test set using the commands:

```
xtest = testdata[,1:2]
ytest = testdata[,3]
pred = predict(model, xtest, type="class")
```

3.3.2 Evaluation Metrics

We can produce the confusion matrix and compute useful metrics with the following commands (questions (b), (c)):

```
library(MLmetrics)
cm = ConfusionMatrix(pred, ytest)
accuracy = Accuracy(pred, ytest)
precision = Precision(ytest, pred, 'versicolor')
recall = Recall(ytest, pred, 'versicolor')
f1 = F1_Score(ytest, pred, 'versicolor')
data.frame(precision, recall, f1)
```

(alternatively we can compute TP, FP, TN, and FN and find precision as TP/(TP + FP) and recall as TP/(TP + FN))

Finally, the F-measure for the 3 models is shown in the following array (question (d)):

Decision Tree	F-Measure
minsplit = 10	0.833
minsplit = 20	0.947
minsplit = 30	0.857

3.4 Exercise

You are given the training data of the following array for a binary classification problem.

CustomerID	Sex	CarType	Budget	Insurance
1	M	Family	Low	No
2	M	Sport	Medium	No
3	M	Sport	Medium	No
4	M	Sport	High	No
5	M	Sport	VeryHigh	No
6	M	Sport	VeryHigh	No
7	F	Sport	Low	No
8	F	Sport	Low	No
9	F	Sport	Medium	No
10	F	Sedan	High	No
11	M	Family	High	Yes
12	M	Family	VeryHigh	Yes
13	M	Family	Medium	Yes
14	M	Sedan	VeryHigh	Yes
15	F	Sedan	Low	Yes
16	F	Sedan	Low	Yes
17	F	Sedan	Medium	Yes
18	F	Sedan	Medium	Yes
19	F	Sedan	Medium	Yes
20	F	Sedan	High	Yes

Calculate the Gini index for:

a) all the training instances

b) the feature CustomerID

c) the feature Sex

d) the feature CarType using multiway split

e) the feature Budget using multiway split

Which of these should be used for the first split according to the Gini index? Hint-question: Explain why CustomerID should not be used as a feature for splitting even though it has the lowest Gini index.

Chapter 4 - Classification with Naive Bayes

4.1 Introduction

4.1.1 Introduction to Naive Bayes

Naive Bayes is a machine learning algorithm that is based on the Bayes theorem as well as the assumption of independence between the features of the dataset. Given a dataset with a set of features $x_1, x_2, \ldots, x_n$ and a class feature c, the Bayes theorem defines the probability that an instance $\{x_1, x_2, \ldots, x_n\}$ belongs to c as the combination of the probabilities $P(x_1|c), P(x_2|c), \ldots, P(x_n|c)$:

$$P(c|x) = \frac{P(x_1|c) \cdot P(x_2|c) \cdot \cdots \cdot P(x_n|c) \cdot P(c)}{P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)}$$

where we notice that the probability of the set of features is given by their product, as their values are independent from one another.

4.1.2 Naive Bayes in R

We may use the e1071 library to build the probabilistic Naive Bayes model in R:

```
library(e1071)
```

Building a model requires executing the `naiveBayes` command:

```
model <- naiveBayes(Target ~ ., data = ..., laplace = ...)
```

where parameter `laplace` is used to select laplace smoothing.

Given a trained model, we can predict the class for a new instance using the command:

```
predict(model, trvalue)
```

while if we further add the parameter `type = "raw"`, we are also provided with the posterior probabilities.

4.2 Naive Bayes Model Construction and Classification

We will use the data of the following table for a problem of binary classification:

Weather	Day	HighTraffic
Hot	Vacation	No
Cold	Work	Yes
Normal	Work	No
Cold	Weekend	Yes
Normal	Weekend	Yes
Cold	Work	No
Hot	Work	No
Hot	Vacation	Yes

We will answer to the following questions:

a) Using the Naive Bayes Classifier, in which class would you classify a new instance with values (Weather, Day) = (Hot, Vacation)?

b) Using the Naive Bayes Classifier, in which class would you classify a new instance with values (Weather, Day) = (Hot, Weekend)?

c) Repeat question (b) using Laplace smoothing.

d) Build the full model in R and repeat question (a).

e) Build the full model in R using laplace smoothing and repeat question (b).

4.2.1 Data and Library Imports

Initially, we read the data and import the required librares:

```
traffic = read.csv("traffic.txt")
library(e1071)
```

4.2.2 Probability Calculation

For question (a), we compute the probability using the following formulas:

$$P(Yes|Hot, Vacation) = \frac{P(Hot|Yes) \cdot P(Vacation|Yes) \cdot P(Yes)}{P(Hot) \cdot P(Vacation)} = \frac{1/4 \cdot 1/4 \cdot 1/2}{3/8 \cdot 2/8} = 1/3$$

$$P(No|Hot, Vacation) = \frac{P(Hot|No) \cdot P(Vacation|No) \cdot P(No)}{P(Hot) \cdot P(Vacation)} = \frac{2/4 \cdot 1/4 \cdot 1/2}{3/8 \cdot 2/8} = 2/3$$

Since $P(No|Hot, Vacation) > P(Yes|Hot, Vacation)$, the algorithm classifies the instance to No.

Similarly, for question (b):

$$P(Yes|Hot, Weekend) = \frac{P(Hot|Yes) \cdot P(Weekend|Yes) \cdot P(Yes)}{P(Hot) \cdot P(Weekend)} = \frac{1/4 \cdot 2/4 \cdot 1/2}{3/8 \cdot 2/8} = 2/3$$

$$P(No|Hot, Weekend) = \frac{P(Hot|No) \cdot P(Weekend|No) \cdot P(No)}{P(Hot) \cdot P(Weekend)} = \frac{2/4 \cdot 0/4 \cdot 1/2}{3/8 \cdot 2/8} = 0$$

Since $P(No|Hot, Weekend) < P(Yes|Hot, Weekend)$, the algorithm classifies the instance to Yes.

Concerning question (c), we will re-compute the probabilities, however adding 1 as the Laplacian parameter. So, given the new probabilities are $P(Hot|Yes) = (1+1)/(4+3) = 2/7$, $P(Weekend|Yes) = (2+1)/(4+3) = 3/7$, $P(Hot|No) = (2+1)/(4+3) = 3/7$, and $P(Weekend|No) = (0+1)/(4+3) = 1/7$, we compute:

$$P(Yes|Hot, Weekend) = \frac{P(Hot|Yes) \cdot P(Weekend|Yes) \cdot P(Yes)}{P(Hot) \cdot P(Weekend)} = \frac{2/7 \cdot 3/7 \cdot 1/2}{3/8 \cdot 2/8} = 32/49$$

$$P(No|Hot, Weekend) = \frac{P(Hot|No) \cdot P(Weekend|No) \cdot P(No)}{P(Hot) \cdot P(Weekend)} = \frac{3/7 \cdot 1/7 \cdot 1/2}{3/8 \cdot 2/8} = 16/49$$

Since $P(No|Hot, Weekend) < P(Yes|Hot, Weekend)$, the algorithm classifies the instance to Yes.

4.2.3 Building the Naive Bayes Model

4.2.3.1 Building Simple Model

For question (d), we build a naive bayes model:

```
model <- naiveBayes(HighTraffic ~ ., data = traffic)
```

Furthermore, we can print the model:

```
print(model)
```

The output is shown below:

```
Naive Bayes Classifier for Discrete Predictors
Call: naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:
Y
 No Yes
0.5 0.5

Conditional probabilities:
     Weather
Y     Cold  Hot Normal
  No  0.25 0.50   0.25
  Yes 0.50 0.25   0.25

     Day
Y     Vacation Weekend Work
  No      0.25    0.00 0.75
  Yes     0.25    0.50 0.25
```

We may now re-compute question (a):

```
trvalue <- data.frame(Weather = factor("Hot", levels(traffic$Weather)),  Day = fa\
ctor("Vacation", levels(traffic$Day)))
predict(model, trvalue)
```

We can also get the relevant probabilities:

```
predict(model, trvalue, type = "raw")
```

4.2.3.2 Building Model with Laplace Smoothing

For question (c), we build a naive bayes model with laplace smoothing:

```
model <- naiveBayes(HighTraffic ~ ., data = traffic, laplace = 1)
```

We may now re-compute question (a):

```
trvalue <- data.frame(Weather = factor("Hot", levels(traffic$Weather)),  Day = fa\
ctor("Weekend", levels(traffic$Day)))
predict(model, trvalue)
```

We can also get the relevant probabilities:

```
predict(model, trvalue, type = "raw")
```

4.3 Naive Bayes Application with Evaluation

We will apply the Naive Bayes classifier on the dataset HouseVotes84. Initially, we import the dataset, ignoring any instances with missing values:

```
data(HouseVotes84, package = "mlbench")
votes = na.omit(HouseVotes84)
```

Furthermore, we import the libraries required to execute and evaluate the model:

```
library(e1071)
library(MLmetrics)
library(ROCR)
```

After that, we split the dataset into training and testing data:

```
trainingdata = votes[1:180,]
testingdata = votes[181:232,]
```

We will answer to the following questions:

a) Build a Naive Bayes model using the training data and apply it on the test data.

b) Compute the precision, the recall and the F-measure for the test data for the class democrat.

c) Build the ROC curve for the model.

4.3.1 Building and Applying the Naive Bayes Model

For question (a), we train a the naive bayes model:

```
model <- naiveBayes(Class ~ ., data = trainingdata)
```

After that, we can apply the model on the test set:

```
xtest = testingdata[,-1]
ytest = testingdata[,1]
pred = predict(model, xtest)
predprob = predict(model, xtest, type = "raw")
```

4.3.2 Metrics Computation and ROC Curve Plotting

We can see the confusion matrix and calculate useful metrics with the following commands (question (b)):

```
ConfusionMatrix(ytest, pred)
Precision(ytest, pred, "democrat")
Recall(ytest, pred)
```

Plotting the ROC curve (question (c)) initially requires computing TPR and FRP with the following commands:

```
pred_obj = prediction(predprob[,1], ytest, label.ordering = c("republican", "demo\
crat"))
ROCcurve <- performance(pred_obj, "tpr", "fpr")
```

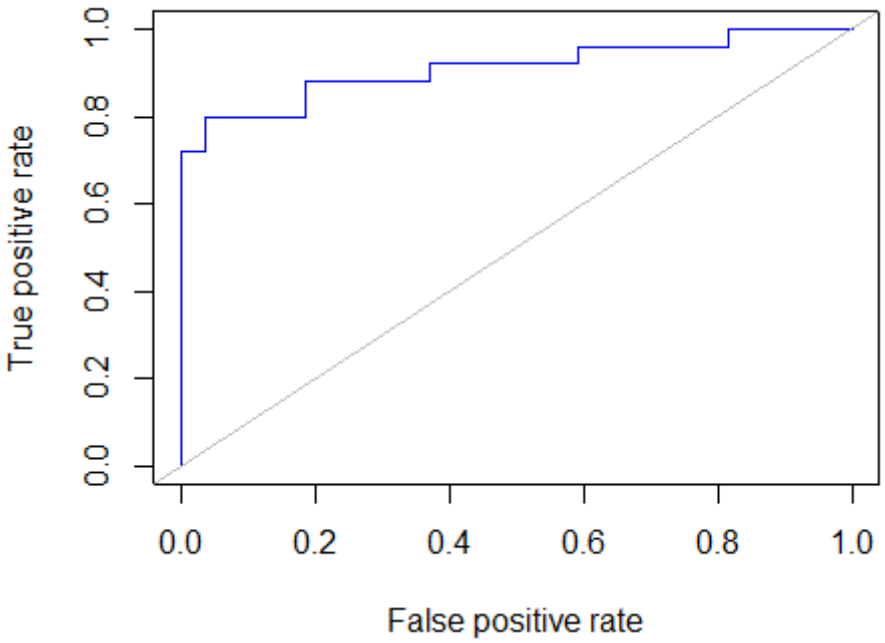
If we plot the `ROCcurve` object, we shall see TPR and FRP and the corresponding thresholds.

Finally, we can plot the curve as follows:

```
plot(ROCcurve, col = "blue")
abline(0,1, col = "grey")
```

And find the area under the curve using the command:

```
performance(pred_obj, "auc")
```



ROC curves

In statistics, a receiver operating characteristic (ROC), or ROC curve, is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings

An ROC curve demonstrates several things:

- It shows the tradeoff between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in specificity).
- The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the classifier.
- The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test.
- The slope of the tangent line at a cutpoint gives the likelihood ratio (LR) for that value of the test.
- The area under the curve is a measure of text accuracy. This is discussed further in the next section.

Accuracy is measured by the area under the ROC curve. An area of 1 represents a perfect test; an area of .5 represents a worthless test. A rough guide for classifying the accuracy of a diagnostic test is the traditional academic point system:

- .90-1 = excellent (A)
- .80-.90 = good (B)
- .70-.80 = fair (C)
- .60-.70 = poor (D)
- .50-.60 = fail (F)

Chapter 5 - Classification with k-Nearest Neighbors

5.1 Introduction

5.1.1 Introduction to k-Nearest Neighbors

k-Nearest Neighbors (kNN) is a machine learning algorithm, where the prediction for a new instance depends on its k nearest instances. As a result, the class for a new instance is determined by majority vote given the class of its k nearest instances.

5.1.2 k-Nearest Neighbors in R

We may use the class library to apply the kNN model in R:

```
library(class)
```

A new instance may be classified using the command `knn`:

```
knn(X_train, X_test, Y_train, k = 1, prob = TRUE)
```

where we have to provide the training data (`X_train`, `Y_train`), the new instances (`X_test`), and the value of k. If we also add the parameter `prob = TRUE`, then the probabilities for the class are also given.

5.2 k-Nearest Neighbors Model Construction and Classification

We will use the data of the following table for a problem of binary classification:

X1	X2	Y
0.7	0.7	A
0.7	0.8	A
0.6	0.6	A
0.5	0.5	A
0.5	0.6	A
0.5	0.7	A
0.5	0.8	A
0.7	0.5	B
0.8	0.7	B
0.8	0.5	B

X1	X2	Y
0.8	0.6	B
1.0	0.3	B
1.0	0.5	B
1.0	0.6	B

We will answer to the following questions:

a) Plot the data with different colors/symbols for each class.

b) Using kNN with `k = 1`, in which class would you classify a new instance with values `(X1, X2) = (0.7, 0.4)`?

c) Repeat question (b) using `k = 5`.

d) Using kNN with `k = 1`, in which class would you classify a new instance with values `(X1, X2) = (0.7, 0.6)`?

5.2.1 Data and Library Imports

Initially, we read the data and import the required librares:

```
knndata = read.csv("knndata.txt")
library(class)
```

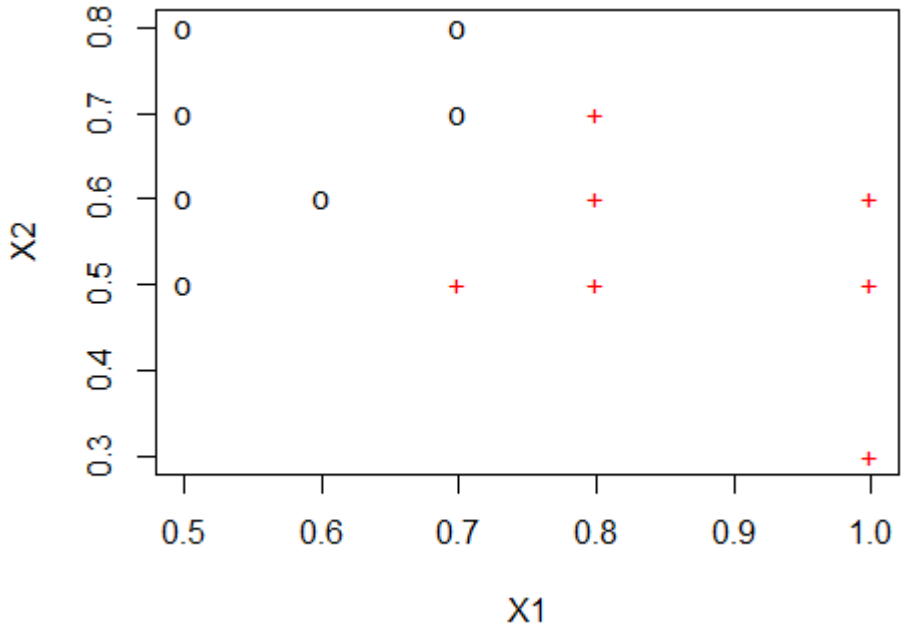
5.2.2 Algorithm Application

We initially split the dataset as follows:

```
X_train = knndata[,c("X1","X2")]
Y_train = knndata$Y
```

For question (a) we execute the command:

```
plot(X_train, col = Y_train, pch = c("o","+")[Y_train])
```



When `k = 1` (question (b)), we have to find the nearest instance (using euclidean distance) to (0.7, 0.4), which is (0.7, 0.5) that belongs to class B. So, we execute the command:

```
knn(X_train, c(0.7, 0.4), Y_train, k = 1, prob = TRUE)
```

and the new instance is classified to class B with probability 1.

When `k = 5` (question (c)), we have to find the 5 nearest instances to (0.7, 0.4), which are (0.7, 0.5), (0.8, 0.5), (0.6, 0.6), (0.8, 0.6), (0.7, 0.7) and belong to classes B, B, A, B, A respectively. We note that 3 of them (out of 5) belong to class B. We may execute the command:

```
knn(X_train, c(0.7, 0.4), Y_train, k = 5, prob = TRUE)
```

and the new instance is classified to class B with probability 0.5.

For question (d), we would like to find the nearest instance to (0.7, 0.6). We observe that instances (0.7, 0.5), (0.7, 0.5) (0.6, 0.6), (0.8, 0.6), (0.7, 0.7), (0.7, 0.7) (that belong to classes B, B, A, B, A, B respectively) all have the same distance from (0.7, 0.6). So (and since the parameter `use.all` of kNN is `TRUE`), the algorithm will use all of these instances. We note that 4 of them (out of 6) belong to class B. We may execute the command:

```
knn(X_train, c(0.7, 0.6), Y_train, k = 1, prob = TRUE)
```

and the new instance is classified to class B with probability 0.66.

5.3 k-Nearest Neighbors Real World Example

In this sub-section we will apply k-NN in the breast cancer Wisconsin dataset.

Breast Cancer Wisconsin (Original) Data Set

For this example we are going to use the Breast Cancer Wisconsin (Original) Data Set[9] and in particular the breast-cancer-wisconsin.data file[10] from the UCI Machine Learning Repository[11].

5.3.1 Data Handling

First we are going to load the dataset as a dataframe. We are assuming that the current working directory is in the same directory where the dataset is stored. We add the `sep` option because the default separator is the empty string. In addition, as one can observe from the dataset instructions, the missing values are denoted with `?`. To check the documentation of the `read.table` function use the command `?read.table`. Next we will split the dataset into training and validation datasets. The steps for loading and splitting the dataset to training and validation are the following:

```
rm(list=ls())
# download the dataset
fileURL <- "http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cance\
r-wisconsin/breast-cancer-wisconsin.data"
download.file(fileURL, destfile="breast-cancer-wisconsin.data", method="curl")
# read the data
data <- read.table("breast-cancer-wisconsin.data", na.strings = "?", sep=",")
# remove the id column
data <- data[,-1]
# put names in the columns (attributes)
names(data) <- c("ClumpThickness",
"UniformityCellSize",
"UniformityCellShape",
"MarginalAdhesion",
"SingleEpithelialCellSize",
"BareNuclei",
"BlandChromatin",
"NormalNucleoli",
"Mitoses",
"Class")
# make the class a factor
data$Class <- factor(data$Class, levels=c(2,4), labels=c("benign", "malignant"))
# set the seed
set.seed(1234)
# split the dataset
ind <- sample(2, nrow(iris), replace=TRUE, prob=c(0.7, 0.3))
trainData <- data[ind==1,]
validationData <- data[ind==2,]
```

5.3.2 Training & Prediction

We load the k-NN algorithm as before.

[9] http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29
[10] http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data
[11] http://archive.ics.uci.edu/ml/index.html

```
library(class)
```

Because all the attributes are between 1 and 10 there is no need to do normalization between 0 and 1, since no attribute will dominate the others in the distance calculation of kNN. Because kNN accepts the training and testing datasets without the target column, which puts in a 3rd argument, we are going to do some data manipulation to have the data the way the knn function likes them (look the manual with ?knn). Also, because no missing values are allowed in kNN, let's remove those too.

```
trainData <- trainData[complete.cases(trainData),]
validationData <- validationData[complete.cases(validationData),]
trainDataX <- trainData[,-ncol(trainData)]
trainDataY <- trainData$Class
validationDataX <- validationData[,-ncol(trainData)]
validationDataY <- validationData$Class
```

Lets predict, since there is no need to training when using kNN. The training instances are the model.

```
prediction = knn(trainDataX, validationDataX, trainDataY, k = 1)
```

You can play with the values of k to look for a better model.

5.3.3 Evaluation

Make the predictions for the validation dataset and print the confusion matrix:

```
cat("Confusion matrix:\n")
xtab = table(prediction, validationData$Class)
print(xtab)
cat("\nEvaluation:\n\n")
accuracy = sum(prediction == validationData$Class)/length(validationData$Class)
precision = xtab[1,1]/sum(xtab[,1])
recall = xtab[1,1]/sum(xtab[1,])
f = 2 * (precision * recall) / (precision + recall)
cat(paste("Accuracy:\t", format(accuracy, digits=2), "\n",sep=" "))
cat(paste("Precision:\t", format(precision, digits=2), "\n",sep=" "))
cat(paste("Recall:\t\t", format(recall, digits=2), "\n",sep=" "))
cat(paste("F-measure:\t", format(f, digits=2), "\n",sep=" "))
```

and the output:

```
## Confusion matrix:
##
## prediction  benign malignant
##   benign        99         4
##   malignant      6        60
##
## Evaluation:
## Accuracy:     0.94
## Precision:    0.94
## Precision:    0.94
## F-measure:    0.95
```

5.3.4 Cross Validation

For performing cross-validation we will use the `caret` package. Here[12] you can find a quick guide to caret.

5.3.4.1 Split data in two groups

The function `createDataParitiion` does a stratified random split of the data. Similar to what we did above ourselves (not stratified though). Then we will use the `train` function to build the kNN model.

```
library(caret)
library(mlbench)
data(Sonar)
set.seed(107)
inTrain <- createDataPartition(y = Sonar$Class, p = .75, list = FALSE)
training <- Sonar[ inTrain,]
testing <- Sonar[-inTrain,]
kNNFit <- train(Class ~ .,
                data = training,
                method = "knn",
                preProc = c("center", "scale"))
print(kNNFit)
```

```
## k-Nearest Neighbors
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 157, 157, 157, 157, 157, 157, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.7683412  0.5331399
##   7  0.7559100  0.5053696
##   9  0.7378175  0.4715006
```

[12] https://cran.r-project.org/web/packages/caret/vignettes/caret.pdf

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

We can also search for the best k value given the training dataset.

```
kNNFit1 <- train(Class ~ .,
                 data = training,
                 method = "knn",
                 tuneLength = 15,
                 preProc = c("center", "scale"))
print(kNNFit1)
```

```
## k-Nearest Neighbors
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 157, 157, 157, 157, 157, 157, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.7477453  0.4840235
##    7  0.7189901  0.4225515
##    9  0.7211797  0.4275156
##   11  0.7140987  0.4150135
##   13  0.7031182  0.3932055
##   15  0.7034819  0.3945755
##   17  0.6914316  0.3698916
##   19  0.6855830  0.3588189
##   21  0.6847459  0.3619330
##   23  0.6821917  0.3571894
##   25  0.6626137  0.3186673
##   27  0.6551801  0.3042504
##   29  0.6660760  0.3291024
##   31  0.6643681  0.3273283
##   33  0.6697744  0.3389183
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 5.
```

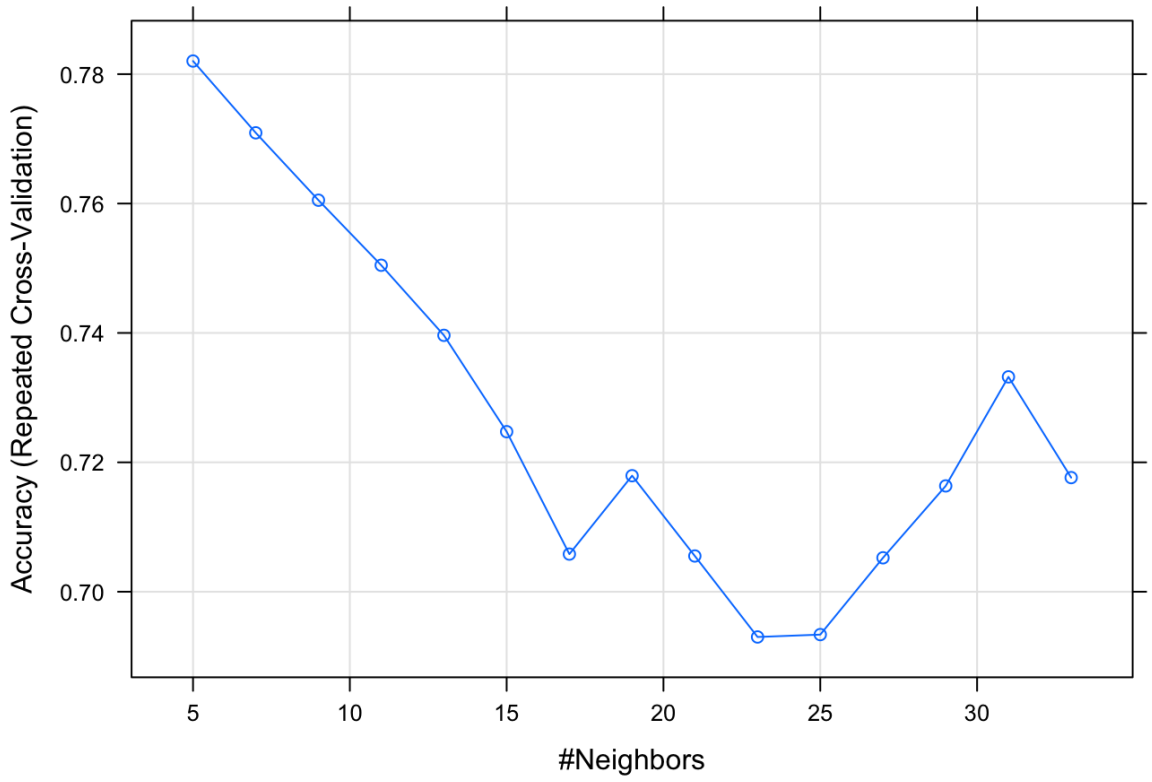
To create a 10-fold cross-validation based search of k, repeated 3 times we have to use the function `trainControl`:

```
ctrl <- trainControl(method = "repeatedcv", repeats = 3)
kNNFit2 <- train(Class ~ .,
                 data = training,
                 method = "knn",
                 tuneLength = 15,
                 trControl = ctrl,
                 preProc = c("center", "scale"))
print(kNNFit2)
plot(kNNFit2)
```

the output:

```
## k-Nearest Neighbors
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 142, 142, 142, 142, 141, 142, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.7820261  0.5548350
##    7  0.7709314  0.5321546
##    9  0.7605147  0.5111479
##   11  0.7504657  0.4924263
##   13  0.7396324  0.4705332
##   15  0.7247386  0.4401370
##   17  0.7058170  0.4006357
##   19  0.7179330  0.4246010
##   21  0.7055229  0.4003491
##   23  0.6930065  0.3779050
##   25  0.6933742  0.3791319
##   27  0.7052451  0.4028364
##   29  0.7163562  0.4277789
##   31  0.7332108  0.4629731
##   33  0.7176389  0.4335254
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 5.
```

and the plot:



For predictions:

```
knnPredict <- predict(kNNFit2, newdata = testing )
#Get the confusion matrix to see accuracy value and other parameter values
confusionMatrix(knnPredict, testing$Class )
```

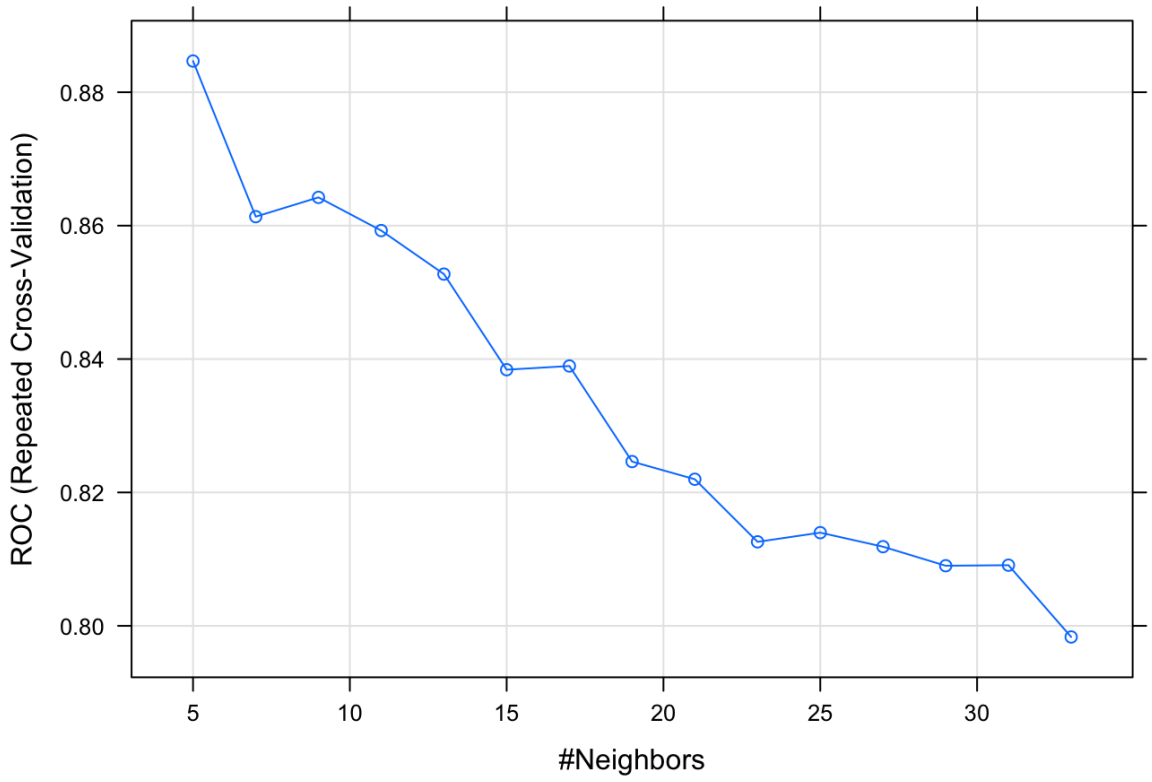
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 25  9
##          R  2 15
##
##                Accuracy : 0.7843
##                  95% CI : (0.6468, 0.8871)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 0.0001502
##
##                   Kappa : 0.56
##  Mcnemar's Test P-Value : 0.0704404
##
##             Sensitivity : 0.9259
##             Specificity : 0.6250
##          Pos Pred Value : 0.7353
##          Neg Pred Value : 0.8824
##              Prevalence : 0.5294
##          Detection Rate : 0.4902
```

```
##    Detection Prevalence : 0.6667
##       Balanced Accuracy : 0.7755
##
##        'Positive' Class : M
##
```

Adding more information in the output:

```
ctrl <- trainControl(method = "repeatedcv", repeats = 3, classProbs=TRUE, summary\
Function = twoClassSummary)
kNNFit4 <- train(Class ~ .,
                 data = training,
                 method = "knn",
                 tuneLength = 15,
                 trControl = ctrl,
                 preProc = c("center", "scale"))
kNNFit4
plot(kNNFit4)
```

```
## k-Nearest Neighbors
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 142, 142, 142, 141, 141, 141, ...
## Resampling results across tuning parameters:
##
##   k   ROC        Sens       Spec
##    5  0.8846768  0.8773148  0.6988095
##    7  0.8613467  0.8726852  0.6678571
##    9  0.8642361  0.8601852  0.6500000
##   11  0.8592634  0.8518519  0.6494048
##   13  0.8527364  0.8282407  0.6250000
##   15  0.8384011  0.8009259  0.5994048
##   17  0.8389550  0.8004630  0.5904762
##   19  0.8246280  0.8087963  0.5988095
##   21  0.8219783  0.8125000  0.6125000
##   23  0.8125951  0.8055556  0.6125000
##   25  0.8139716  0.7898148  0.6351190
##   27  0.8118676  0.7893519  0.6535714
##   29  0.8090112  0.7842593  0.6577381
##   31  0.8090939  0.7958333  0.6625000
##   33  0.7983300  0.7606481  0.6815476
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was k = 5.
```



And finally how to use bootstrap with the caret package:

```
knnFit5 <- train(Class ~ .,
                 data = training,
                  method = "knn",
                  preProcess = c("center", "scale"),
                  tuneLength = 10,
                  trControl = trainControl(method = "boot"))
knnPredict2 <- predict(knnFit5, newdata = testing)
#Get the confusion matrix to see accuracy value and other parameter values
confusionMatrix(knnPredict2, testing$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 25  9
##          R  2 15
##
##                Accuracy : 0.7843
##                  95% CI : (0.6468, 0.8871)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 0.0001502
##
##                   Kappa : 0.56
##  Mcnemar's Test P-Value : 0.0704404
##
```

```
##             Sensitivity : 0.9259
##             Specificity : 0.6250
##          Pos Pred Value : 0.7353
##          Neg Pred Value : 0.8824
##              Prevalence : 0.5294
##          Detection Rate : 0.4902
##    Detection Prevalence : 0.6667
##       Balanced Accuracy : 0.7755
##
##        'Positive' Class : M
##
```

Reference

http://rstudio-pubs-static.s3.amazonaws.com/16444_caf85a306d564eb490eebdbaf0072df2.html[13]

[13] http://rstudio-pubs-static.s3.amazonaws.com/16444_caf85a306d564eb490eebdbaf0072df2.html

Chapter 6 - Classification with Support Vector Machines

6.1 Introduction

6.1.1 Introduction to Support Vector Machines

Support Vector Machines (SVM) constitute an algorithm that finds a linear hyperplane that separates the data. The problem can be defined as maximizing the distance between the hyperplane and the data (margin):

Maximize $Margin = 2/||w^2||$ given that $f(x) = \{1 if w \cdot x + b \geq 1, -1 if w \cdot x + b \leq 1\}$

If the data are not linearly separable, then they can be mapped to a space of different (usually larger) dimensionality, in order to make them linearly separable.

6.1.2 Support Vector Machines in R

We may use the e1071 library to build an SVM model in R:

```
library(e1071)
```

We can execute the command `svm` to build a model:

```
model = svm(y ~ ., data, kernel = "radial", type = "C-classification")
```

where parameter `kernel` controls which kernel is going to be used (one of `linear`, `polynomial`, `radial`), while we may also define more parameters for each kernel, such as `degree`, `gamma`, `coef0`. By executing `?svm`, we can see all the parameters.

Having a model, a new instance may be classified using the command:

```
predict(model, test)
```

If we also want to view the probabilities for each class, then we have to add the parameter `probability = TRUE`. Note that the parameter `probability = TRUE` must be added both while training (command `svm`) and while running (command `predict`).

6.2 Support Vector Machines Model Construction and Classification

We will use the data of the file `alldata.txt` for a problem of binary classification:

```
       X1                    X2                    y
 Min.   :-3.25322007   Min.   :-4.664161   Min.   :1.0
 1st Qu.:-0.70900886   1st Qu.: 0.625361   1st Qu.:1.0
 Median :-0.01162501   Median : 1.641370   Median :1.5
 Mean   : 0.03493570   Mean   : 1.939284   Mean   :1.5
 3rd Qu.: 0.73337179   3rd Qu.: 3.115350   3rd Qu.:2.0
 Max.   : 3.63957363   Max.   : 9.784076   Max.   :2.0
```

Initially, we import the dataset, and split it into training and testing data::

```
alldata = read.csv("alldata.txt")
trainingdata = alldata[1:600, ]
testdata = alldata[601:800, ]
```

We will answer to the following questions:

a) Plot the training data with different colors/symbols for each class.

b) Apply SVM with RBF kernel and gamma equal to 1, and plot the hyperplane on the figure of question (a).

c) Repeat question (b) for gamma value equal to 0.01 and equal to 100. What are your observations?

Using the gamma values `gammavalues = c(0.01, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000)`, we will also answer to the following questions:

d) Compute the training error and the testing error.

e) Plot the two errors on the same diagram. What are your observations?

f) Apply 10-fold cross validation to find the best value for gamma.

6.2.1 Library Imports

Initially, we import the required librares:

```
library(MLmetrics)
library(e1071)
```

6.2.2 Algorithm Application and Hyperplane Diagram

For question (a), we execute

```
plot(trainingdata[, c(1:2)], col = trainingdata$y, pch = c("o","+")[trainingdata$\
y])
```

For questions (b), (c), we initially build a grid, on top of which we are going to plot the hyperplanes:

```
X1 = seq(min(trainingdata[, 1]), max(trainingdata[, 1]), by = 0.1)
X2 = seq(min(trainingdata[, 2]), max(trainingdata[, 2]), by = 0.1)
mygrid = expand.grid(X1, X2)
colnames(mygrid) = colnames(trainingdata)[1:2]
```

For question (b), we train the SVM with `gamma = 1` and apply it on the grid:

```
svm_model = svm(y ~ ., kernel="radial", type="C-classification", data = trainingd\
ata, gamma = 1)
pred = predict(svm_model, mygrid)
Y = matrix(pred, length(X1), length(X2))
contour(X1, X2, Y, add = TRUE, levels = 1.5, labels = "gamma = 1", col = "blue")
```

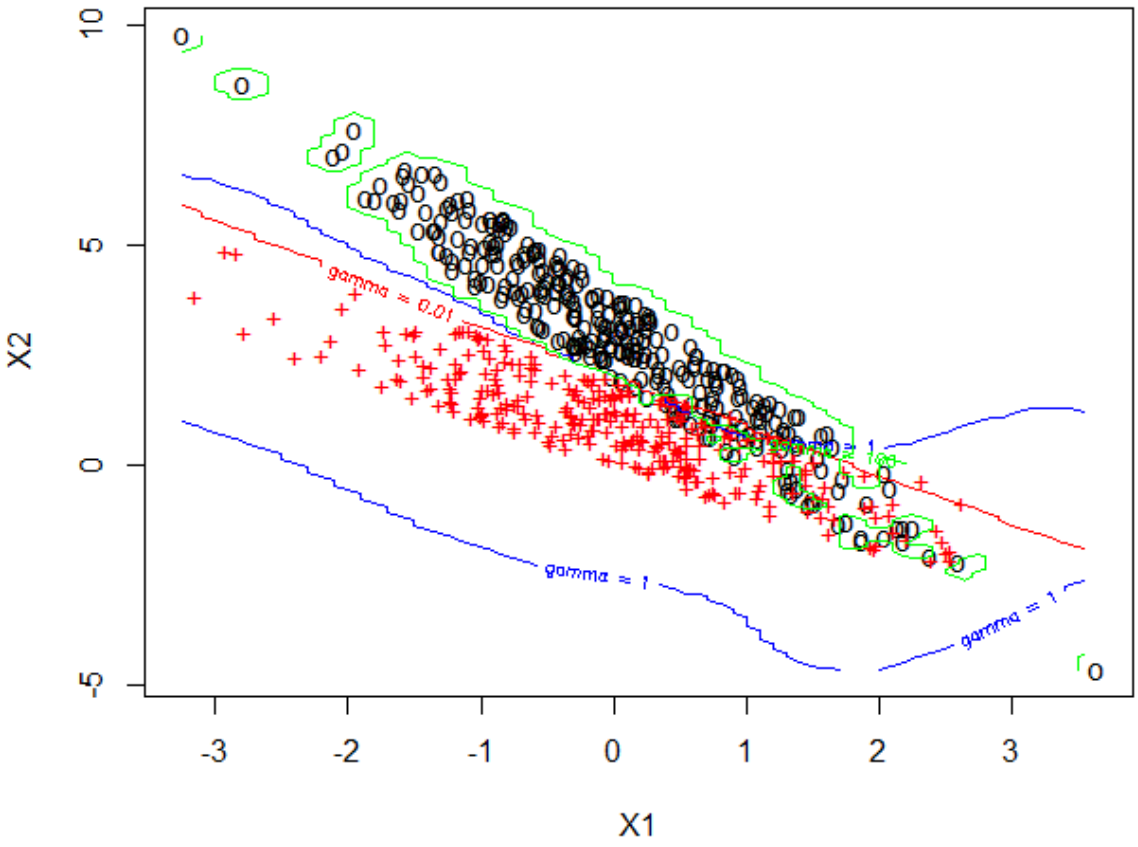
Similarly, for question (c), when `gamma = 0.01`:

```
svm_model = svm(y ~ ., kernel="radial", type="C-classification", data = trainingd\
ata, gamma = 0.01)
pred = predict(svm_model, mygrid)
Y = matrix(pred, length(X1), length(X2))
contour(X1, X2, Y, add = TRUE, levels = 1.5, labels = "gamma = 0.01", col = "red")
```

and when `gamma = 100`:

```
svm_model = svm(y ~ ., kernel="radial", type="C-classification", data = trainingd\
ata, gamma = 100)
pred = predict(svm_model, mygrid)
Y = matrix(pred, length(X1), length(X2))
contour(X1, X2, Y, add = TRUE, levels = 1.5, labels = "gamma = 100", col = "green\
")
```

Finally, the diagram is shown below:



6.2.3 Training and Testing Error Curves

For question (d), we iterate over all gamma values and calculcate the training error:

```
training_error = c()
for (gamma in gammavalues) {
  svm_model = svm(y ~ ., kernel="radial", type="C-classification", data = trainin\
gdata, gamma = gamma)
  pred = predict(svm_model, trainingdata[, c(1:2)])
  training_error = c(training_error, 1 - Accuracy(trainingdata$y, pred))
}
```

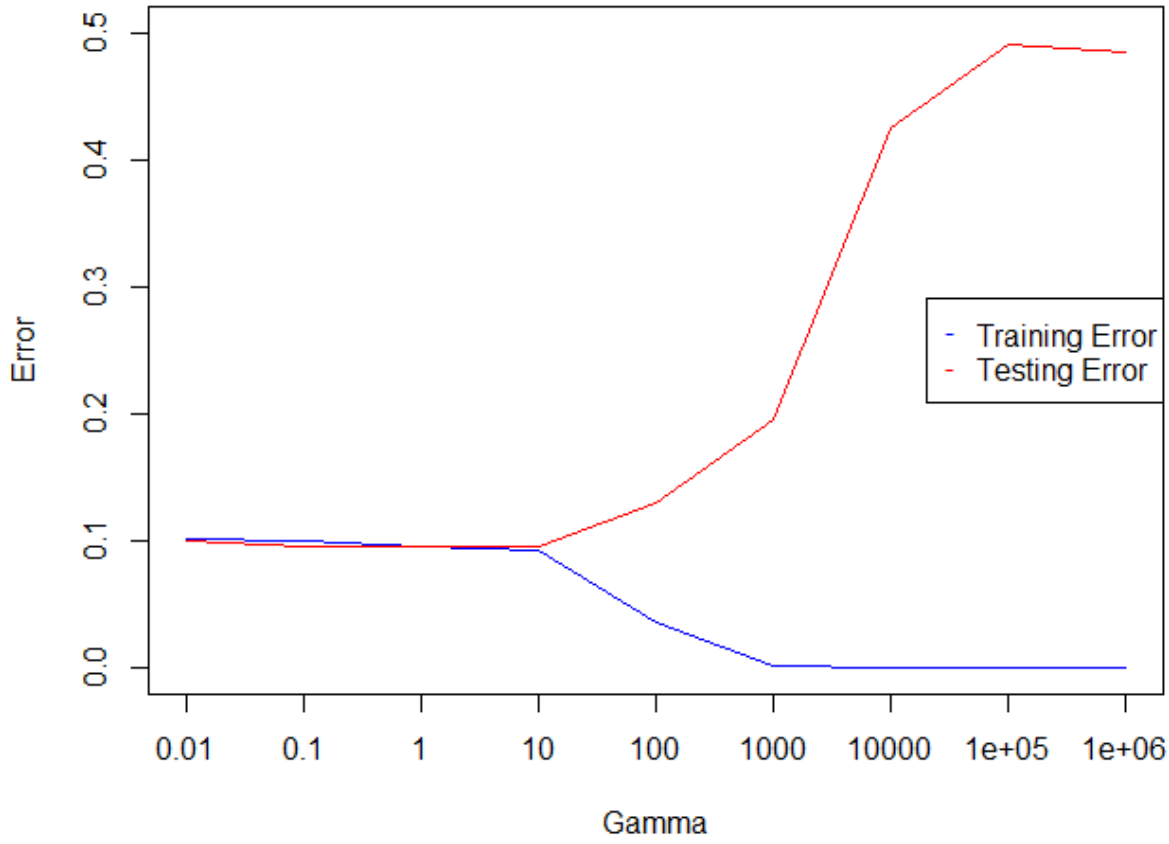
Similarly, for the testing error:

```
testing_error = c()
for (gamma in gammavalues) {
  svm_model = svm(y ~ ., kernel="radial", type="C-classification", data = trainin\
gdata, gamma = gamma)
  pred = predict(svm_model, testdata[, c(1:2)])
  testing_error = c(testing_error, 1 - Accuracy(testdata$y, pred))
}
```

We may plot the two errors on the same diagram using the commands:

```
plot(training_error, type = "l", col="blue", ylim = c(0, 0.5), xlab = "Gamma", yl\
ab = "Error", xaxt = "n")
axis(1, at = 1:length(gammavalues), labels = gammavalues)
lines(testing_error, col="red")
legend("right", c("Training Error", "Testing Error"), pch = c("-","-"),  col = c(\
"blue", "red"))
```

Finally, we are provided with the following diagram:



6.2.4 Cross-validation

We will apply k-fold cross validation to compute the best value for gamma (question (f)). Initially, we construct k folds:

```
k = 10
dsize = nrow(trainingdata)
folds = split(sample(1:dsize), ceiling(seq(dsize) * k / dsize))
```

After that, we make a loop over gamma, and inside the loop we create a new (nested) loop over the folds:

```
accuracies <- c()
for (gamma in gammavalues) {
  predictions <- data.frame()
  testsets <- data.frame()
  for(i in 1:k){
    # Select 9 out of 10 folds for training and 1 for validation
    trainingset <- trainingdata[unlist(folds[-i]),]
    validationset <- trainingdata[unlist(folds[i]),]
    # Train and apply the model
    svm_model = svm(y ~ ., kernel="radial", type="C-classification", data = train\
ingset, gamma = gamma)
    pred = predict(svm_model, validationset[, c(1:2)])
    # Save predictions and testsets
    predictions <- rbind(predictions, as.data.frame(pred))
    testsets <- rbind(testsets, as.data.frame(validationset[,3]))
  }
  # Calculate the new accuracy and add it to the previous ones
  accuracies = c(accuracies, Accuracy(predictions, testsets))
}
```

We calculate the accuracy for each fold, and, finally, select the gamma value with the maximum accuracy:

```
print(accuracies)
bestgamma = gammavalues[which.max(accuracies)]
```

6.3 Exercise

You are given the following data where Y is the class vector of a binary classification problem.

X1	X2	Y
2	2	1
2	-2	1
-2	-2	1
-2	2	1
1	1	2
1	-1	2
-1	-1	2
-1	1	2

You can enter the data in R using the following commands:

```
X1 = c(2, 2, -2, -2, 1, 1, -1, -1)
X2 = c(2, -2, -2, 2, 1, -1, -1, 1)
Y = c(1, 1, 1, 1, 2, 2, 2, 2)
alldata = data.frame(X1, X2, Y)
```

Answer to the following questions:

a) Plot the training data with different colors/symbols for each class.

b) Apply SVM with RBF kernel, and plot the hyperplane on the figure of question (a).

c) Using your jmodel, in which class would you classify a new instance with values `(4, 5)`?

Part III - Data Processing

Chapter 7 - Feature Selection

7.1 Introduction

What is feature selection? What are filter methods? What are wrapper methods?

7.2 Filter Methods

In this section we will use the **Correlation Feature Selection (CFS)** method to select attributes to be used when training and testing the machine learning models. We will use the Iris dataset for this purpose.

```
data = iris
# split into training and validation datasets
set.seed(1234)
ind <- sample(2, nrow(data), replace=TRUE, prob=c(0.7, 0.3))
trainData <- data[ind==1,]
validationData <- data[ind==2,]
# keep only instances that do not have missing values.
trainData <- trainData[complete.cases(trainData),]
validationData <- validationData[complete.cases(validationData),]
```

7.2.1 Using the CFS method

The CFS method is found in the `FSelector` package and to use it we call the `cfs` method.

```
library(FSelector)
subset <- cfs(Species ~ ., trainData)
f <- as.simple.formula(subset, "Species")
print(f)
```

```
## Species ~ Petal.Length + Petal.Width
## <environment: 0x114e18a70>
```

The output is a formula that we can use in various classification algorithms and says that according to the CFS algorithm and the training dataset, the best features in order to predict the Species of the Iris flowers are the Petal Length and the Petal Width.

For example in the Naive Bayes algorithm, we will use both the formula that includes all the attributes for predicting the Species and the formula derived from CFS.

```
library(e1071)
model <- naiveBayes(Species ~ ., data=trainData, laplace = 1)
simpler_model <- naiveBayes(f, data=trainData, laplace = 1)

pred <- predict(model, validationData)
simpler_pred <- predict(simpler_model, validationData)

library(MLmetrics)
train_pred <- predict(model, trainData)
train_simpler_pred <- predict(simpler_model, trainData)
paste("Accuracy in training all attributes",
      Accuracy(train_pred, trainData$Species), sep=" - ")
```

```
## [1] "Accuracy in training all attributes - 0.964285714285714"
```

```
paste("Accuracy in training CFS attributes",
      Accuracy(train_simpler_pred, trainData$Species), sep=" - ")
```

```
## [1] "Accuracy in training CFS attributes - 0.955357142857143"
```

```
paste("Accuracy in validation all attributes",
      Accuracy(pred, validationData$Species), sep=" - ")
```

```
## [1] "Accuracy in validation all attributes - 0.947368421052632"
```

```
paste("Accuracy in validation CFS attributes",
      Accuracy(simpler_pred, validationData$Species), sep=" - ")
```

```
## [1] "Accuracy in validation CFS attributes - 0.973684210526316"
```

The accuracy in the training set is increased when using all the attrbitues, but decreased in the validation set in relation to the simpler model that only used 2 attributes.

7.2 Wrapper Methods

7.2.1 Recurrent Feature Elimination

As a demonstration of a wrapper method, we will use the **Recurrent Feature Elimination (RFE)** method to select attributes in the Pima Indians Diabetes dataset. For this purpose the library `caret` will be used.

First, as always, we will split the dataset in train and validation datasets.

```
# ensure the results are repeatable
set.seed(1234)
# load the required libraries
library(mlbench)
library(caret)
# load the data
data(PimaIndiansDiabetes)
data = PimaIndiansDiabetes
# split into training and validation datasets
set.seed(1234)
ind <- sample(2, nrow(data), replace=TRUE, prob=c(0.7, 0.3))
trainData <- data[ind==1,]
validationData <- data[ind==2,]
trainData <- trainData[complete.cases(trainData),]
validationData <- validationData[complete.cases(validationData),]
```

Next we will use the `rfe` method of the `caret` package, setting it up using the `rfeControl` method.

```
# define the control using a random forest selection function
control <- rfeControl(functions=nbFuncs, method="cv", number=10)
# run the RFE algorithm
results <- rfe(trainData[,1:8], trainData[,9], sizes=c(1:8), rfeControl=control)
```

Let's see the results.

```
# summarize the results
print(results)
```

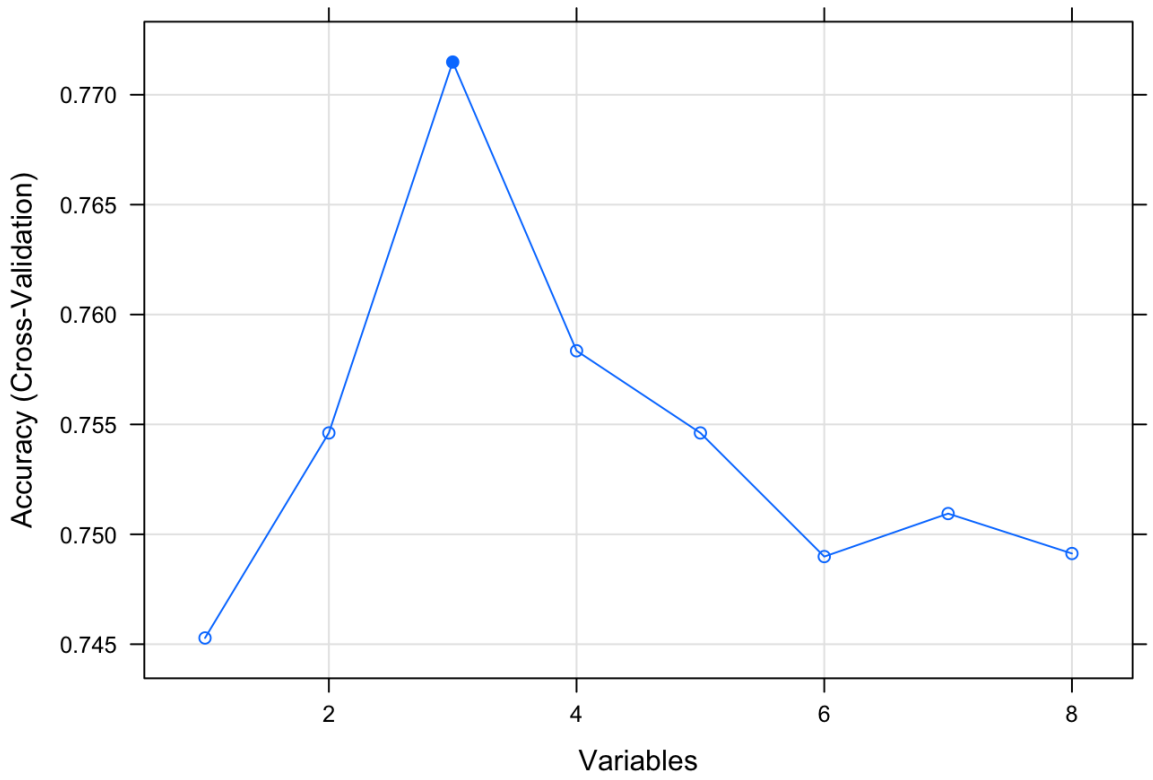
```
##
## Recursive feature selection
##
## Outer resampling method: Cross-Validated (10 fold)
##
## Resampling performance over subset size:
##
##  Variables Accuracy  Kappa AccuracySD KappaSD Selected
##          1   0.7453 0.3681    0.04390 0.12449
##          2   0.7546 0.4182    0.04598 0.12317
##          3   0.7715 0.4707    0.05759 0.14388        *
##          4   0.7584 0.4479    0.04145 0.10478
##          5   0.7546 0.4457    0.04429 0.10095
##          6   0.7490 0.4344    0.04393 0.09394
##          7   0.7509 0.4290    0.04686 0.11122
##          8   0.7491 0.4201    0.03507 0.07767
##
## The top 3 variables (out of 3):
##    glucose, age, mass
```

The RFE out of the 8 variables selected three (glucose, age, mass) that provided the best accuracy under 10-fold cross-validation.

```
# list the chosen features
predictors(results)
```

```
## [1] "glucose" "age"     "mass"
```

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



We will use the 3 top variables that come out of RFE in the Naive Bayes algorithm:

```
library(e1071)
(f <- as.formula(paste("diabetes", paste(results$optVariables, collapse=" + "), s\
ep=" ~ ")))
```

```
## diabetes ~ glucose + age + mass
```

Outer parentheses

Note the usage of the outer parenthesis. We both set the variable *f* equal to the T> formula and print it to the console.

```
model <- naiveBayes(diabetes ~ ., data=trainData, laplace = 1)
simpler_model <- naiveBayes(f, data=trainData, laplace = 1)

pred <- predict(model, validationData)
simpler_pred <- predict(simpler_model, validationData)

library(MLmetrics)
train_pred <- predict(model, trainData)
train_simpler_pred <- predict(simpler_model, trainData)
paste("Accuracy in training all attributes",
      Accuracy(train_pred, trainData$diabetes), sep=" - ")
```

```
## [1] "Accuracy in training all attributes - 0.760299625468165"
```

```
paste("Accuracy in training RFE attributes",
      Accuracy(train_simpler_pred, trainData$diabetes), sep=" - ")
```

```
## [1] "Accuracy in training RFE attributes - 0.764044943820225"
```

```
paste("Accuracy in validation all attributes",
      Accuracy(pred, validationData$diabetes), sep=" - ")
```

```
## [1] "Accuracy in validation all attributes - 0.764957264957265"
```

```
paste("Accuracy in validation RFE attributes",
      Accuracy(simpler_pred, validationData$diabetes), sep=" - ")
```

```
## [1] "Accuracy in validation RFE attributes - 0.799145299145299"
```

Using a simpler formula with 3 out of 8 attibutes, we were able to get better generalization results in the validation dataset.

7.2.2 Forward Search

The standard wrapper methods are forward and backward search. In this sub-section we will apply forward search and decision trees as a model in the breast cancer Wisconsin dataset (see Chapter 5).

```
# Downloading the file
fileURL <- "http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cance\
r-wisconsin/breast-cancer-wisconsin.data"
download.file(fileURL, destfile="breast-cancer-wisconsin.data", method="curl")
data <- read.table('breast-cancer-wisconsin.data', na.strings = "?", sep=",")
data <- data[,-1]
names(data) <- c("ClumpThickness",
                 "UniformityCellSize",
                 "UniformityCellShape",
                 "MarginalAdhesion",
                 "SingleEpithelialCellSize",
                 "BareNuclei",
                 "BlandChromatin",
                 "NormalNucleoli",
                 "Mitoses",
                 "Class")
data$Class <- factor(data$Class, levels=c(2,4), labels=c("benign", "malignant"))
set.seed(1234)
ind <- sample(2, nrow(data), replace=TRUE, prob=c(0.7, 0.3))
trainData <- data[ind==1,]
validationData <- data[ind==2,]
# remove cases with missing data
trainData <- trainData[complete.cases(trainData),]
validationData <- validationData[complete.cases(validationData),]
```

We will use the `forward.search` method of the `FSelector` package and the `rpart` decision trees training method of the homonymous package. The `forward.search` method needs an evaluation function that evaluates a subset of attributes. In our case the `evaluator` function performs 5-fold cross-validation on the training dataset.

```
library(FSelector)
library(rpart)

evaluator <- function(subset) {
  #k-fold cross validation
  k <- 5
  splits <- runif(nrow(trainData))
  results = sapply(1:k, function(i) {
    test.idx <- (splits >= (i - 1) / k) & (splits < i / k)
    train.idx <- !test.idx
    test <- trainData[test.idx, , drop=FALSE]
    train <- trainData[train.idx, , drop=FALSE]
    tree <- rpart(as.simple.formula(subset, "Class"), train)
    error.rate = sum(test$Class != predict(tree, test, type="c")) / nrow(test)
    return(1 - error.rate)
  })
  print(subset)
  print(mean(results))
  return(mean(results))
}

subset <- forward.search(names(trainData)[-10], evaluator)
```

```
## [1] "ClumpThickness"
## [1] 0.8535402
## [1] "UniformityCellSize"
## [1] 0.8982188
## [1] "UniformityCellShape"
## [1] 0.9132406
## [1] "MarginalAdhesion"
## [1] 0.8564007
## [1] "SingleEpithelialCellSize"
## [1] 0.9055005
## [1] "BareNuclei"
## [1] 0.9079083
## [1] "BlandChromatin"
## [1] 0.9027608
## [1] "NormalNucleoli"
## [1] 0.895089
## [1] "Mitoses"
## [1] 0.7942549
## [1] "ClumpThickness"      "UniformityCellShape"
## [1] 0.9297849
## [1] "UniformityCellSize"  "UniformityCellShape"
## [1] 0.9335359
## [1] "UniformityCellShape" "MarginalAdhesion"
## [1] 0.9167509
## [1] "UniformityCellShape"      "SingleEpithelialCellSize"
## [1] 0.9404055
## [1] "UniformityCellShape" "BareNuclei"
## [1] 0.9384218
## [1] "UniformityCellShape" "BlandChromatin"
## [1] 0.9283328
## [1] "UniformityCellShape" "NormalNucleoli"
## [1] 0.944158
## [1] "UniformityCellShape" "Mitoses"
## [1] 0.9162663
## [1] "ClumpThickness"      "UniformityCellShape" "NormalNucleoli"
## [1] 0.9405466
## [1] "UniformityCellSize"  "UniformityCellShape" "NormalNucleoli"
## [1] 0.9322648
## [1] "UniformityCellShape" "MarginalAdhesion"    "NormalNucleoli"
## [1] 0.9387149
## [1] "UniformityCellShape"      "SingleEpithelialCellSize"
## [3] "NormalNucleoli"
## [1] 0.9314805
## [1] "UniformityCellShape" "BareNuclei"          "NormalNucleoli"
## [1] 0.9470806
## [1] "UniformityCellShape" "BlandChromatin"      "NormalNucleoli"
## [1] 0.9267064
## [1] "UniformityCellShape" "NormalNucleoli"      "Mitoses"
## [1] 0.9401233
## [1] "ClumpThickness"      "UniformityCellShape" "BareNuclei"
## [4] "NormalNucleoli"
## [1] 0.9335856
## [1] "UniformityCellSize"  "UniformityCellShape" "BareNuclei"
## [4] "NormalNucleoli"
## [1] 0.9498017
## [1] "UniformityCellShape" "MarginalAdhesion"    "BareNuclei"
```

```
## [4] "NormalNucleoli"
## [1] 0.9418352
## [1] "UniformityCellShape"      "SingleEpithelialCellSize"
## [3] "BareNuclei"               "NormalNucleoli"
## [1] 0.9415623
## [1] "UniformityCellShape" "BareNuclei"          "BlandChromatin"
## [4] "NormalNucleoli"
## [1] 0.9506024
## [1] "UniformityCellShape" "BareNuclei"          "NormalNucleoli"
## [4] "Mitoses"
## [1] 0.9382617
## [1] "ClumpThickness"      "UniformityCellShape" "BareNuclei"
## [4] "BlandChromatin"      "NormalNucleoli"
## [1] 0.9515459
## [1] "UniformityCellSize"  "UniformityCellShape" "BareNuclei"
## [4] "BlandChromatin"      "NormalNucleoli"
## [1] 0.930654
## [1] "UniformityCellShape" "MarginalAdhesion"    "BareNuclei"
## [4] "BlandChromatin"      "NormalNucleoli"
## [1] 0.9454745
## [1] "UniformityCellShape"      "SingleEpithelialCellSize"
## [3] "BareNuclei"               "BlandChromatin"
## [5] "NormalNucleoli"
## [1] 0.9385037
## [1] "UniformityCellShape" "BareNuclei"          "BlandChromatin"
## [4] "NormalNucleoli"      "Mitoses"
## [1] 0.935793
## [1] "ClumpThickness"      "UniformityCellSize"  "UniformityCellShape"
## [4] "BareNuclei"          "BlandChromatin"      "NormalNucleoli"
## [1] 0.9460072
## [1] "ClumpThickness"      "UniformityCellShape" "MarginalAdhesion"
## [4] "BareNuclei"          "BlandChromatin"      "NormalNucleoli"
## [1] 0.9494458
## [1] "ClumpThickness"           "UniformityCellShape"
## [3] "SingleEpithelialCellSize" "BareNuclei"
## [5] "BlandChromatin"           "NormalNucleoli"
## [1] 0.9506376
## [1] "ClumpThickness"      "UniformityCellShape" "BareNuclei"
## [4] "BlandChromatin"      "NormalNucleoli"      "Mitoses"
## [1] 0.939423
```

After the search we get the following formula, where 5 out of the 9 variables were kept.

```
f <- as.simple.formula(subset, "Class")
print(f)
```

```
## Class ~ ClumpThickness + UniformityCellShape + BareNuclei + BlandChromatin +
##     NormalNucleoli
## <environment: 0x132db8c38>
```

The fact that we performed forward search using decision trees in order to get a formula with a subset of attributes, doesn't stop us from using another classification model for training and prediction. For example as in the previous examples in tehi chapter, we can use the Naive Bayes algorithm to evaluate the forward selection algorithm both in the training and the validation datasets under the accuracy metric.

```
library(e1071)
model <- naiveBayes(Class ~ ., data=trainData, laplace = 1)
simpler_model <- naiveBayes(f, data=trainData, laplace = 1)

pred <- predict(model, validationData)
simpler_pred <- predict(simpler_model, validationData)

library(MLmetrics)
train_pred <- predict(model, trainData)
train_simpler_pred <- predict(simpler_model, trainData)
paste("Accuracy in training all attributes",
      Accuracy(train_pred, trainData$Class), sep=" - ")
```

```
## [1] "Accuracy in training all attributes - 0.957805907172996"
```

```
paste("Accuracy in training forward search attributes",
      Accuracy(train_simpler_pred, trainData$Class), sep=" - ")
```

```
## [1] "Accuracy in training forward search attributes - 0.953586497890295"
```

```
paste("Accuracy in validation all attributes",
      Accuracy(pred, validationData$Class), sep=" - ")
```

```
## [1] "Accuracy in validation all attributes - 0.976076555023923"
```

```
paste("Accuracy in validation forward search attributes",
      Accuracy(simpler_pred, validationData$Class), sep=" - ")
```

```
## [1] "Accuracy in validation forward search attributes - 0.971291866028708"
```

In the breast cancer Wisconsin dataset, the feature selection algorithm did not outperform the use of all attributes. The obvious cause for this is that in the dataset we have 9 attributes, handpicked by domain experts and have indeed a combined predictive power that cannot outperform any subset of them. So removing features in this case some does not produce better results.

Backward Search

One can also alter the `forward.search` method with `backward.search` to perform the backward search wrapper selection method. *How many attributes are kept in the case of Backward Search?*

Chapter 8 - Dimensionality Reduction

8.1 Principal Components Analysis

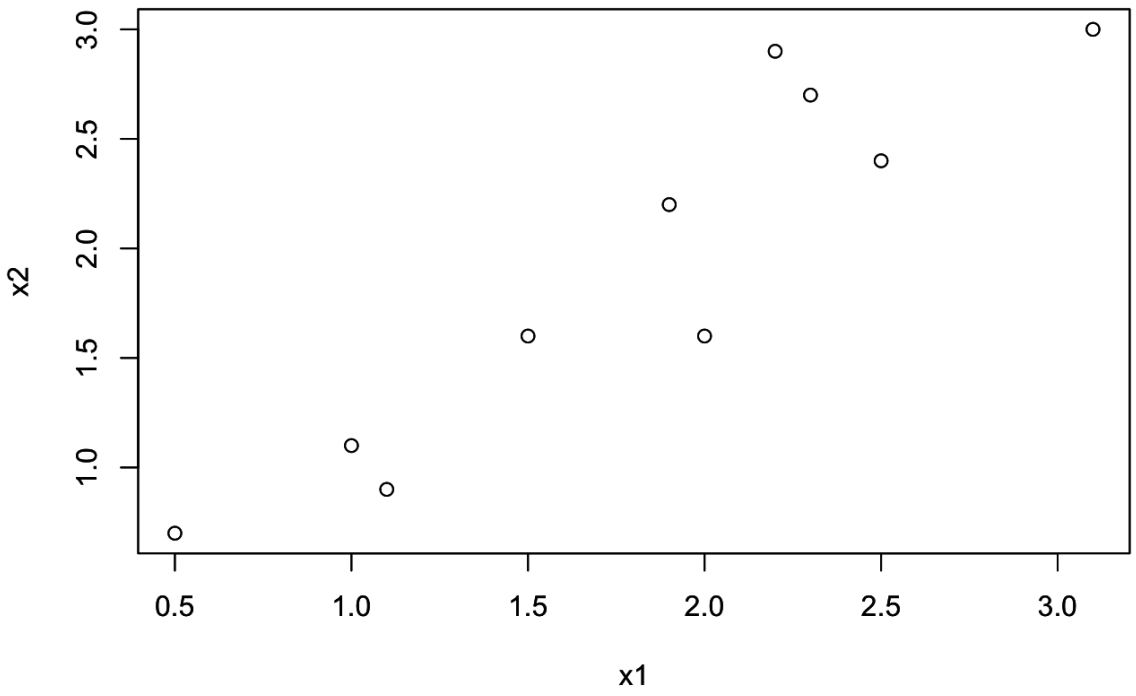
8.1.1 "Raw" Principal Components Analysis**

First we will try to perform Principal Components analysis (PCA) without using a ready-made R function from a library.

Step 1: Data

First we are going to make a dataset and plot it.

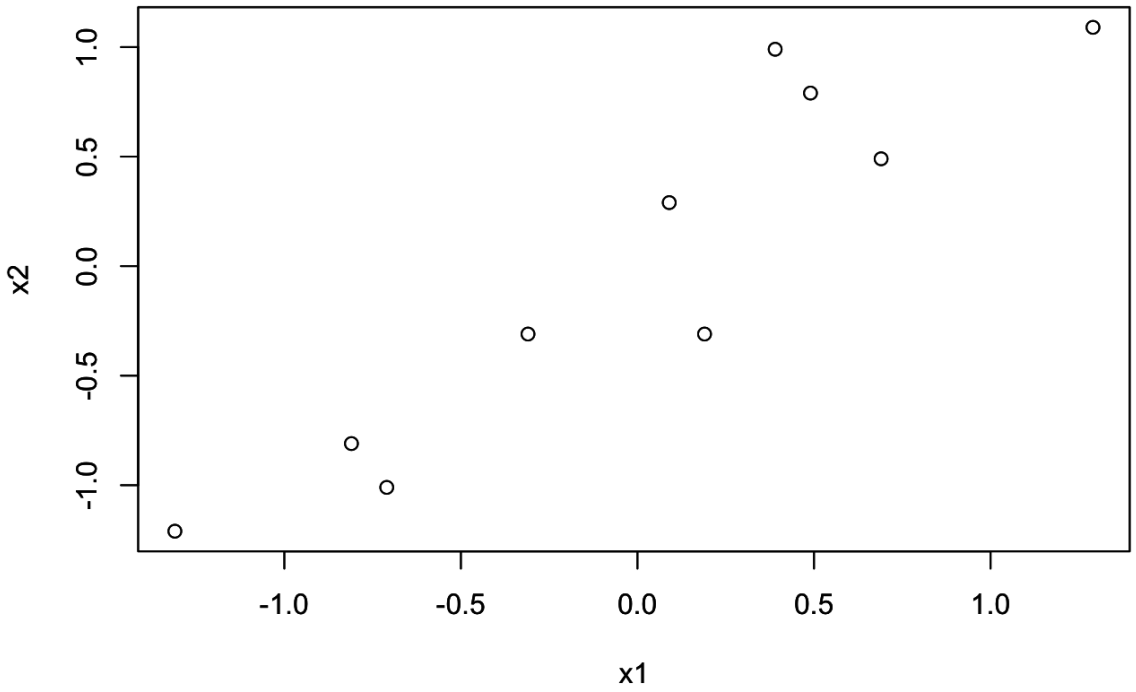
```
d <- c(2.5, 2.4, 0.5, 0.7, 2.2, 2.9, 1.9, 2.2, 3.1, 3.0, 2.3,
       2.7, 2, 1.6, 1, 1.1, 1.5, 1.6, 1.1, 0.9)
data <- matrix(d, ncol=2, byrow = T)
plot(data, xlab="x1", ylab="x2")
```



Step 2: Subtract the mean

We subtract the mean of each attribute (x1, x2) from the respective values (centering the data) using the function `scale`.

```
data_norm <- scale(data, scale=F)
plot(data_norm, xlab="x1", ylab="x2")
```



Step 3: Calculate the covariance matrix

We calculate the covariance matrix using the function `cov`.

```
S <- cov(data_norm)
print(S)
```

```
##           [,1]      [,2]
## [1,] 0.6165556 0.6154444
## [2,] 0.6154444 0.7165556
```

Step 4: Computer the eigenvectors of the covariance matrix

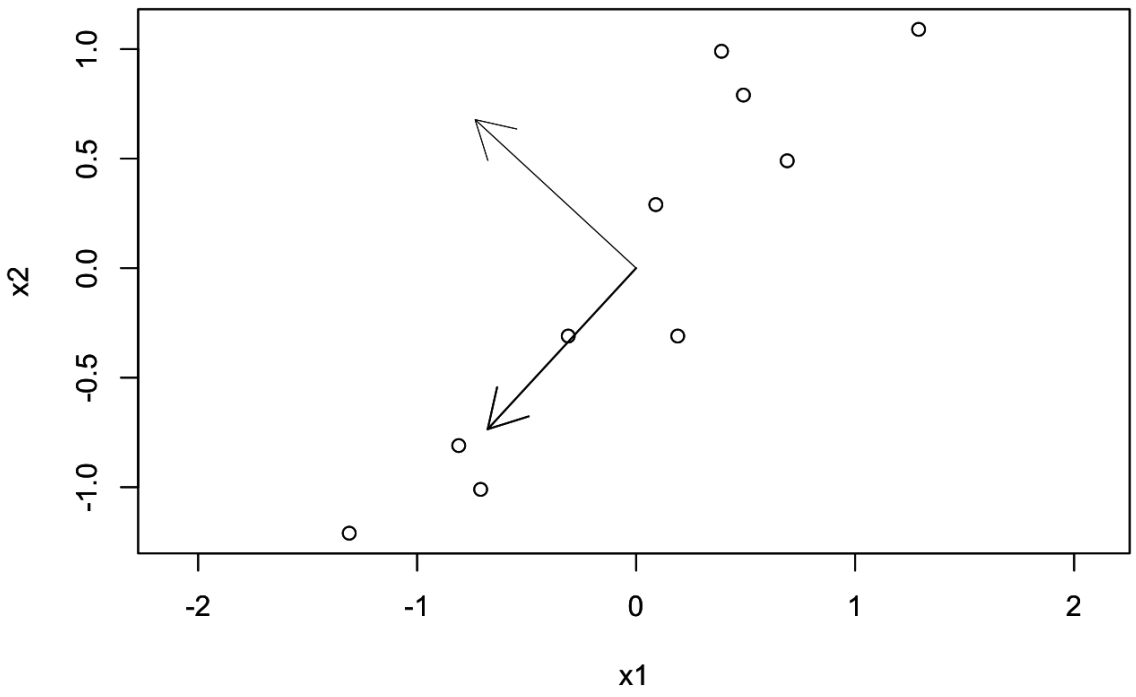
Using the `svd` method (Singular Value Decomposition) we derive the eigenvectors of the covariance matrix. First we print the decomposition:

```
udv <- svd(S)
print(udv)
```

```
## $d
## [1] 1.2840277 0.0490834
##
## $u
##            [,1]       [,2]
## [1,] -0.6778734 -0.7351787
## [2,] -0.7351787  0.6778734
##
## $v
##            [,1]       [,2]
## [1,] -0.6778734 -0.7351787
## [2,] -0.7351787  0.6778734
```

and then we plot the eigenvectors:

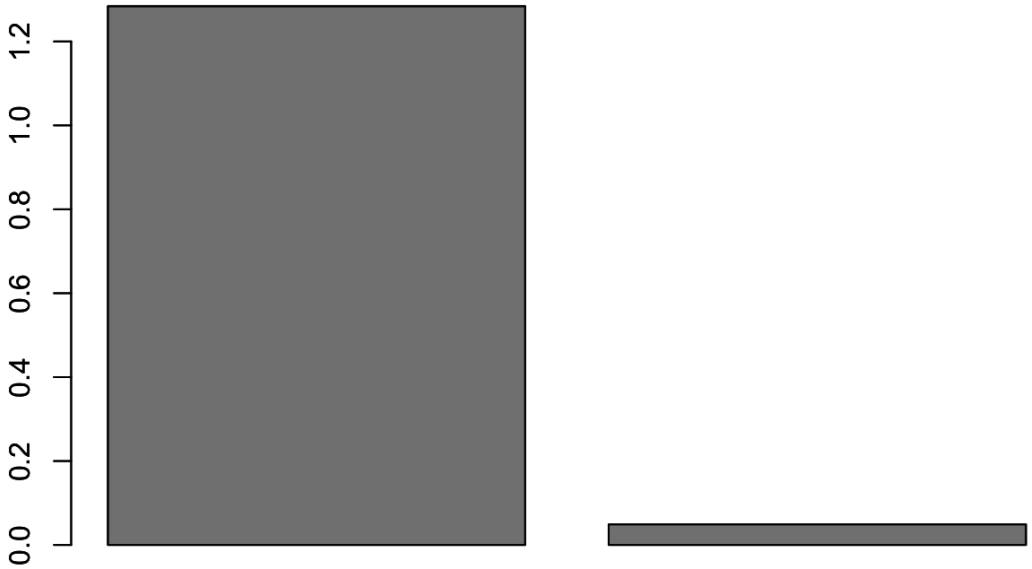
```
# asp=1 keep the aspect ratio to 1 so that the eigenvectors are
# displayed appropriatelly, perpendicular
plot(data_norm, asp=1, xlab="x1", ylab="x2")
arrows(0,0,udv$u[1,1],udv$u[2,1], lwd=1)
arrows(0,0,udv$u[1,2],udv$u[2,2], lwd=0.5)
```



Step 5: Choosing components

We can use a barplot to display the variance accounted for each component.

```
barplot(udv$d)
print(cumsum(udv$d)/sum(udv$d))
```



We can see that the 1st component accounts for more than 95% of the variance.

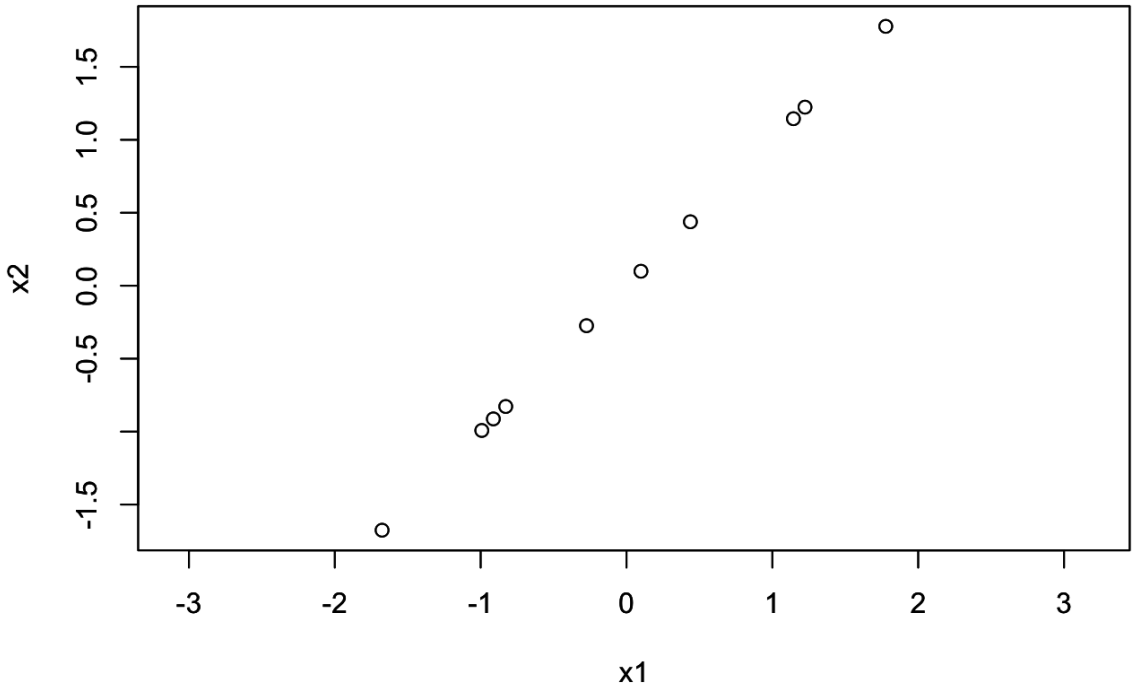
Step 6: Picking the 1st component

We transform the 2D dataset into a 1D dataset using just the 1st Principal Component (PC).

```
data_new <- t(udv$u[,1,drop=FALSE]) %*% t(data_norm)
data_new
```

```
##            [,1]    [,2]       [,3]       [,4]      [,5]       [,6]
## [1,] -0.8279702 1.77758 -0.9921975 -0.2742104 -1.675801 -0.9129491
##            [,7]     [,8]      [,9]    [,10]
## [1,] 0.09910944 1.144572 0.4380461 1.223821
```

```
plot(data_new,data_new,asp=1,xlab="x1", ylab="x2")
```



8.1.2 PCA using an R function

In this subsection, we will use the `prcomp` function that performs PCA in one step. We will apply it on the Iris dataset and test the gains or losses in terms of accuracy using the kNN algorithm.

```
# download the file
data = iris
set.seed(1234)
ind <- sample(2, nrow(data), replace=TRUE, prob=c(0.7, 0.3))
trainData <- data[ind==1,]
validationData <- data[ind==2,]

library(class)
library(stats)

trainData <- trainData[complete.cases(trainData),]
validationData <- validationData[complete.cases(validationData),]

trainDataX <- trainData[,-ncol(trainData)]
logTrainDataX <- log(trainDataX)
train.pca <- prcomp(logTrainDataX, center = TRUE, scale. = TRUE)
summary(train.pca)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4
## Standard deviation     1.7226 0.9313 0.36484 0.17916
## Proportion of Variance 0.7419 0.2168 0.03328 0.00802
## Cumulative Proportion  0.7419 0.9587 0.99198 1.00000
```

From the summary, we observe that with 2 PCs, we can explain more than 95% of the variance.

```
trainDataY <- trainData$Species
validationDataX <- validationData[,-ncol(trainData)]
# Let's also transform the validation data
logValidationDataX <- log(validationDataX)
validation.pca <- predict(train.pca, newdata=logValidationDataX)
validationDataY <- validationData$Species

# no pca prediction
prediction = knn(trainDataX, validationDataX, trainDataY, k = 3)
# So let's predict using only the 7 principal components
prediction_pca = knn(train.pca$x[,1:2], validation.pca[,1:2], trainDataY, k = 3)

cat("Confusion matrix:\n")
```

```
 ## Confusion matrix:
```

```
xtab = table(prediction, validationData$Species)
print(xtab)
```

```
##
## prediction   setosa versicolor virginica
##   setosa         10          0         0
##   versicolor      0         12         1
##   virginica       0          0        15
```

```
cat("\nEvaluation:\n\n")
```

```
##
## Evaluation:
```

```
accuracy = sum(prediction == validationData$Species)/length(validationData$Specie\
s)
cat(paste("Accuracy:\t", format(accuracy, digits=2), "\n",sep=" "))
```

```
## Accuracy:     0.97
```

```
cat("Confusion matrix PCA:\n")
```

```
## Confusion matrix PCA:
```

```
xtab = table(prediction_pca, validationData$Species)
print(xtab)
```

```
##
## prediction_pca setosa versicolor virginica
##     setosa         10          0         0
##     versicolor      0         10         3
##     virginica       0          2        13
```

```
cat("\nEvaluation PCA:\n\n")
```

```
##
## Evaluation PCA:
```

```
accuracy = sum(prediction_pca == validationData$Species)/length(validationData$Sp\
ecies)
cat(paste("Accuracy PCA:\t", format(accuracy, digits=2), "\n",sep=" "))
```

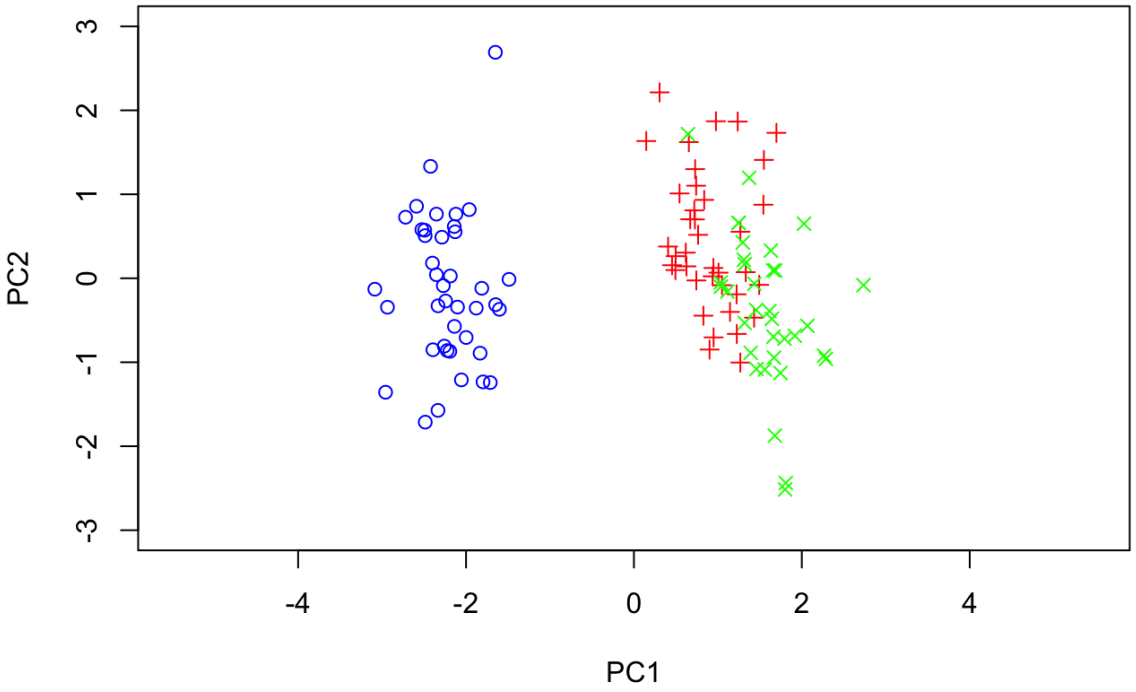
```
## Accuracy PCA:     0.87
```

Even though the validation dataset is small, one can observe that using 50% less predictor attributes, the algorithm made only 4 extra mistakes. Of course in real world scenarios, cross-validation should be used with such small datasets.

7.2.3 PCA for Plotting

Using the PCA we can also plot in 2 dimensions, high-dimensional data, by using just the first two components.

```
plot(train.pca$x[trainData$Species == 'setosa',1:2], col="blue", ylim = c(-3, 3),\
 xlim=c(-3,3), asp=1)
points(train.pca$x[trainData$Species == 'versicolor',1:2], pch = 3, col="red")
points(train.pca$x[trainData$Species == 'virginica',1:2], pch = 4, col="green")
```



In this case the PCA is not the best method for dimensionality reduction to improve the accuracy. From the plot one cas observe that the `versicolor` and `virginica` species are intermixed in 2D and thus the k-NN algorithm will have trouble classifying between the two.

Part III - Clustering

Chapter 9 - Centroid-based clustering and Evaluation

Centroid-based clustering is a subset of clustering techniques where clusters are represented by a central vector, which may not necessarily be a member of the data set. When the number of clusters is fixed to a given number ***k***, k-Means clustering gives a formal definition of the following optimization problem:

Given a dataset containing `N` samples:

$$X = \{x_1, x_2, x_3, \ldots, x_N\}$$

and `K` as the number of selected clusters:

$$Clusters : \{C_1, C_2, \ldots, C_K\} \quad with \quad Centers : \{m_1, m_2, \ldots, m_K\}$$

k-Means Algorithm Purpose: Find the distribution of all samples among clusters so as to minimize the sum of squares of the distances between each sample and the center of the cluster it belongs. This metric corresponds to the ***Within cluster Sum of Squares â€“ WSS or SSE*** and is given by the following formula:

$$SSE = \sum_{i-1}^{K} \sum_{x \in C_i} \left\| x - m_i \right\|^2$$

In order to solve the aforementioned optimization problem, k-Mean algorithm involves two distinc steps:

1. Given certain centers, formulate clusters by assigning each sample to the closest cluster
2. Calculate the new centers based on the formulate clusters

In the initialization step, the algorithm randomly selects `K` samples as the initial centers and then using these centers assigned each sample in the closest cluster. Once every sample has been assigned to a certain cluster, the new center of each cluster is calculated as the mean of every sample that belongs to the cluster. This iterative process is continuoued until there is no change in the clusters in to consecutive iterations.

The most famous and used variation of the k-Means algorithm is ***k-Medoids***, which is mainly used for categorical data and allows the use of various different distance functions. It is worth noticing that unlike k-Means, k-Medoids always uses a certain sample as the ***medoid*** of each cluster.

9.1 - k-Means in R

For performing k-Means clustering in R, we can use the following command:

```
# Perform k-Means to form 3 clusters
model = kmeans(data, centers = 3)
```

The `centers` parameter can either take the number of the initial centers (and thus the number of clusters) or the centers themselves as argument.

We can use the following commands in order to show the final centroids and the distribution of the samples into the formulated clusters:

```
# Get centroids
model$centers

# Get distribution of samples into clusters
model$cluster
```

9.2 - k-Medoids in R

In order to perform k-Medoids clustering in R, we need to import the library cluster:

```
library(cluster)
```

and then we can use the following command:

```
# Perform k-Medoids to form 3 clusters
model = pam(conferences, 3)
```

We can use the following commands in order to show the final medoids and the distribution of the samples into the formulated clusters:

```
# Get centroids
model$medoids

# Get distribution of samples into clusters
model$clustering
```

If the dataset contains categorical variables, then we can show the medoids using the respective variable with the following command:

```
data[model$id.med,]
```

9.3 - Clustering Evaluation in R

In order to evaluate our clustering in R, we can use the library cluster which enables us to easily calculate several evaluation metrics. Once we have perform the clutering and get the final distribution of the samples into the clusters, then we can compute a list of metrics that enable us evaluate the result. Some of those metrics are the following:

- ***Cohesion (or within-cluster sum of squares)***

$$WSS = \sum_{i-1}^{K} \sum_{x \in C_i} \left\| x - m_i \right\|^2$$

- ***Separation (or between-cluster sum of squares)***

$$BSS = \sum_{i-1}^{K} |C_i| \left\| m - m_i \right\|^2$$

- ***Silhouette coefficient***

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

where `a(i)` is the average distance between the `i-th` sample and all other samples within the same cluster and `b(i)` is the lowest average distance of the `i-th` sample to all samples in any other clusters, of which `i` is not a member.

We can compute the aforementioned metrics using the following R commands:

```
# Compute WSS
cohesion = model$tot.withinss

# Compute BSS
separation = model$betweenss

# Compute Silhouette
model_silhouette = silhouette(model$cluster, dist(data))
```

In the computation of silhouette coefficient, `data` stands for the original data. In order to plot the silhouette coeffiecient and the mean silhouette we can use the following commands:

```
# Plot Silhouette
plot(model_silhouette)

# Compute mean silhouette
mean(model_silhouette[, 3])
```

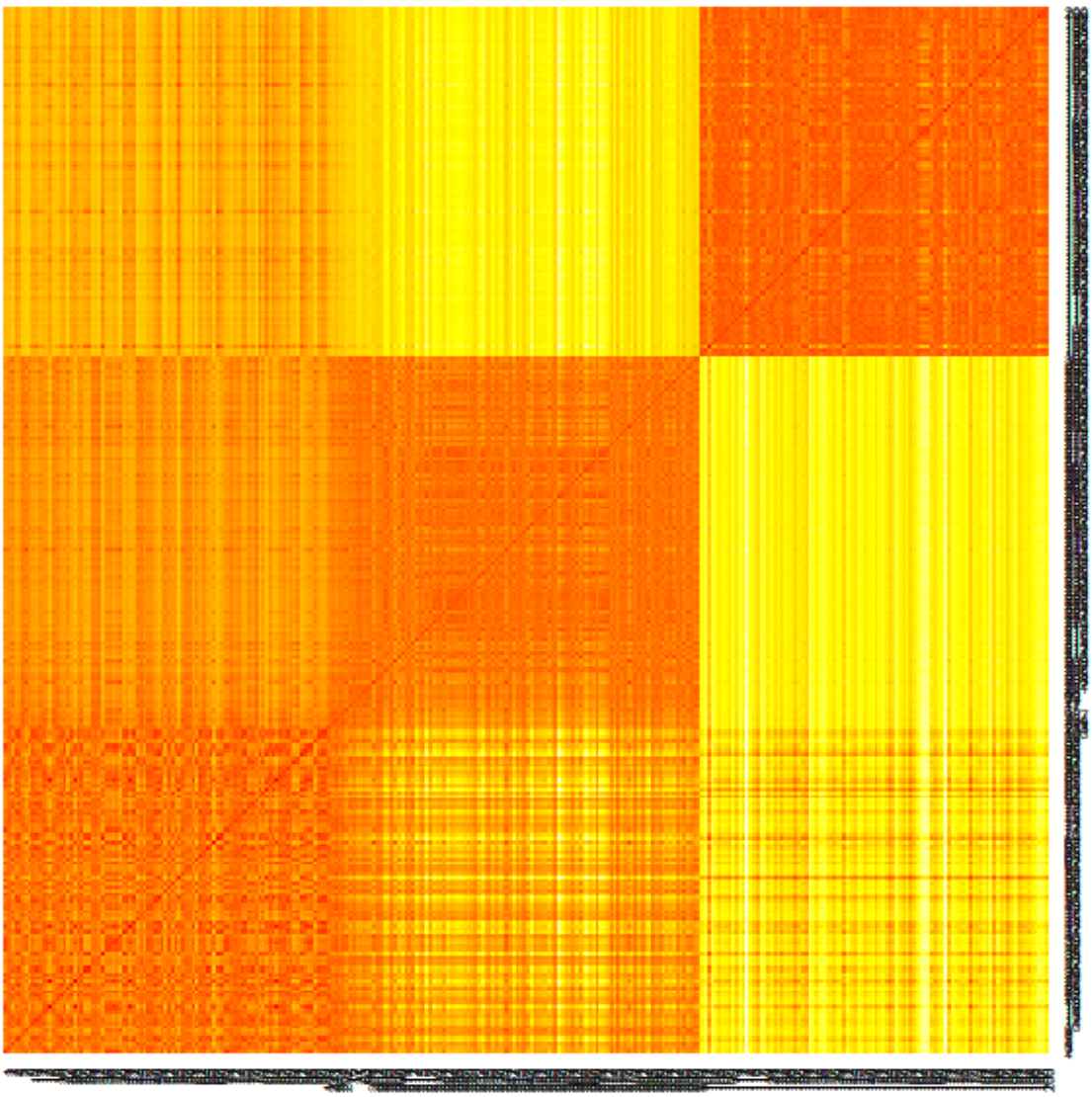
Another way that can provide us with an overview of the clustering and enables us to evaluate its performance is the construction of heatmaps. At first, we can order out data samples using the following command

```
data_ord = data[order(model$cluster),]
```

and then constuct our heatmap

```
heatmap(as.matrix(dist(data_ord)), Rowv = NA, Colv = NA,
              col = heat.colors(256), revC = TRUE)
```

The following figure illustrates a sample heatmap constructed by using data samples that are distributed among 3 clusters. The color of the heatmap refers to the distance of the ordered data samples. The red color denotes small distance while yellow high distance.



9.4 - k-Means clustering overview

The following table contains the dataset that we are going to use in order to solve a clustering problem using k-Means algorithm.

	X	Y
x1	7	1
x2	3	4
x3	1	5
x4	5	8
x5	1	3
x6	7	8
x7	8	2
x8	5	9

Using the above dataset, we will answer the following questions:

(a) Plot the data in a two dimentions.

(b) Apply k-Means algorithm in order to distribute the data points into 3 clusters. Initialize the algorithm by using the data points x1, x2 and x3 as the initial centers.

(c) Calculate the cohesion and the separation of the clustering performed in question (b)

(d) Plot the clustering results using a different color for the data points included in each cluster and also mark the centroids

9.4.1 - Data Construction

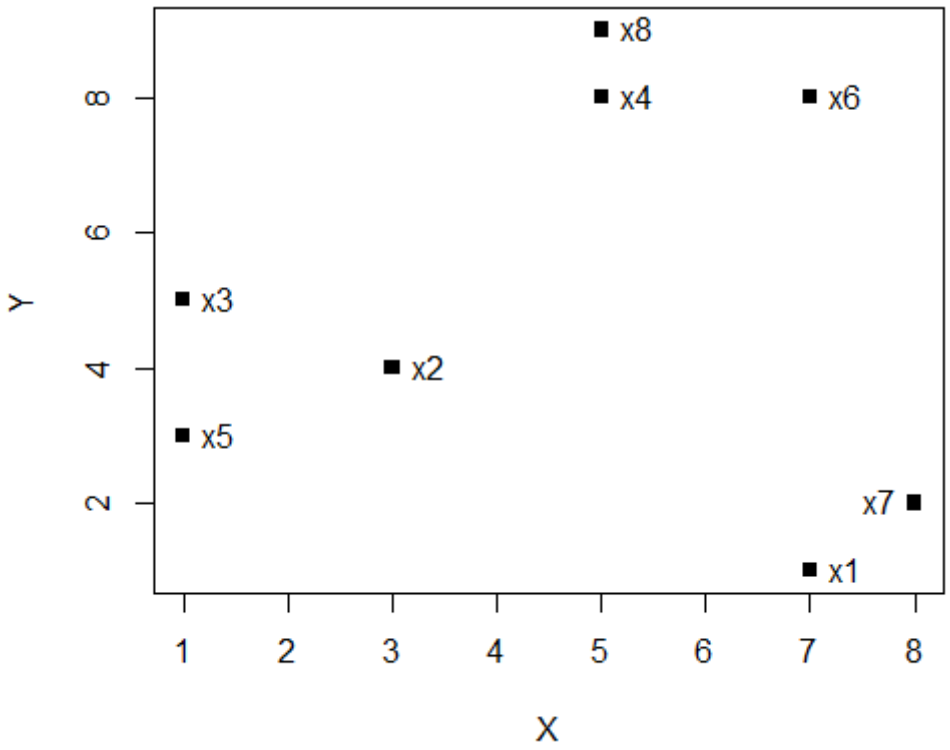
First of all, we need to properly construct our data frame containing the above dataset. In order to do so, we can use the following commands:

```
# Construct vector containing X-axis data
X = c(7, 3, 1, 5, 1, 7, 8, 5)

# Construct vector containing Y-axis data
Y = c(1, 4, 5, 8, 3, 8, 2, 9)

# Construct vector containing the data labels which correspond to the row names
rnames = c("x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8")

# Create the data frame
kdata = data.frame(X, Y, row.names = rnames)
```

Once having created our data frame, we can plot our data using the following commands:

```
# Plot the data
plot(kdata, pch = 15)

# Add the data labels
text(hdata, labels = row.names(hdata),
                pos = c(4, 2, 4, 4, 4, 4, 4, 4))
```

Our data is illutrated in the following figure:



9.4.2 - Clustering using k-Means algorithm

In order to answer question (b), we apply k-Means algorithm setting the data points x1, x2 and x3 as the initial centers:

$$C_1^0 = \begin{bmatrix} 7 \\ 1 \end{bmatrix} \quad C_2^0 = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \quad C_3^0 = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$

1st iteration:

At first, we will compute the distances between all data points contained in the dataset and the initial centers. These distances are given in the following distance matrix:

	C_1^0	C_2^0	C_3^0
x1	**0.00**	5.00	7.21
x2	5.00	**0.00**	2.24
x3	7.21	2.24	**0.00**
x4	7.28	**4.47**	5.00
x5	6.32	2.24	**2.00**
x6	7.00	**5.66**	6.71
x7	**1.41**	5.39	7.62
x8	8.25	**5.39**	5.66

After having computed the distances, we find the minimum distance for each data point in order to assign it into the respective cluster. Once all data point are assigned to one cluster, we compute the new centers.

$$C_1^1 = \frac{x_1 + x_7}{2} = \begin{bmatrix} 7.5 \\ 1.5 \end{bmatrix}$$

$$C_2^1 = \frac{x_2 + x_4 + x_6 + x_8}{4} = \begin{bmatrix} 7 \\ 7.25 \end{bmatrix} \quad C_3^1 = \frac{x_3 + x_5}{2} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$$

2nd iteration:

The following table contains the distances of the data points from the updated centers.

	C_1^1	C_2^1	C_3^1
x1	**0.71**	6.56	6.71
x2	5.15	3.82	**2.00**
x3	7.38	4.59	**1.00**
x4	6.96	**0.75**	5.66
x5	6.67	5.84	**1.00**
x6	6.52	**2.14**	7.21
x7	**0.71**	6.05	7.28
x8	7.91	**1.75**	6.40

The new centers are:

$$C_1^2 = \frac{x_1 + x_7}{2} = \begin{bmatrix} 7.5 \\ 1.5 \end{bmatrix}$$

$$C_2^2 = \frac{x_4 + x_6 + x_8}{3} = \begin{bmatrix} 5.67 \\ 8.33 \end{bmatrix} \quad C_3^2 = \frac{x_2 + x_3 + x_5}{3} = \begin{bmatrix} 1.67 \\ 4 \end{bmatrix}$$

Given that the centers have changed, we the execution of the algorithm continuoues in the next iteration

3rd iteration:

Once again, we compute the updated distances.

	C_1^2	C_2^2	C_3^3
x1	**0.71**	7.45	6.12
x2	5.15	5.09	**1.33**
x3	7.38	5.73	**1.20**
x4	6.96	**0.75**	5.21
x5	6.67	7.09	**1.20**
x6	6.52	**1.37**	6.67
x7	**0.71**	6.75	6.64
x8	7.91	**0.94**	6.01

The new centers are:

$$C_1^3 = \frac{x_1 + x_7}{2} = \begin{bmatrix} 7.5 \\ 1.5 \end{bmatrix}$$

$$C_2^3 = \frac{x_4 + x_6 + x_8}{3} = \begin{bmatrix} 5.67 \\ 8.33 \end{bmatrix} \quad C_3^3 = \frac{x_2 + x_3 + x_5}{3} = \begin{bmatrix} 1.67 \\ 4 \end{bmatrix}$$

As we can see, the centers have not changes and thus the algorithm converges after the 3rd iteration.

We can simply perform the aforementioned manual procedure with R using the following commands:

```
# Perform k-Means clustering
model = kmeans(kdata, centers = kdata[1:3,])

# Get final centers
model$centers

# Get the distribution of data points among clusters
model$cluster
```

9.4.3 - Compute Cohesion and Separation

In order to answer the question (c), we have to compute the cohesion and the sepeartion of the clustering performed in the previous subsection.

At first, we will compute these metrics manually and then we will see how they can be computed using R. In the computation, we will use the above clusters.

$$C_1: \quad m_1 = \begin{bmatrix} 7.5 \\ 1.5 \end{bmatrix} \quad C_2: \quad m_2 = \begin{bmatrix} 5.67 \\ 8.33 \end{bmatrix} \quad C_3: \quad m_3 = \begin{bmatrix} 1.67 \\ 4 \end{bmatrix}$$

- ***Cohesion computation***

The Cohesion metric is given by the following equation:

$$WSS = \sum_{i-1}^{K} \sum_{x \in C_i} \|x - m_i\|^2$$

At this point, we will compute the cohesion for every cluster:

1st Cluster:

$$WSS(C_1) = \sum_{x \in C_1} \|x - m_1\|^2 = \|x_1 - m_1\|^2 + \|x_7 - m_1\|^2$$

$$= \sqrt{(7 - 7.5)^2 + (1 - 1.5)^2}^2 + \sqrt{(8 - 7.5)^2 + (2 - 1.5)^2}^2$$

$$= 0.25 + 0.25 + 0.25 + 0.25 = 1$$

2nd Cluster:

$$WSS(C_2) = \sum_{x \in C_2} \|x - m_2\|^2 = \|x_4 - m_2\|^2 + \|x6 - m_2\|^2 + \|x_8 - m_2\|^2$$

$$= \sqrt{(5 - 5.67)^2 + (8 - 8.33)^2}^2 + \sqrt{(7 - 5.67)^2 + (8 - 8.33)^2}^2$$

$$+ \sqrt{(5 - 5.67)^2 + (9 - 8.33)^2}^2$$

$$= 0.4489 + 0.1089 + 1.7689 + 0.1089 + 0.4489 + 0.4489 = 3.3334$$

3rd Cluster:

$$WSS(C_3) = \sum_{x \in C_3} \|x - m_3\|^2 = \|x_2 - m_2\|^2 + \|x3 - m_2\|^2 + \|x_5 - m_2\|^2$$

$$= \sqrt{(3 - 1.67)^2 + (4 - 4)^2}^2 + \sqrt{(1 - 1.67)^2 + (5 - 4)^2}^2$$

$$+ \sqrt{(1 - 1.67)^2 + (3 - 4)^2}^2$$

$$= 1.7689 + 0 + 0.4489 + 1 + 0.4489 + 1 = 4.6667$$

The total cohesion is given by the following equation:

$$WSS_{Total} = \sum_i WSS(C_i) = WSS(C_1) + WSS(C_2) + WSS(C_3)$$

$$= 1 + 3.3334 + 4.6667 = 9$$

- ***Separation computation***

The Separation (Between cluster Sum of Squares) metric is given by the following equation:

$$BSS_{Total} = \sum_{i-1}^{K} |C_i| \cdot \|x - m_i\|^2$$

So, our first setp is to compute the center of our dataset, which is given by the mean of all data points.

$$m = \frac{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{8} = \begin{bmatrix} 4.625 \\ 5 \end{bmatrix}$$

$$BSS_{Total} = 2 \cdot \sqrt{(4.625 - 7.5)^2 + (5 - 1.5)^2}^2$$

$$+ \quad 3 \cdot \sqrt{(4.625 - 5.67)^2 + (5 - 8.33)^2}^2$$

$$+ \quad 3 \cdot \sqrt{(4.625 - 1.67)^2 + (5 - 4)^2}^2$$

$$= 41.031 + 36.589 + 29.255 = 106.875$$

We can easily compute the above metrics using R with the following commands:

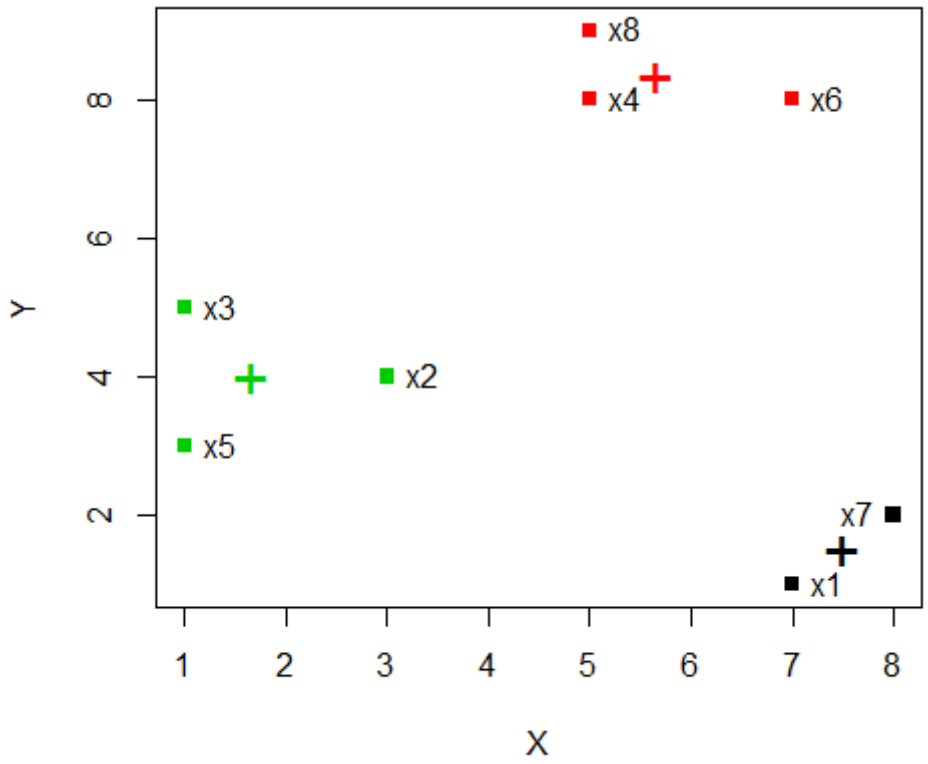
```
# Perform k-Means clustering
model = kmeans(kdata, centers = kdata[1:3,])

# Compute cohesion
cohesion = model$tot.withinss

# Compute separation
separation = model$betweenss
```

9.4.4 - Visualize k-Means Clustering

Finally, in order to answer question (d), we can use the following commands:

```
# Plot data using a different color for each cluster
plot(kdata, col = model$cluster, pch = 15)

# Add data labels
text(kdata, labels = row.names(kdata),
           pos = c(4, 4, 4, 4, 4, 4, 2, 4))

# Display centers
points(model$centers, col = 1:length(model$centers),
                      pch = "+", cex = 2)
```

The above command will result in the following figure:



9.5 - k-Means clustering and evaluation in real life Application

In this section, we are going to demonstrate the application of k-Means algorithm in a real life application example. In this context, we have a dataset that contains 300 different 2D data points that originate from three different distributions. A summary of the dataset is given in the following table:

	X1	X2
Min.	-4.720	-3.974
1st Qu.	-1.237	-1.059
Median	1.757	2.096
Mean	1.322	1.532
3rd Qu.	3.592	3.675
Max.	6.077	6.689

We will use the aforementioned data in order to answer the following questions:

(a) Plot the data using a different color for each cluster, apply the k-Means algorithm and identify the optimal number of clusters based on the SSE metric.

(b) Apply k-Means in order to formulate 3 clusters (plot the outcome using a different color for each cluster along with the centroids) and compute the cohesion and the separation.

(c) Calculate silhouette coeffiecient and create the silhouette plot.

(d) Plot the heatmap for the clustered data.

9.5.1 - Identify optimal number of clusters

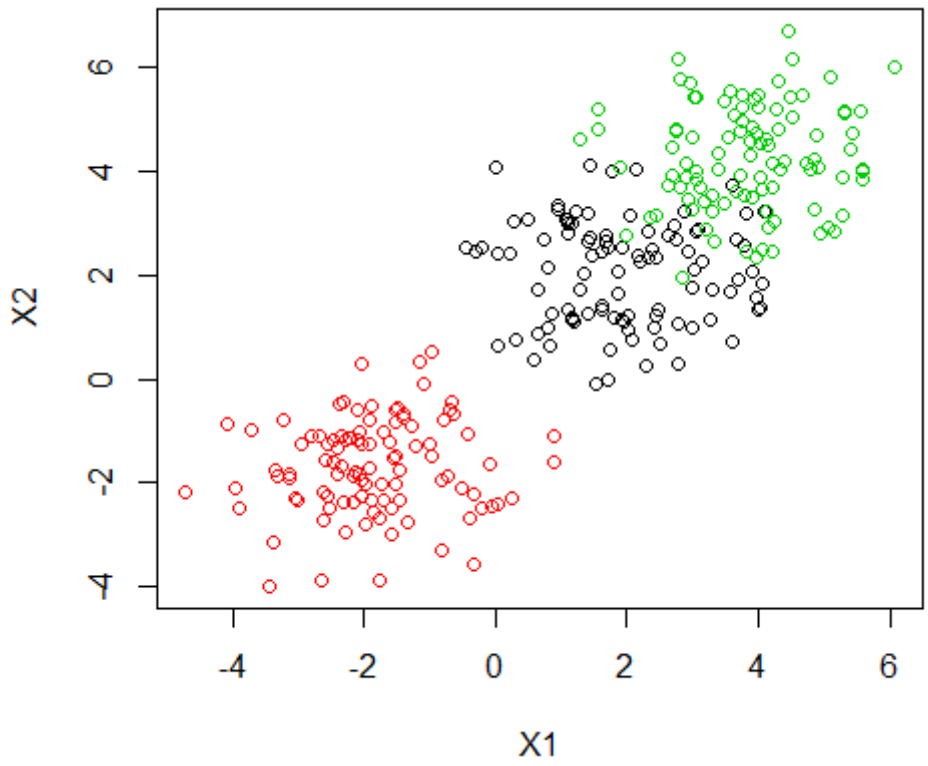
At first, we will read the given dataset using the following commands:

```
# Import cluster library
library(cluster)

# Read data
cdata = read.csv("cdata.txt")

# The third column contains the cluster for each data point
target = cdata[, 3]

# The first two columns contain the data coordinates
cdata = cdata[, 1:2]
```

After that, we can plot the dataset using the following command:

```
# Plot data
plot(cdata, col = target)
```

The output is the following figure, where tha data point for each cluster are illustrated with different color (red, black, green):



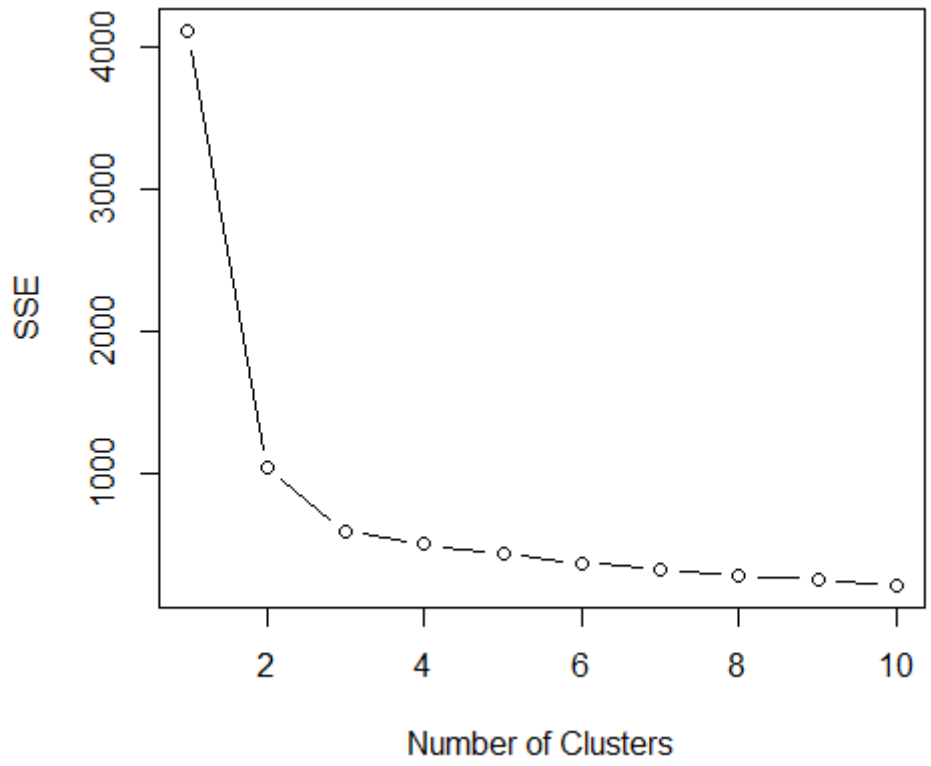
In order to identify the optimal number of clusters for the given dataset, we will be based on the values of SSE metric. To that end, we will perform k-Means clustering setting the number of

clusters from 1 to 10 and calculate SSE for each different clustering approach. This process can be done using the following commands:

```
# Initialize a vector to hold the SSE values
SSE <- (nrow(cdata) - 1) * sum(apply(cdata, 2, var))

# Calculate SSE values for the different clusterings
for (i in 2:10)
    SSE[i] <- kmeans(cdata, centers = i)$tot.withinss
```

Once finished, we can plot the SSE values using the command:

```
plot(1:10, SSE, type="b", xlab="Number of Clusters",
                          ylab="SSE")
```



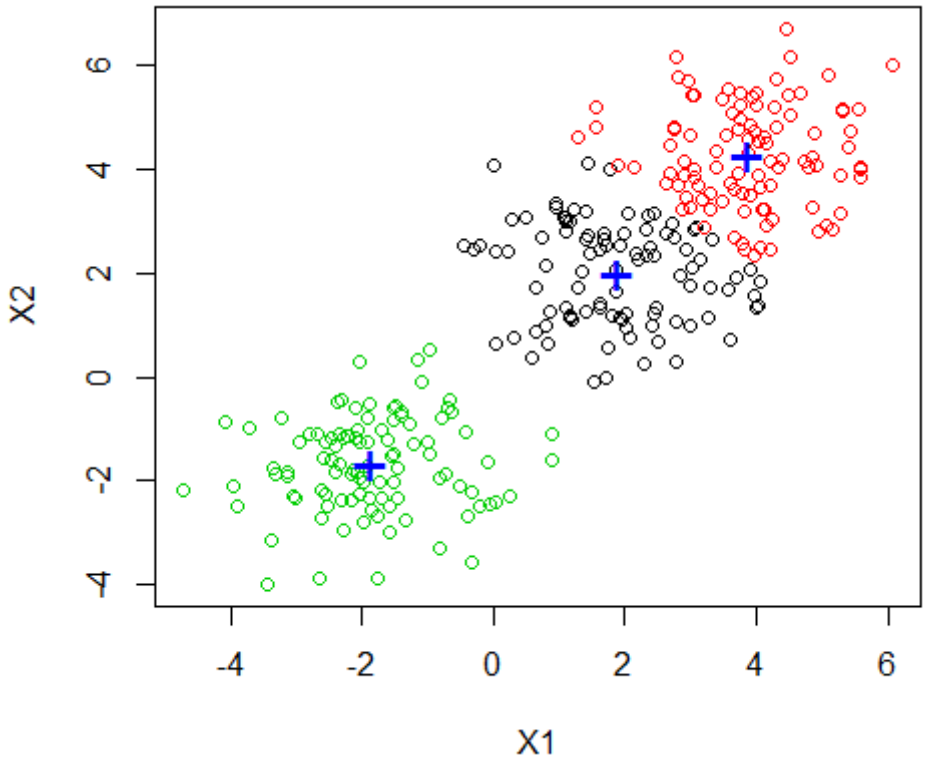
As we can see from the SSE values, following the elbow-method, the optimal number of clusters for the given dataset is 2 or 3.

9.5.2 - k-Means modeling

In order to answer question (b), we can use the following commands:

```
# Create k-Means model
model = kmeans(cdata, centers = 3)

# Get clusters' centroids
model$centers

# Get distribution of data point into clusters
model$cluster

# Calculate cohesion
cohesion = model$tot.withinss

# Calculate separation
separation = model$betweenss

# Plot clustering result
plot(cdata, col = model$cluster)
points(model$centers, col = 4, pch = "+", cex = 2)
```

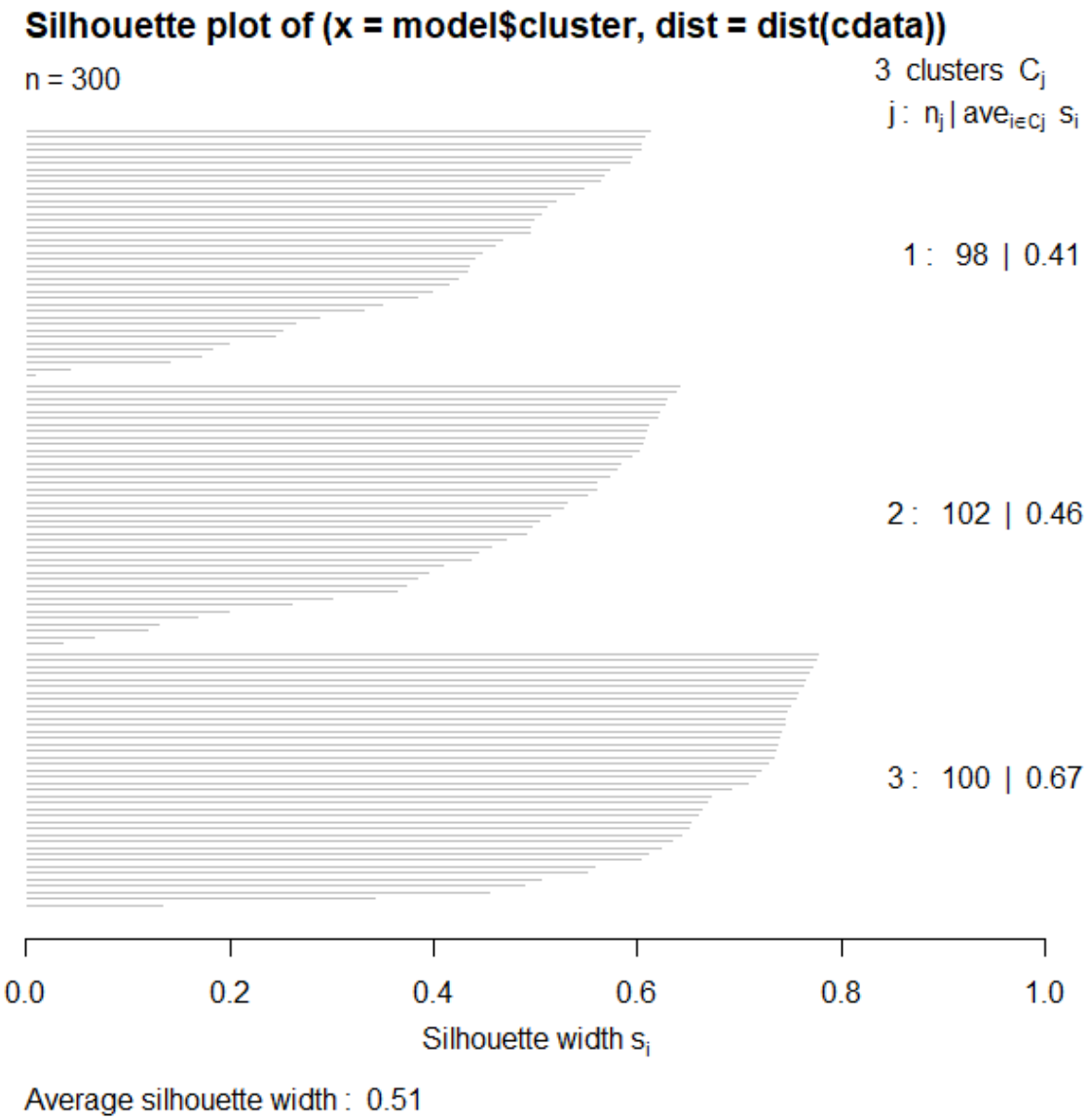
The following figure depicts the clustering outcome:



9.5.3 - Silhouette computation

For computing and plotting the silhouette coefficient (question (c)), we can use the following commands:

```
# Compute silhouette
model_silhouette = silhouette(model$cluster, dist(cdata))

# Plot silhouette
plot(model_silhouette)
```



In addition, we can compute the mean silhouette using the following command:

```
mean_silhouette = mean(model_silhouette[, 3])
```

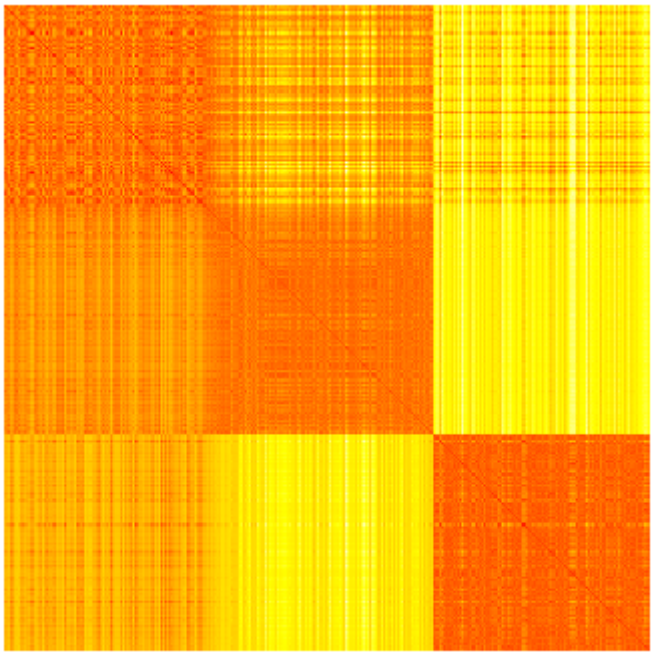
9.5.3 - Heatmap construction

In order to construct the heatmap, we first sort the data based on the cluster they belong using the following command:

```
cdata_ord = cdata[order(model$cluster),]
```

Then, we plot the heatmap using the command:

```
heatmap(as.matrix(dist(cdata_ord)), Rowv = NA, Colv = NA,
                col = heat.colors(256), revC = TRUE)
```



9.6 - k-Medoids Application

For applying the k-Medoids algorithm, we will use the data contained in the following table:

	Rank	Topic
Conference_1	High	SE
Conference_2	Low	SE
Conference_3	High	ML
Conference_4	Low	DM
Conference_5	Low	ML
Conference_6	High	SE

The above dataset presents the rank and the topic of 6 different conferences. SE satnds for Software Engineering, ML for Machine Learning and DM for Data Mining.

At first, we will construct our data frame containing the above dataset with the following commands:

```
# Vector that holds Rank data
Rank = c("High", "Low", "High", "Low", "Low", "High")

# Vector that holds Topic data
Topic = c("SE", "SE", "ML", "DM", "ML", "SE")

# Data frame
conferences = data.frame(Rank, Topic)
```

We can apply k-Medoids using the following commands:

```
# Import library
library(cluster)

# Create k-Medoids model
model = pam(conferences, 3)
```

We can display the medoids and the distribution of the data instances into clusters using the following commands:

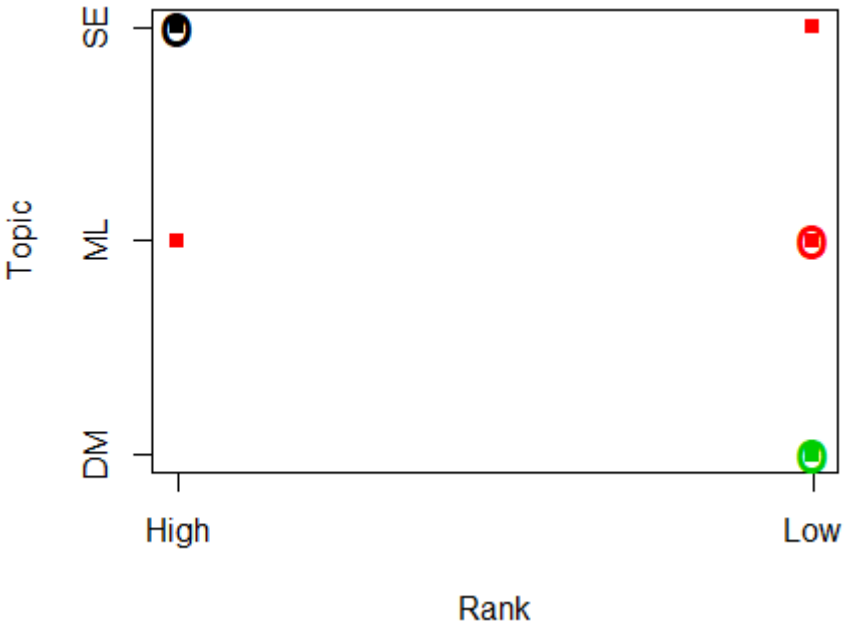
```
# Get medoids
model$medoids

# Get distribution of samples into clusters
model$clustering
```

We can also display the medoids by their names using the following command:

```
conferences[model$id.med,]
```

Finally, we can plot the data using the following commands:

```
# Get the different ranks
L1 = levels(conferences$Rank)

# Get the different topics
L2 = levels(conferences$Topic)

# Plot data
plot(model$data, xaxt = "n", yaxt = "n", pch = 15, col = model$cluster)

# Insert axes labels
axis(1, at = 1:length(L1), labels = L1)
axis(2, at = 1:length(L2), labels = L2)

# Display medoids
points(conferences[model$id.med,], col = 1:3, pch = "o", cex = 2)
```


SE
ML
DM
Topic
High
Low
Rank

Chapter 10 - Connectivity-based clustering

Connectivity-based clustering, also known as ***hierarchical clustering***, is based on the core idea that every instance of a given dataset is related with every other instance. The main idea behind the aforementioned assumption is that the degree of this relation is stronger for the "nearby" objects than to objects farther away. The term **nearby** implies that these algorithms connect *data instances* to form *clusters* based on their distance. A cluster can be described largely by the maximum distance needed to connect parts of the cluster. At different distances, different clusters will form, which can be represented using a dendrogram, which explains where the common name "hierarchical clustering" comes from: these algorithms do not provide a single partitioning of the data set, but instead provide an extensive hierarchy of clusters that merge with each other at certain distances.

10.1 - Hierarchical clustering in R

10.1.1 - Data Construction

In order to provide a detailed overview of how hierarchical clustering can be performed using R packages, let's consider the following simple dataset containing five instances of the 2-dimentional space.

	X	Y
x1	2	0
x2	8	4
x3	0	6
x4	7	2
x5	6	1

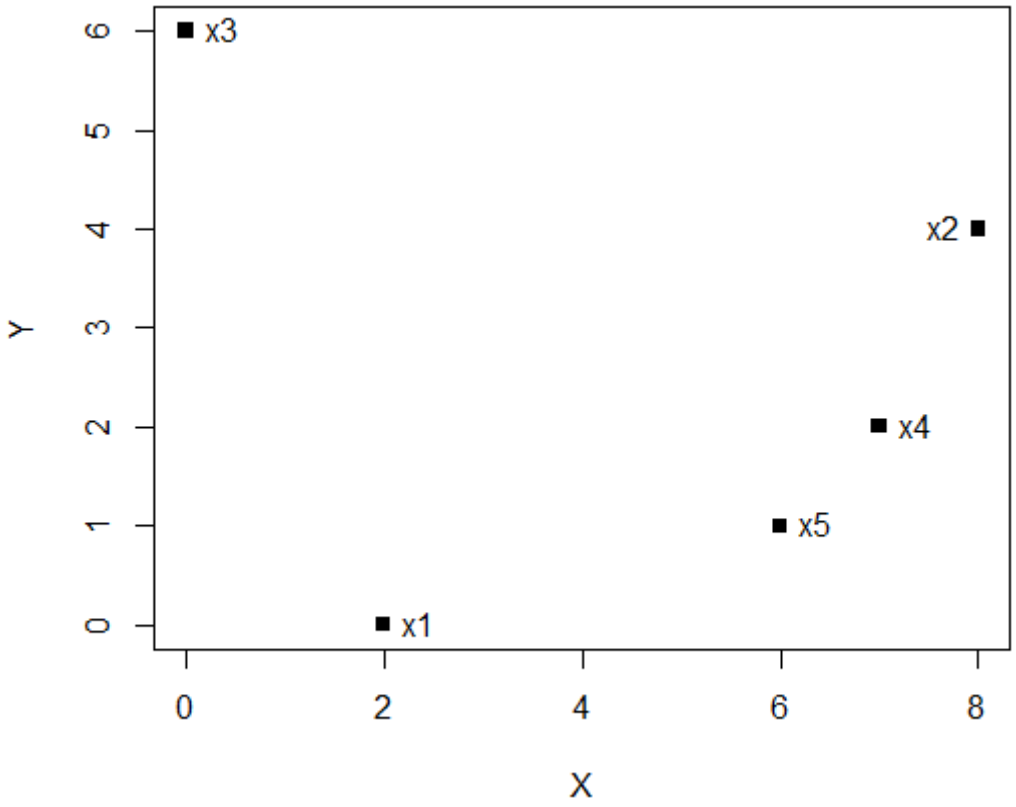
First of all, we need to properly construct our data frame containing the above dataset. In order to do so, we can use the following commands:

```
# Construct vector containing X-axis data
X = c(2, 8, 0, 7, 6)

# Construct vector containing Y-axis data
Y = c(0, 4, 6, 2, 1)

# Construct vector containing the data labels which correspond to the row names
rnames = c("x1", "x2", "x3", "x4", "x5")

# Create the data frame
hdata = data.frame(X, Y, row.names = rnames)
```

Once having created our data frame, we can plot our data using the following commands:

```
# Plot the data
plot(hdata, pch = 15)

# Add the data labels
text(hdata, labels = row.names(hdata), pos = c(4, 2, 4, 4, 4))
```

Our data is illutrated in the following figure:



10.1.2 - Hierarchical Clustering using Single and Complete Linkage

In this subsection, we will perform hierarchical clustering using single and complete linkage using the aforementioned dataset.

As already noted, hierarchical clustering is an iterative process that starts with the individual elements of a given dataset in order to progressively form clusters. These clusters are formed by merging elements that appear to have a strong connection. This connection is expressed using a distance metric. A very common distance metric is the Euclidean distance, which is given by the following equation:

$$\|x_1 - x_2\|_2 = \sqrt{\sum_i (x_1 - x_2)^2}$$

At first, we will go through the single linkage algorithm:

1st iteration:

The following table presents the distnce matrix for the aforementioned dataset.

	x1	**x2**	**x3**	**x4**	**x5**
x1	0	7.21	6.32	5.39	4.12
x2	7.21	0	8.25	2.24	3.61
x3	6.32	8.25	0	8.06	7.81
x4	5.39	2.24	8.06	0	***1.41***
x5	4.12	3.61	7.81	***1.41***	0

As shown in the table, the points x4 and x5 are the ones having the smallest distance. As a result, we link them together and thus we formulate our first cluster.

2nd iteration:

Since we are using single linkage, the distance between the cluster containing the points x4 and x5 and every other point is the minimum of the distance between this point and the points x4 and x5.

The following table presents the updated distance matrix.

	x1	**x2**	**x3**	**x4 - x5**
x1	0	7.21	6.32	4.12
x2	7.21	0	8.25	***2.24***
x3	6.32	8.25	0	7.81
x4 - x5	4.12	***2.24***	7.81	0

Again, using the same strategy, we link the point x2 with the cluster (x4 - x5).

3rd iteration:

In the next iteration, the distance matrix is updated again and the point x1 is linked with the cluster (x2 - x4 - x5).

	x1	**x2 - x4 - x5**	**x3**
x1	0	***4.12***	6.32
x2 - x4 - x5	***4.12***	0	7.81
x3	6.32	7.81	0

4th iteration:

Finally, the clustering is completed in the 4th iteration where the final point x3 is linked with the cluster (x1 - x2 - x4 - x5).

	x1 - x2 - x4 - x5	**x3**
x1 - x2 - x4 - x5	0	***6.32***
x3	***6.32***	0

At this point we will go through the same iterative process again, but for the case of complete linkage.

1st iteration:

The first step is the same as in the case of single linkage, where we compute the distance matrix of our original dataset and link the points x4 and x5 which are the ones having the smallest distance.

	x1	**x2**	**x3**	**x4**	**x5**
x1	0	7.21	6.32	5.39	4.12
x2	7.21	0	8.25	2.24	3.61
x3	6.32	8.25	0	8.06	7.81
x4	5.39	2.24	8.06	0	***1.41***
x5	4.12	3.61	7.81	***1.41***	0

2nd iteration:

As opposed to the case of single linkage, in complete linkage the distance between the cluster containing the points x4 and x5 and every other point is the maximum of the distance between this point and the points x4 and x5.

The following table presents the updated distance matrix.

	x1	**x2**	**x3**	**x4 - x5**
x1	0	7.21	6.32	5.39
x2	7.21	0	8.25	***3.61***
x3	6.32	8.25	0	8.06
x4 - x5	5.39	***3.61***	8.06	0

Based on the updated distances, we link the point x2 with the cluster (x4 - x5). Up to this point, the results are the same with the case of single linkage.

3rd iteration:

In the next iteration, the distance matrix is updated again and this time the closest point are found to be x1 and x3. As a result, they are linked together and formulate the cluster (x1 - x3)

	x1	**x2 - x4 - x5**	**x3**
x1	0	7.21	***6.32***
x2 - x4 - x5	7.21	0	8.25
x3	***6.32***	8.25	0

4th iteration:

Finally, the clustering is completed in the 4th iteration where the clusters (x1 - x3) and (x2 - x4 - x5) are linked together.

	x1 - x3	**x2 - x4 - x5**
x1 - x2 - x4 - x5	0	***8.25***
x3	***8.25***	0

10.1.3 - Modeling with R

We can use the following R command in order to compute the distance matrix.

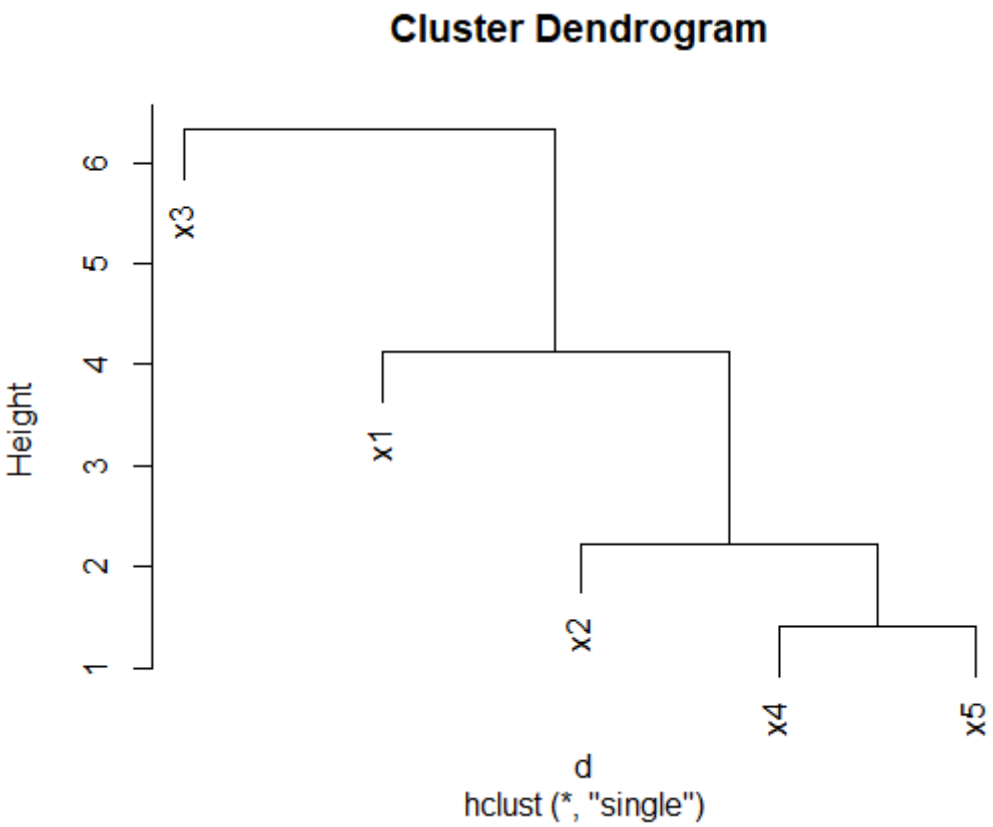
```
d = dist(hdata)
```

Have in mind that `hdata` is the already constructed data frame that contains our dataset (see Data Construction section).

We can use R commands in order to perform hierarchical clustering and plot the respective dendrogram that illustrates the formulated clusters.

```
# Perform clustering using single linkage
hc_single = hclust(d, method = "single")

# Plot the respective dendrogram
plot(hc_single)
```

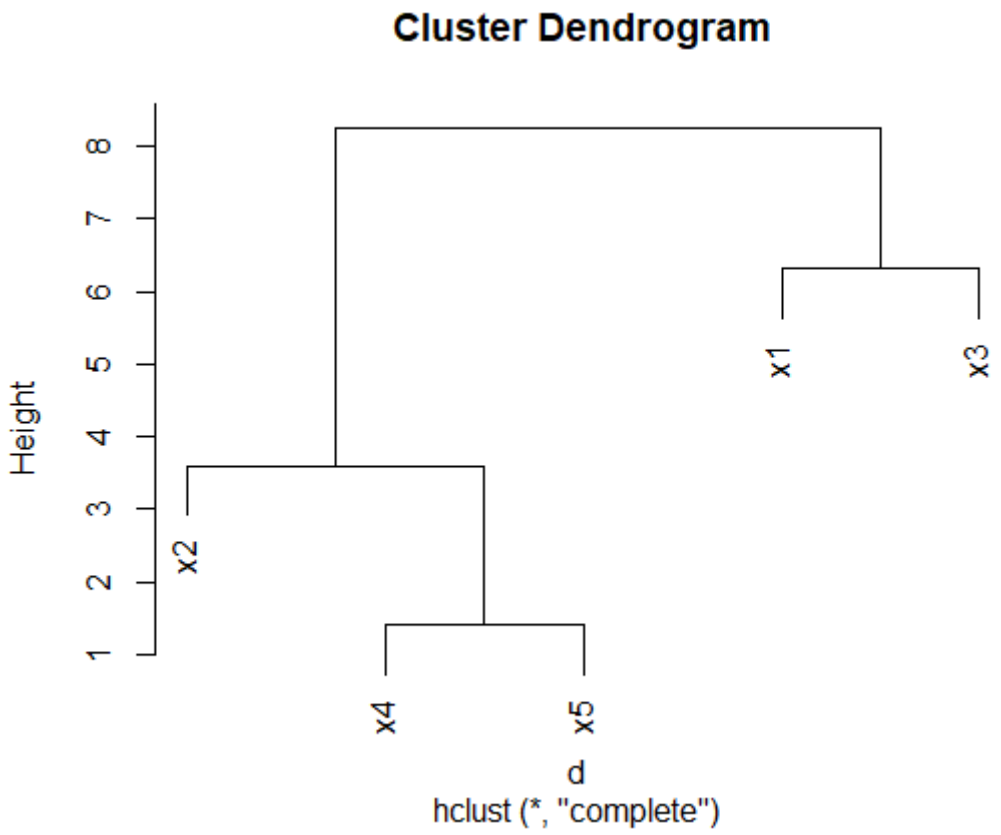
The following figure depicts the produced dengrogram.



We can use the same commands for the case of complete linkage.

```
# Perform clustering using complete linkage
hc_single = hclust(d, method = "complete")

# Plot the respective dendrogram
plot(hc_single)
```

The following figure depicts the produced dengrogram for the case of complete linkage.



It is worth noticing that the Y-axis of the dendrograms refers to the distance metric.

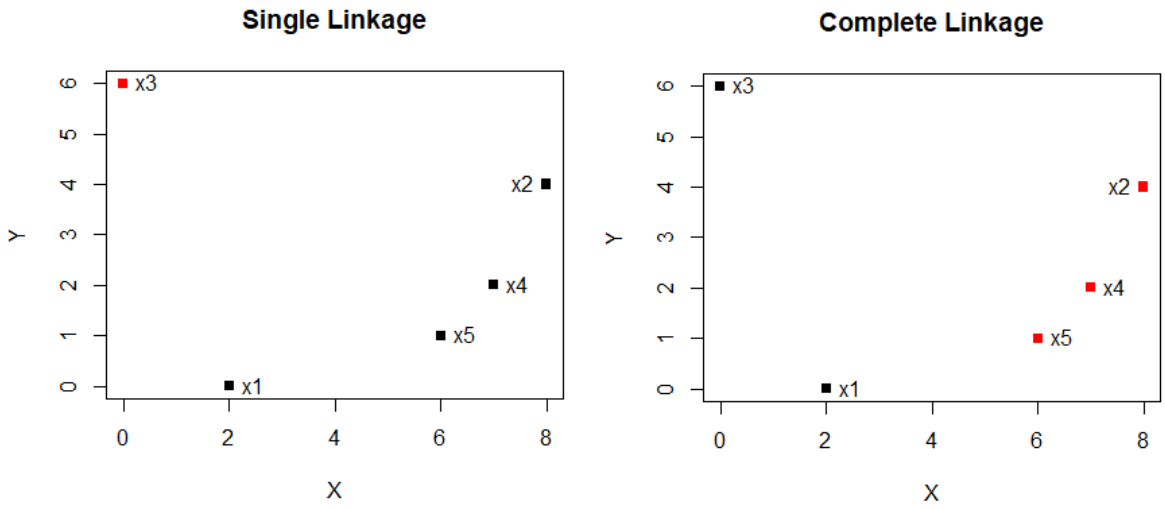
Once having performed the hierarchical clustering, We can split the original data into a number of clusters based on our preference using the following command:

```
# Split data into clusters. k refers to the number of clusters
clusters = cutree(hc_single, k = 2)
#       -------- OR --------
clusters = cutree(hc_complete, k = 2)
```

In order to demonstrate the formulated clusters from the two aforementioned approaches (single and complete linkage), we can plot using the following commands:

```
# Single linkage result
plot(hdata, col = clusters, pch = 15, main = "Single Linkage")
text(hdata, labels = row.names(hdata), pos = c(4, 2, 4, 4, 4))

# Complete linkage result
plot(hdata, col = clusters, pch = 15, main = "Complete Linkage")
text(hdata, labels = row.names(hdata), pos = c(4, 2, 4, 4, 4))
```

The following figure depicts the two formulated clusters for both the cases of single and complete linkage.



10.2 - Hierarchical Clustering Application and Evaluation

In order to apply hierarchical clustering into a real world example, we will use the dataset europe.txt. This dataset contains statistics regarding the Gross Domestic Product (GDP), the Inflation and the Unemployment in 28 EU countries. The first 6 entries are presented in the table below:

	GDP	Inflation	Unemployment
Austria	41600	3.5	4.2
Belgium	37800	3.5	7.2
Bulgaria	13800	4.2	9.6
Croatia	18000	2.3	17.7
Czech Republic	27100	1.9	8.5
Denmark	37000	2.8	6.1
...	...	...	...

We will this dataset in order to answer the following questions:

(a) Apply hierarchical clustering using complete linkage and plot the respective dendrogram

(b) Select the appropriate number of clusters based on silhouette coefficient

(c) Use the given data in order to form 7 clusters and produce the respective dendrogram that illustrates the results

(d) Create a 3D plot of the data using a different color for each cluster

(e) Calculate and plot the silhouette coefficient

10.2.1 - Create Hierarchical Clustering Model

At first, in order to answer (a), we will perform hierarchical clustering using the given data.

Our initial step involves importing the libraries necessary for answering the above questions along with importing the given data. This is done using the following commands:

```
# Import libraries
library(cluster)
library(scatterplot3d)

# Read data from file
europe = read.csv("europe.txt")
```

After reading the given dataset, we scale the data and we compute the distance matrix using Eucledean distance as our distance metric.

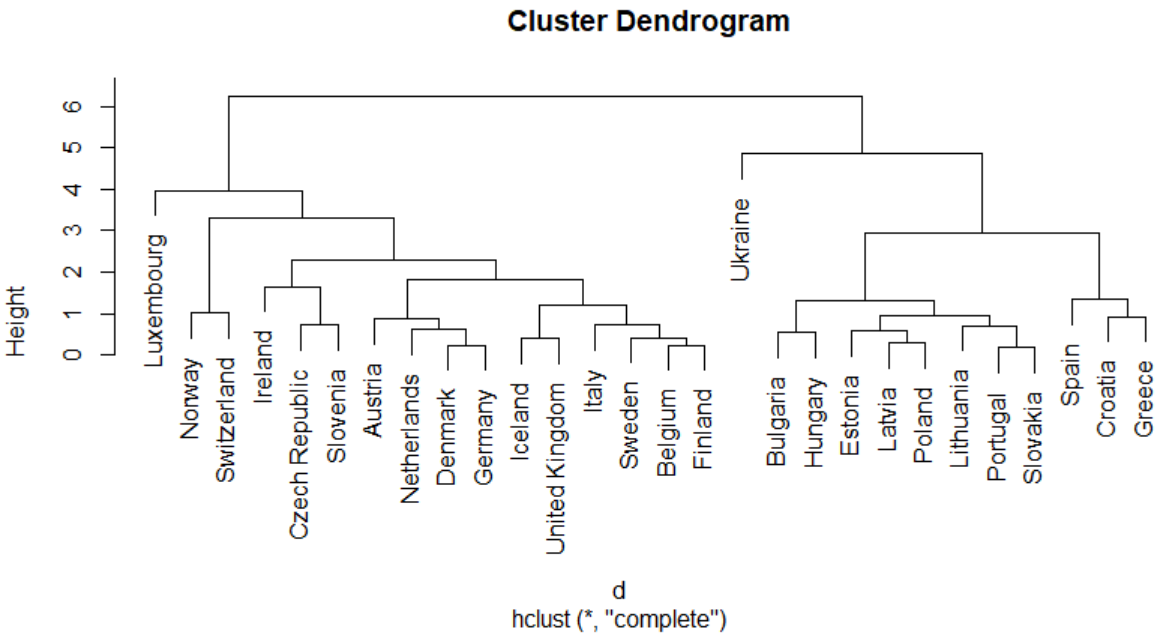
```
# Get distance matrix
d = dist(scale(europe))
```

After the computation of the distance matrix, we are ready to perform hierarchical clustering using complete linkage.

```
# Perform hierarchical clustering using complete linkage
hc <- hclust(d, method = 'complete')
```

We can plot the respective dendrogram (illustrated in the following Figure) using the following command:

```
# Plot dendrogram
plot(hc)
```



10.2.2 - Selecting Optimal Number of Clusters

Our next goal is to determine the optimal number of clusters. Our selection will be based on the values of silhouette coefficient which is used as a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation).

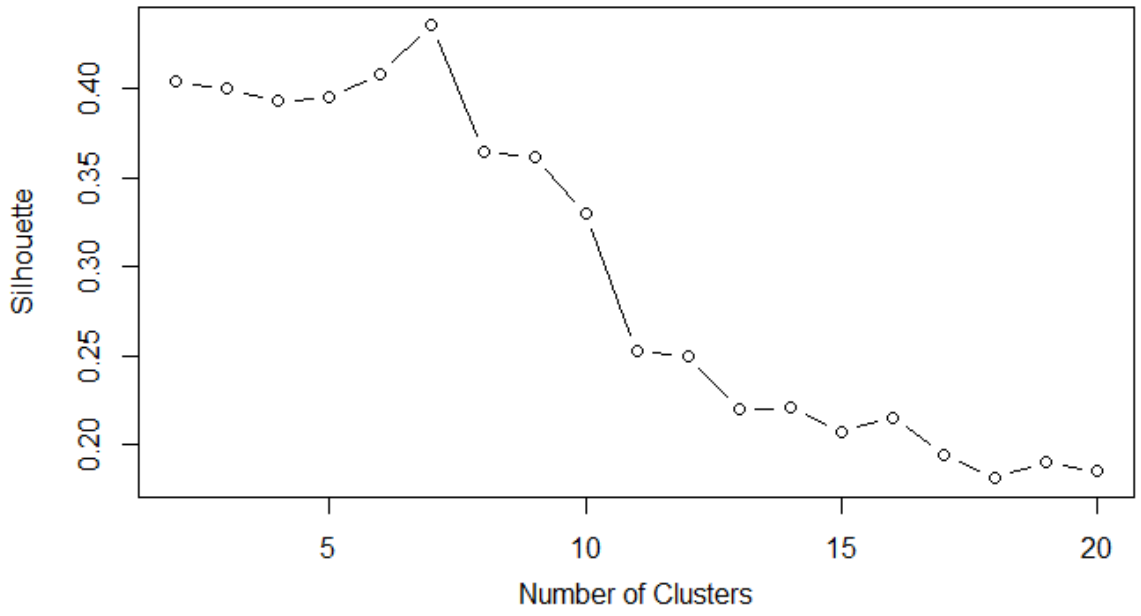
In order to determine the optimal number of clusters, we will use the constructed (from question (a)) hierarchical clustering model in order to form $2, \ldots, 20$ clusters and then select the case that maximizes silhouette. The following R code can be used to create the different splits and store the silhouette coeffiecient values:

```
# Initialize a vector that will hold the silhouette values
slc = c()

# Iterate over the number of clusters
for (i in 2:20){
    # Create clusters
    clusters = cutree(hc, k = i)

    # Calculate and store silhouette values
    slc [i-1] = mean(silhouette(clusters, d)[, 3])
}
```

After the computation of the silhouette values for the different clusterings, we can plot them using the following command:

```
# Initialize a vector that will hold the silhouette values
plot(2:20, slc, type="b", xlab="Number of Clusters",
                          ylab="Silhouette")
```

The following figure depicts the mean silhouette values for the different number of clusters. As shown form the graph, the optimal number of clusters that maximizes silhouette coefficient is 7.



10.2.3 - Visualizing Clusters

In order to answer question (c), we will first use the constructed hierarchical clustering model in order to form 7 clusters and then use these clusters to visualize the data of the original dataset into a 3D plot.

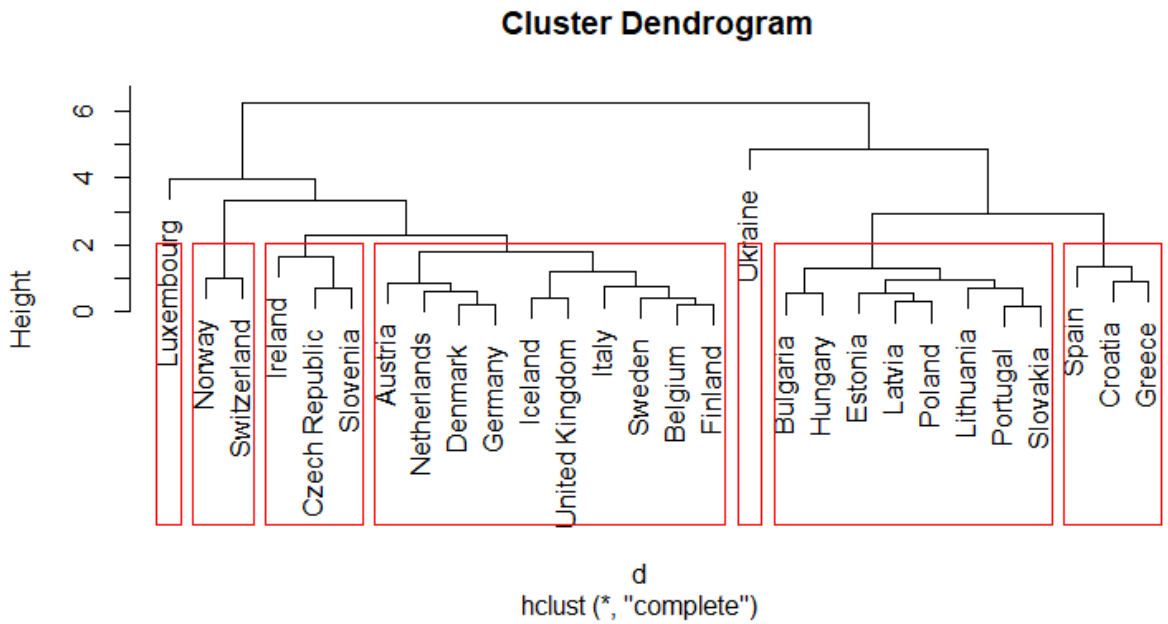
```
# Form the 7 clusters
clusters = cutree(hc, k = 7)
```

Uusing the aforementioned command, we have split our data into 7 clusters. The variable `clusters` holds the information of how the clusters are distributed among the data instances.

In order to display the constructed clusters on the produced dendrogram, we can use the following commands:

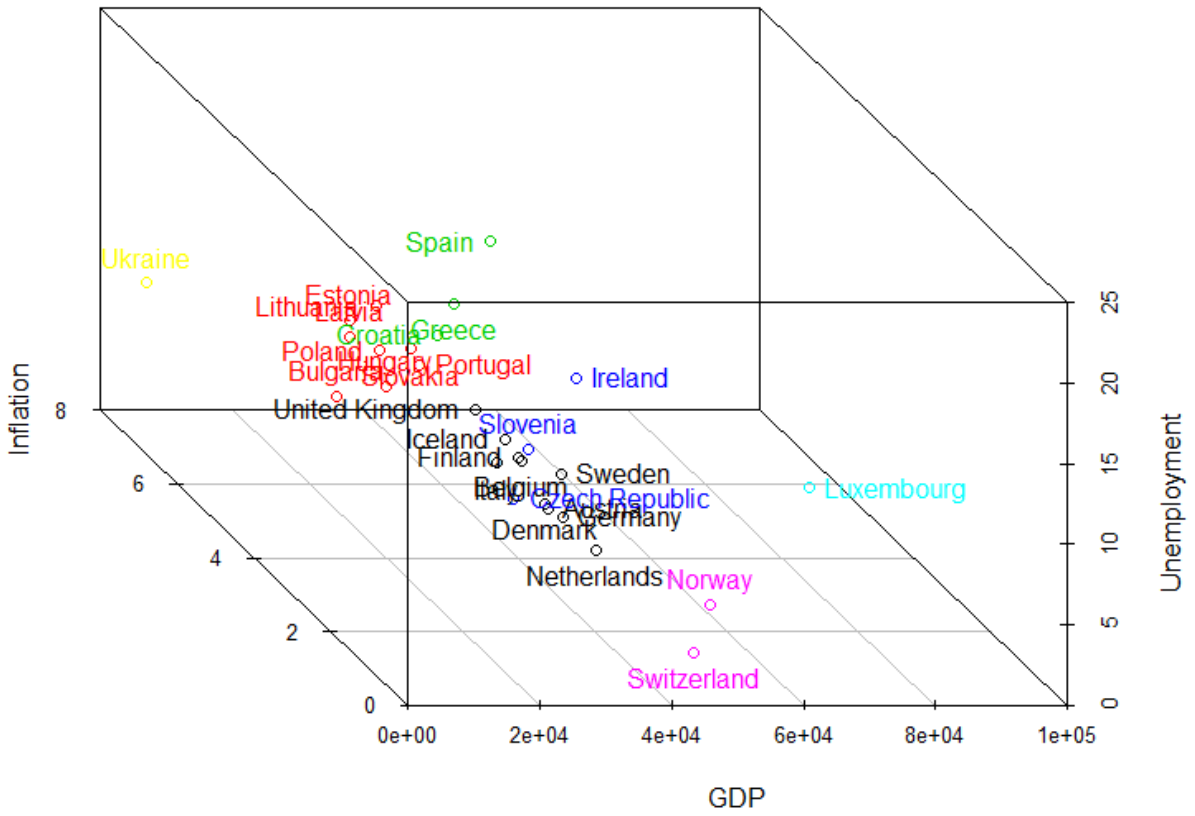
```
# Plot dendrogram
plot(hc)

# Display clusters
rect.hclust(hc, k = 7)
```

The following figure is the output of the aforementioned commands, where each red rectangle refers to each cluster.



We can also create a 3D plot of the clusters using the following commands:

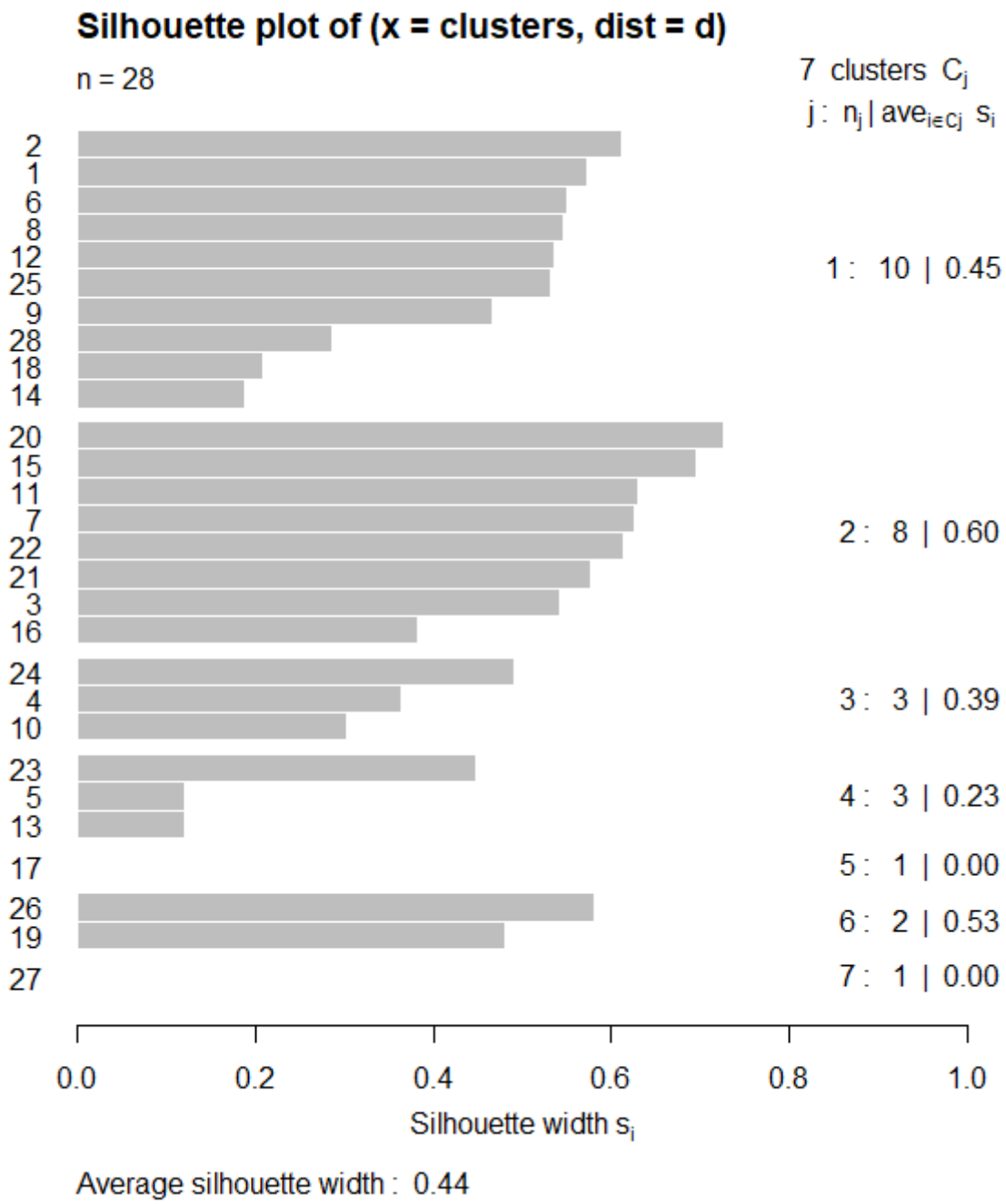
```
# Plot dendrogram
s3d = scatterplot3d(europe, angle = 125, scale.y = 1.5,
                                         color = clusters)

coords <- s3d$xyz.convert(europe)

text(coords$x, coords$y, labels=row.names(europe),
               pos=sample(1:4), col = clusters)
```



10.2.4 - Silhouette Plot

Finally, in order to calculate and plot the silhouette coefficient (question (d)), we can use the following commands:

```
# Calculate silhouette
model_silhouette = silhouette(clusters, d)

# Plot silhouette
plot(model_silhouette)
```


Silhouette plot of (x = clusters, dist = d)
n = 28
7 clusters C_j
j : n_j | $ave_{i \in Cj}$ s_i
2
1
6
8
12
25
9
28
18
14
20
15
11
7
22
21
3
16
24
4
10
23
5
13
17
26
19
27
1 : 10 | 0.45
2 : 8 | 0.60
3 : 3 | 0.39
4 : 3 | 0.23
5 : 1 | 0.00
6 : 2 | 0.53
7 : 1 | 0.00
0.0
0.2
0.4
0.6
0.8
1.0
Silhouette width s_i
Average silhouette width : 0.44

Chapter 11 - Density-based clustering

Density-based clustering is based on the ground truth that given a cetrain dataset, clusters can be defined as areas of higher density than the remainder of the dataset. Data instances within the sparse areas, which are required to separate clusters, are usually considered to be noise. The most famous and widely used density-based algorithm is ***DBSCAN*** (Density-based Spatial Clustering of Applications with Noise)which was introduced by Martin Ester, Hans-Peter Kriegel, Jorg Sander and Xiaowei Xu in 1996. The main idea behind the DBSCAN algorithm is that given a set of points in some n-dimantional space, points that are closely packed together (points with many nearby neighbors and thus of higher density) are grouped into clusters, while points that lie alone in low-density regions are considered as outliers.

11.1 - DBSCAN in R

In order to use DBSCAN in R, we need to use the cluster library which can be imported with the following command:

```
library(dbscan)
```

In order to perform clustering using DBSCAN and get the distribution of the input dataset into clusters, we can use the following commands:

```
# Perform clustering using DBSCAN
model = dbscan(ddata, eps = 1.0, minPts = 4)

# Get distribution of the input dataset into clusters
clusters = model$cluster
```

DBSCAN algorithm requires setting 2 main parameters:

- ***epsilon (eps)***, which specifies the maximum distance that two data points can have in order to be considered as part of the same cluster.

and

- ***minPoints (minPts)***, which specifies the minimum number of data points required to form a cluster.

11.2 - DBSCAN model construction

In an effort to construct a density-based clustering model using DBSCAN algorithm, we will use the data of the following table:

	X	Y
x1	2	10
x2	2	5
x3	8	4
x4	5	8
x5	7	5
x6	6	4
x7	1	2
x8	4	9

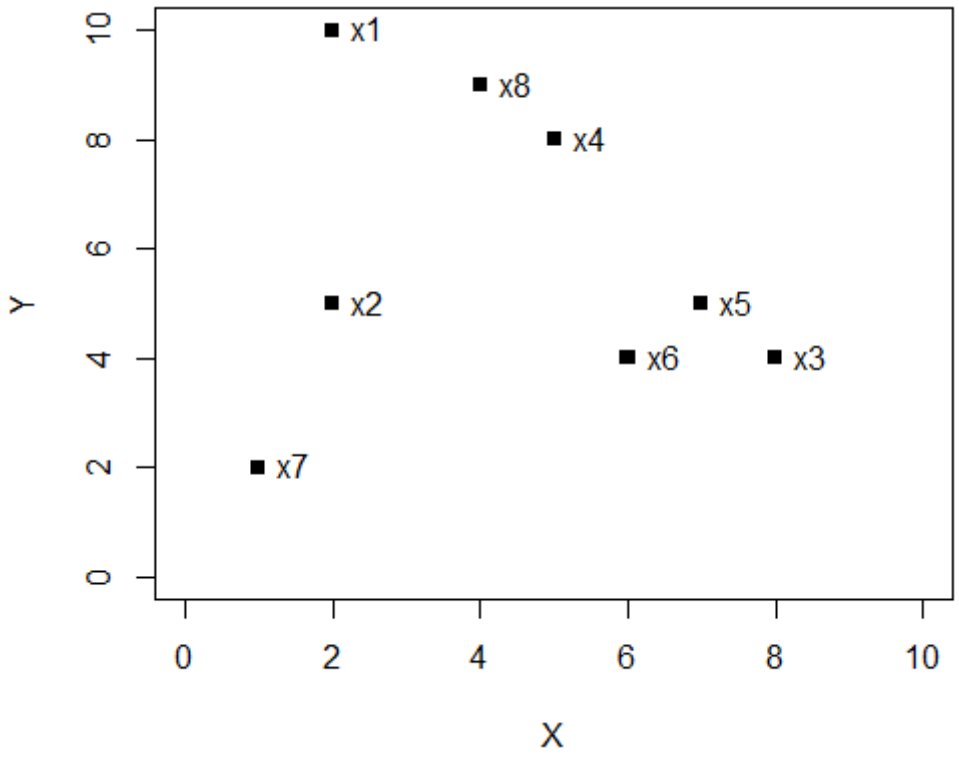
In order to visualize the data using R, we will first need to create the resperctive data frame usinf the following commands:

```
# Construct vector containing X-axis data
X = c(2, 2, 8, 5, 7, 6, 1, 4)

# Construct vector containing Y-axis data
Y = c(10, 5, 4, 8, 5, 4, 2, 9)

# Construct vector containing the data labels which correspond to the row names
rnames = c("x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8")

# Create the data frame
ddata = data.frame(X, Y, row.names = rnames)
```

Once having created our data frame, we can plot our data using the following commands:

```
# Plot the data
plot(ddata, pch = 15)

# Add the data labels
text(ddata, labels = row.names(hdata),
                pos = c(4, 2, 4, 4, 4, 4, 4, 4))
```

Our data is illutrated in the following figure:



11.3 - DBSCAN calculation

In this subsection, we will apply DBSCAN algorithm using minPoints = 2 and epsilon = {2, 3.5} in order to distribute the given data into clusters.

- ***minPoints = 2 and epsilon = 2***

We first need to compute the distance matrix for the given dataset which is given in the following table:

	x1	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**
x1	0.00							
x2	5.00	0.00						
x3	8.49	6.08	0.00					
x4	3.61	4.24	5.00	0.00				
x5	7.07	5.00	**1.41**	3.61	0.00			
x6	7.21	4.12	**2.00**	4.12	**1.41**	0.00		
x7	8.06	3.16	7.28	7.21	6.71	5.39	0.00	
x8	2.24	4.47	6.40	**1.41**	5.00	5.39	7.62	0.00

Given the fact that minPoints value is 2, each cluster should contain at least two points. In addition, based on the epsilon values, which is 2, DBSCAN forms the following clusters:

$$C_1 : \{x_3, x_5, x_6\} \quad and \quad C_2 : \{x_4, x_8\}$$

The remaining points are considered as noise.

- ***minPoints = 2 and epsilon = 3.5***

	x1	x2	x3	x4	x5	x6	x7	x8
x1	0.00							
x2	5.00	0.00						
x3	8.49	6.08	0.00					
x4	3.61	4.24	5.00	0.00				
x5	7.07	5.00	**1.41**	3.61	0.00			
x6	7.21	4.12	**2.00**	4.12	**1.41**	0.00		
x7	8.06	**3.16**	7.28	7.21	6.71	5.39	0.00	
x8	**2.24**	4.47	6.40	**1.41**	5.00	5.39	7.62	0.00

In the case of epsilon = 3.5, DBSCAN forms the following clusters:

$$C_1 : \{x_1, x_4, x_8\}, \quad C_2 : \{x_2, x_7\} \quad and \quad C_3 : \{x_3, x_5, x_6\}$$

As we can see no point is considered as noise.

11.4 - DBSCAN clustering with R

In order to perform clustering using DBSCAN with R, we can use the following command:

```
model = dbscan(ddata, eps = 2, minPts = 2)
```

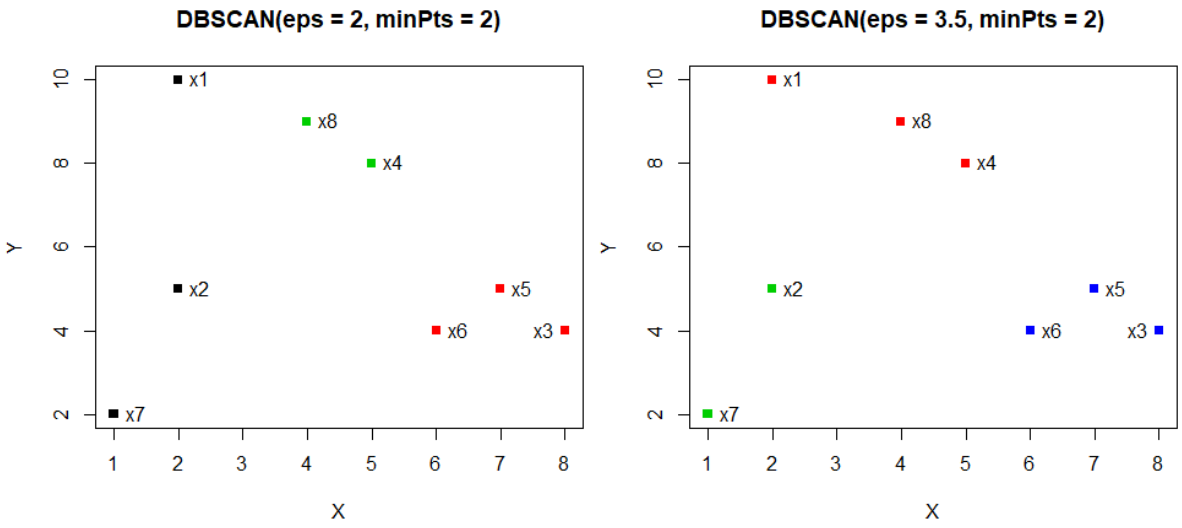
In order to get the distribution of the data point into clusters, we can use the following command:

```
clusters = model$cluster
```

Finally, we can plot the clustering results using the following commands:

```
# Plot clusters
plot(ddata, col=clusters+1, pch=15, main="DBSCAN(eps = 2, minPts = 2)")

# Add data labels
text(ddata, labels = row.names(ddata), pos = 4)
```

In the following figure, we can see the formulated clusters from the previous subsection.



11.5 - Density-based Clustering Application

In this section, we are going to demonstrate the application of DBSCAN algorithm in a real life application example. In this context, we have a dataset that contains 2,000 different 2D data points. A summary of the dataset is given in the following table:

	X1	X2
Min.	5.09	7.54
1st Qu.	10.45	9.78
Median	13.30	12.01
Mean	13.28	12.03
3rd Qu.	16.13	14.27
Max.	21.43	16.48

We will use the aforementioned data in order to answer the following questions:

(a) Plot the given data points

(b) Apply k-Means algorithm in order to create two clusters and plot the result using different color for each cluster. Is the result satisfying? Do the created clusters correspond to the actual ones?

(c) Optimize esp parameter for DBSCAN using kNN distance (use k = 10).

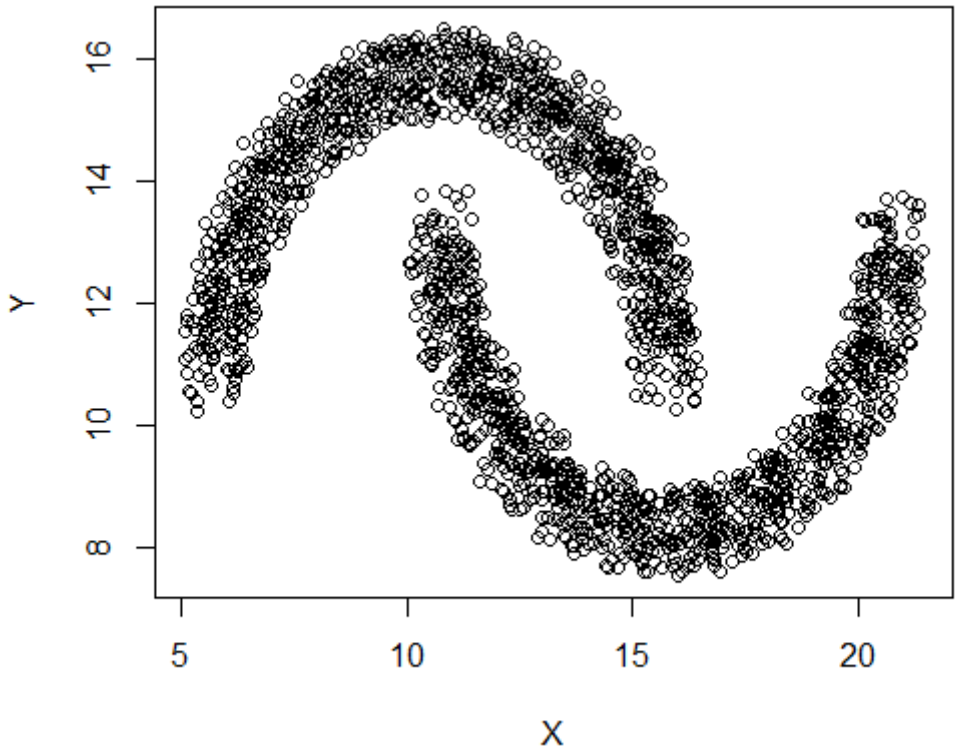
(d) Apply DBSCAN using eps = 0.4 and minPts = 10 into the given dataset. Plot the result using different color for each cluster and black for the data points considered as noise.

11.5.1 - Modeling using k-Means

Our first step is to read and plot the given dataset using the following commands:

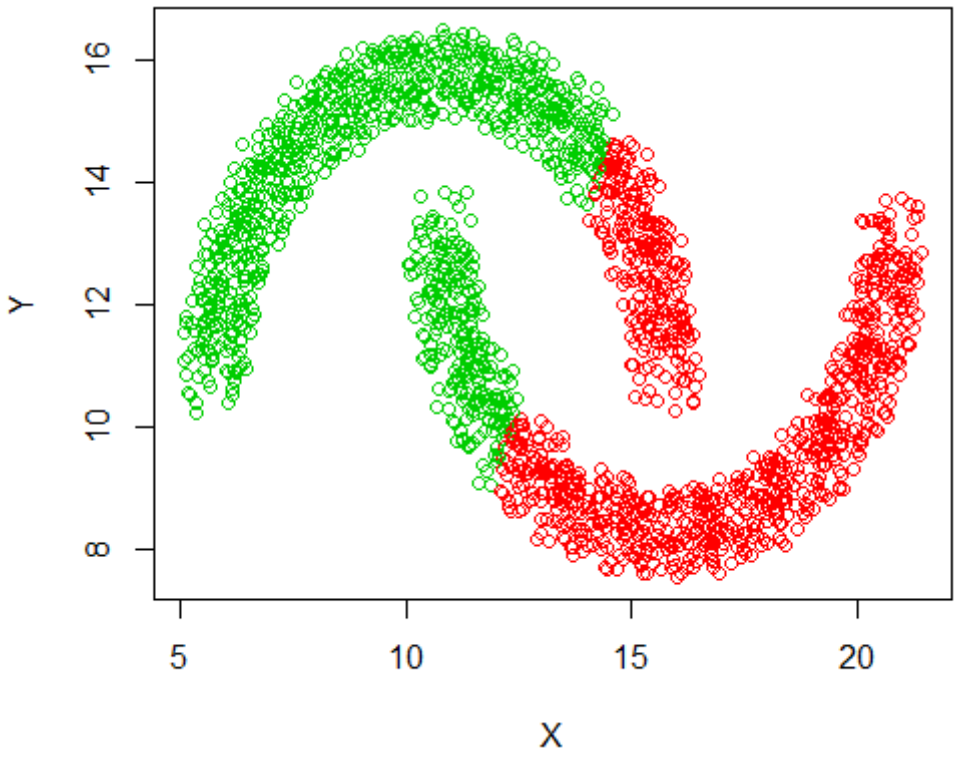
```
# Plot library
library(dbscan)

# Read data
mdata = read.csv("mdata.txt")

# Plot data
plot(mdata)
```

The following figure illustrates the given dataset. As we can see from the plot, we can identify two distinguished clusters.



Then, we apply k-Means and plot the clustering results using the following commands:

```
# Construct k-Means model
model = kmeans(mdata, 2)

# Plot data
plot(mdata, col = model$cluster + 1)
```



As we can see, k-Means is not able to suffieciently split the data into clusters. This result originates from the fact that k-Means relies on the Euclidean distance and thus cannot distingush clusters organized in complex structures.

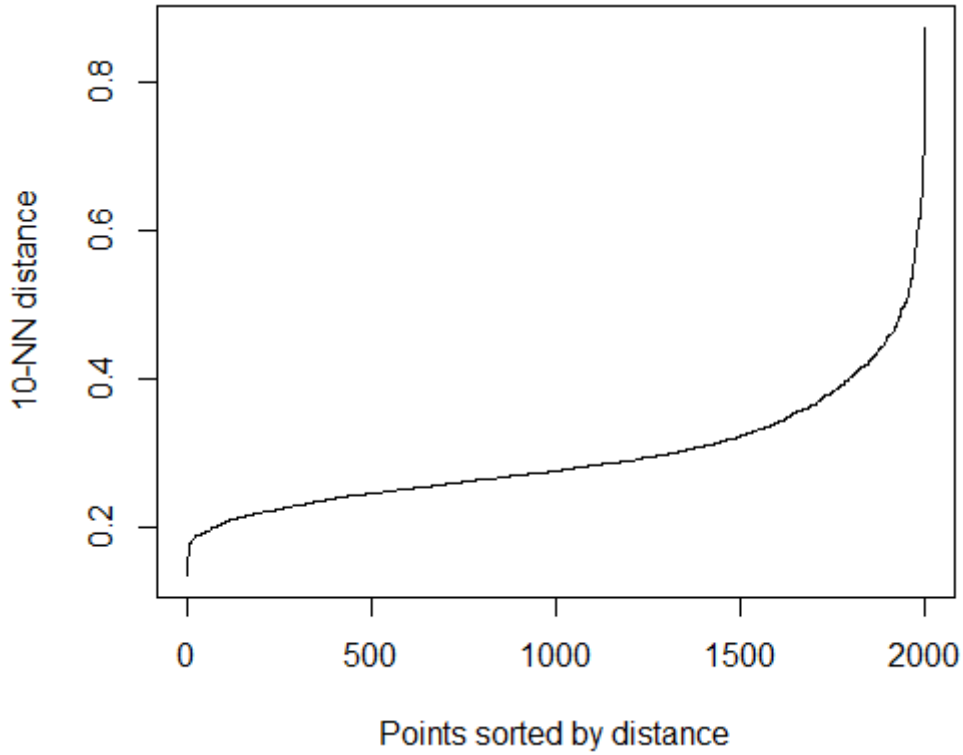
11.5.2 - Optimizing parameter eps

In order to optimize the parameter eps of the DBSCAN algorithm (quaestion (c)), we will use kNN distance. We can use the following R command in order to compute the destance of every data point with its 10 nearest neighbours:

```
knndist = kNNdist(mdata, k = 10)
```

Then, we can plot the respective graph after sorting the data into ascending order:

```
# Get the distances of the 10 nearest neighbours for each point
kdist = knndist[, 10]

# Plot distances
plot(sort(kdist), type = 'l', xlab = "Points sorted by distance",
                              ylab = "10-NN distance")
```



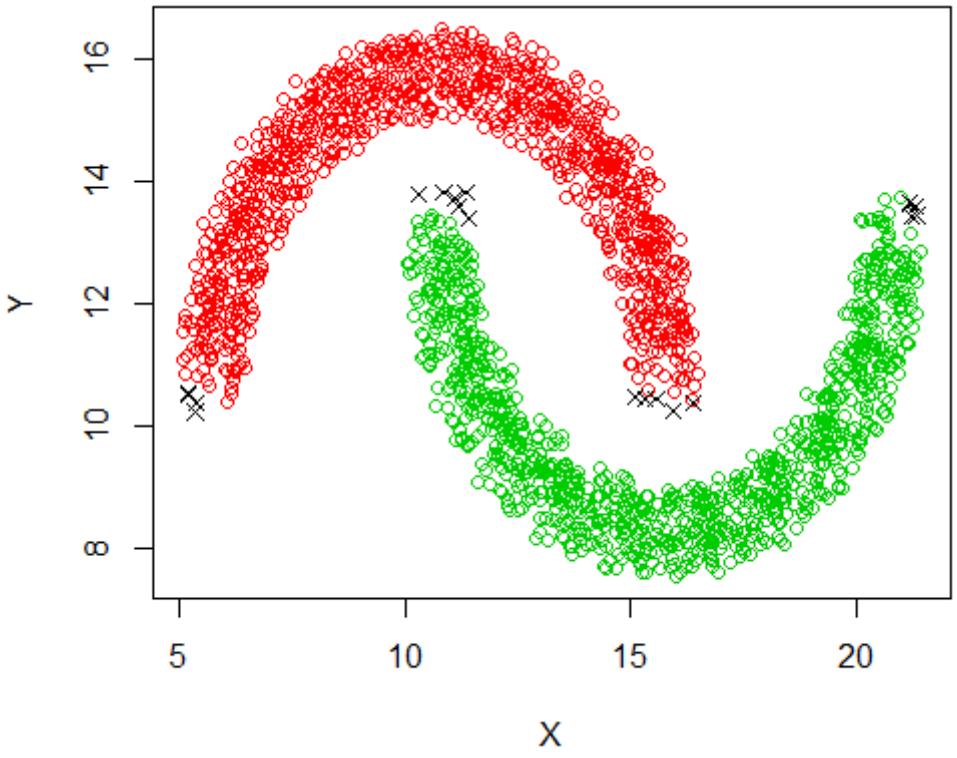
In order to optimize eps, we follow the elbow method. As we can see from the graph, the optimized eps valus are in the interval [0.35, 0.45].

11.5.3 - Modeling using DBSCAN

Lastly, we can answer question (d) by applying DBSCAN on the given dataset using eps = 0.4 and minPoints = 10.

```
# Construct DBSCAN model
model = dbscan(mdata, eps = 0.4, minPts = 10)

# Plot distances
plot(mdata, col = model$cluster + 1, pch = ifelse(model$cluster, 1, 4))
```

The following figure depicts the clustering result. As we can see, DBSCAN is able to efficiently split the given dataset into clusters.


Y
16
14
12
10
8
5
10
15
20
X

Chapter 12 - Distribution-based clustering

12.1 - Theoretical background

Distribution-based clustering is closely related to statistics and is based on the fact that clusters can easily be defined as objects including data instances belonging most likely to the same distribution. While the theoretical foundation of the methods that apply to this clustering category is excellent, they suffer from one key problem known as *overfitting*, unless cetrain constraints are set into the complexity of the model. A more complex model will usually be able to explain the data better, which makes choosing the appropriate level of complexity inherently difficult. One prominent and widely used method is known as ***Gaussian Mixture Models*** or ***GMMS*** (using the ***Expectation-Maximization algorithm*** or ***EM***).

12.1.1 - The Expectation-Maximization algorithm

In statistics, an Expectation-Maximization (EM) algorithm is an iterative method to find maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models, where the model depends on unobserved latent variables. In practice EM algorithm maximizes the following expression:

$$X : dataset, \quad Z : latent \;\; variables, \quad \theta : set \;\; of \;\; parameters$$

$$\ln P(X/\theta) = \ln \sum_{Z} P(X, Z/\theta)$$

The EM iteration alternates between performing an expectation step (E-step), which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and a maximization step (M-step), which computes parameters maximizing the expected log-likelihood found on the E step. These parameter estimates are then used to determine the distribution of the latent variables in the next E-step.

E-step:

$$Given \;\; \theta_{old} : \quad find \quad P(Z/X, \theta_{old})$$

M-step:

$$\theta_{new} = argmax_{\theta}(\sum_{Z} P(Z/X, \theta_{old}) \ln P(X, Z/\theta))$$

The EM algorithm reaches convergence when the log-likelihood value is no longer updated or the updat is smaller than a given threshold.

12.1.2 - Gaussian Mixture Models - GMMs

A Gaussian Mixture Model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

Generally, a mixture of *k* normal distributions is given by the following equation:

$$P(x) = \sum_{k=1}^{K} \pi_k N(x/\mu_k, \Sigma_k)$$

$$where$$

$$\pi_k : Mixing \;\; Coefficients \in [0,1] \;\; and \;\; \sum_{k=1}^{K} \pi_k = 1$$

$$\mu_k, \Sigma_k : Mean \;\; and \;\; Covariance \;\; of \;\; distribution \;\; k$$

$$N : N(x/\mu, \Sigma) = \frac{1}{2\pi^{n/2}\left|\Sigma\right|^{1/2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

GMMs use EM as the optimization algorithm that enables calculating the latent variables of the distributions that describe the goven data. As a result, the EM algorithm in the case of GMMs maximizes the following quantity:

$$\ln P(X/\pi, \mu, \Sigma) = \sum_{n=1}^{N} \ln(\sum_{k=1}^{K} \pi_k N(x_n/\mu_k, \Sigma_k))$$

$$where \quad X = \{x_1, x_2, \ldots, x_n\} \longrightarrow Given \quad dataset$$

12.2 - Modeling Gaussian Mixture Models using EM

In order to perform clustering using GMMs we will use a dataset that contains 1,000 1D data instances that originate from two different distributions. Regarding the distributions, we know that their standard deviation is 1. A summary of the data is given in the following table:

	D1_X	D2_X
Min.	-1.905	3.992
1st Qu.	0.223	6.322
Median	0.907	6.944
Mean	1.000	6.985
3rd Qu.	1.664	7.681
Max.	3.659	10.810

Using the above data, we will answer the following questions:

(a) Plot the data (1 one dimention) along with the probability density function using a different color for each distribution.

(b) Calculate the means (μ_1, μ_2) and the latent variables λ_1, λ_2 for each one of the normal distributions.

(d) Plot the estimated and the real probability density function of the given data in the same figure.

12.2.1 - Data Plot

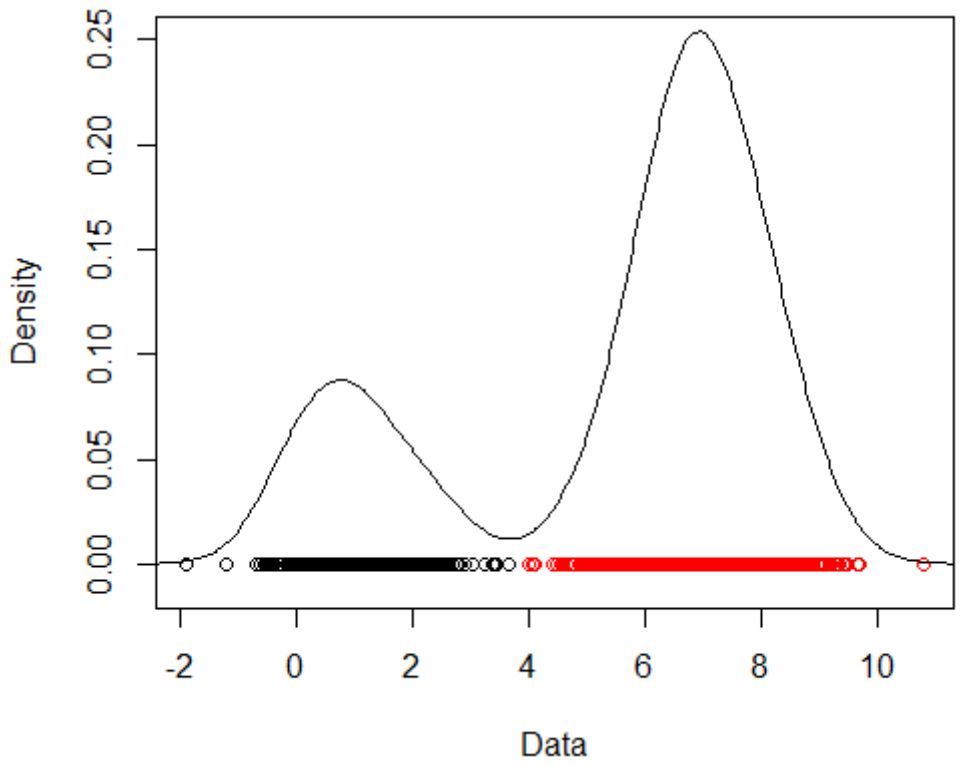
In order to answer the first question, we will first read the data using the following commands:

```
# Read data
gdata = read.csv("gdata.txt")

# Save 1D data in the vector x
x = gdata[, 1]

# Save cluster index in the vector y
y = gdata[, 2]
```

In order to plot the data along with the probability density function, we can use the following commands:

```
# Plot data
plot(data.frame(x, 0), ylim = c(-0.01, 0.25), col = y,
                                    xlab = "Data",
                                    ylab = "Density")

# Plot probability density function
lines(density(x), col = y)
```

The following figure illustrates the given dataset along with the probability density function. The data from each cluster are illustrated with a different color (black and red)



12.2.2 - Manual calcultion of the parameters of the normal distibutions

In this subsection, we will use the given dataset in order to calculate the parameters of the normal distributions using EM algorithm. At first, we will choose random initial values for the parameters μ and λ:

$$Initialization: \mu_1 = 0, \mu_2 = 1, \lambda_1 = 0.5, \lambda_2 = 0.5$$

$$\Theta = \{\theta_1, \theta_2\} = \{(0, 1, 0.5), (1, 1, 0.5)\}$$

E-step:

In this step, we want to calculate the probability for a given data point to originate from a certain normal distibution. This probability is given by the following equation:

$$P(D_j/x_i, \theta_j) = \frac{\lambda_j \cdot P(x_i/\theta_j)}{\lambda_1 \cdot P(x_i/\theta_1) + \lambda_2 \cdot P(x_i/\theta_2)}$$

$$where: j = 1, 2 \quad \lambda_{init} = \{\lambda_1, \lambda_2\} = \{0.5, 0.5\}$$

At this point, we will compute the above probability for the given point $x_i = 2$:

$$P(x_i/\theta_j) = \frac{1}{\sqrt{2\pi\sigma_j}} e^{-\frac{(x_i - \mu_j)^2}{2\sigma_j}}$$

$$P(2/\theta_1) = \frac{1}{\sqrt{2\pi \cdot 1}} e^{-\frac{(2-0)^2}{2 \cdot 1}} = \frac{1}{\sqrt{2\pi}} e^{-2} = 0.054$$

$$P(2/\theta_2) = \frac{1}{\sqrt{2\pi \cdot 1}} e^{-\frac{(2-1)^2}{2 \cdot 1}} = \frac{1}{\sqrt{2\pi}} e^{-1/2} = 0.242$$

The probabilities of the given point to originate from distributions 1 and 2 are given by the following two equations:

$$P(D_1/2, \theta_1) = \frac{0.5 \cdot 0.054}{0.5 \cdot 0.054 + 0.5 \cdot 0.242} = 0.182$$

$$P(D_2/2, \theta_2) = \frac{0.5 \cdot 0.242}{0.5 \cdot 0.054 + 0.5 \cdot 0.242} = 0.818$$

Given the aforementioned results, the point $x_i = 2$ originates from distribution 1 with probability 0.182 and from distribution 2 with probability 0.818. Once having computed the aforementioned probabilities for all 1,000 given data points, we will proceed in the maximization step of the EM algorithm.

M-step:

In this step, we will calculate the parameters μ_1, μ_2 that maximize the probabilities $P(D_1/X_i, \theta_1), P(D_2/X_i, \theta_2)$. In order to compute the parameters μ_1, μ_2, we will use the following equations:

$$\mu_1 = \frac{\sum_{i=1}^{1000} x_i \cdot P(D_1/x_i, \theta_1)}{\sum_{i=1}^{1000} P(D_1/x_i, \theta_1)}$$

$$\mu_2 = \frac{\sum_{i=1}^{1000} x_i \cdot P(D_2/x_i, \theta_2)}{\sum_{i=1}^{1000} P(D_2/x_i, \theta_2)}$$

The above equations calculate the weighted average of all data points, where the weights are actually the probabilities for each data point to originate from one of the two distributions.

The mixing coefficients are given by the following equations:

$$\lambda_1 = \frac{\sum_{i=1}^{1000} P(D_1/x_i, \theta_1)}{1000}$$

$$\lambda_2 = \frac{\sum_{i=1}^{1000} P(D_2/x_i, \theta_2)}{1000}$$

After computing the values of the parameters $\mu_1, \mu_2, \lambda_1, \lambda_2$, the log-likelihood of the given dataset is given by the following equation:

$$\ln(P(X/\theta)) = \sum_{i=1}^{1000} \ln(\sum_{j=1}^{2} \lambda_j \cdot P(x_i/\theta_j))$$

$$= \sum_{i=1}^{1000} \ln(\lambda_1 \cdot P(x_i/\theta_1) + \lambda_2 \cdot P(x_i/\theta_2))$$

Once having computed the log-likelihood of the dataset, we compare it with the value of the previous iteration. If the value has not changed more than a given threshold (espilon value), then the algorithm converges. Otherwise, we return back in the E-step to re-calculate the new parameters' values and continue the execution.

12.2.3 - Calculting the parameters of the normal distibutions using R

The R algorithm that enables us to compute the parameters of the normal distibutions is the following and involves two main steps, the initialization and the loop until convergence:

- ***Step 1: Initialize the means and the latent variables***

```
# Initialize means
mu = c(0, 1)

# Initialize lambdas
lambda = c(0.5, 0.5)

# Set stopping criterion
epsilon = 1e-08

# Compute initial log-likelihood
log_likelihood = sum(log(lambda[1] * dnorm(x, mean = mu[1], sd = 1) +
                         lambda[2] * dnorm(x, mean = mu[2], sd = 1)))
```

- ***Step 2: Loop until convergence***

```
repeat {
  # ------ Expectation step ------
    # Find distributions given mu, lambda (and sigma)
    T1 <- dnorm(x, mean = mu[1], sd = 1)
    T2 <- dnorm(x, mean = mu[2], sd = 1)
    P1 <- lambda[1] * T1 / (lambda[1] * T1 + lambda[2] * T2)
    P2 <- lambda[2] * T2 / (lambda[1] * T1 + lambda[2] * T2)

  # ------ Maximization step ------
    # Find mu, lambda (and sigma) given the distributions
    mu[1] <- sum(P1 * x) / sum(P1)
```

```
    mu[2] <- sum(P2 * x) / sum(P2)
    lambda[1] <- mean(P1)
    lambda[2] <- mean(P2)

    # Calculate the new log likelihood (to be maximized)
    new_log_likelihood = sum(log(lambda[1] * dnorm(x, mean = mu[1], sd = 1) +
                      lambda[2] * dnorm(x, mean = mu[2], sd = 1)))

    # Print the current parameters and the log likelihood
    cat("mu =", mu, " lambda =", lambda, " log_likelihood =", new_log_likelihood,\
 "\n")

    # Break if the algorithm converges
    if (new_log_likelihood - log_likelihood <= epsilon) break
    log_likelihood = new_log_likelihood
}
```

12.2.4 - GMMs model construction using mixtools

In order to answer question (c), we will construct a GMMs clustering model using R. The library we are going to use is mixtools:

```
library(mixtools)
```

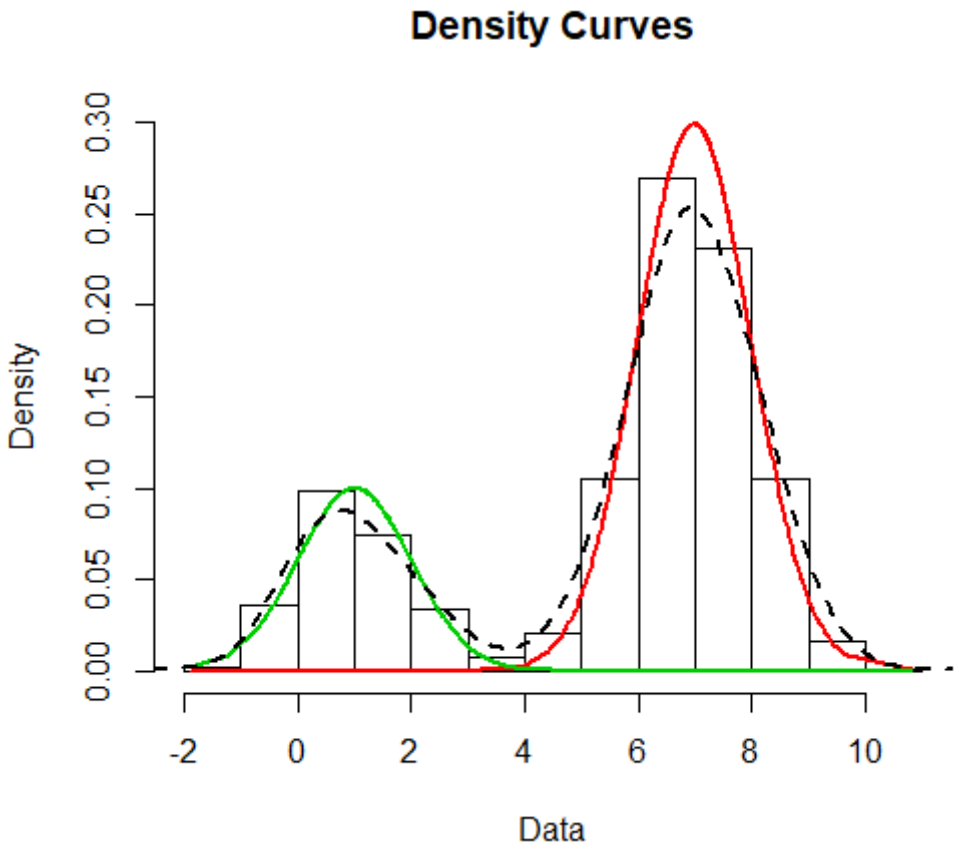
We can use the expectation maximization (EM) algorithm and overview the final calculated parameters using the following commands:

```
# Apply EM using the aforementioned initialization
model <- normalmixEM(x, mu = c(0,1), sd.constr = c(1,1))

# Get means of the distrubitions
model$mu

# Get lambda values
model$lambda

# Get final log-likelihood
model$loglik
```

We can plot both the original and the estimated probability density functions (into the same plot) using the following commands:

```
# Plot estimated probability density function
plot(model, which = 2)

# Plot original probability density function
lines(density(x), lty = 2, lwd = 2)
```

The following figure illustrates the probability density functions. The dashed line refers to the actual density, while the straight line to the estimated. As we can see, the estimation lies very close to the actual density.



12.3 - GMMs Clustering Application

In this section, we are going to demonstrate the application of GMMs in a real life application example. In this context, we have a dataset that contains 300 different 2D data points that originate from three different distributions. A summary of the two distibutions is given in the following table:

	D1_X1	D1_X2	D1_X1	D1_X2
Min.	3.33	5.57	17.64	5.47
1st Qu.	8.29	9.01	22.43	8.56
Median	10.07	10.22	24.91	10.02
Mean	1.32	1.53	-4.72	-3.97
3rd Qu.	11.88	11.38	27.44	11.48
Max.	17.32	14.80	31.27	23.47

Then, we will use the given dataset in order to answer the following questions:

(a) Plot the given dataset using a different color for each cluster.

(b) Constuct a GMMs model inorder to cluster the provided dataset into 3 clusters. Set epsilon value to 0.1.

(c) Show how the EM algorithm converges and plot the data using the estimated probability density function

(d) Assign each data point to the cluster with the highest probability and plot the formulated clusters along with their centroids.

(e) Calculate and plot silhouette coefficient

(f) Show the heatmap of the data after sorting them based on the above clustering results

12.3.1 - Dataset construction

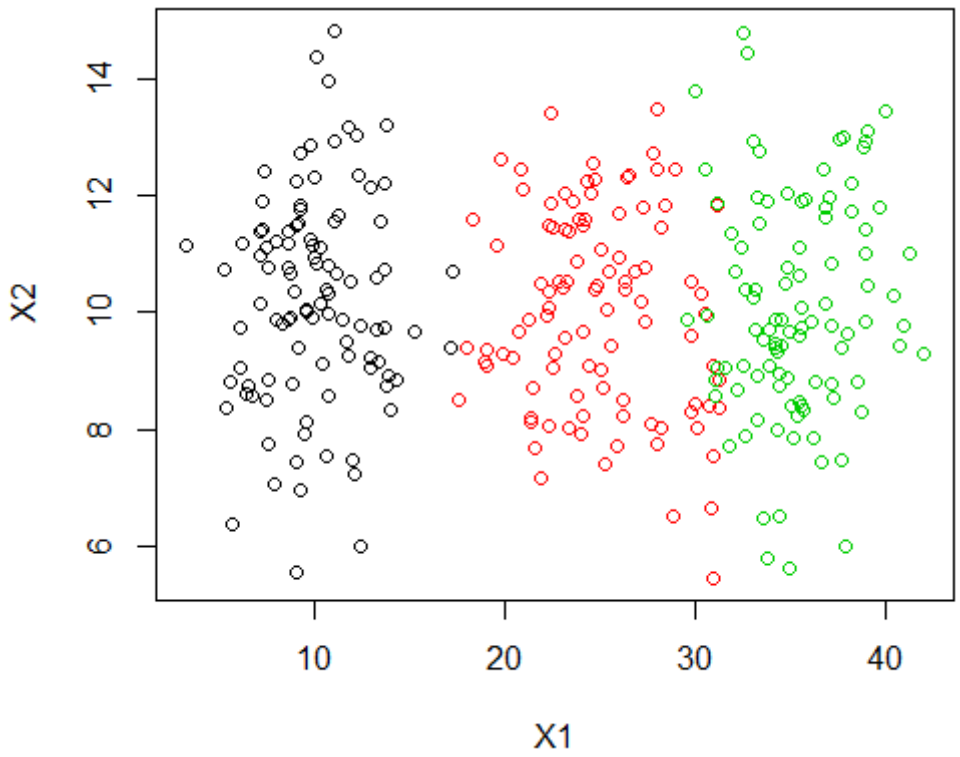
At first, we will read the given dataset using the following commands:

```
# Read data
gsdata = read.csv("gsdata.txt")

# The third column contains the cluster for each data point
target = gsdata[, 3]

# The first two columns contain the data coordinates
gsdata = gsdata[, 1:2]
```

We can plot tha data using the following command:

```
plot(gsdata, col = target)
```

The output is the following figure, where tha data points for each cluster are illustrated with different colors (red, black, green):



12.3.2 - GMMs model construction

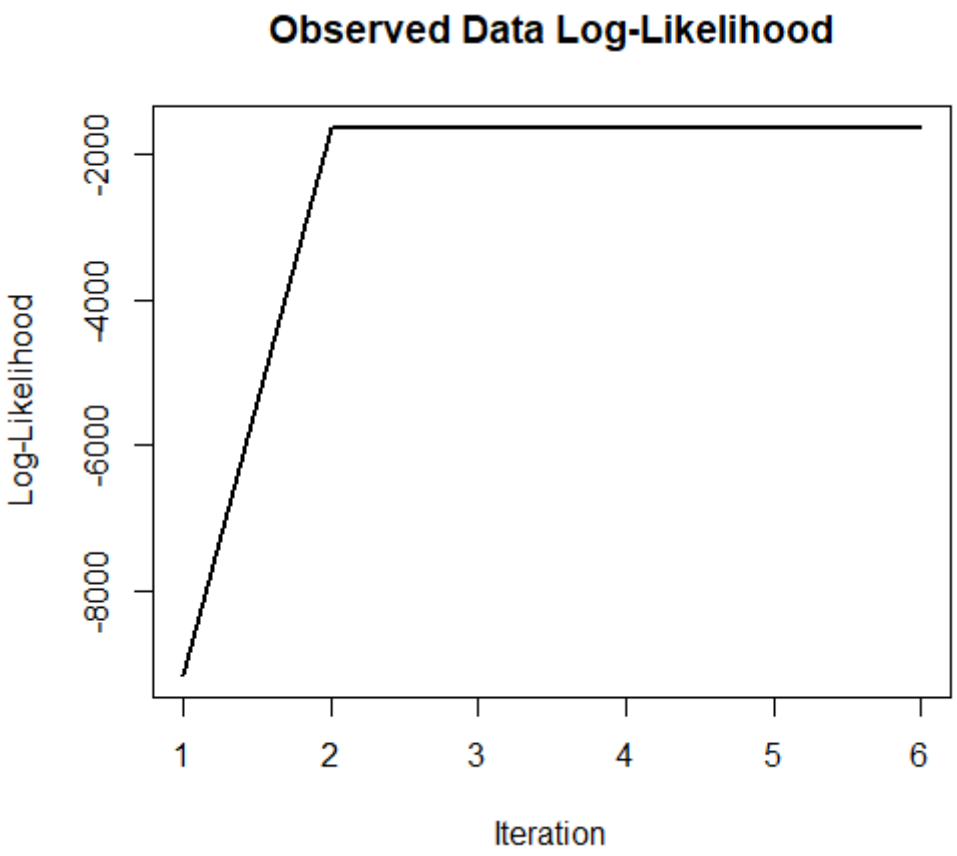
In order to construct the GMMs model, we will use the following commands:

```
# Import library mixtools
library(mixtools)

# Construct model
model = mvnormalmixEM(gsdata, k = 3 , epsilon = 0.1)
```

Once we have constructed the model, we can use the following command to demonstrate how the algorithm converges:

```
plot(model, which = 1)
```

The convergence can be shown in the following figure:

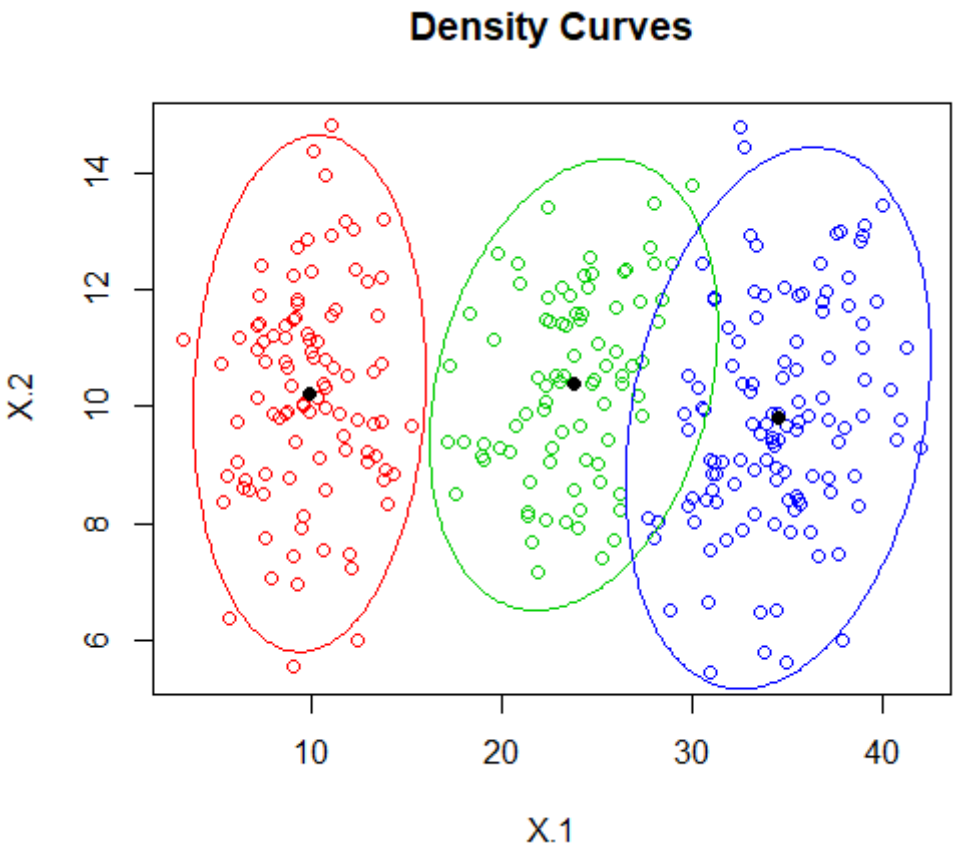


As shown in the graph, it is worth noticing that the EM algorith converges very fast. Since the 2nd iteration, the log-likelihood is almost the same.

Finally, we can plot the given dataset along with the probability density function for each cluster using the following command:

```
plot(model, which = 2)
```

The output is given in the following figure:



Using the ***model$posterior*** command, we can see the probability for each data point to originate from each one of the three clusters (soft-assigments). However, in order to answer this question, we have made *hard-assigments* using the following commands:

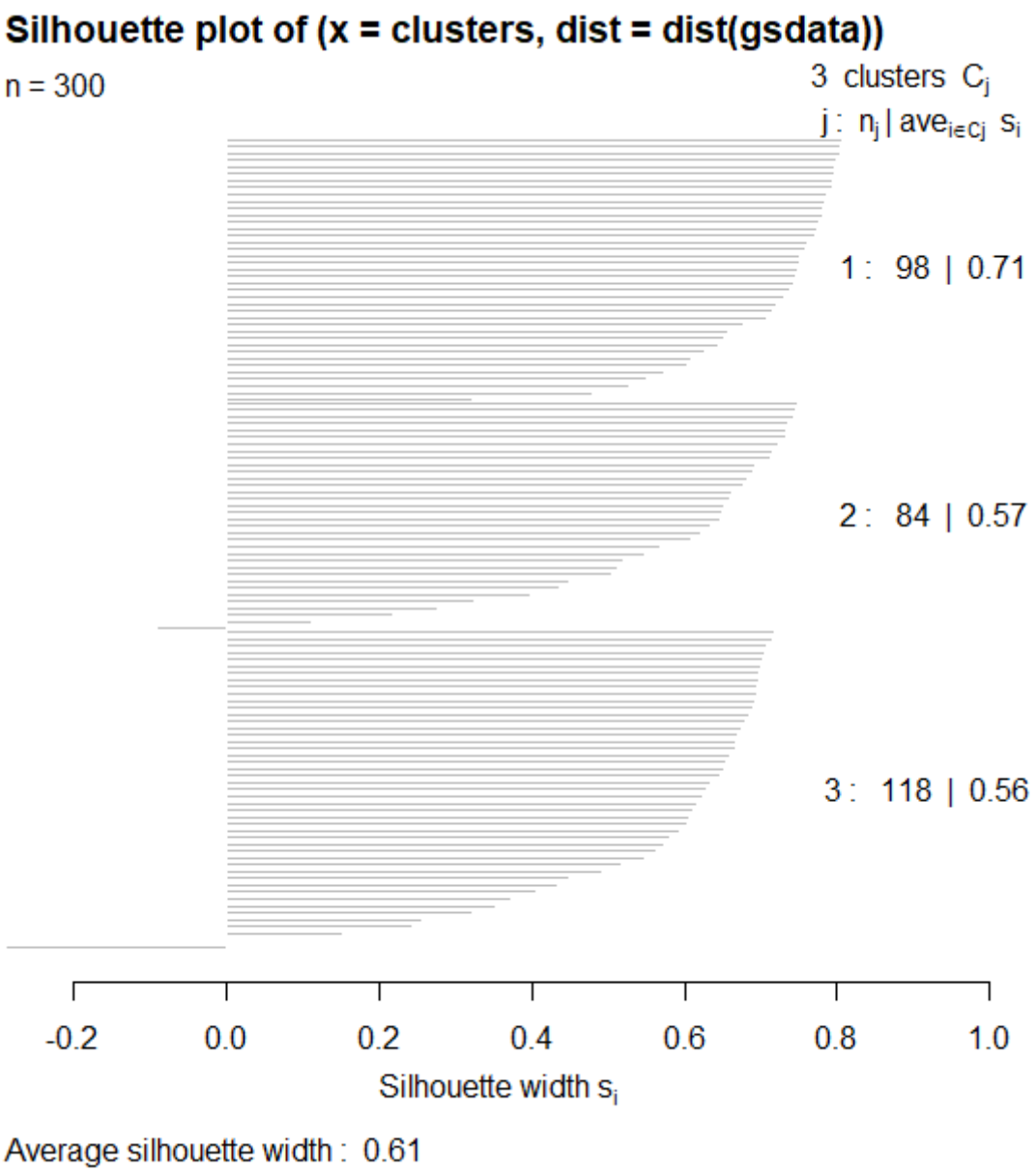
```
# Assign each point to the cluster with the highest probability
clusters = max.col(model$posterior)

# Calculate centers
centers = matrix(unlist(model$mu), byrow = TRUE, ncol = 2)
```

Using the above hard-assigmentsl, we can compute the silhouette coeffiecient and construct the heatmap.

12.3.3 - Silhouette computation

In order to calculate and plot silhouette, we can use the following commands:

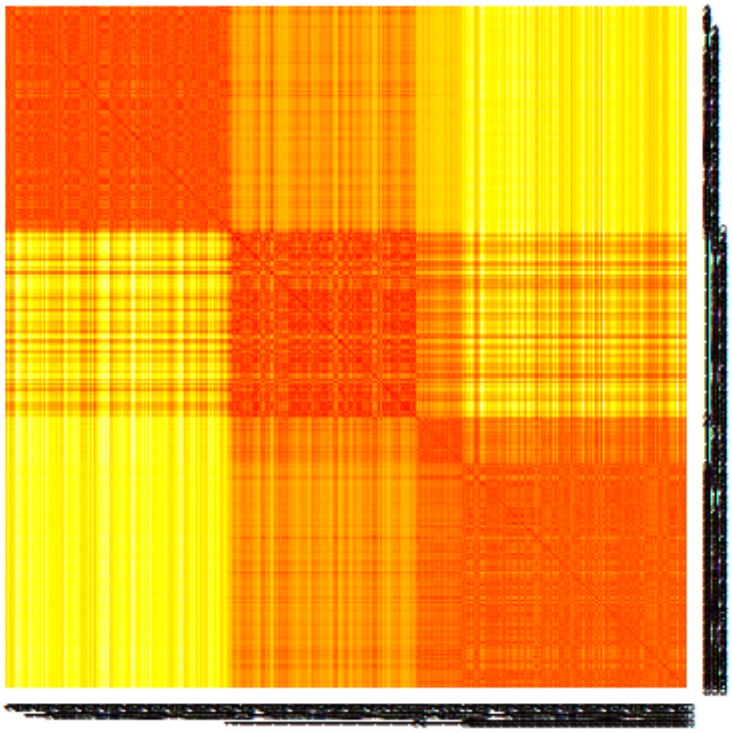
```
# Calculate silhouette
model_silhouette = silhouette(clusters, dist(gsdata))

# Plot silhouette
plot(model_silhouette)
```



12.3.4 Heatmap constuction

In order to construct the heatmap, we can use the following R commands:

```
# Order data based on the cluster value
gsdata_ord = gsdata[order(clusters),]

# Construct heatmap
heatmap(as.matrix(dist(gsdata _ord)), Rowv = NA, Colv = NA,
                    col = heat.colors(256), revC = TRUE)
```



12.4 - GMMs Application with Information Criteria

In this section, we will perfrom clustering using GMMs based on information criteria in order to select the optimal number of clusters. Towards this direction, we will use a dataset containing 300 1D data points. A summary of the given dataset is given in the following table:

	X
Min.	-2.2239
1st Qu.	0.6396
Median	5.1139
Mean	5.0336
3rd Qu.	9.1177
Max.	12.0908

We will use the given dataset in order to answer the following questions:

(a) Plot the data along with the probability density function

(b) Perform clustering using the EM algorithm setting the number of clusters to 2, 3, 4 and 5. For each case, plot the probability density function and calculate the information criteria AIC and BIC.

(c) Plot the AIC and BIC values for the different number of clusters and select the optimal number of clusters

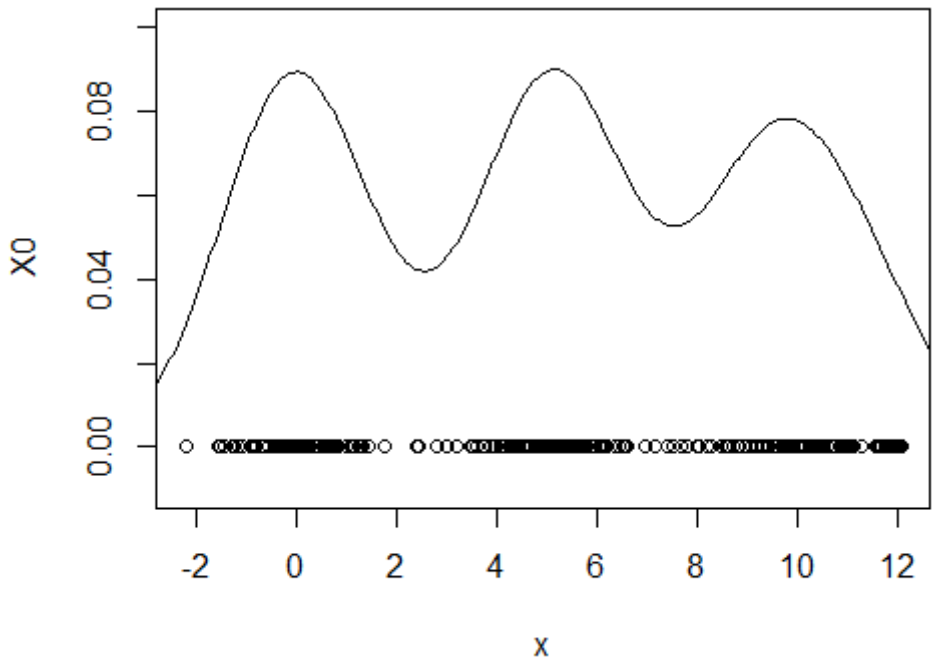
12.4.1 - Data Construction

At first, we will read the given dataset using the following commands:

```
# Import mixtools library
library(mixtools)

# Read data
icdata = read.csv("icdata.txt")

# The first column contains the value of each data point
x = icdata[, 1]

# The second column contains the cluster for each data point
y = icdata[, 2]
```

After that, we can plot the dataset along with the probability density function using the following command:

```
# Plot data
plot(data.frame(x, 0), ylim = c(-0.01, 0.1))

# Plot probability density function
lines(density(x))
```



12.4.2 - Usage of Information Criteria for selecting the optimal number of clusters

In order to answer question (b), we need to compute the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC). For a given model that has t parameters, is applied in

N data samples and has log-likelihood L, the aforementioned criteria are given by the following equations:

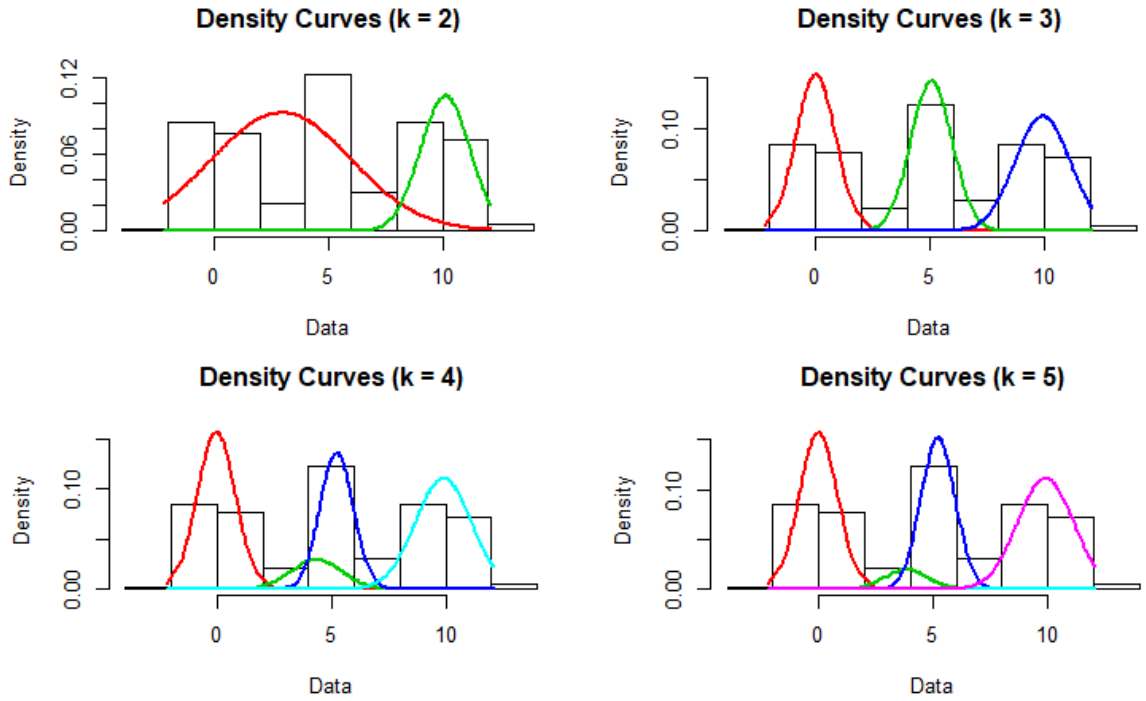
$$AIC = 2 \cdot t - 2 \cdot \ln(L)$$

$$BIC = t \cdot \ln(N) - 2 \cdot \ln(L)$$

We can cluster the data using 2, 3, 4 and 5 distributions using the following code:

```
# Vector for holding AIC values
AIC = c()

# Vector for holding BIC values
BIC = c()

# Set a 2x2 plot grid
par(mfrow = c(2, 2))

# Iterate over the number of clusters
for (k in 2:5) {
  # Construct model
  model <- normalmixEM(x, k = k, epsilon = 0.0001)

  # Plot clustering results
  plot(model, which = 2 , main2 = paste("Density (k = ", k, ")", sep = ""))

  # Calculate number of parameters
  numparams = length(model$mu) + length(model$stdev) + length(model$lambda)

  # Store AIC and BIC values
  AIC[k] = 2 * numparams - 2 * model$loglik
  BIC[k] = log(length(x)) * numparams - 2 * model$loglik
}
```

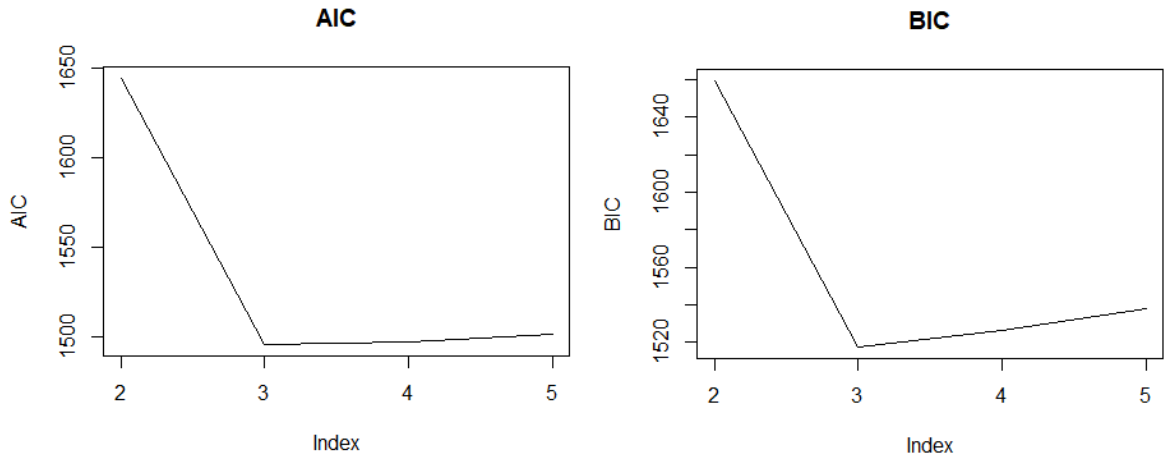
The following figure depicts the clustering results:



Finally, we can plot AIC and BIC values using the following commands:

```
# Plot AIC values
plot(AIC[2:5], type = 'l', xaxt = "n", main = "AIC", ylab = "AIC")
axis(1, at = 1:4, labels = 2:5)

# Plot BIC values
plot(BIC[2:5], type = 'l', xaxt = "n", main = "BIC", ylab = "BIC")
axis(1, at = 1:4, labels = 2:5)
```

The AIC and the BIC values in relation to the number of clusters are shown in the following figure:



Part V - Extended Topics

Chapter 13 - Association Rules

The association rules extraction algorithm is included in the `arules` library. The supermarkt transaction data to be used for executing market basket analysis can be found in the Grocery Shopping datasets page of ACM RecSys. More specifically we will use the Belgium retail market dataset.

First let's load the data, read it with the `read.transactions` function and inspect the first 10 of them:

```
library(arules)
fileURL <- "http://fimi.ua.ac.be/data/retail.dat.gz"
download.file(fileURL, destfile="retail.data.gz", method="curl")
# Read the data in basket format
trans = read.transactions("retail.data.gz", format = "basket", sep=" ");
inspect(trans[1:10])
```

```
##      items
## [1]  {0,
##       1,
##       10,
##       11,
##       12,
##       13,
##       14,
##       15,
##       16,
##       17,
##       18,
##       19,
##       2,
##       20,
##       21,
##       22,
##       23,
##       24,
##       25,
##       26,
##       27,
##       28,
##       29,
##       3,
##       4,
##       5,
##       6,
##       7,
##       8,
##       9}
## [2]  {30,
```

```
##       31,
##       32}
## [3]  {33,
##       34,
##       35}
## [4]  {36,
##       37,
##       38,
##       39,
##       40,
##       41,
##       42,
##       43,
##       44,
##       45,
##       46}
## [5]  {38,
##       39,
##       47,
##       48}
## [6]  {38,
##       39,
##       48,
##       49,
##       50,
##       51,
##       52,
##       53,
##       54,
##       55,
##       56,
##       57,
##       58}
## [7]  {32,
##       41,
##       59,
##       60,
##       61,
##       62}
## [8]  {3,
##       39,
##       48}
## [9]  {63,
##       64,
##       65,
##       66,
##       67,
##       68}
## [10] {32,
##       69}
```

One can see that each transaction containsa list of item IDs. The function `summary` given a dataset read by the `read.tansactions` function will provide a specialized summary for transactional data:

```
summary(trans)
```

```
## transactions as itemMatrix in sparse format with
##  88162 rows (elements/itemsets/transactions) and
##  16470 columns (items) and a density of 0.0006257289
##
## most frequent items:
##      39      48      38      32      41 (Other)
##   50675   42135   15596   15167   14945  770058
##
## element (itemset/transaction) length distribution:
## sizes
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
## 3016 5516 6919 7210 6814 6163 5746 5143 4660 4086 3751 3285 2866 2620 2310
##   16   17   18   19   20   21   22   23   24   25   26   27   28   29   30
## 2115 1874 1645 1469 1290 1205  981  887  819  684  586  582  472  480  355
##   31   32   33   34   35   36   37   38   39   40   41   42   43   44   45
##  310  303  272  234  194  136  153  123  115  112   76   66   71   60   50
##   46   47   48   49   50   51   52   53   54   55   56   57   58   59   60
##   44   37   37   33   22   24   21   21   10   11   10    9   11    4    9
##   61   62   63   64   65   66   67   68   71   73   74   76
##    7    4    5    2    2    5    3    3    1    1    1    1
##
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    4.00    8.00   10.31   14.00   76.00
##
## includes extended item information - examples:
##   labels
## 1      0
## 2      1
## 3     10
```

After reading succesfully the transactions we go ahead with our analysis:

```
rules <- apriori(trans, parameter = list(support = 0.01, confidence = 0.6))
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##         0.6    0.1    1 none FALSE            TRUE       5    0.01      1
##  maxlen target   ext
##      10  rules FALSE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 881
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[16470 item(s), 88162 transaction(s)] done [0.22s].
## sorting and recoding items ... [70 item(s)] done [0.01s].
```

```
## creating transaction tree ... done [0.06s].
## checking subsets of size 1 2 3 4 done [0.01s].
## writing ... [84 rule(s)] done [0.00s].
## creating S4 object  ... done [0.03s].
```

```
quality(rules) <- round(quality(rules), digits=3)
rules
```

```
## set of 84 rules
```

84 rules were generated. To see them use `inspect`.

```
inspect(rules)
```

```
##      lhs            rhs  support confidence lift  count
## [1]  {37}        => {38} 0.012   0.974      5.505  1046
## [2]  {286}       => {38} 0.013   0.943      5.333  1116
## [3]  {12925}     => {39} 0.011   0.639      1.112   938
## [4]  {1146}      => {39} 0.011   0.689      1.199   983
## [5]  {79}        => {39} 0.013   0.694      1.208  1111
## [6]  {1327}      => {39} 0.013   0.647      1.126  1156
## [7]  {438}       => {39} 0.014   0.676      1.177  1260
## [8]  {60}        => {39} 0.011   0.660      1.149   983
## [9]  {255}       => {48} 0.012   0.717      1.500  1057
## [10] {255}       => {39} 0.012   0.717      1.248  1057
## [11] {533}       => {39} 0.010   0.620      1.079   922
## [12] {270}       => {39} 0.014   0.689      1.198  1194
## [13] {2238}      => {39} 0.015   0.750      1.306  1287
## [14] {110}       => {38} 0.031   0.975      5.513  2725
## [15] {110}       => {39} 0.020   0.630      1.095  1759
## [16] {147}       => {39} 0.013   0.639      1.112  1137
## [17] {271}       => {39} 0.016   0.685      1.191  1434
## [18] {413}       => {48} 0.013   0.604      1.263  1135
## [19] {413}       => {39} 0.013   0.601      1.046  1130
## [20] {36}        => {38} 0.032   0.950      5.372  2790
## [21] {36}        => {39} 0.023   0.694      1.207  2037
## [22] {475}       => {48} 0.016   0.659      1.379  1428
## [23] {475}       => {39} 0.017   0.692      1.204  1500
## [24] {170}       => {38} 0.034   0.978      5.529  3031
## [25] {170}       => {39} 0.023   0.664      1.156  2059
## [26] {101}       => {39} 0.016   0.626      1.089  1400
## [27] {310}       => {48} 0.019   0.652      1.365  1692
## [28] {310}       => {39} 0.021   0.714      1.242  1852
## [29] {237}       => {39} 0.022   0.636      1.107  1929
## [30] {225}       => {39} 0.027   0.722      1.256  2351
## [31] {89}        => {48} 0.032   0.729      1.526  2798
## [32] {89}        => {39} 0.031   0.716      1.246  2749
## [33] {65}        => {39} 0.032   0.623      1.084  2787
## [34] {38}        => {39} 0.117   0.663      1.154 10345
## [35] {41}        => {48} 0.102   0.603      1.263  9018
## [36] {41}        => {39} 0.129   0.764      1.329 11414
## [37] {48}        => {39} 0.331   0.692      1.203 29142
```

```
##  [38] {110,48}    => {38} 0.015   0.986      5.575  1361
##  [39] {110,38}    => {39} 0.020   0.639      1.111  1740
##  [40] {110,39}    => {38} 0.020   0.989      5.592  1740
##  [41] {110,48}    => {39} 0.012   0.751      1.307  1037
##  [42] {36,48}     => {38} 0.015   0.960      5.429  1360
##  [43] {36,38}     => {39} 0.022   0.697      1.213  1945
##  [44] {36,39}     => {38} 0.022   0.955      5.398  1945
##  [45] {36,48}     => {39} 0.013   0.788      1.371  1116
##  [46] {475,48}    => {39} 0.012   0.765      1.330  1092
##  [47] {39,475}    => {48} 0.012   0.728      1.523  1092
##  [48] {170,48}    => {38} 0.017   0.988      5.584  1538
##  [49] {170,38}    => {39} 0.023   0.666      1.159  2019
##  [50] {170,39}    => {38} 0.023   0.981      5.543  2019
##  [51] {170,48}    => {39} 0.014   0.775      1.348  1206
##  [52] {101,48}    => {39} 0.011   0.722      1.255   946
##  [53] {101,39}    => {48} 0.011   0.676      1.414   946
##  [54] {310,48}    => {39} 0.015   0.796      1.385  1347
##  [55] {310,39}    => {48} 0.015   0.727      1.522  1347
##  [56] {237,48}    => {39} 0.014   0.740      1.287  1244
##  [57] {237,39}    => {48} 0.014   0.645      1.349  1244
##  [58] {225,48}    => {39} 0.016   0.806      1.403  1400
##  [59] {48,89}     => {39} 0.024   0.759      1.321  2125
##  [60] {39,89}     => {48} 0.024   0.773      1.617  2125
##  [61] {48,65}     => {39} 0.020   0.711      1.236  1797
##  [62] {39,65}     => {48} 0.020   0.645      1.349  1797
##  [63] {32,38}     => {39} 0.021   0.649      1.130  1840
##  [64] {38,41}     => {48} 0.027   0.609      1.275  2374
##  [65] {38,41}     => {39} 0.035   0.783      1.362  3051
##  [66] {38,48}     => {39} 0.069   0.768      1.336  6102
##  [67] {32,41}     => {48} 0.023   0.645      1.351  2063
##  [68] {32,41}     => {39} 0.027   0.738      1.284  2359
##  [69] {32,48}     => {39} 0.061   0.672      1.170  5402
##  [70] {32,39}     => {48} 0.061   0.639      1.337  5402
##  [71] {41,48}     => {39} 0.084   0.817      1.421  7366
##  [72] {39,41}     => {48} 0.084   0.645      1.350  7366
##  [73] {110,38,48} => {39} 0.012   0.758      1.318  1031
##  [74] {110,39,48} => {38} 0.012   0.994      5.620  1031
##  [75] {36,38,48}  => {39} 0.012   0.794      1.382  1080
##  [76] {36,39,48}  => {38} 0.012   0.968      5.471  1080
##  [77] {170,38,48} => {39} 0.014   0.776      1.349  1193
##  [78] {170,39,48} => {38} 0.014   0.989      5.592  1193
##  [79] {32,38,48}  => {39} 0.014   0.751      1.306  1236
##  [80] {32,38,39}  => {48} 0.014   0.672      1.406  1236
##  [81] {38,41,48}  => {39} 0.023   0.839      1.459  1991
##  [82] {38,39,41}  => {48} 0.023   0.653      1.365  1991
##  [83] {32,41,48}  => {39} 0.019   0.798      1.388  1646
##  [84] {32,39,41}  => {48} 0.019   0.698      1.460  1646
```

For example the first rule `{37} => {38}` says that if someone buys item with code 37, he or she will probably buy item with code 38 with support 1.2% and confidence 97.4% and a lift of 5.5.

For a more detailed analysis of the `arules` library check the Introduction to arules[14] vignette.

[14] https://mran.revolutionanalytics.com/web/packages/arules/vignettes/arules.pdf