P.E.S. Education Trust (R), Mandya

P.E.S. COLLEGE OF ENGINEERING

(An Autonomous Institution Affiliated to VTU, Belagavi, Grant -in- Aid Institution (Government of Karnataka), World Bank Funded College (TEQIP), Accredited by NBA & NAAC and Approved by AICTE, New Delhi.)

Mandya - 571401, Karnataka

Data Science Laboratory [P18ISL76] Manual

Department of Information Science and Engineering

P.E.S. College of Engineering

Mandya - 571401, Karnataka

Contents

Course Title: Data Science Laboratory			
Course Code: P18ISL76	**Semester: VII**	**L-T-P-H : 0:0:3:3**	**Credits: 1.5**
Contact Period: Lecture: 36Hrs, Exam: 3 Hrs		**Weightage: CIE:50%, SEE: 50%**	

Course Learning Objectives (CLOs)

This course aims to

Explore Data science process on structured data using *R*.

Course Content

1. Program to perform Data exploration and Pre-processing on a given dataset.
2. Program to implement Linear regression for a given dataset.
3. Program to implement Multiple Linear regression for a given dataset.
4. Program to implement *K*-NN algorithm on a given dataset.
5. Build model to perform clustering using *K*-means and also determine the optimal value of *K* using Elbow method.
6. Program to implement Naive Bayes classifier on a given dataset.
7. Build models using Decision trees.
8. Build your own recommendation system.

Note: The above programs / models have to be implemented using *R Studio*.

Course Outcomes

After learning all programs of the course, the student is able to

1. Develop codes using *R* programming language.
2. Implement Data science process on structured data using *R*.

Course Articulation Matrix (CAM)															
Course Outcomes	**Program Outcomes (PO's)**												**PSO's**		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
CO1	2	1	1		2							1			1
CO2	2	1	1	1	2							1			1

Installing R and R Packages

To Install R and R Packages

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for WINDOWS" link at the top of the page.
5. Click on the install R for the first time & then click Download R 4.1.1 for Windows
6. Save the .exe file, double-click it to open, and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

To Install RStudio

1. Go to www.rstudio.com and click on the "Download Free Desktop IDE" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Mac version, save the .exe file on your computer, double-click it to open, and follow the installation instructions.

To Install R Packages

The capabilities of Rare extended through user-created packages, which allow specialized statistical techniques, graphical devices, import/export capabilities, reporting tools (knitr, Sweave), etc. These packages are developed primarily in R, and sometimes in Java, C, C++, and Fortran. The R packaging system is also used by researchers to create compendia to organise research data, code and report files in a systematic way for sharing and public archiving.

A core set of packages is included with the installation of R, with more than 12,500 additional packages (as of May 2018[update]) available at the Comprehensive R Archive Network (CRAN).

Packages are collections of R functions, data, and compiled code in a well- defined format. The directory where packages are stored is called the library. R comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used.

.libPaths() # get library location

library() # see all packages installed

search() # see packages currently loaded

Adding R Packages

You can expand the types of analyses you do be adding other packages. A complete list of contributed packages is available from CRAN. Follow these steps:

1. Download and install a package (you only need to do this once).
2. To use the package, invoke the **library(**package**)** command to load it into the current session. (You need to do this once in each session, unless you customize your environment to automatically load it each time.)

Installing and Loading Packages

It turns out the ability to estimate ordered logistic or probit regression is included in the MASS package. To install this package, you run the following command:

> install.packages ("MASS")

You will be asked to pick a CRAN mirror from which to download (generally the closer the faster) and R will install the package to your library. R will still be clueless. To actually tell R to use the new package you have to tell R to load the package's library each time you start an R session, just like so:

> library ("MASS")

R now knows all the functions that are canned in the MASS package. To see what functions are implemented in the MASS package, type:

> library (help = "MASS")

Maintaining your Library

Packages are frequently updated. Depending on the developer this could happen very often. To keep your packages updated enter this every once in a while:

> update.packages( )

The Workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions). At the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started. Commands are entered interactively at the R user prompt. **Up** and **down arrow keys** scroll through your command history.

You will probably want to keep different projects in different physical directories. Here are some standard commands for managing your workspace.

```
getwd( ) # print the current working directory
ls ( ) # list the objects in the current workspace.
setwd (mydirectory) # change to my directory
setwd ("c:/docs/mydir") # note / instead of \ in windows
# view and set options for the session
help(options)# learn about available options
options( ) # view current option settings
```

RStudio

RStudio is an open-source integrated development environment that facilitates statistical modelling as well as graphical capabilities for R programming.

The four RStudio Windows

When you open RStudio, you'll see the following four windows (also called panes) shown in Figure 1. However, your windows might be in a different order that those in Figure 1. If you'd like, you can change the order of the windows under RStudio preferences. You can also change their shape by either clicking the minimize or maximize buttons on the top right of each panel, or by clicking and dragging the middle of the borders of the windows.

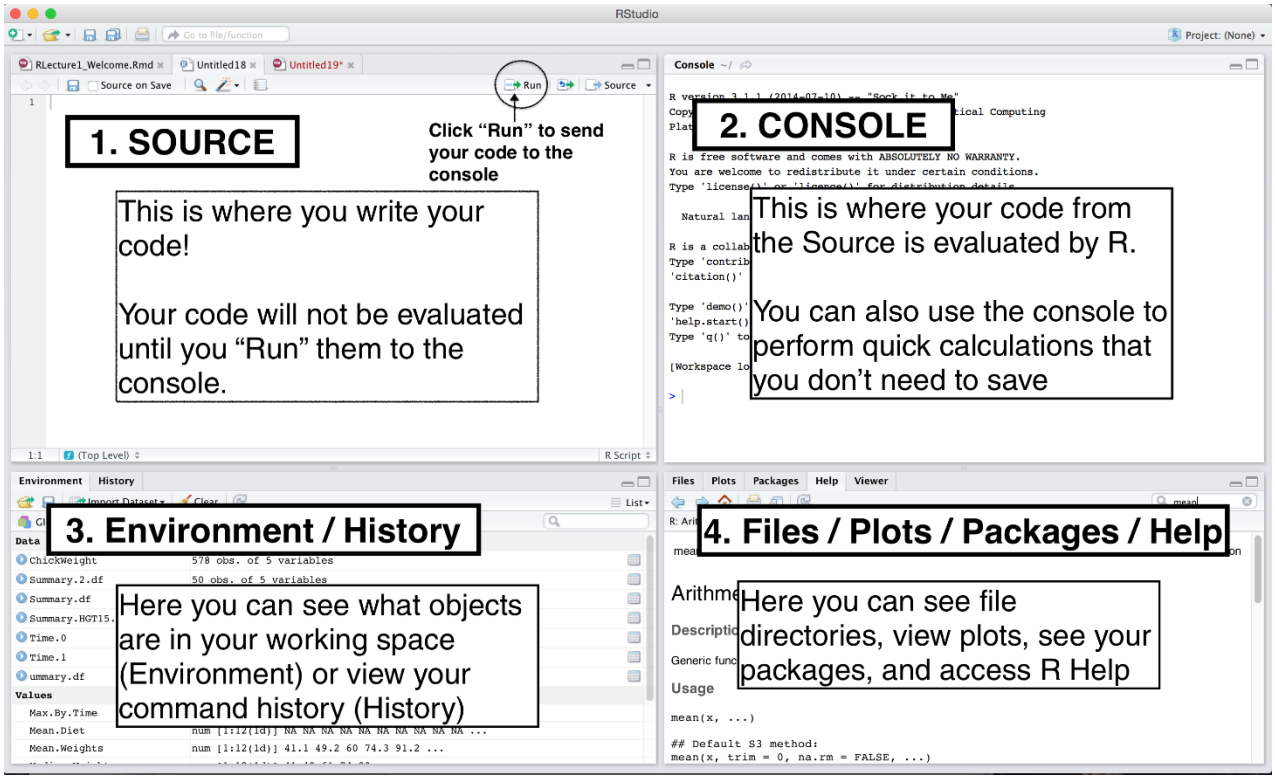


Figure 1: The four panes of RStudio.

Basics of R Programming

Entering Input

At the R prompt we type expressions. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <- ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
> x <- 5 ## nothing printed
> x ## auto-printing occurs
[1] 5
> print(x) ## explicit printing
[1] 5
```

The [1] shown in the output indicates that x is a vector and 5 is its first element. Typically, with interactive work, we do not explicitly print objects with the print function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings. When an R vector is printed you will notice that an index for the vector is printed in square brackets [ ] on the side. For example, see this integer sequence of length 20.

```
> x <- 10:30
> x
[1] 10 11 12 13 14 15 16 17 18 19 20 21
[13] 22 23 24 25 26 27 28 29 30
```

The numbers in the square brackets are not part of the vector itself, they are merely part of the *printed output*. With R, it's important that one understand that there is a difference between the

actual R object and the manner in which that R object is printed to the console. Often, the printed output may have additional bells and whistles to make the output more friendly to the users. However, these bells and whistles are not inherently part of the object. Note that the**:** operator is used to create integer sequences.

R Objects

R has five basic or "atomic" classes of objects:

• character

• numeric (real numbers)

• integer

• complex

• logical (True/False)

The most basic type of R object is a vector. Empty vectors can be created with the vector( ) function. There is really only one rule about vectors in R, which is that **A vector can only contain objects of the same class**.

But of course, like any good rule, there is an exception, which is a *list*, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that's usually why we use them. There is also a class for "raw" objects, but they are not commonly used directly in data analysis and I won't cover them here.

Numbers

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like "1" or "2" in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like "1.00" or "2.00"). This isn't important most of the time…except when it is. If you explicitly want an integer, you need to specify the L suffix. So, entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object. There is also a special number Inf which represents infinity. This allows us to represent entities like 1 / 0. This way, Inf can be used in ordinary calculations; e.g. 1 / Inf is 0. The value NaN represents an undefined value ("not a number"); e.g. 0 / 0; NaN can also be thought of as a missing value (more on that later).

Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are

• names, dimnames

• dimensions (e.g. matrices, arrays)

• class (e.g. integer, numeric)

• length

• other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the attributes() function. Not all R objects contain attributes, in which case the attributes() function returns NULL.

Creating Vectors

The c() function can be used to create vectors of objects by concatenating things together.

```
> x <- c(0.5, 0.6)          ## numeric
> x <- c(TRUE, FALSE)       ## logical
> x <- c(T, F)              ## logical
> x <- c("a", "b", "c")     ## character
> x <- 9:29                 ## integer
> x <- c(1+0i, 2+4i)        ## complex
```

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values. The T and F values are primarily there for when you're feeling lazy. You can also use the vector() function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

```
> y <- c(1.7, "a")          ## character
> y <- c(TRUE, 2)           ## numeric
> y <- c("a", TRUE)         ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.
In the example above, we see the effect of *implicit coercion*. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the as.* functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

```
> x <- c("a", "b", "c")
> as.numeric(x)
Warning: NAs introduced by coercion
[1] NA NA NA
> as.logical(x)
[1] NA NA NA
> as.complex(x)
Warning: NAs introduced by coercion
[1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9  10
> dim(m)  <-  c(2, 5)
> m
      [,1]  [,2]  [,3]  [,4]  [,5]
[1,]     1     3     5     7     9
[2,]     2     4     6     8    10
```

Matrices can be created by *column-binding* or *row-binding* with the cbind() and rbind() functions.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
     x  y
[1,] 1  10
[2,] 2  11
[3,] 3  12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various "apply" functions discussed later, make for a powerful combination. Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1
```

```
[[2]]
[1] "a"
[[3]]
[1] TRUE
[[4]]
[1] 1+4i
```

We can also create an empty list of a prespecified length with the vector() function

```
> x <- vector("list", length = 5)
> x
[[1]]
NULL
[[2]]
NULL
[[3]]
NULL
[[4]]
NULL
[[5]]
NULL
```

Factors

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*. Factors are important in statistical modelling and are treated specially by modelling functions like lm() and glm(). Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

Factor objects can be created with the factor() function.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
no yes
2   3
> ## See the underlying representation of factor
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no" "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often default to creating factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x ## Levels are put in alphabetical order
[1] yes yes no yes no
Levels: no yes
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+ levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

Missing Values

Missing values are denoted by NA or NaN for q undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

Data Frames

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr has an

optimized set of functions designed to work efficiently with data frames. Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows. Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric). In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame. Data frames are usually created by reading in a dataset using the read.table() or read.csv(). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists. Data frames can be converted to a matrix by calling data.matrix(). While it might seem that the as.matrix() function should be used to coerce a data frame to a matrix, almost always, what you want is the result of data.matrix().

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
foo bar
1 1 TRUE
2 2 TRUE
3 3 FALSE
4 4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Names

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x
      New York       Seattle   Los Angeles
             1             2             3
> names(x)
[1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
$`Los Angeles`
[1] 1
$Boston
[1] 2
```

```
$London
[1] 3
> names(x)
[1] "Los Angeles" "Boston" "London"
```

Matrices can have both column and row names.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Column names and row names can be set separately using the colnames() and rownames() functions.

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
  h f
x 1 3
z 2 4
```

Note that for data frames, there is a separate function for setting the row names, the row.names() function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the names() function. Yes, I know its confusing. Here's a quick summary:

Object	Set column names	Set row names
data frame	names()	row.names()
matrix	colnames()	rownames()

R – Decision making

R provides the following types of decision making statements:

1. if statement:

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax: if(boolean_expression) {

// statement(s) will execute if the boolean expression is true.

}

2. if...else statement

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax: if(boolean_expression) {

// statement(s) will execute if the boolean expression is true.

} else {

// statement(s) will execute if the boolean expression is false.

}

3. switch statement

A switch statement allows a variable to be tested for equality against a list of values.

Syntax: switch(expression, case1, case2, case3....)

R – Loops

1. Repeat loop

The Repeat loop executes the same code again and again until a stop condition is met.

Syntax: repeat {

```
commands
if(condition) {
break
}
}
```

2. while loop

The While loop executes the same code again and again until a stop condition is met.

Syntax: while (test_expression) {

```
 statement
}
```

3. for loop

A For loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax: for (value in vector) {

```
statements
}
```

R – Functions

A function is a set of statements organized together to perform a specific task. In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows –

```
function_name <- function(arg_1, arg_2, ...) {
  Function body
}
```

Built-in Function

Simple examples of in-built functions are **seq(), mean(), max(), sum(x) and paste(...) etc.** They are directly called by user written programs. You can refer most widely used R functions.

Ex:- # Create a sequence of numbers from 32 to 44.

```
print(seq(32,44))
```

User-defined Function

Ex:- # Create a function to print squares of numbers in sequence.

```
new.function <- function(a) {
for(i in 1:a) {
        b <- i^2
        print(b)
        }
}
```

Calling a Function

Ex:- # Create a function to print squares of numbers in sequence.

```
new.function <- function(a) {
for(i in 1:a) {
        b <- i^2
        print(b)
        }
}
# Call the function new.function supplying 6 as an argument.
new.function(6)
```

Calling a Function with Argument Values (by position and by name)

Ex:- # Create a function with arguments.

```
new.function <- function(a,b,c) {
        result <- a * b + c
        print(result)
}
# Call the function by position of arguments.
new.function(5,3,11)
# Call the function by names of the arguments.
new.function(a = 11, b = 5, c = 3)
```

Calling a Function with Default Argument

Ex: - # Create a function with arguments.

```
new.function <- function(a = 3, b = 6) {
        result <- a * b
        print(result)
}
# Call the function without giving any argument.
new.function()
# Call the function with giving new values of the argument.
new.function(9,5)
```

Video Link for R Tutorial: https://youtu.be/eDrhZb2onWY

1. Program to perform Data exploration and Pre-processing on a given dataset.

Dataset: Cars (cars_multi.csv & cars_price.csv**)**

URL for the Dataset: https://shorturl.at/esvB6

Objective: To understand how the attributes / columns / features in the dataset relate to each other, to uncover interesting things, and to communicate those findings.

Assumption: The Cars dataset i.e., CSV files are present in the current working directory

Program:

Program1.R

```
#if the library has not been installed then you can install the library using the
below syntax [Here: corrplot is the library]
#install.packages("corrplot")
# Importing library
library(dplyr)   ## For data manipulation and visualization

library(ggplot2) # For plots
library(corrplot) ##provides a visual exploratory tool on correlation matrix

# Load data
cars_multi <- read.csv("cars_multi.csv")
cars_price <- read.csv("cars_price.csv")

#Exploratory Data Analysis
# Head of the dataset cars_multi
head(cars_multi)

##   ID mpg cylinders displacement horsepower weight acceleration model origin
## 1  1  18         8          307        130   3504         12.0    70      1
## 2  2  15         8          350        165   3693         11.5    70      1
## 3  3  18         8          318        150   3436         11.0    70      1
## 4  4  16         8          304        150   3433         12.0    70      1
## 5  5  17         8          302        140   3449         10.5    70      1
## 6  6  15         8          429        198   4341         10.0    70      1
##                   car_name
## 1 chevrolet chevelle malibu
## 2         buick skylark 320
## 3        plymouth satellite
## 4             amc rebel sst
## 5               ford torino
## 6          ford galaxie 500

head(cars_price)

##   ID    price
## 1  1 25561.59
## 2  2 24221.42
## 3  3 27240.84
## 4  4 33684.97
## 5  5 20000.00
## 6  6 30000.00

dim(cars_multi)

## [1] 398  10
```

```r
dim(cars_price)
```

```
## [1] 398   2
```

```r
# Join two dataset
cars <- left_join(cars_multi, cars_price, by = "ID")

#Display names of columns / attributes / features / variables
colnames(cars)
```

```
##  [1] "ID"           "mpg"          "cylinders"    "displacement" "horsepower"
##  [6] "weight"       "acceleration" "model"        "origin"       "car_name"
## [11] "price"
```

```r
# Checking missing cases
sum(!complete.cases(cars))
```

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

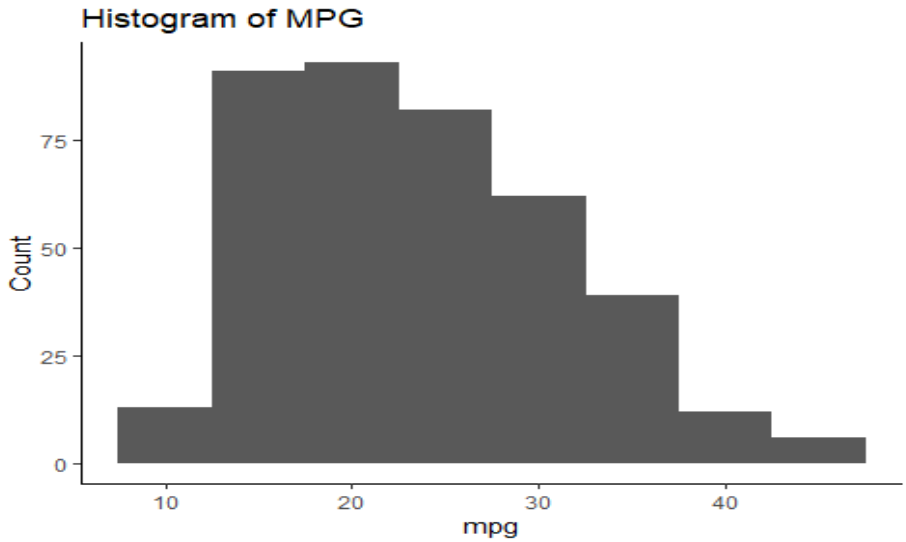
```r
#Overview of the dataset
summary(cars)
```

```
##        ID              mpg          cylinders      displacement
##  Min.   :  1.0   Min.   : 9.00   Min.   :3.000   Min.   : 68.0
##  1st Qu.:100.2   1st Qu.:17.50   1st Qu.:4.000   1st Qu.:104.2
##  Median :199.5   Median :23.00   Median :4.000   Median :148.5
##  Mean   :199.5   Mean   :23.51   Mean   :5.455   Mean   :193.4
##  3rd Qu.:298.8   3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:262.0
##  Max.   :398.0   Max.   :46.60   Max.   :8.000   Max.   :455.0
##   horsepower            weight      acceleration       model
##  Length:398         Min.   :1613   Min.   : 8.00   Min.   :70.00
##  Class :character   1st Qu.:2224   1st Qu.:13.82   1st Qu.:73.00
##  Mode  :character   Median :2804   Median :15.50   Median :76.00
##                     Mean   :2970   Mean   :15.57   Mean   :76.01
##                     3rd Qu.:3608   3rd Qu.:17.18   3rd Qu.:79.00
##                     Max.   :5140   Max.   :24.80   Max.   :82.00
##      origin        car_name             price
##  Min.   :1.000   Length:398         Min.   : 1598
##  1st Qu.:1.000   Class :character   1st Qu.:23110
##  Median :1.000   Mode  :character   Median :30000
##  Mean   :1.573                      Mean   :29684
##  3rd Qu.:2.000                      3rd Qu.:36430
##  Max.   :3.000                      Max.   :53746
```

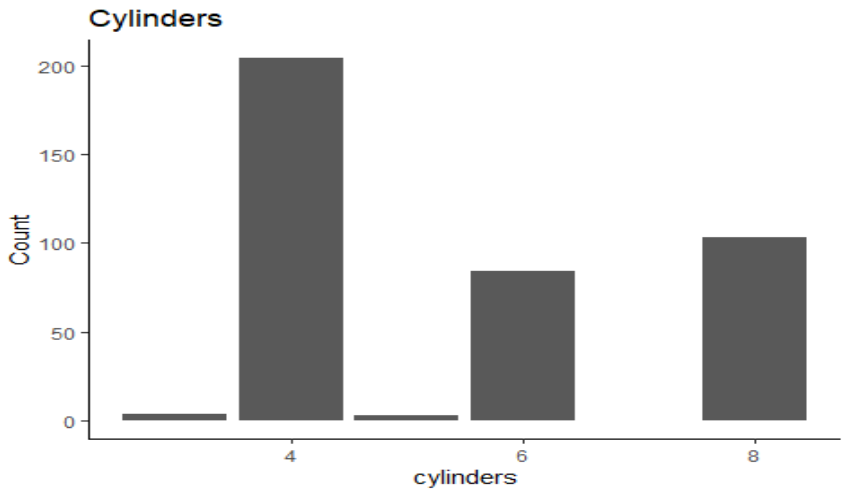
```r
#Structure of the dataset
str(cars)
```

```
## 'data.frame':    398 obs. of  11 variables:
##  $ ID          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ mpg         : num  18 15 18 16 17 15 14 14 14 15 ...
##  $ cylinders   : int  8 8 8 8 8 8 8 8 8 8 ...
##  $ displacement: num  307 350 318 304 302 429 454 440 455 390 ...
##  $ horsepower  : chr  "130" "165" "150" "150" ...
##  $ weight      : int  3504 3693 3436 3433 3449 4341 4354 4312 4425 3850 ...
##  $ acceleration: num  12 11.5 11 12 10.5 10 9 8.5 10 8.5 ...
##  $ model       : int  70 70 70 70 70 70 70 70 70 70 ...
##  $ origin      : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ car_name    : chr  "chevrolet chevelle malibu" "buick skylark 320" "plymouth
satellite" "amc rebel sst" ...
##  $ price       : num  25562 24221 27241 33685 20000 ...
```

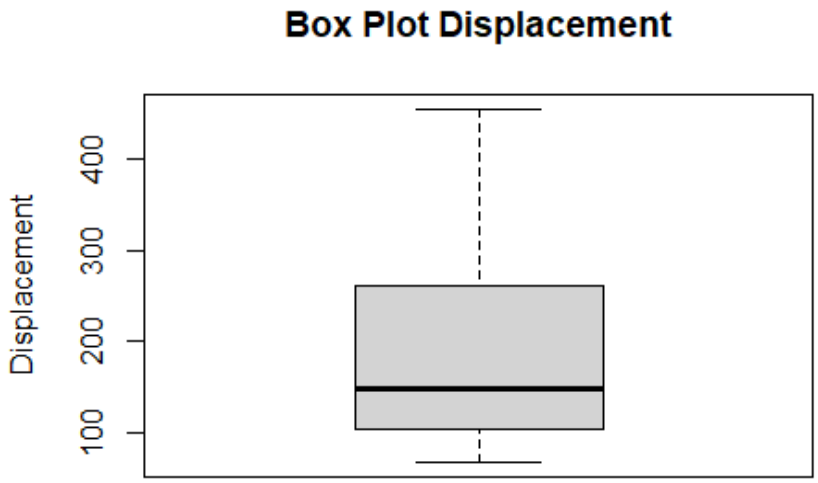
```
#Looking at each variable / attribute / feature

#1. MPG: Mpg means Miles per gallon and we want to know the most common value

ggplot(cars, aes(mpg)) +
  geom_histogram(binwidth = 5) +
  labs(title = "Histogram of MPG", y = "Count") +
  theme_classic()
```



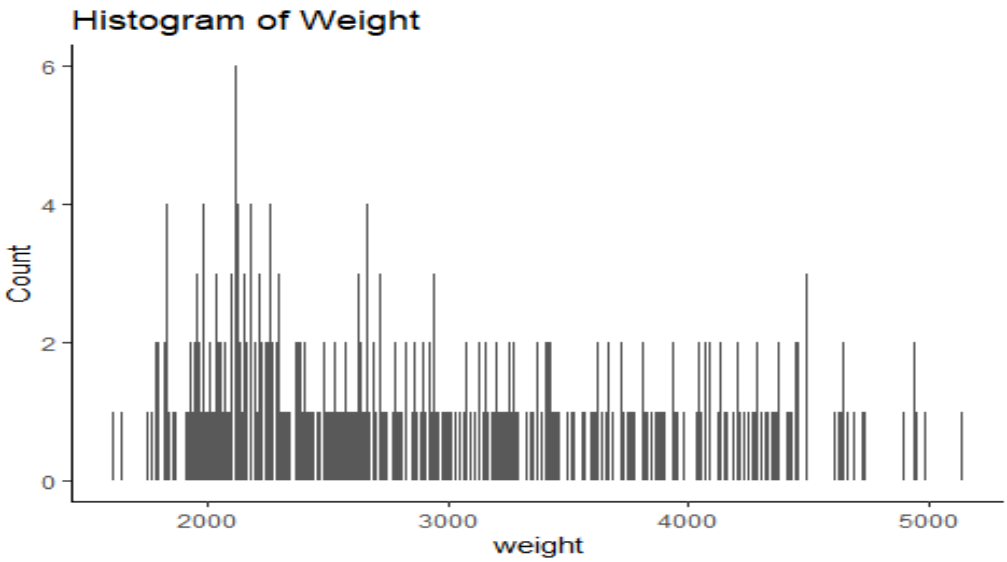

```
#Finding: We can see that the most common mpg is something between 15 and 20 mpg

#2. Cylinders

ggplot(cars, aes(cylinders)) +
  geom_bar() +
  labs(title = "Cylinders", y = "Count") +
  theme_classic()
```



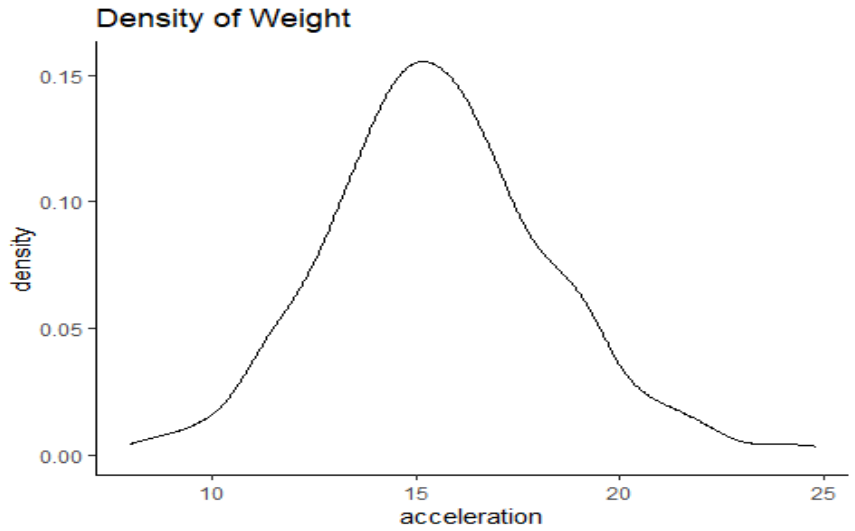

```
#Finding: For cylinders we can see that 4 cylinders is 2 times more often than 8 c
ylinders

#3. Displacement

boxplot(cars$displacement, data=cars$displacement, main="Box Plot Displacement",
        xlab="", ylab="Displacement")
```



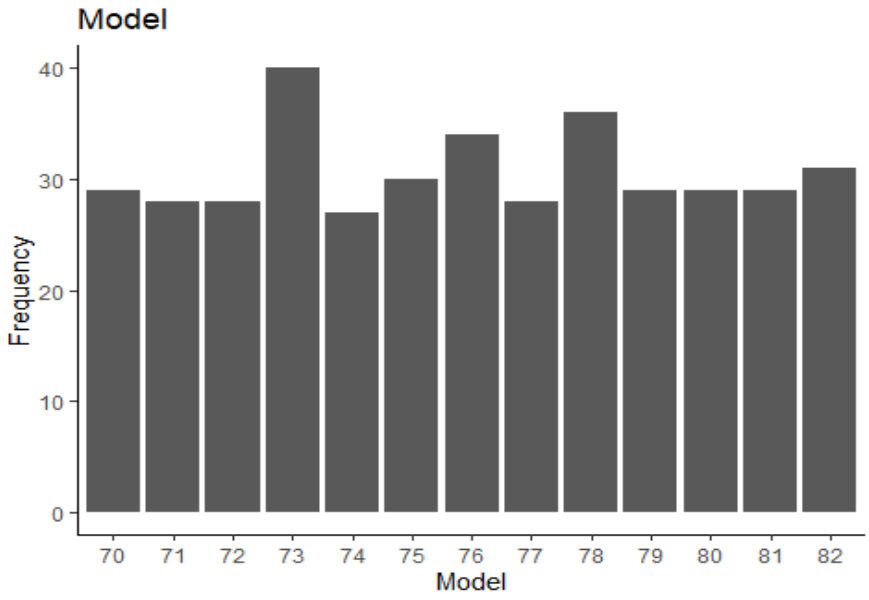

```
#Finding: There is more data/value greater than the average

#4: Horsepower
count(cars[as.character(cars$horsepower) == "?",])

##   n
## 1 6

#Finding: In fact we have 6 missing values at horsepower.

#5: Weight
ggplot(cars, aes(weight)) +
  geom_histogram(binwidth = 5) +
  labs(title = "Histogram of Weight", y = "Count") +
  theme_classic()
```



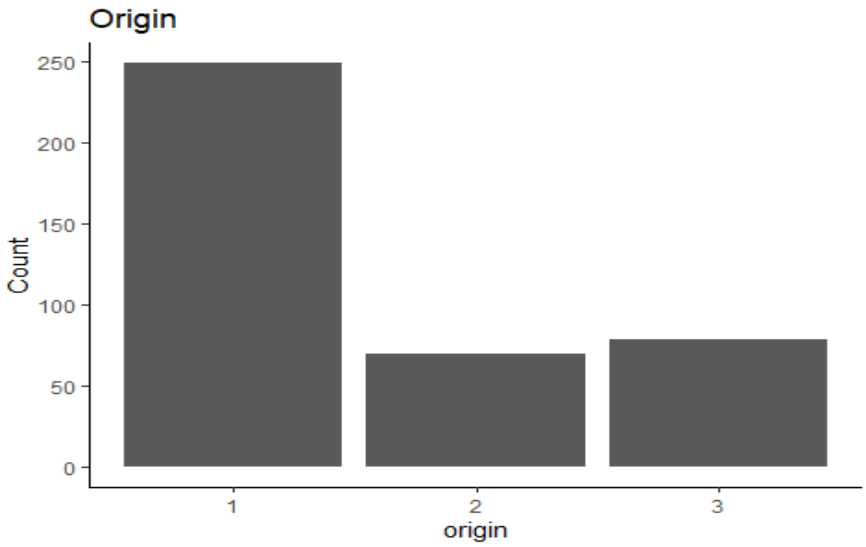

```
#Finding: For Weight we see that the most common weight is something between 2000
and 3000. But most important we saw that we have the only one unique weight for th
e majority of the cars

#6: Acceleration

ggplot(cars, aes(acceleration)) +
  geom_density() +
  labs(title = "Density of Weight") +
  theme_classic()
```



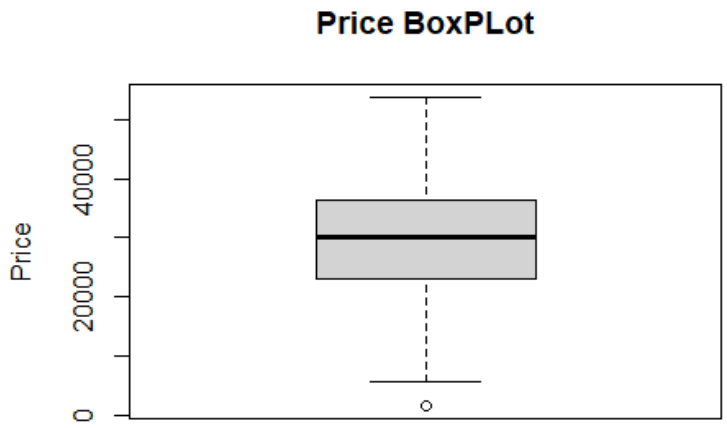

```
#Finding: We see that the density of acceleration is more concentrate at 15

#7. Model
to_Plot <- as.data.frame(table(cars$model))
colnames(to_Plot) <- c("Model", "Frequency")

ggplot(to_Plot, aes(x = Model, y = Frequency)) +
  geom_bar(stat = "identity") +
  labs(title = "Model") +
  theme_classic()
```



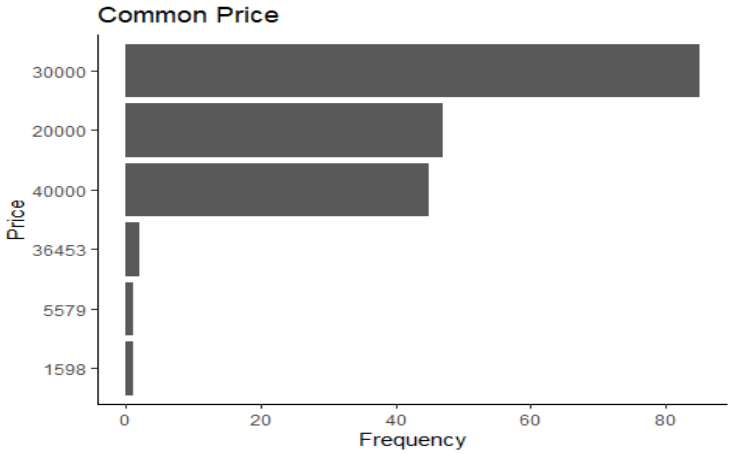

```
#Finding: As we can see we have a good a balance sample for model

#8. Origin
ggplot(cars, aes(origin)) +
  geom_bar() +
  labs(title = "Origin", y = "Count") +
  theme_classic()
```



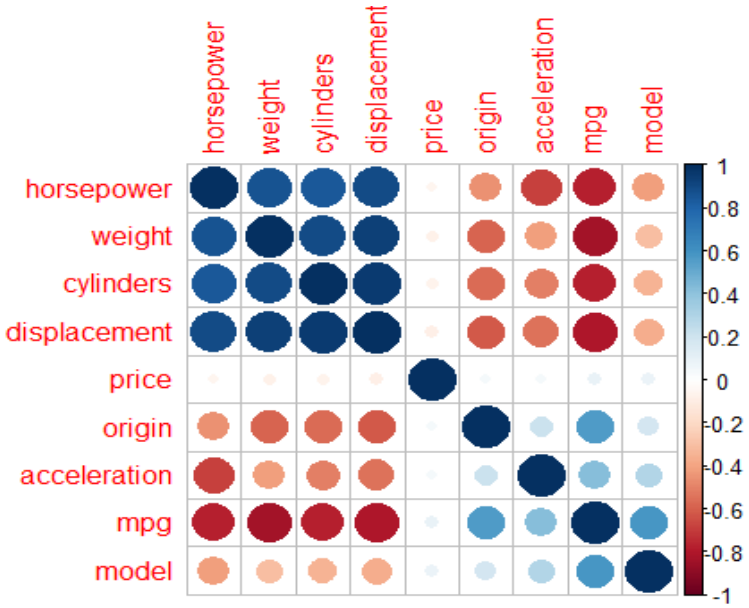

```
#Finding: We have the majority of the cars from origin 1

#9. Price

#For price we decide to do some pre-processing to make more simpler. We are going
to keep only the value before the point. For example: If we have
1598.07337 we are going to keep only 1598.

#We made this decision because we believe that the value after the point is meanin
gless
cars$price <- as.integer(cars$price)

boxplot(cars$price, data=cars$price, main="Price BoxPLot",
        xlab="", ylab="Price")
```




```
#Finding: Looking at the box plot of price we can see that we have only one

Outlier
to_Plot <- as.data.frame(table(cars$price))
colnames(to_Plot) <- c("Price", "Frequency")

ggplot(head(to_Plot[ order(-to_Plot[,2]), ]), aes(x = reorder(Price, Frequency), y
= Frequency)) +
  geom_bar(stat = "identity") +
  labs(title = "Common Price", x = "Price") +
  theme_classic() +
  coord_flip()
```




```
#Finding: With this visualization we can see that we have 3 price that repeat more
than 40 times. We have 219 unique prices. This could be a problem if we have to pr
edict the price of the cars because we have unbalanced data

#Correlation
#At this plot we can see the correlation between all features.
# Transforming from factor to numeric

cars$horsepower <- as.numeric(as.character(cars$horsepower))
## Warning: NAs introduced by coercion
## Warning: NAs introduced by coercion
# Removing not complete row
cars <- cars[complete.cases(cars),]

# Removing the ID
cars <- cars[,-1]

nums <- sapply(cars, is.numeric)
correlations <- cor(cars[,nums])
corrplot(correlations, order = "hclust")
```




```
#Finding: Looking at the MPG we can see that MPG has a negative correlation with
horsepower, weight, cylinders and displacement which make total sense. On the
other hand, MPG has a positive correlation with origin, acceleration and model.
#The correlation between price and mpg is neutral.
```

2. Program to implement Linear regression for a given dataset.

Dataset: Marketing

Source of the Dataset: "datarium" Package

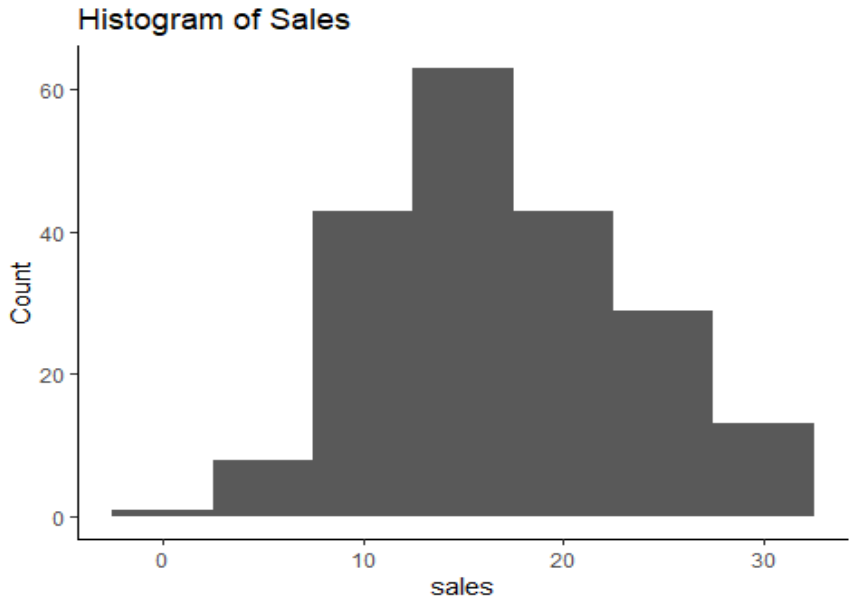
Objective: To predicting the sales units, given the amount of money spent in the advertising media named youtube.

Program:

Program2.R

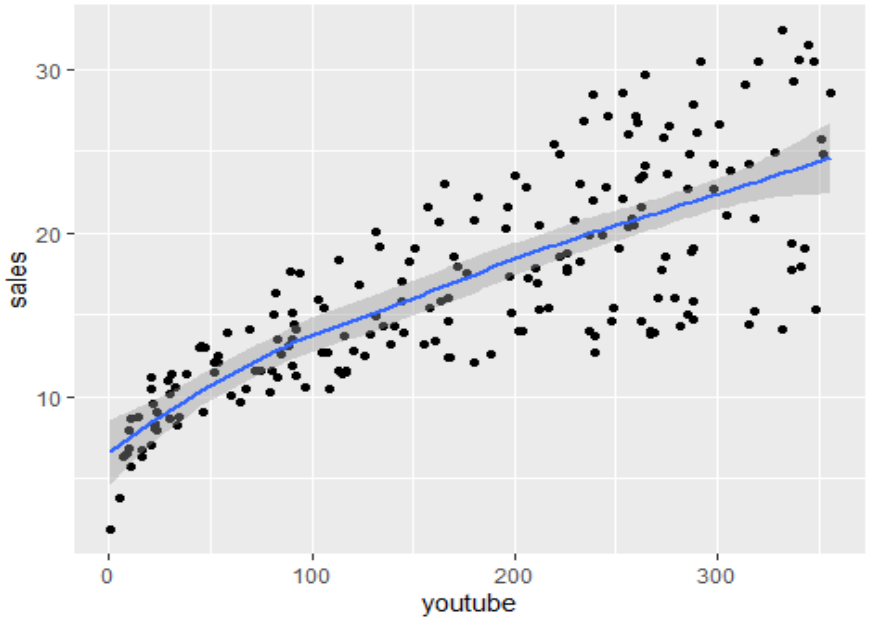
```
#install.packages("caret")
library(dplyr) ## For data manipulation and visualization
library(ggplot2)   ## For plots
library(caret)  ## For createDataPartition function
library(datarium) ## For Marketing dataset

data("marketing", package = "datarium")

#Exploratory Data Analysis
head(marketing)

##   youtube facebook newspaper sales
## 1  276.12    45.36     83.04 26.52
## 2   53.40    47.16     54.12 12.48
## 3   20.64    55.08     83.16 11.16
## 4  181.80    49.56     70.20 22.20
## 5  216.96    12.96     70.08 15.48
## 6   10.44    58.68     90.00  8.64

#dimension of the dataset
dim(marketing)

## [1] 200   4

#Overview of the dataset
summary(marketing)

##     youtube          facebook       newspaper          sales
##  Min.   :  0.84   Min.   : 0.00   Min.   :  0.36   Min.   : 1.92
##  1st Qu.: 89.25   1st Qu.:11.97   1st Qu.: 15.30   1st Qu.:12.45
##  Median :179.70   Median :27.48   Median : 30.90   Median :15.48
##  Mean   :176.45   Mean   :27.92   Mean   : 36.66   Mean   :16.83
##  3rd Qu.:262.59   3rd Qu.:43.83   3rd Qu.: 54.12   3rd Qu.:20.88
##  Max.   :355.68   Max.   :59.52   Max.   :136.80   Max.   :32.40

#Structure of the dataset
str(marketing)

## 'data.frame':    200 obs. of  4 variables:
##  $ youtube  : num  276.1 53.4 20.6 181.8 217 ...
##  $ facebook : num  45.4 47.2 55.1 49.6 13 ...
##  $ newspaper: num  83 54.1 83.2 70.2 70.1 ...
##  $ sales    : num  26.5 12.5 11.2 22.2 15.5 ...

# Checking missing cases
sum(!complete.cases(marketing))

## [1] 0
```

```
##Looking at Sales attribute
ggplot(marketing, aes(sales)) +
  geom_histogram(binwidth = 5) +
  labs(title = "Histogram of Sales", y = "Count") +
  theme_classic()
```




```
# Visualize the data (displaying the sales in thousands of dollars vs. Youtube
advertising budget
ggplot(marketing, aes(x = youtube, y = sales)) +
  geom_point() +
  stat_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
#Correlation coefficient
#Measures the level of association between X and Y
#~0: weak relationship between the variables
cor(marketing$sales, marketing$youtube)
```

```
## [1] 0.7822244
```

```
# Split the data into training and test set
# Here seed is set to ensure reproducibility; Value for seed can any positive
integer value
set.seed(123)
training.samples <- marketing$sales %>% createDataPartition(p = 0.8, list = FALSE)

train.data <- marketing[training.samples,]
test.data <- marketing[-training.samples,]

#Simple Linear Regression; First Parameter represent Formula and Second Parameter
represent data
model <- lm(sales ~ youtube, data = train.data)

#Summarize the model
summary(model)$coef
##               Estimate  Std. Error  t value     Pr(>|t|)
## (Intercept) 8.58914961 0.616044182 13.94242 1.987874e-29
## youtube     0.04671639 0.003003398 15.55451 8.019035e-34

#The output above shows the estimate of the regression beta coefficients (column
Estimate) and their significance
#levels (column Pr(>|t|)). The intercept (b0) is 8.38 and the coefficient of
youtube variable is 0.046.
#The estimated regression equation can be written as:sales = 8.38 + 0.046*youtube.

#Predictions can be easily made using the R function predict().
#Make Predictions using Test data
predictions <- model %>% predict(test.data)

# Model Performance can be measured using any one of the following metrics
#(a) Compute the prediction error, RMSE; RMSE value must be lower for the better
model
RMSE(predictions, test.data$sales)

## [1] 3.687985

#(b) Compute R-Square; R-Square value must be near to 1 for the better model
R2(predictions, test.data$sales)

## [1] 0.6582824

#In the following example, we predict sales units for two youtube advertising
budget: 0 and 1000.
# Making Predictions for new data
newdata <- data.frame(youtube=c(0,1000))

predictions <- model %>% predict(newdata)
predictions
## [1] 8.58915

## [2] 55.30554
```

3. Program to implement Multiple Linear regression for a given dataset.

Dataset: Marketing

Source of the Dataset: "datarium" Package

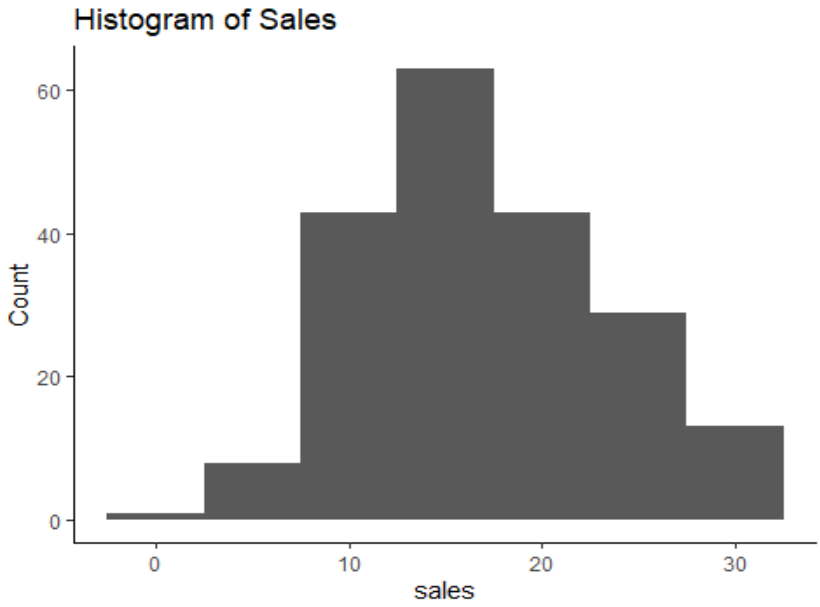
Objective: To predicting the sales units, given the amount of money spent in the advertising medias like youtube, facebook and newspaper.

Program:

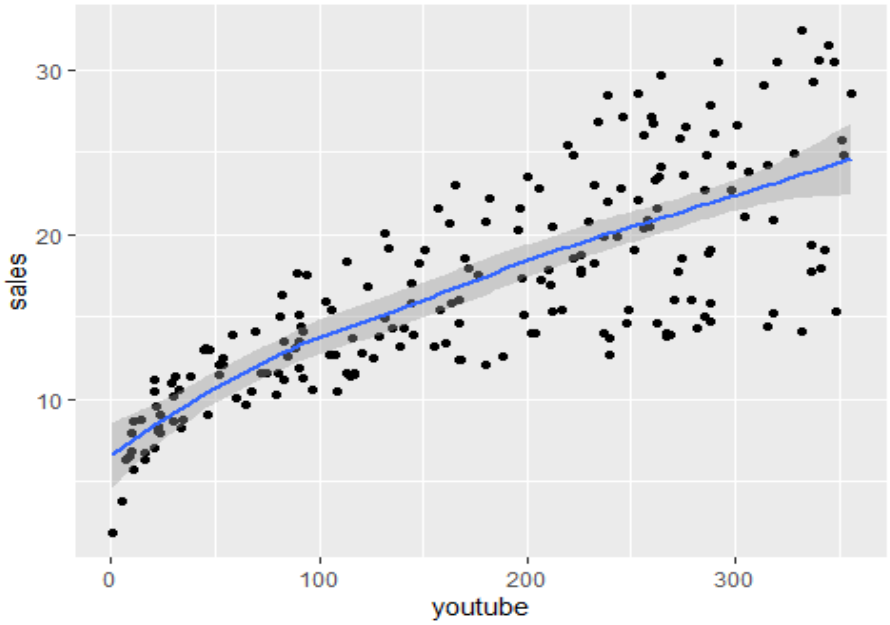
Program3.R

```
#install.packages("caret")
library(tidyverse) ## For data manipulation and visualization
library(ggplot2)   ## For plots
library(caret)      ## For createDataPartition function
library(datarium)  ## For Marketing dataset

data("marketing", package = "datarium")

#Exploratory Data Analysis
head(marketing)

##   youtube facebook newspaper sales
## 1  276.12    45.36     83.04 26.52
## 2   53.40    47.16     54.12 12.48
## 3   20.64    55.08     83.16 11.16
## 4  181.80    49.56     70.20 22.20
## 5  216.96    12.96     70.08 15.48
## 6   10.44    58.68     90.00  8.64

#dimension of the dataset
dim(marketing)

## [1] 200   4

#Overview of the dataset
summary(marketing)

##     youtube          facebook       newspaper          sales
##  Min.   :  0.84   Min.   : 0.00   Min.   :  0.36   Min.   : 1.92
##  1st Qu.: 89.25   1st Qu.:11.97   1st Qu.: 15.30   1st Qu.:12.45
##  Median :179.70   Median :27.48   Median : 30.90   Median :15.48
##  Mean   :176.45   Mean   :27.92   Mean   : 36.66   Mean   :16.83
##  3rd Qu.:262.59   3rd Qu.:43.83   3rd Qu.: 54.12   3rd Qu.:20.88
##  Max.   :355.68   Max.   :59.52   Max.   :136.80   Max.   :32.40

#Structure of the dataset
str(marketing)
## 'data.frame':    200 obs. of  4 variables:
##  $ youtube  : num  276.1 53.4 20.6 181.8 217 ...
##  $ facebook : num  45.4 47.2 55.1 49.6 13 ...
##  $ newspaper: num  83 54.1 83.2 70.2 70.1 ...
##  $ sales    : num  26.5 12.5 11.2 22.2 15.5 ...

# Checking missing cases
sum(!complete.cases(marketing))

## [1] 0
```

```
##Looking at Sales attribute
ggplot(marketing, aes(sales)) +
  geom_histogram(binwidth = 5) +
  labs(title = "Histogram of Sales", y = "Count") +
  theme_classic()
```




```
# Visualize the data (displaying the sales in thousands of dollars vs. Youtube
advertising budget
ggplot(marketing, aes(x = youtube, y = sales)) +
  geom_point() +
  stat_smooth()

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
#Correlation coefficient
#Measures the level of association between X and Y
#~0: weak relationship between the variables
cor(marketing$sales, marketing$youtube)

## [1] 0.7822244
```

```
cor(marketing$sales, marketing$facebook)
```

```
## [1] 0.5762226
```

```
cor(marketing$sales, marketing$newspaper)
```

```
## [1] 0.228299
```

```
# Split the data into training and test set
# Here seed is set to ensure reproducibility; Value for seed can any positive
integer value
set.seed(123)
training.samples <- marketing$sales %>% createDataPartition(p = 0.8, list = FALSE)

train.data <- marketing[training.samples,]
test.data <- marketing[-training.samples,]

# Multiple Linear Regression; First Parameter represent Formula and Second
Parameter represent data
model <- lm(sales ~ youtube + facebook + newspaper, data = train.data)
model <- lm(sales ~ ., data = train.data)
#Summarize the model
summary(model)$coef
##                Estimate  Std. Error    t value     Pr(>|t|)
## (Intercept) 3.594141778 0.420814685  8.5409134 1.054115e-14
## youtube     0.044635905 0.001552128 28.7578721 1.125356e-64
## facebook    0.188823227 0.009528828 19.8159966 1.090250e-44
## newspaper   0.002839757 0.006441963  0.4408217 6.599447e-01

#The output above shows the estimate of the regression beta coefficients (column
Estimate) and their significance
#levels (column Pr(>|t|)). The intercept (b0) is 8.38 and the coefficient of
youtube variable is 0.046.
#The estimated regression equation can be written as:sales = 8.38 + 0.046*youtube.

#Predictions can be easily made using the R function predict().
#Make Predictions using Test data
predictions <- model %>% predict(test.data)

# Model Performance can be measured using any one of the following metrics
#(a) Compute the prediction error, RMSE; RMSE value must be lower for the better
model
RMSE(predictions, test.data$sales)
```

```
## [1] 1.965508
```

```
#(b) Compute R-Square; R-Square value must be near to 1 for the better model
R2(predictions, test.data$sales)
```

```
## [1] 0.9049049
```

```
#In the following example, we predict sales units for youtube advertising budget 2
000, facebook advertising budget 1000, and newspaper advertising budget 1000.
newdata <- data.frame(youtube=2000, facebook=1000, newspaper=1000)

#Make Predictions
predictions <- model %>% predict(newdata)
predictions
```

```
## [1] 284.5289
```

4. Program to implement K-NN algorithm on a given dataset.

Dataset: credit_data

URL for the Dataset: https://shorturl.at/noHX3

Objective: To study a bank credit dataset and use the K-NN algorithm to classify the applicant's loan request into two classes namely "Approved" and "Disapproved".

Assumption: The credit dataset i.e., CSV file are present in the current working directory

Program:

Program4.R

```
#Import the dataset
loan <- read.csv("credit_data.csv")

#Structure
str(loan)

## 'data.frame':    1000 obs. of  21 variables:
##  $ Creditability                   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Account.Balance                 : int  1 1 2 1 1 1 1 1 4 2 ...
##  $ Duration.of.Credit..month.      : int  18 9 12 12 12 10 8 6 18 24 ...
##  $ Payment.Status.of.Previous.Credit: int  4 4 2 4 4 4 4 4 4 2 ...
##  $ Purpose                         : int  2 0 9 0 0 0 0 0 3 3 ...
##  $ Credit.Amount                   : int  1049 2799 841 2122 2171 2241 3398 13
61 1098 3758 ...
##  $ Value.Savings.Stocks            : int  1 1 2 1 1 1 1 1 1 3 ...
##  $ Length.of.current.employment    : int  2 3 4 3 3 2 4 2 1 1 ...
##  $ Instalment.per.cent             : int  4 2 2 3 4 1 1 2 4 1 ...
##  $ Sex...Marital.Status            : int  2 3 2 3 3 3 3 3 2 2 ...
##  $ Guarantors                      : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Duration.in.Current.address     : int  4 2 4 2 4 3 4 4 4 4 ...
##  $ Most.valuable.available.asset   : int  2 1 1 1 2 1 1 1 3 4 ...
##  $ Age..years.                     : int  21 36 23 39 38 48 39 40 65 23 ...
##  $ Concurrent.Credits              : int  3 3 3 3 1 3 3 3 3 3 ...
##  $ Type.of.apartment               : int  1 1 1 1 2 1 2 2 2 1 ...
##  $ No.of.Credits.at.this.Bank      : int  1 2 1 2 2 2 2 1 2 1 ...
##  $ Occupation                      : int  3 3 2 2 2 2 2 2 1 1 ...
##  $ No.of.dependents                : int  1 2 1 2 1 2 1 2 1 1 ...
##  $ Telephone                       : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Foreign.Worker                  : int  1 1 1 2 2 2 2 2 1 1 ...

#Data Cleaning

#From the structure of the dataset, we can see that there are 21 predictor variabl
es that will help us
#to decide whether or not an applicant's loan must be approved.
#Some of these variables are not essential in predicting the loan of an applicant,
for example,
#variables such as Telephone, Concurrent. Credits, Duration.in.Current.address,
Type.of.apartment, etc.
#Such variables must be removed because they will only increase the complexity of
the Machine Learning model.

#Also note that, the 'Creditability' variable is our output variable or the target
variable.
```

```
loan.subset <- loan[c('Creditability','Age..years.','Sex...Marital.Status','Occupa
tion','Account.Balance','Credit.Amount','Length.of.current.employment','Purpose')]

#Again see the structure
#Now we have narrowed down 21 variables to 8 predictor variables that are signific
ant for building the model.
str(loan.subset)
```

```
## 'data.frame':    1000 obs. of  8 variables:
##  $ Creditability              : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Age..years.                : int  21 36 23 39 38 48 39 40 65 23 ...
##  $ Sex...Marital.Status       : int  2 3 2 3 3 3 3 3 2 2 ...
##  $ Occupation                 : int  3 3 2 2 2 2 2 2 1 1 ...
##  $ Account.Balance            : int  1 1 2 1 1 1 1 1 4 2 ...
##  $ Credit.Amount              : int  1049 2799 841 2122 2171 2241 3398 1361 10
98 3758 ...
##  $ Length.of.current.employment: int  2 3 4 3 3 2 4 2 1 1 ...
##  $ Purpose                    : int  2 0 9 0 0 0 0 0 3 3 ...
```

```
#Data Normalization

#You must always normalize the data set so that the output remains unbiased. To
explain this, let's take a look at the first few observations in our data set.
head(loan.subset)
```

```
##   Creditability Age..years. Sex...Marital.Status Occupation Account.Balance
## 1             1          21                    2          3               1
## 2             1          36                    3          3               1
## 3             1          23                    2          2               2
## 4             1          39                    3          2               1
## 5             1          38                    3          2               1
## 6             1          48                    3          2               1
##   Credit.Amount Length.of.current.employment Purpose
## 1          1049                           2       2
## 2          2799                           3       0
## 3           841                           4       9
## 4          2122                           3       0
## 5          2171                           3       0
## 6          2241                           2       0
```

```
#Notice the Credit amount variable, its value scale is in 1000s, whereas the rest
of the variables are in single digits or 2 digits. If the data isn't normalized it
will lead to a biased outcome.

#Normalization function
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x))) }

#In the below code snippet, we're storing the normalized data set in the 'loan.sub
set.n' variable and
#also we're removing the 'Credibility' variable since it's the response variable t
hat needs to be predicted.

loan.subset.n <- as.data.frame(lapply(loan.subset[,2:8], normalize))
#Normalized dataset
head(loan.subset.n)
```

```
##   Age..years. Sex...Marital.Status Occupation Account.Balance Credit.Amount
## 1  0.03571429            0.3333333  0.6666667       0.0000000    0.04396390
## 2  0.30357143            0.6666667  0.6666667       0.0000000    0.14025531
## 3  0.07142857            0.3333333  0.3333333       0.3333333    0.03251898
## 4  0.35714286            0.6666667  0.3333333       0.0000000    0.10300429
## 5  0.33928571            0.6666667  0.3333333       0.0000000    0.10570045
## 6  0.51785714            0.6666667  0.3333333       0.0000000    0.10955211
##   Length.of.current.employment Purpose
## 1                         0.25     0.2
## 2                         0.50     0.0
## 3                         0.75     0.9
## 4                         0.50     0.0
## 5                         0.50     0.0
## 6                         0.25     0.0
```

```r
#Data Spliting
set.seed(123)
dat.d <- sample(1:nrow(loan.subset.n),size=nrow(loan.subset.n)*0.7,replace=FALSE)
#random selection of 70% data.

train.loan <- loan.subset[dat.d,] # 70% training data
test.loan <- loan.subset[-dat.d,] # remaining 30% test data

#Creating seperate dataframe for 'Creditability' feature which is our target.
train.loan_labels <- loan.subset[dat.d,1]
test.loan_labels <-loan.subset[-dat.d,1]


#Install class package
#install.packages('class')
# Load class package
library(class)                # For K-NN


#Next, we're going to calculate the number of observations in the training data
set.
#The reason we're doing this is that we want to initialize the value of 'K' in the
K-NN model.
#One of the ways to find the optimal K value is to calculate the square root of
the total number of
#observations in the data set. This square root will give you the 'K' value.

#Find the number of observation
NROW(train.loan_labels)
```

```
## [1] 700
```

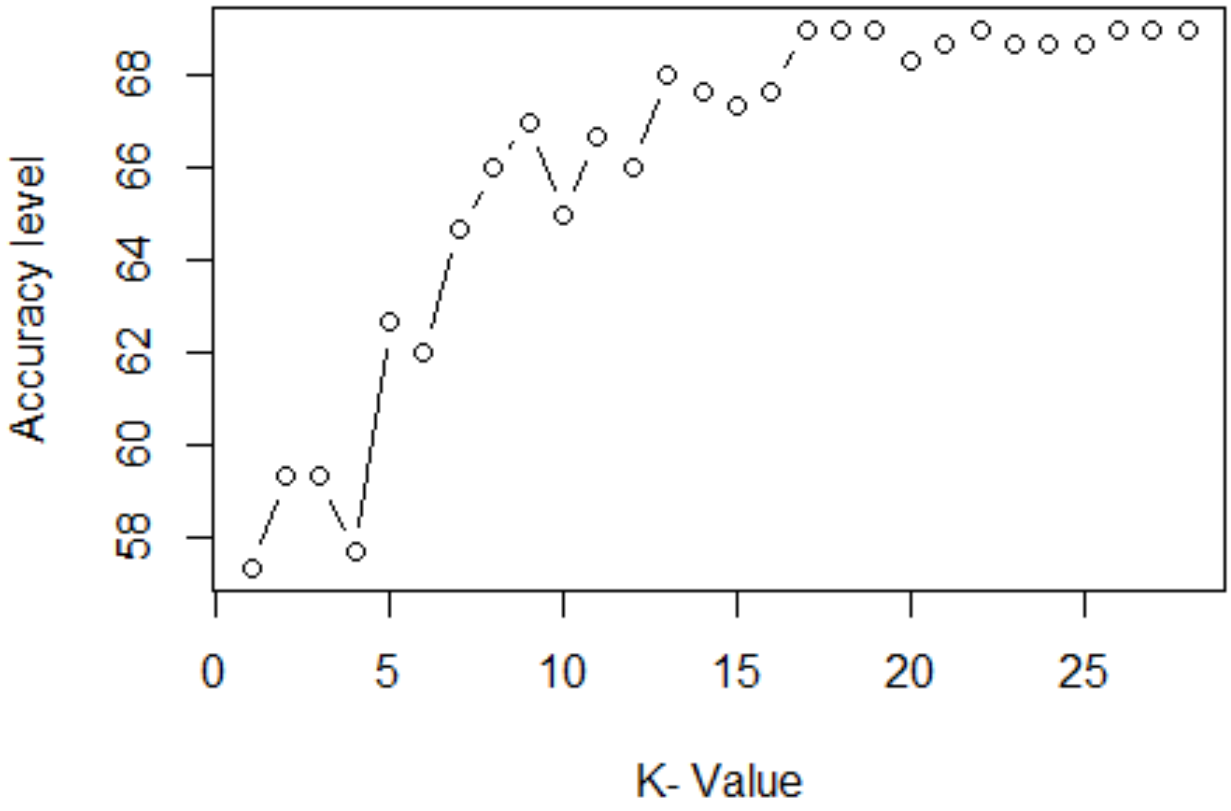
```r
#So, we have 700 observations in our training data set. The square root of 700 is
around 26.45,
#therefore we'll create two models. One with 'K' value as 26 and the other model w
ith a 'K' value as 27.

knn.26 <- knn(train=train.loan, test=test.loan, cl=train.loan_labels, k=26)
knn.27 <- knn(train=train.loan, test=test.loan, cl=train.loan_labels, k=27)
```

```
#Model Evaluation

#Calculate the proportion of correct classification for k = 26, 27
ACC.26 <- 100 * sum(test.loan_labels == knn.26)/NROW(test.loan_labels)
ACC.27 <- 100 * sum(test.loan_labels == knn.27)/NROW(test.loan_labels)

ACC.26

## [1] 68.66667

ACC.27

## [1] 69

#Finding: The accuracy for K = 26 is 68.66 and for K = 27 it is 69. We can also
check the predicted outcome against the actual value in tabular form:

# Check prediction against actual value in tabular form for k=26
table(knn.26 ,test.loan_labels)

##       test.loan_labels
## knn.26   0   1
##      0   8   7
##      1  87 198

knn.26

##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
##  [75] 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
## [112] 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [149] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [223] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1
## [260] 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1 1 1
## [297] 1 1 1 0
## Levels: 0 1

# Check prediction against actual value in tabular form for k = 27
table(knn.27 ,test.loan_labels)

##       test.loan_labels
## knn.27   0   1
##      0   8   6
##      1  87 199

knn.27

##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
##  [75] 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [112] 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [149] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [223] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1
## [260] 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1 1 1
## [297] 1 1 1 0
## Levels: 0 1
```

```
#You can also use the confusion matrix to calculate the accuracy.
#To do this we must first install the Caret package:
#install.packages('caret')


library(caret)

## Loading required package: ggplot2

## Loading required package: lattice

confusionMatrix(table(knn.26 ,test.loan_labels))

## Confusion Matrix and Statistics
##
##       test.loan_labels
## knn.26   0   1
##      0   8   7
##      1  87 198
##
##                Accuracy : 0.6867
##                  95% CI : (0.6309, 0.7387)
##     No Information Rate : 0.6833
##     P-Value [Acc > NIR] : 0.4783
##
##                   Kappa : 0.0647
##
##  Mcnemar's Test P-Value : 3.693e-16
##
##             Sensitivity : 0.08421
##             Specificity : 0.96585
##          Pos Pred Value : 0.53333
##          Neg Pred Value : 0.69474
##              Prevalence : 0.31667
##          Detection Rate : 0.02667
##    Detection Prevalence : 0.05000
##       Balanced Accuracy : 0.52503
##
##        'Positive' Class : 0
##

#So, from the output, we can see that our model predicts the outcome with an
accuracy of 68.67% which
#is good since we worked with a small data set. A point to remember is that the
more data (optimal data) you feed the machine, the more efficient the model
will be.

# Create a loop that calculates the accuracy of the K-NN model for 'K' values rang
ing from 1 to 28.
#This way you can check which 'K' value will result in the most accurate model:


i=1
k.optm=1
for (i in 1:28){
  knn.mod <- knn(train=train.loan, test=test.loan, cl=train.loan_labels, k=i)
  k.optm[i] <- 100 * sum(test.loan_labels == knn.mod)/NROW(test.loan_labels)
  k=i
  cat(k,'=',k.optm[i],'') }
```

```
## 1 = 57.33333 2 = 59.33333 3 = 59.33333 4 = 57.66667 5 = 62.66667 6 = 62 7 = 64.
66667 8 = 66 9 = 67 10 = 65 11 = 66.66667 12 = 66 13 = 68 14 = 67.66667 15 = 67.33
333 16 = 67.66667 17 = 69 18 = 69 19 = 69 20 = 68.33333 21 = 68.66667 22 = 69 23 =
68.66667 24 = 68.66667 25 = 68.66667 26 = 69 27 = 69 28 = 69
```

```
#Accuracy plot; Graphical representation
plot(k.optm, type="b", xlab="K- Value",ylab="Accuracy level")
```




```
#Finding: From the output you can see that for K = 27, we achieve the maximum
accuracy, i.e. 69%.
```

5. Build model to perform clustering using K-means and also determine the optimal value of K using Elbow method.

Dataset: Iris

Source of the Dataset: "dplyr" package

Objective: To draw data patterns and cluster data observations into different groups based on their similarities.

Program:

Program5.R

```
library(dplyr) # for Iris dataset
library(ggplot2)

#Loading iris dataset
data(iris)

#Display first few rows in a iris dataset
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

#summary of iris dataset
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
##

#Now, let's remove the last column from the data and keep only the numeric columns
for the analysis
#Removing "Species column"
iris_2<-iris[-5]
head(iris_2)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
## 4          4.6         3.1          1.5         0.2
## 5          5.0         3.6          1.4         0.2
## 6          5.4         3.9          1.7         0.4
```

```r
#Once, we have dataset ready let's standardize our data
#Standardize data
iris_3<-as.data.frame(scale(iris_2))

head(iris_3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1   -0.8976739  1.01560199    -1.335752   -1.311052
## 2   -1.1392005 -0.13153881    -1.335752   -1.311052
## 3   -1.3807271  0.32731751    -1.392399   -1.311052
## 4   -1.5014904  0.09788935    -1.279104   -1.311052
## 5   -1.0184372  1.24503015    -1.335752   -1.311052
## 6   -0.5353840  1.93331463    -1.165809   -1.048667
```

```r
#Checking Mean and SD of data before and after standardization
sapply(iris_2,mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

```r
sapply(iris_2,sd)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##    0.8280661    0.4358663    1.7652982    0.7622377
```

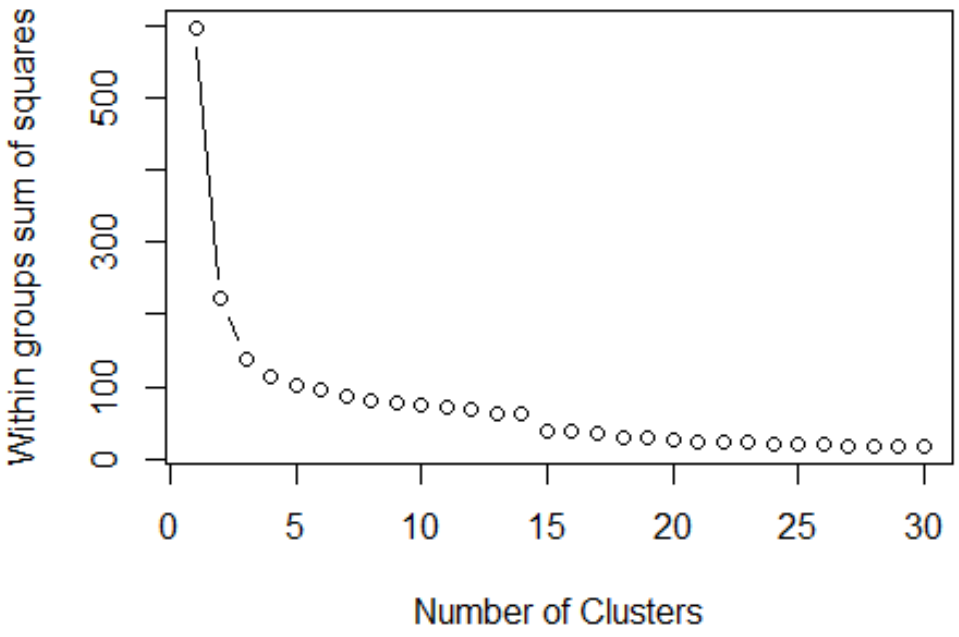
```r
sapply(iris_3,mean)
```

```
##  Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
## -4.484318e-16  2.034094e-16 -2.895326e-17 -3.663049e-17
```

```r
sapply(iris_3,sd)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##            1            1            1            1
```

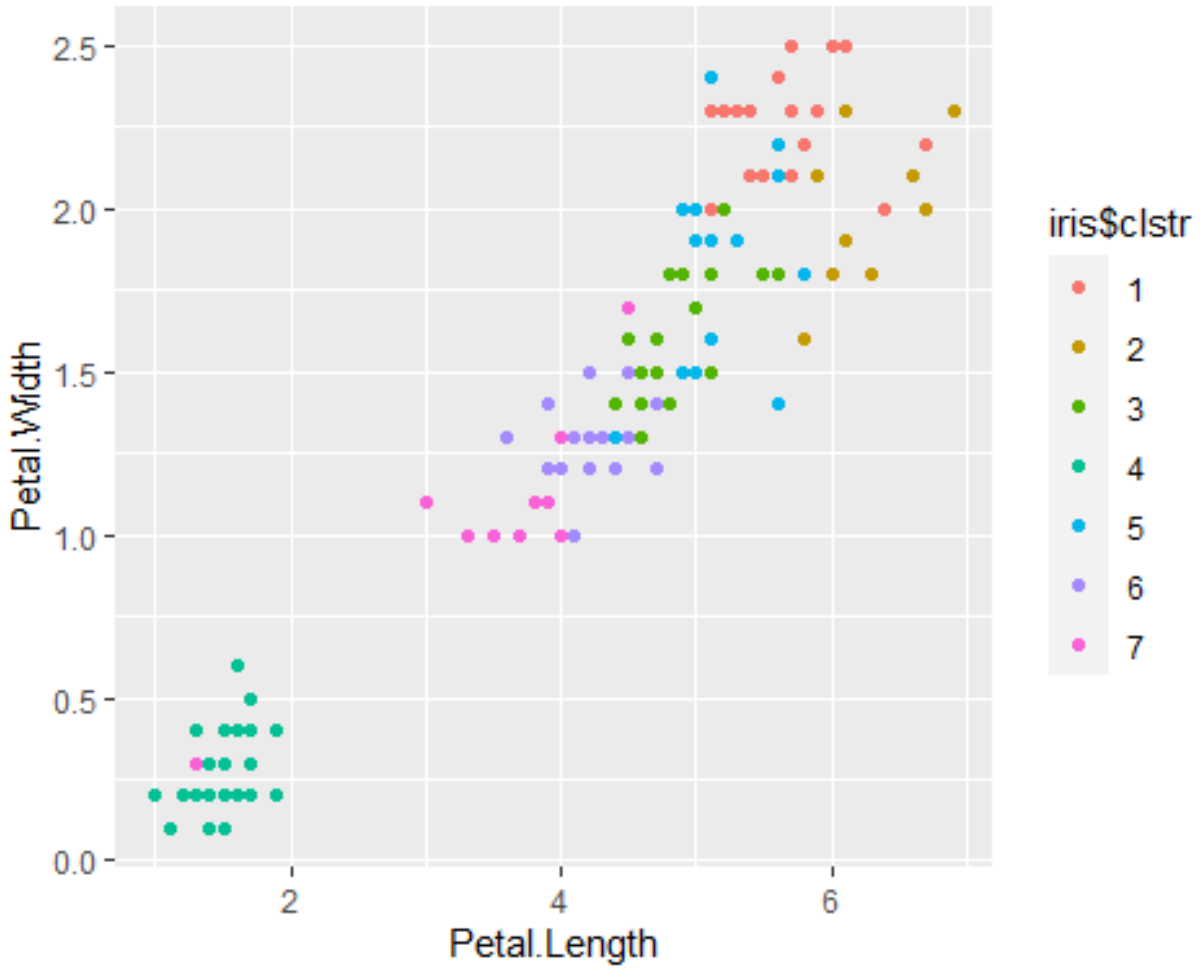
```r
#install.packages("NbClust")
library(NbClust)

# creating function wssplot
wssplot <- function(data, nc=15, seed=1234){
  wss <- (nrow(data)-1)*sum(apply(data,2,var))
  for (i in 2:nc){
    set.seed(seed)
    wss[i] <- sum(kmeans(data, centers=i)$withinss)}
  plot(1:nc, wss, type="b", xlab="Number of Clusters",
       ylab="Within groups sum of squares")}


#Now, let's call the "wssplot" function which we created.

wssplot(iris_3,nc=30,seed=1234)
```




```
#Here,
#Iris_3 - dataset which we want to segment
#nc - maximum number of clusters we are giving
#seed - random initialization of clusters (Keep it constant if you want to
retrieve the same clusters later point of time)

#Here, we have plotted WSS with number of clusters. From here we can see that
there is not much decrease
#in WSS even if we increase the number of clusters beyond 7. This graph is also
known as "Elbow Curve"
#where the bending point (E.g, nc = 7 in our case) is known as "Elbow Point". From
the above plot we can
#conclude that if we keep number of clusters = 3, we should be able to get good
clusters with good
#homogeneity within themselves. Let's fix the cluster size to "7" and call the
kmeans() function to give
#the clusters.

# fitting the clusters
iris_kmeans<-kmeans(iris_3,7)
iris_kmeans$centers
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1    1.1351750  0.50576163    1.0875090   1.4002632
## 2    1.9201365 -0.30998294    1.4211008   1.0358391
## 3    0.6932514 -0.05173771    0.6099721   0.5142373
## 4   -0.9987207  0.90322901   -1.2987572  -1.2521493
## 5    0.3435045 -1.04925145    0.7570644   0.7953192
## 6   -0.1066743 -0.51009528    0.2673769   0.1451866
## 7   -0.6747262 -1.56105274   -0.1330784  -0.1706850
```

```
iris_kmeans$size

## [1] 18  9 23 49 18 20 13

iris$clstr<-iris_kmeans$cluster


# cross-validation with original species available in data

iris$clstr<-iris_kmeans$cluster
table(iris$Species,iris$clstr)

##
##                1  2  3  4  5  6  7
##   setosa       0  0  0 49  0  0  1
##   versicolor   0  0 15  0  4 20 11
##   virginica   18  9  8  0 14  0  1

#Visualization

iris$clstr <- as.factor(iris$clstr)
ggplot(iris, aes(Petal.Length, Petal.Width, color = iris$clstr)) + geom_point()
```




```
#Seeing the results it seems that the clusters are quite good as our clusters
don't have mix of different species (except few exceptions).
```

6. Program to implement Naive Bayes classifier on a given dataset.

Dataset: Iris

Source of the Dataset: "dplyr" package

Objective: Given Sepal and Petal lengths and width predict the class of Iris.

Program:

Program6.R

```
library(e1071)   # for navieBayes
library(caret)   #for Classification And REgression Training
library(dplyr)
data(iris)
#Display first few rows in a iris dataset
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

#summary of iris dataset
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
# Split the data into training and test set
set.seed(123) # Here seed is set to ensure reproducibility;
indexes = createDataPartition(iris$Species, p = .8, list = F)
train = iris[indexes, ]
test = iris[-indexes, ]

#Building the Model using training data
nb_class = naiveBayes(Species~., data = train)
print(nb_class)
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##     setosa versicolor  virginica
##  0.3333333  0.3333333  0.3333333
##
## Conditional probabilities:
```

```
##             Sepal.Length
## Y              [,1]      [,2]
##   setosa     4.9800 0.3567661
##   versicolor 5.9400 0.4903165
##   virginica  6.6375 0.6949221
##
##             Sepal.Width
## Y              [,1]      [,2]
##   setosa     3.3700 0.3450752
##   versicolor 2.7700 0.3267556
##   virginica  3.0125 0.3123012
##
##             Petal.Length
## Y              [,1]      [,2]
##   setosa     1.4650 0.1717930
##   versicolor 4.2325 0.4676112
##   virginica  5.6225 0.5775667
##
##             Petal.Width
## Y              [,1]      [,2]
##   setosa     0.2400 0.0928191
##   versicolor 1.3275 0.2087662
##   virginica  2.0700 0.2662176
#Make predictions
pred = predict(nb_class, test, type="class")
#Confusion Matrix
cm = confusionMatrix(test$Species, pred)
print(cm)
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         10          0         0
##   versicolor      0         10         0
##   virginica       0          2         8
##
## Overall Statistics
##
##                Accuracy : 0.9333
##                  95% CI : (0.7793, 0.9918)
##     No Information Rate : 0.4
##     P-Value [Acc > NIR] : 1.181e-09
##
##                   Kappa : 0.9
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                 1.0000            0.8333           1.0000
## Specificity                 1.0000            1.0000           0.9091
## Pos Pred Value              1.0000            1.0000           0.8000
## Neg Pred Value              1.0000            0.9000           1.0000
## Prevalence                  0.3333            0.4000           0.2667
## Detection Rate              0.3333            0.3333           0.2667
## Detection Prevalence        0.3333            0.3333           0.3333
## Balanced Accuracy           1.0000            0.9167           0.9545
```

7. Build models using Decision trees.

Dataset: Car

URL for the Dataset: https://shorturl.at/bdzPY

Objective: To model a classifier for evaluating the acceptability of car using its given features.

Assumption: The dataset i.e., data file is present in the current working directory

Program:

Program7.R

```
library(caret)
library(rpart.plot)
## Loading required package: rpart

#In case if you face any error while running the code. Frist install the package
rplot.plot using the command install.packages("rpart.plot")

car_df <- read.csv("car.data", sep = ',', header = FALSE)

#For importing data into an R data frame, we can use read.csv() method with
parameters as a file name and whether our dataset consists of the 1st row with a
header or not. If a header row exists then, the header should be set TRUE else
header should set to FALSE.

str(car_df)

## 'data.frame':    1728 obs. of  7 variables:
##  $ V1: chr  "vhigh" "vhigh" "vhigh" "vhigh" ...
##  $ V2: chr  "vhigh" "vhigh" "vhigh" "vhigh" ...
##  $ V3: chr  "2" "2" "2" "2" ...
##  $ V4: chr  "2" "2" "2" "2" ...
##  $ V5: chr  "small" "small" "small" "med" ...
##  $ V6: chr  "low" "med" "high" "low" ...
##  $ V7: chr  "unacc" "unacc" "unacc" "unacc" ...

head(car_df)

##      V1    V2 V3 V4    V5   V6    V7
## 1 vhigh vhigh  2  2 small  low unacc
## 2 vhigh vhigh  2  2 small  med unacc
## 3 vhigh vhigh  2  2 small high unacc
## 4 vhigh vhigh  2  2   med  low unacc
## 5 vhigh vhigh  2  2   med  med unacc
## 6 vhigh vhigh  2  2   med high unacc

#All the features are categorical, so normalization of data is not needed.
#Splitting the dataset
set.seed(123)
intrain <- createDataPartition(y = car_df$V7, p= 0.7, list = FALSE)
training <- car_df[intrain,]
testing <- car_df[-intrain,]

#check dimensions of train & test set
dim(training); dim(testing);

## [1] 1211    7
## [1] 517   7
```

```r
# checking whether our data contains missing values or not?
anyNA(car_df)
```

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

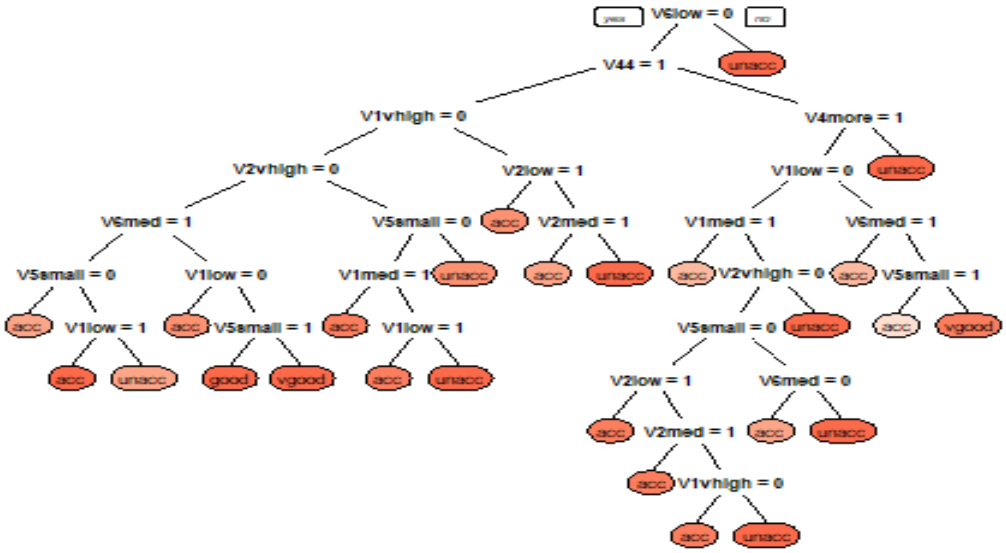
```r
summary(car_df)
```

```
##       V1                 V2                 V3                 V4
##  Length:1728        Length:1728        Length:1728        Length:1728
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##       V5                 V6                 V7
##  Length:1728        Length:1728        Length:1728
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
```

```r
#Training the Decision Tree classifier with criterion as information gain
trctrl <- trainControl(method = "repeatedcv", number = 10, repeats = 3)
set.seed(123)
dtree_fit <- train(V7 ~., data = training, method = "rpart",
                   parms = list(split = "information"),
                   trControl=trctrl,
                   tuneLength = 10)

#Trained Decision Tree classifier results
dtree_fit
```

```
## CART
##
## 1211 samples
##    6 predictor
##    4 classes: 'acc', 'good', 'unacc', 'vgood'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 1091, 1090, 1089, 1089, 1090, 1089, ...
## Resampling results across tuning parameters:
##
##   cp           Accuracy   Kappa
##   0.005494505  0.8695322  0.7215983
##   0.008241758  0.8576747  0.6948754
##   0.010302198  0.8439429  0.6637965
##   0.010989011  0.8343075  0.6441204
##   0.013736264  0.8246562  0.6213005
##   0.019230769  0.8144811  0.5978687
##   0.024725275  0.8130991  0.5936106
##   0.052197802  0.7845025  0.5559936
##   0.063186813  0.7808983  0.5582887
##   0.079670330  0.7379758  0.2680527
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.005494505.
```

```r
#Ploting Decision Tree
prp(dtree_fit$finalModel, box.palette = "Reds", tweak = 1.2)
```




```
#Prediction
testing[1,]
##      V1    V2 V3 V4    V5   V6    V7
## 3 vhigh vhigh  2  2 small high unacc
predict(dtree_fit, newdata = testing[1,])
## [1] unacc
## Levels: acc good unacc vgood
#For our 1st record of testing data classifier is predicting class variable as
"unacc".  Now, its time to predict target variable for the whole test set.
test_pred <- predict(dtree_fit, newdata = testing)
confusionMatrix(test_pred,as.factor(testing$V7))  #check accuracy
## Confusion Matrix and Statistics
##
##           Reference
## Prediction acc good unacc vgood
##      acc   102   14    31     7
##      good    1    3     0     0
##      unacc   7    0   332     0
##      vgood   5    3     0    12
##
## Overall Statistics
##
##                Accuracy : 0.8685
##                  95% CI : (0.8363, 0.8964)
##     No Information Rate : 0.7021
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7211
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##                      Class: acc Class: good Class: unacc Class: vgood
## Sensitivity              0.8870    0.150000       0.9146      0.63158
## Specificity              0.8706    0.997988       0.9545      0.98394
## Pos Pred Value           0.6623    0.750000       0.9794      0.60000
## Neg Pred Value           0.9642    0.966862       0.8258      0.98592
## Prevalence               0.2224    0.038685       0.7021      0.03675
## Detection Rate           0.1973    0.005803       0.6422      0.02321
## Detection Prevalence     0.2979    0.007737       0.6557      0.03868
## Balanced Accuracy        0.8788    0.573994       0.9346      0.80776
```

8. Build your own recommendation system.

Dataset: data

URL for the Dataset: https://shorturl.at/mEFY4

Objective: To model item-based collaborative filtering recommendation.

Assumption: The dataset i.e., CSV file are present in the current working directory

Program:

Program8.R

```
#library(methods)
#library(knitr)
#The following libraries were used
library(recommenderlab) # To test and develop recommender algorithms
library(ggplot2)
library(data.table)

#Some pre-processing of the data available is required before creating the
recommendation system
df_data <- fread('data.csv')
df_data[ ,InvoiceDate := as.Date(InvoiceDate)]

#There is negative Quantity & Unit Price, also NULL/NA Customer ID. We will delete
all the NA Row

df_data[Quantity<=0,Quantity:=NA]
df_data[UnitPrice<=0,UnitPrice:=NA]
df_data <- na.omit(df_data)

#Create a Item Dictionary which allows an easy search of a Item name by any of its
StockCode
setkeyv(df_data, c('StockCode', 'Description'))
itemCode <- unique(df_data[, c('StockCode', 'Description')])
setkeyv(df_data, NULL)

#Convert from transactional to binary metrix, 0 for no transaction and vice versa
df_train_ori <- dcast(df_data, CustomerID ~ StockCode, value.var = 'Quantity',fun.
aggregate = sum, fill=0)

CustomerId <- df_train_ori[,1] #!

df_train_ori <- df_train_ori[,-c(1,3504:3508)]

#Fill NA with 0
for (i in names(df_train_ori))
  df_train_ori[is.na(get(i)), (i):=0]
```

```
#In order to use the ratings data for building a recommendation engine with
recommenderlab,
#we must convert buying matrix into a sparse matrix of type realRatingMatrix.

df_train <- as.matrix(df_train_ori)
df_train <- df_train[rowSums(df_train) > 5,colSums(df_train) > 5]
df_train <- binarize(as(df_train, "realRatingMatrix"), minRatin = 1)

#Training
#We will use Item Base Collaboratife Filtering or IBCF. Jaccard is used because
our data is binary
#Dataset is split Randomly with 80% for training and 20% for test

which_train <- sample(x = c(TRUE, FALSE), size = nrow(df_train),replace = TRUE,
prob = c(0.8, 0.2))
y <- df_train[!which_train]
x <- df_train[which_train]

#Training parameter
#Let's have a look at the default parameters of IBCF model. Here, k is the number
of items to compute
#the similarities among them in the first step. After, for each item, the algorith
m identifies its k
#most similar items and stores the number. method is a similarity funtion, which
is Cosine by default,
#may also be pearson. Let us create the model using the default parameters of
method = Cosine and k=30.

recommender_models<-recommenderRegistry$get_entries(dataType="binaryRatingMatrix")
recommender_models$IBCF_binaryRatingMatrix$parameters
## $k
## [1] 30
##
## $method
## [1] "Jaccard"
##
## $normalize_sim_matrix
## [1] FALSE
##
## $alpha
## [1] 0.5


#Training Dataset
method <- 'IBCF'
parameter <- list(method = 'Jaccard')
n_recommended <- 5
n_training <- 1000
recc_model <- Recommender(data = x, method = method, parameter = parameter)
model_details <- getModel(recc_model)
```

```
#Predict
#Test Dataset is split randomly, We only use 20% for test.Return value of
prediction is top-N-List of
#recommendation item for each user in test dataset.


recc_predicted <-predict(object = recc_model, newdata = y, n = n_recommended,
type="topNList")

#Recomendation item for first 5 user in training dataset
as(recc_predicted,"list")[1:5]

## $`1`
## [1] "20724" "20723" "23204" "22355" "22697"
##
## $`6`
## [1] "22697" "20972" "72709" "22057" "37450"
##
## $`17`
## [1] "22383" "20723" "20725" "22355" "20727"
##
## $`25`
## [1] "22261" "47590A" "22961" "84970L" "23243"
##
## $`28`
## [1] "22386" "20723" "20724" "23203" "22411"

user_1 <- CustomerId[as.integer(names(recc_predicted@items[1]))]

#these are the recommendations for user_1
vvv <- recc_predicted@items[[1]]
vvv <- rownames(model_details$sim)[vvv]
itemCode[vvv]

##    StockCode                         Description
## 1:     20724     RED RETROSPOT CHARLOTTE BAG
## 2:     20723        STRAWBERRY CHARLOTTE BAG
## 3:     23204     CHARLOTTE BAG APPLES DESIGN
## 4:     22355       CHARLOTTE BAG SUKI DESIGN
## 5:     22697 GREEN REGENCY TEACUP AND SAUCER
```