Appendix A

Programming Machine Learning in R

A.1 PRE-REQUISITES

Before starting with machine learning programming in R, we need to fulfil certain pre-requisites. Quite understandably, the first and foremost activity is to install R and get started with the basic programming interface of R, i.e. R console. Then, we need to become familiar with the R commands and scripting in R. In this section, we will have a step-by-step guide of fulfilling these pre-requisites.

A.1.1 Install R in Your System

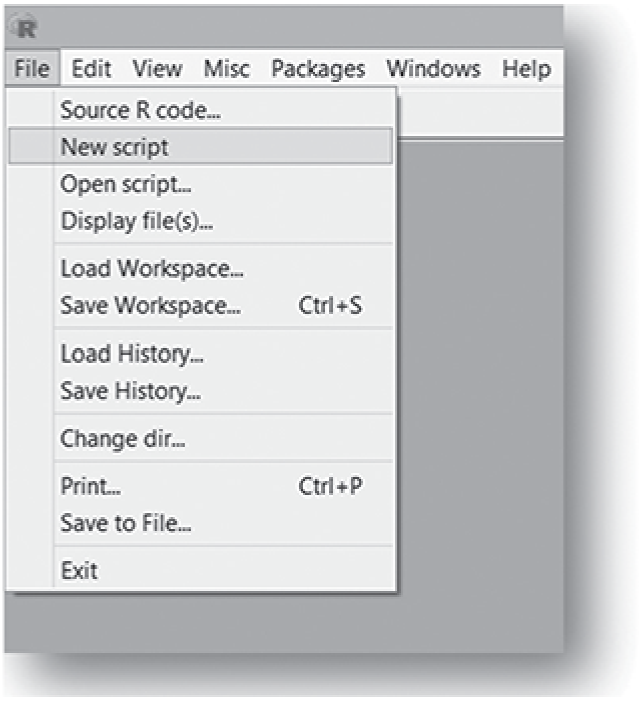
- R 3.5.0 or higher (https://cran.r-project.org/bin/windows/base/)
- RStudio 1.1.453 or higher (*Optional*, only if you want to leverage the advantage of using an integrated development environment (IDE). Otherwise, R console is sufficient.) (https://www.rstudio.com/products/rstudio/download/)

A.1.1.1 Note

- The Comprehensive R Archive Network (or CRAN) is a worldwide network of ftp and web servers that store identical and up-to-date versions of code and documentation for R.
- RStudio is a free and open-source IDE for R.

A.1.2 Know How to Manage R scripts

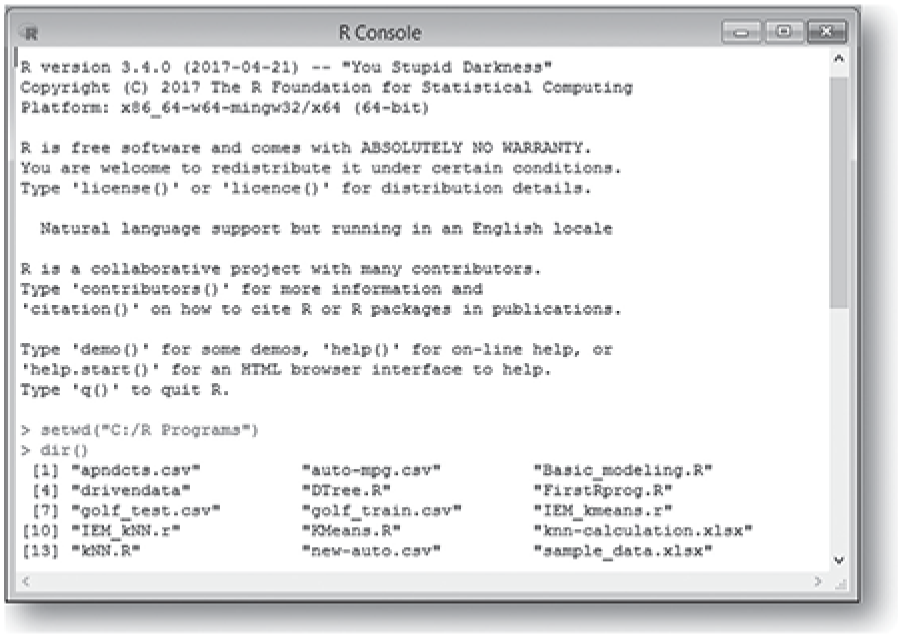
- Open new / pre-existing scripts in R console as shown in Figure A.1:



Copyright © The R Foundation

FIG. A.1 Opening a script in R console

- Scripts can be written / edited on the R editor window, and console commands can be directly executed on the R console window as shown in Figure A.2:



```
R version 3.4.0 (2017-04-21) -- "You Stupid Darkness"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> setwd("C:/R Programs")
> dir()
 [1] "apndcts.csv"          "auto-mpg.csv"          "Basic_modeling.R"
 [4] "drivendata"           "DTree.R"               "FirstRprog.R"
 [7] "golf_test.csv"        "golf_train.csv"        "IEM_kmeans.r"
[10] "IEM_kNN.r"            "KMeans.R"              "knn-calculation.xlsx"
[13] "kNN.R"                "new-auto.csv"          "sample_data.xlsx"
```

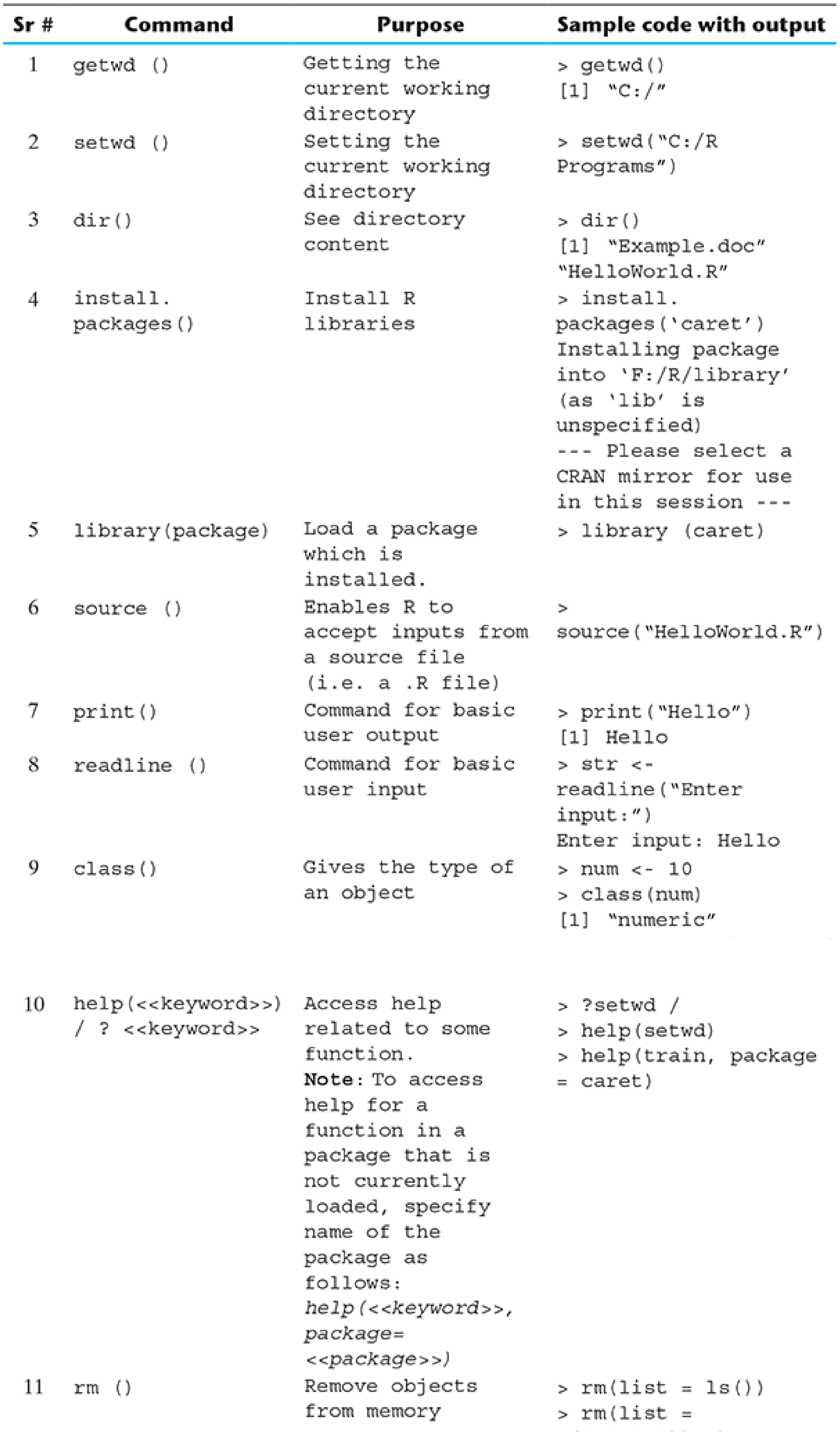
Copyright © The R Foundation

FIG. A.2 Writing code in R console

A.1.3 Know How to do Basic Programming Using R

A.1.3.1 Introduction to basic R commands

Try each of the following commands from the command prompt in R console (or from RStudio, if you want).



Sr #	Command	Purpose	Sample code with output

1	`getwd ()`	Getting the current working directory	`> getwd()`
			`[1] "C:/"`

2	`setwd ()`	Setting the current working directory	`> setwd("C:/R Programs")`

3	`dir()`	See directory content	`> dir()`
			`[1] "Example.doc" "HelloWorld.R"`

4	`install. packages()`	Install R libraries	`> install. packages('caret')`
			`Installing package into 'F:/R/library' (as 'lib' is unspecified)`
			`--- Please select a CRAN mirror for use in this session ---`

5	`library(package)`	Load a package which is installed.	`> library (caret)`
6	`source ()`	Enables R to accept inputs from a source file (i.e. a .R file)	`> source("HelloWorld.R")`

7	`print()`	Command for basic user output	`> print("Hello")`
			`[1] Hello`

8	`readline ()`	Command for basic user input	`> str <- readline("Enter input:")`
			`Enter input: Hello`

9	`class()`	Gives the type of an object	`> num <- 10`
			`> class(num)`
			`[1] "numeric"`

10	`help(<<keyword>>) / ? <<keyword>>`	Access help related to some function. **Note**: To access help for a function in a package that is not currently loaded, specify name of the package as follows: *help(<<keyword>>, package= <<package>>)*	`> ?setwd /`
			`> help(setwd)`
			`> help(train, package = caret)`

11	`rm ()`	Remove objects from memory	`> rm(list = ls())`
			`> rm(list =`

Note:	`c("g","x"))` where g and x are variables created

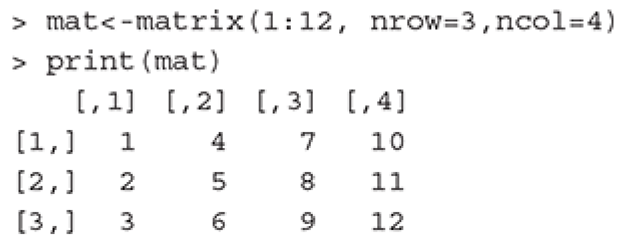
"#" is used for inserting inline comments

<- and = are alternative / synonymous assignment operators

A.1.3.2 Basic data types in R

- **Vector:** This data structure contains similar types of data, i.e. integer, double, logical, complex, etc. The function c () is used to create vectors.
  > num <- c(1,3.4,-2,-10.85) #numeric vector
  > char <- c("a","hello","280") #character vector
  > bool <- c(TRUE,FALSE,TRUE,FALSE,FALSE) #logical vector
  > print(num)
  [1] 1.0 3.4 -2.0 -10.85

- **Matrix:** Matrix is a 2-D data structure and can be created using the matrix () function. The values for rows and columns can be defined using 'nrow' and 'ncol' arguments. However, providing both is not required as the other dimension is automatically acquired with the help of the length parameter (first index : last index).



```
> mat<-matrix(1:12, nrow=3,ncol=4)
> print(mat)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

- **List:** This data structure is slightly different from vectors in the sense that it contains a mixture of different data types A list is created using the list () function.

```
> L <- list(num=10.5, str="Goodbye", matrix=mat)
> print(L)
$num
[1] 10.5
$str
[1] "Goodbye"
$matrix
    [,1] [,2] [,3] [,4]
[1,]   1    4    7   10
[2,]   2    5    8   11
[3,]   3    6    9   12
> print (L[1])#1st component of the list
$num
[1] 10.5 #to truncate '$num', use double indexing, i.e. '[[]]'
```

- **Factor:** The factor stores the nominal values as a vector of integers in the range [1…*k*] (where *k* is the number of unique values in the nominal variable) and an internal vector of character strings (the original values) mapped to these items.

  > data <- c('A','B','B','C','A','B','C','C','B')

  > fact <- factor(data)

  > fact

  [1] A B B C A B C C B

  Levels: A B C

  > table(fact) #arranges argument item in a tabular format fact

  A B C #unique items mapped to frequencies

  2 4 3

- **Data frame:** This data structure is a special case of list where each component is of the same length. Data frame is created using the frame () function.

```
> DF <- data.frame("Num" = 3:5, "Name" = c("Nolan","Kubrick",
"Tarantino"), "Color" = c("Blue",NA,"Red"))
> print(DF)
 Num       Name  Color
1  3      Nolan   Blue
2  4    Kubrick   <NA>
3  5  Tarantino    Red
```

A.1.3.3 Loops

For loop

Syntax:

```
for (variable in sequence)

{

   (loop_body)

}
```

Example: Printing squares of all integers from 1 to 3.

```
for (i in c(1:3))

{

   j = i*i

   print(j)

}

[1] 1

[1] 4

[1] 9
```

While loop

Syntax:

```
while (condition)
```

```
{
   (loop_body)
}
```

Example: Printing squares of all integers from 1 to 3.

```
i <- 1
while(i<=3)
{
   sqr <- i*i
   print(sqr)
   i <- i+1
}
[1] 1
[1] 4
[1] 9
```

If-else statement

Syntax:

```
if (condition 1)
{
   Statement 1
}
else if (condition 2)
{
Statement 2
}
else
{
Statement 4
}
```

Example:

```
x = 0
if (x > 0)
{
print("positive")
} else if (x == 0)
{
print("zero")
} else
{
```

```
print("negative")

}

[1] "zero"
```

A.1.3.4 Writing functions

Writing a function (in a script):

Syntax:

```
function_name <- function(argument_list)

{

  (function_body)

}
```

Example: Function to calculate factorial of an input number *n*.

```
factorial <- function (n)

{

  fact<-1

for(i in 1:n)

{

  fact <-fact*i

}

return(fact)
}
```

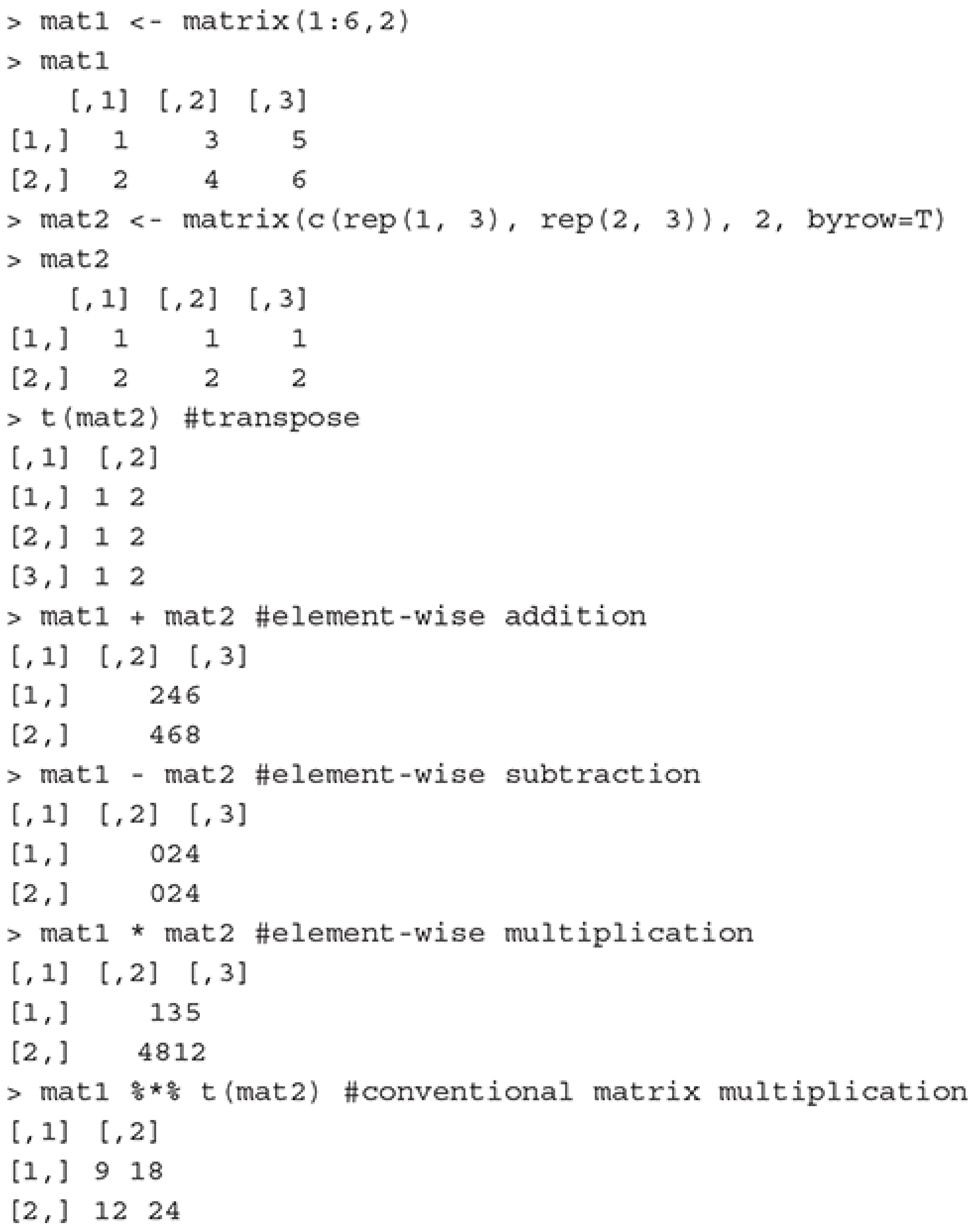
Running the function (after compiling the script by using source (‘script_name’)):

```
> f <- factorial(6)

> print(f)

[1] 720
```

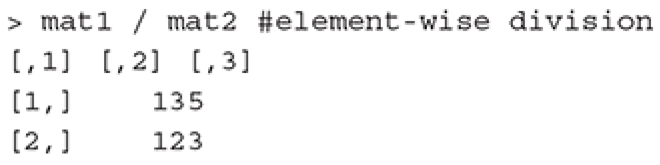
A.1.3.5 Mathematical operations on data types

- **Vectors:**
  ```
  > n <- 10
  > m <- 5
  > n + m #addition
  [1] 15
  > n – m #subtraction
  [1] 5
  > n * m #multiplication
  [1] 50
  > n / m #division
  [1] 2
  ```

- **Matrices:**



```
> mat1 <- matrix(1:6,2)
> mat1
   [,1] [,2] [,3]
[1,]  1    3    5
[2,]  2    4    6
> mat2 <- matrix(c(rep(1, 3), rep(2, 3)), 2, byrow=T)
> mat2
   [,1] [,2] [,3]
[1,]  1    1    1
[2,]  2    2    2
> t(mat2) #transpose
[,1] [,2]
[1,] 1 2
[2,] 1 2
[3,] 1 2
> mat1 + mat2 #element-wise addition
[,1] [,2] [,3]
[1,]    246
[2,]    468
> mat1 - mat2 #element-wise subtraction
[,1] [,2] [,3]
[1,]    024
[2,]    024
> mat1 * mat2 #element-wise multiplication
[,1] [,2] [,3]
[1,]    135
[2,]   4812
> mat1 %*% t(mat2) #conventional matrix multiplication
[,1] [,2]
[1,] 9 18
[2,] 12 24
```



```
> mat1 / mat2 #element-wise division
[,1] [,2] [,3]
[1,]    135
[2,]    123
```

A.1.3.6 Basic data handling commands

> data <- read.csv(“auto-mpg.csv”) # Uploads data from a .csv file

> class(data) # To find the type of the data set object loaded

```
[1] “data.frame”

> dim(data) # To find the dimensions, i.e. the number of rows and columns of the data set loaded

[1] 398 9

> nrow(data) # To find only the number of rows

[1] 398

> ncol(data) # To find only the number of columns

[1] 9

> names (data) #

[1] “mpg” “cylinders” “displacement” “horsepower” “weight”

[6] “acceleration” “model.year” “origin” “car.name”

> head (data, 3) # To display the top 3 rows

> tail(data, 3) # To display the bottom 3 rows

> View(data) # To view the data frame contents in a separate UI

> data[1,9] # Will return cell value of the 1st row, 9th column of a data frame

[1] chevrolet chevelle malibu

> write.csv(data, “new-auto.csv”) # To write the contents of a data frame object to a .csv file

> rbind(data[1:15,], data[25:35,]) # Bind sets of rows
```

> cbind(data[,3], data[,5]) # Bind sets of columns, with the same number of rows

> data <- data[!data$model.year > 74,] #Remove all rows with model year greater than 74

Note:

For advanced data manipulation, the **dplyr** library of R (developed by Hadley Wickham et al) can be leveraged. It is the next version of the **plyr** package focused on working with data frames (hence the name "d"plyr).

A.1.3.7 Advanced data manipulation commands

> library("dplyr")

# Functions to project specific columns

> select (data, cylinders) #Selects a specific feature

> select(data, -cylinders) # De-selects a specific feature

> select(data,2) #selects columns by column index

> select(data,2:3) #selects columns by column index range

> select(data,starts_with("Cyl"))#Selects features by pattern match

Some additional options to project data elements on the basis of conditions are as follows:

- **ends_with ()** = Select columns that end with a character string
- **contains ()** = Select columns that contain a character string
- **matches ()** = Select columns that match a regular expression
- **one_of ()** = Select column names that are from a group of names

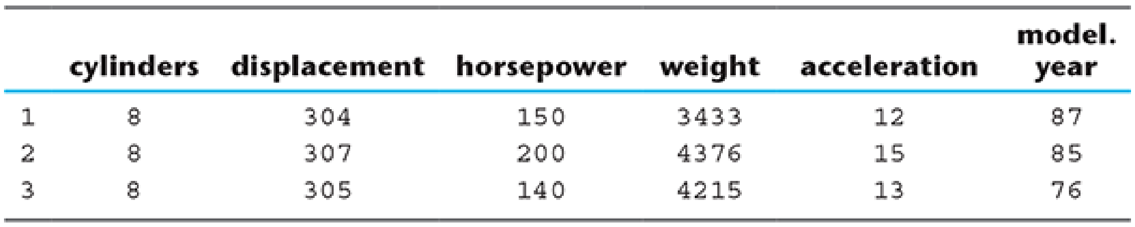
# Functions to select specific rows

```
> filter(data, cylinders == 8) #selects rows based on conditions
```

```
> filter(data, cylinders == 8, model.year > 75)
```

In R, the **pipe** (%>%) operator allows to pipe the output from one function to the input of another function. Instead of nesting functions (reading from inside to outside), the idea of piping is to read the functions from left to right.

```
> data %>% select(2:7) %>% filter(cylinders == 8, model.year > 75) %>%
head(3)
```



	cylinders	displacement	horsepower	weight	acceleration	model. year
1	8	304	150	3433	12	87
2	8	307	200	4376	15	85
3	8	305	140	4215	13	76

```
> arrange(data,model.year) #Sorts ascending rows by a feature
```

```
> arrange(data,- model.year) #Sorts descending rows by a feature
```

```
> mutate(data,Total = mpg*2) #Adds new columns
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model. year	origin
1	18	8	307	130	3504	12.0	70	1
2	15	8	350	165	3693	11.5	70	1
3	18	8	318	150	3436	11.0	70	1

	car.name	total
1	chevrolet chevelle malibu	36
2	buick skylark 320	30
3	Plymouth satellite	36

```
> mutate(data, mpg = mpg*2) #Also, transforms existing columns
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model. year	origin
1	36	8	307	130	3504	12.0	70	1
2	30	8	350	165	3693	11.5	70	1
3	36	8	318	150	3436	11.0	70	1

	car name
1	chevrolet chevelle malibu
2	buick skylark 320
3	plymouth satellite

```
> data %>% select(2:7) %>% filter(cylinders == 8, acceleration >
15.3) %>% group_by(model.year) #Groups rows together according to
attribute values

Source : local data frame [10 x 6]
Groups : model. year [7]
```

	cylinders	displacement	horsepower	weight	acceleration	model. year
	<int>	<dbl>	<fctr>	<int>	<dbl>	<int>
1	8	304	193	4732	18.5	70
2	8	302	140	4294	16.0	72
3	8	302	140	4638	16.0	74
4	8	304	150	4257	15.5	74
5	8	260	110	4060	19.0	77
6	8	260	110	3365	15.5	78
7	8	305	130	3840	15.4	79
8	8	350	125	3900	17.4	79
9	8	260	90	3420	2 2.2	79
10	8	350	105	3725	19.0	81

A.2 PREPARING TO MODEL

Now that we are reasonably familiar with the basic R commands, we have acquired the ability to start machine learning programming in R. But before starting the actual modelling work, we have to first understand the data using the concepts highlighted in Chapter 2. Also, there might be some issues in the data, which we will reveal during data exploration. We have to remediate that too.

So first, let us find out how to do data exploration in R. There are two ways to explore and understand data:

1. By using certain statistical functions to understand the central tendency and spread of the data
2. By visually observing the data in the form of plots or graphs

A.2.1 Basic Statistical Functions for Data Exploration

Let us start with the first approach of understanding the data through statistical techniques. As we have seen in Chapter 2, for any data set, it is critical to understand the central tendency and spread of the data. We have also seen that the standard statistical measures used are as follows:

1. Measures of central tendency – mean, median, mode
2. Measures of data spread
    1. Dispersion of data – variance, standard deviation
    2. Position of the different data values – quartiles, interquartile range (IQR)

In R, there is a function **summary**, which generates the summary statistics of the attributes of a data set. It gives the first basic understanding of the data set, which can trigger thoughts about the data set and the anomalies that may be present. We will use another diagnostic function, str, which compactly provides the structure of a data frame along with the data types of the different attributes. So, let us start exploring a data set **Auto MPG data set** from the University of California, Irvine (UCI) machine learning repository. We will run the *str* and *summary* commands for the Auto MPG data set.

```
 > str(data)
'data. frame' :  398 obs. of 9 variables:
$ mpg         :  num 18 15 18 16 17 15 14 14 14 15 …
$ cylinders   :  int 8888888888 …
$ displacement:  num 307 350 318 304 302 429 454 440 455 3000 …
$ horsepower  :  int 130 165 150 150 140 198 220 215 225 190 …
$ 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.year  :  int 70 70 70 87 70 70 70 70 70 70 …
$ origin      :  int 1111111111 …
$ car.name    :  Factor w/ 305 levels "amc ambassador
                 brougham",..: 50 37 232 1$
```

```
          > summary(data)
      mpg            cylinders        displacement       horsepower         weight
Min.   :9.00     Min.   :3.000   Min.    :  68.0   Min.    :46.0    Min.   :1613
1st Qu.:17.50    1st Qu.:4.000   1st Qu.: 104.2    1st Qu.:75.0     1st Qu.:2224
Median :23.00    Median :4.000   Median : 148.5    Median :93.5     Median :2804
Mean   :23.51    Mean   :5.455   Mean   : 200.0    Mean   :104.5    Mean   :2970
3rd Qu.:29.00    3rd Qu.:8.000   3rd Qu.: 262.0    3rd Qu.:126.0    3rd Qu.:3608
Max    :46.60    Max.   :8.000   Max.   :3000.0    Max.   :230.0    Max.   :5140
                                                   NA's   :6

    acceleration        model.year           origin            car.name
Min.   :8.00       Min.   :60.00     Min.   :1.000     ford pinto    :  6
1st Qu.:13.82      1st Qu.:73.00     1st Qu.:1.000     amc matador   :  5
Median :15.50      Median :76.00     Median :1.000     ford maverick :  5
Mean   :15.57      Mean   :76.07     Mean   :1.573     toyota corolla:  5
3rd Qu.:17.18      3rd Qu.:79.00     3rd Qu :2.000     amc gremlin   :  4
Max.   :24.80      Max.   :90.00     Max.   :3.000     amc hornet    :  4
                                                       (other)       :369
```

Looking closely at the output of the *summary* command, there are six measures listed for the attributes (well, most of them). These are

1. Min. – minimum value of the attribute
2. 1st Qu. – first quartile (for details, refer to Chapter 2)
3. Median
4. Mean
5. 3rd Qu. – third quartile (for details, refer to Chapter 2)
6. Max. – maximum value of the attribute

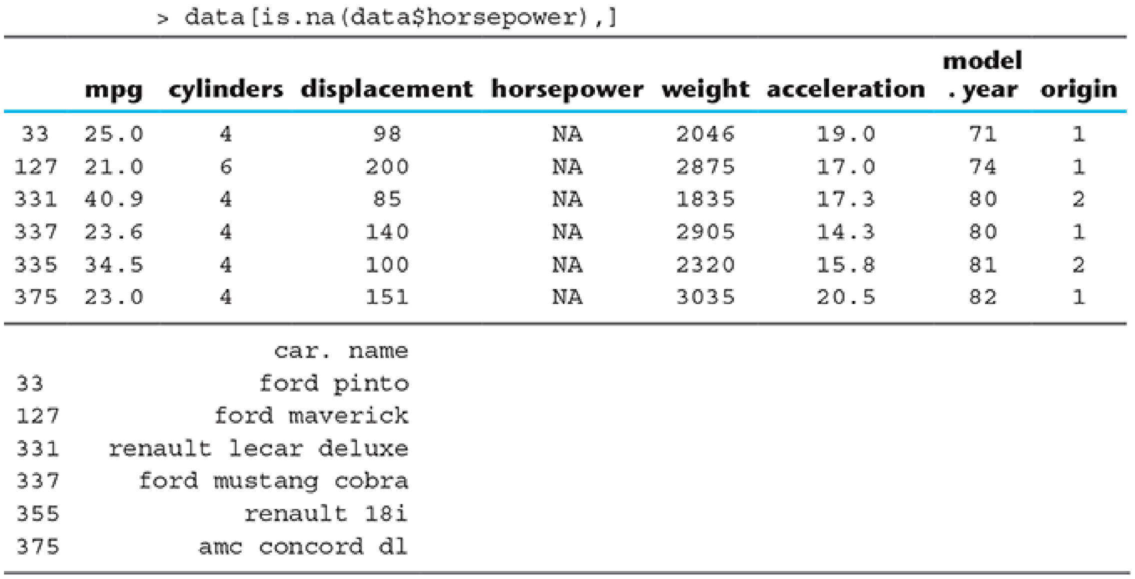
These measures give quite good understanding of the data set attributes. Now, note that the attribute car.name is not showing these values and showing something else. Why is that so and what are the values that it is showing? Let us try to understand the reason for this difference.

The attribute car.name is a nominal, i.e. categorical attribute. As we have already seen in Chapter 2, mathematical or statistical operations are not possible for a nominal variable. Hence, only the unique values for the attribute along with the number of occurrences or frequency are given. We can obtain an exhaustive list of nominal attributes using the following R command.

```
> summary(data$car.name)
           ford pinto          amc matador
                    3                    5
        ford maverick       toyota corolla
                    5                    5
          amc gremlin           amc hornet
                    4                    4

   Chevrolet chevette    Chevrolet impala
                    4                   4
          Peugeot 504       toyota corona
                    4                   4
    Chevrolet caprice  Chevrolet citation
              classic
                    3                   3
       Chevrolet nova      Chevrolet vega
                    3                   3
           dodge colt     ford galaxie 500
                    3                   3
      ford gran torino         honda civic
                    3                   3
```

Next, let us try to explore whether any variable has any issue with the data values where a cleaning may be required. As discussed in Chapter 2, there may be two primary data issues: missing values and outliers.

Let us first try to determine whether there is any missing value for any of the attributes. Let us use a small piece of R code to find out whether there is any missing/ unwanted value for an attribute in the data. If there is such issue, return the rows in which the attribute has missing/unwanted values. By checking all the attributes, we find that the attribute 'horsepower' has missing values.



```
> data[is.na(data$horsepower),]
                                                                   model
     mpg  cylinders  displacement  horsepower  weight  acceleration  . year  origin
 33  25.0      4          98           NA       2046       19.0        71       1
127  21.0      6         200           NA       2875       17.0        74       1
331  40.9      4          85           NA       1835       17.3        80       2
337  23.6      4         140           NA       2905       14.3        80       1
335  34.5      4         100           NA       2320       15.8        81       2
375  23.0      4         151           NA       3035       20.5        82       1

                car. name
33             ford pinto
127         ford maverick
331   renault lecar deluxe
337     ford mustang cobra
355           renault 18i
375         amc concord dl
```

There are six rows in the data set, which have missing values for the attribute 'horsepower'. We will have to remediate these rows before we proceed with the modelling activities. We will do that shortly.

The easiest and most effective method to detect outliers is from the box plot of the attributes. In the box plot, outliers are very clearly highlighted. When we explore the attributes using box plots in a short while, we will have a clear view of this aspect.

Let us quickly see the other R commands for obtaining statistical measures of the numeric attributes.

```
> range(data$mpg) #Gives minimum and maximum values

[1] 9.0 46.6

> diff(range(data$mpg))

[1] 37.6

> quantile(data$mpg)
```

```
0% 25% 50% 75% 100%

9.0 17.5 23.0 29.0 46.6

> IQR(data$mpg)

[1] 11.5

> mean(data$mpg)

[1] 23.51457

> median(data$mpg)

[1] 23

> var(data$mpg)

[1] 61.08961

> sd(data$mpg)

[1] 7.815984
```

Note:

To perform data exploration (as well as data visualization), the **ggplot2** library of R can be leveraged. Created by Hadley Wickham, the ggplot2 library offers a comprehensive graphics module for creating elaborate and complex plots.

A.2.2 Basic Plots for Data Exploration

To start using the library functions of *ggplot2*, we need to load the library as follows:

```
> library(ggplot2)
```

Let us now understand the different graphs that are used for data exploration and how to generate them using R code.

A.2.2.1 Box plot

Syntax: boxplot (x, data, notch, varwidth, names, main)

Usage:

```
> boxplot(iris) # Iris is a popular data set used in machine learning, which comes bundled in R installation
```

A separate window opens in R console with the box plot generated as shown in Figure A.3.

As we can see, Figure A.3 shows the box plot of the entire iris data set, i.e. for all the features in the iris data set, there is a component or box plot in the overall plot. However, if we want to review individual features separately, we can do that too using the following R command.

```
> boxplot(iris$Sepal.Width, main="Boxplot", ylab = "Sepal.Width")
```

The output of the command, i.e. the box plot of an individual feature, sepal width, of the iris data set is shown in Figure A.4.

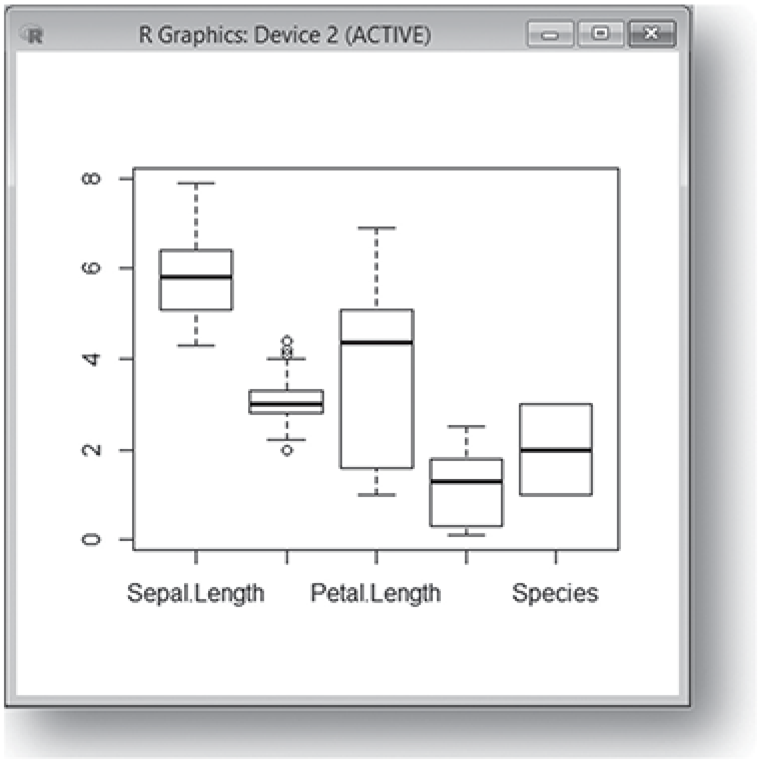


FIG. A.3 Box plot of an entire data set

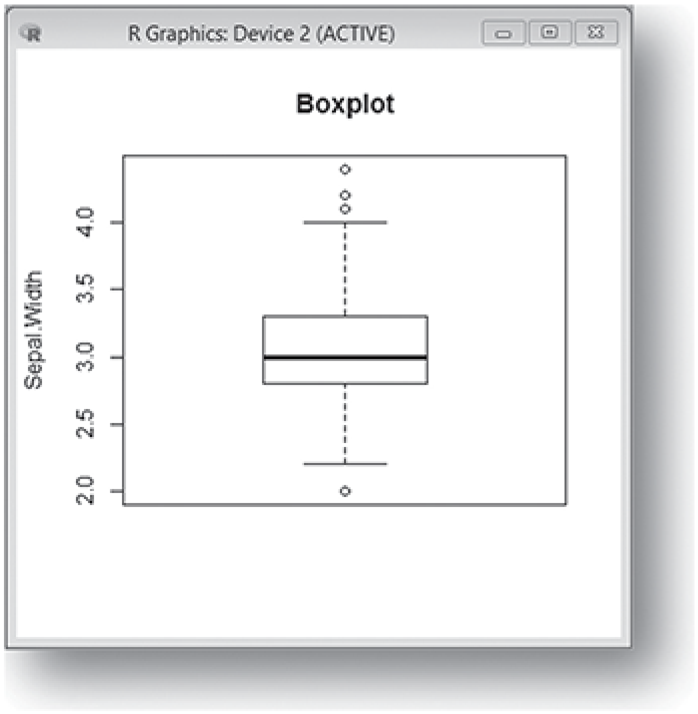


FIG. A.4 Box plot of a specific feature

A.2.2.2 Histogram

Syntax: hist (v, main, xlab, xlim, ylim, breaks, col, border)

Usage:

> hist(iris$Sepal.Length, main = “Histogram”, xlab = “Sepal Length”, col = “blue”, border = “green”)

The output of the command, i.e. the histogram of an individual feature, petal length, of the iris data set is shown in Figure A.5.

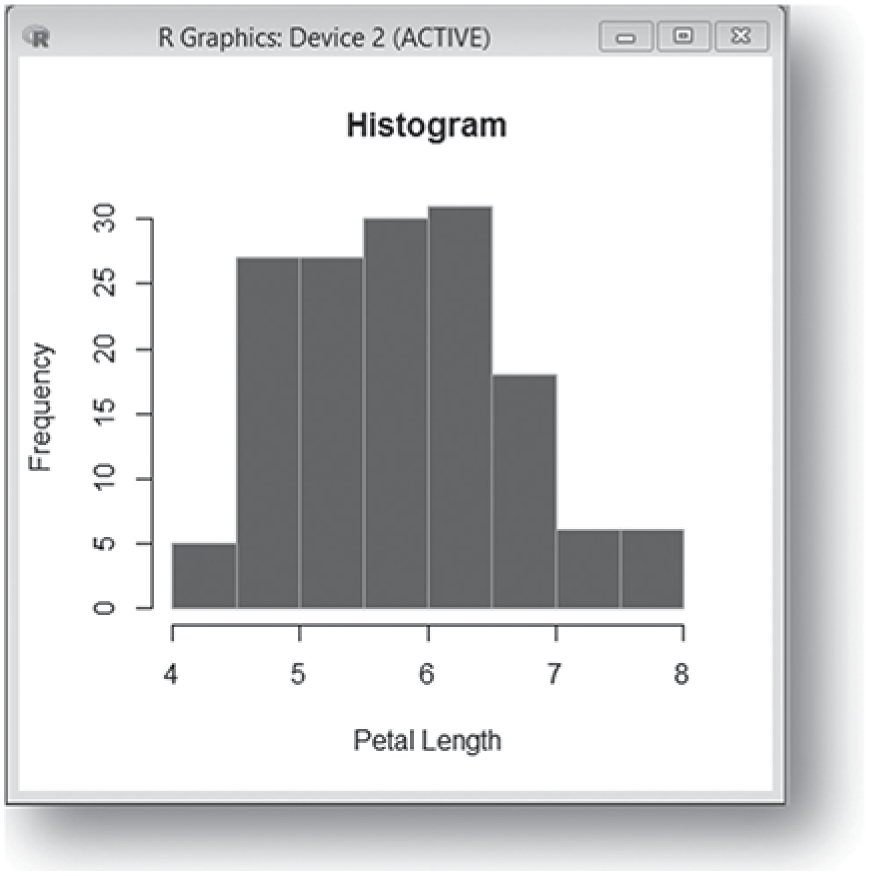


FIG. A.5 Histogram of a specific feature

A.2.2.3 Scatterplot

Syntax: plot (x, y, main, xlab, ylab, xlim, ylim, axes)

Usage:

```
> plot(Sepal.Length~Petal.Length,data=iris,main="Scatter Plot",xlab="Petal Length",ylab="Sepal Length")

> abline(lm(iris $Sepal.Length~ iris$Petal.Length), col="red") # Fit a regression line (red) to show the trend
```

The output of the command, i.e. the scatter plot of the feature pair petal length and sepal length of the iris data set, is shown in Figure A.6.

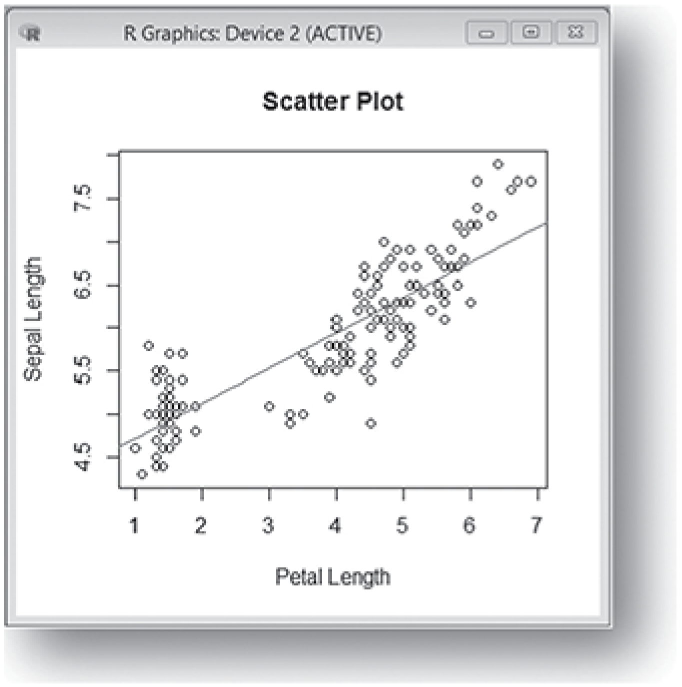


FIG. A.6 Scatter plot of petal length vs sepal length

A.2.3 Data Pre-Processing

The primary data pre-processing activities are remediating data issues related to outliers and missing values. Also, feature subset selection is quite a critical area of data pre-processing. Let us understand how to write programmes for achieving these purposes.

A.2.3.1 Handling outliers and missing values

As we saw in Chapter 2, the primary measures for remediating outliers and missing values are as follows:

- Removing specific rows containing outliers/missing values
- Imputing the value (i.e. outlier/missing value) with a standard statistical measure, e.g. mean or median or mode for that attribute
- Estimate the value (i.e. outlier/missing value) on the basis of value of the attribute in similar records and replace with the estimated value.
- Cap the values within 1.5 times IQR limits

Removing outliers/missing values

We have to first identify the outliers. We have already seen in boxplots that outliers are clearly visible when we draw the box plot of a specific attribute. Hence, we can use the same concept as shown in the following code:

```
> outliers <- boxplot.stats(data$mpg)$out
```

Then, those rows can be removed using the code:

```
> data <- data[!(data$mpg == outliers),]
```

OR,

```
> data <- data[-which(data$mpg == outliers),]
```

For missing value identification and removal, the below code is used:

```
> data1 <- data[!(is.na(data$horsepower)),]
```

Imputing standard values

The code for identification of outliers or missing values will remain the same. For imputation, depending on which statistical function is to be used for imputation, the code will be as follows:

```
> library(dplyr)
```

```
# Only the affected rows are identified and the value of the attribute is transformed to the mean value of the attribute

> imputedrows <- data[which(data$mpg == outliers),] %>% mutate(mpg = mean(data$mpg))

# Affected rows are removed from the data set

> outlier_removed_rows <- data[-which(data$mpg == outliers),]

# Recombine the imputed row and the remaining part of the data set

> data <- rbind(outlier_removed_rows, imputedrows)
```

Almost the same code can be used for imputing missing values with the only difference being in the identification of the relevant rows.

```
> imputedrows <- data[(is.na(data$horsepower)),]%>% mutate (horsepower = mean(data$horsepower))

> missval_removed_rows <- data[!(is.na(data$horsepower)),]

> data <- rbind(outlier_removed_rows, imputedrows)
```

Capping of values

The code for identification of outlier values will remain the same. For capping, generally a value of 1.5 times the IQR is used for imputation, and the code will be as follows:

```
> library(dplyr)

> outliers <- boxplot.stats(data$mpg)$out

> imputedrows <- data[which(data$mpg == outliers),] %>% mutate(mpg = 1.5*IQR(data$mpg))
> outlier_removed_rows <- data[-which(data$mpg == outliers),]
```

```
> data <- rbind(outlier_removed_rows, imputedrows)
```

A.3 MODELLING AND EVALUATION

Note:

The **caret** package (short for Classification And REgression Training) contains functions to streamline the model training process for complex regression and classification problems. The package contains tools for

- data splitting
- different pre-processing functionalities
- feature selection
- model tuning using resampling
- model performance evaluation
  as well as other functionalities.

A.4 MODEL TRAINING

To start using the functions of the *caret* package, we need to include the *caret* as follows:

```
> library(caret)
```

A.4.1 Holdout

The first step before the start of modelling, in the case of supervised learning, is to load the input data, holdout a portion of the input data as test data, and use the remaining portion as training data for building the model. Below is the standard procedure to do it.

```
> inputdata <- read.csv(“btissue.csv”)
```

> split = 0.7 #Ratio in which the input data is to be split to training and test data. A split value = 0.7 indicates 70% of the data will be training data,

i.e. 30% of the input data is retained as test data

> set.seed(123) # This step is optional, needed for result reproducibility

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list = FALSE) # Does a stratified random split

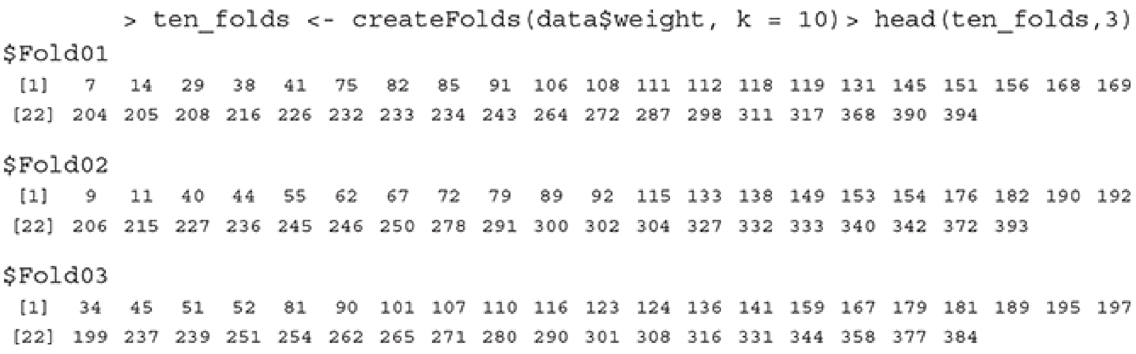
of data into training and test sets

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

A.4.2 K-Fold Cross-Validation

Let us do a 10-fold cross-validation. For creating the cross-validation, functions from the *caret* package can be used as follows:



```
        > ten_folds <- createFolds(data$weight, k = 10)> head(ten_folds,3)
$Fold01
 [1]   7  14  29  38  41  75  82  85  91 106 108 111 112 118 119 131 145 151 156 168 169
[22] 204 205 208 216 226 232 233 234 243 264 272 287 298 311 317 368 390 394

$Fold02
 [1]   9  11  40  44  55  62  67  72  79  89  92 115 133 138 149 153 154 176 182 190 192
[22] 206 215 227 236 245 246 250 278 291 300 302 304 327 332 333 340 342 372 393

$Fold03
 [1]  34  45  51  52  81  90 101 107 110 116 123 124 136 141 159 167 179 181 189 195 197
[22] 199 237 239 251 254 262 265 271 280 290 301 308 316 331 344 358 377 384
```

Next, we perform the data holdout.

train_ds <- inputdata [-ten_folds$Fold01,]

test_ds <- inputdata [ten_folds$Fold01,]

Note:

When we perform data holdout, i.e. splitting of the input data into training and test data sets, the records selected for each set are picked randomly. So, it is obvious that executing the same code or R function may result in different training data sets. The model trained will also be somewhat different.

In R, there is a **set.seed** function which sets the starting point of the random number generator used internally to pick the records. This ensures that random numbers of a specific sequence are used every time, and hence, the same records (i.e. records having the same sequence number) are picked every time and the model is trained in the same way. This is extremely critical for the reproducibility of results, i.e. every time, the same machine learning program generates the same set of results. The code is as follows:

```
> set.seed (5)
```

A.4.3 Bootstrap Sampling

To generate a bootstrap sample for any statistics, R package **boot** can be used. A sample code is given below.

```
> install.packages (“boot”)

> library (boot)

myfunc <- function (){

# body of the function …
return (someval)
}

> bootcorr <- boot(data = mydata, statistic = myfunc, R = 500) # R is the number of bootstrap samples
```

A.4.4 Training the Model

Once the model preparatory steps such as data holdout, etc. are completed, the actual training starts using the following code or similar codes.

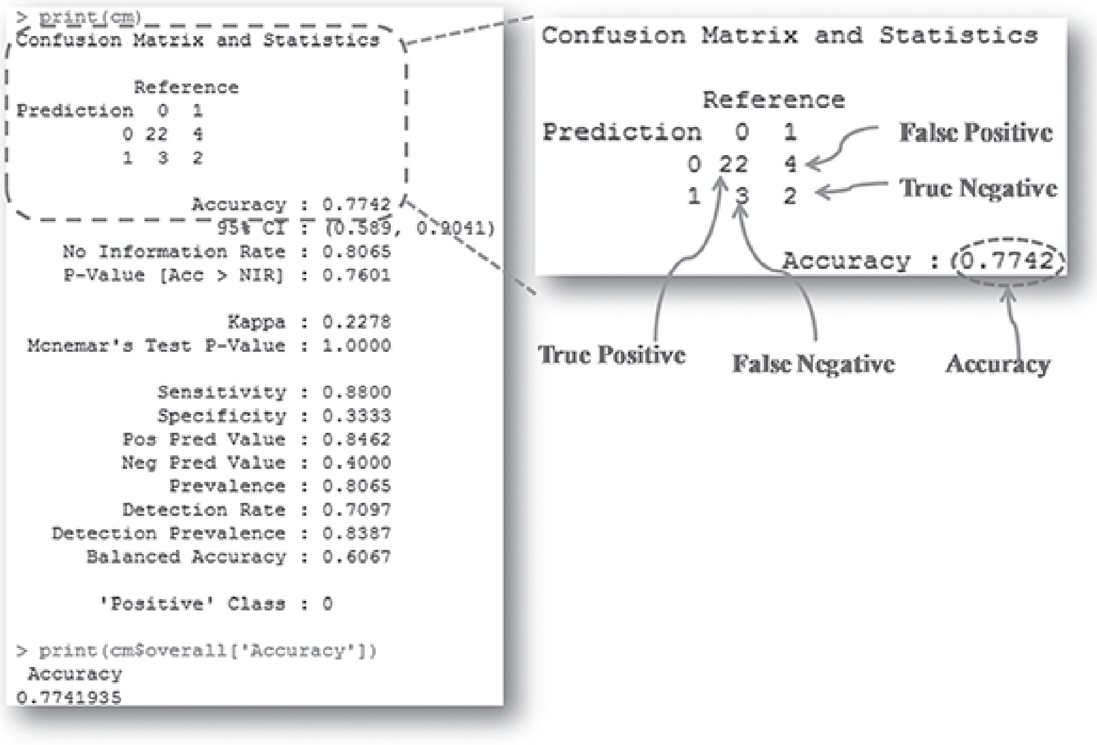
> model <- train(class ~ ., method = “rpart”, data = train_ds) # Code for the decision tree model

A.4.5 Evaluating Model Performance

There are different ways to evaluate the different models in supervised and unsupervised learning. Some of them, as we have seen, have been discussed in Chapter 3. Now, it is time to see how we can implement them through R code.

A.4.5.1 Supervised learning - classification

In supervised learning, model accuracy is the most critical measure for evaluating a model’s performance. There are also other measures such as sensitivity, specificity, precision, recall, etc., each of which we have studied in Chapter 3. R library *caret* gives a function *confusionMatrix* to reveal the confusion matrix of a model, and on the basis of the confusion matrix, values of the different measures, namely accuracy, sensitivity, specificity, precision, recall, etc., are obtained. Figure A.7 presents a snapshot of the *confusionMatrix* output for a specific data set.



```
> print(cm)
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 22  4
         1  3  2

               Accuracy : 0.7742
                 95% CI : (0.589, 0.9041)
    No Information Rate : 0.8065
    P-Value [Acc > NIR] : 0.7601

                  Kappa : 0.2278
 Mcnemar's Test P-Value : 1.0000

            Sensitivity : 0.8800
            Specificity : 0.3333
         Pos Pred Value : 0.8462
         Neg Pred Value : 0.4000
             Prevalence : 0.8065
         Detection Rate : 0.7097
   Detection Prevalence : 0.8387
      Balanced Accuracy : 0.6067

       'Positive' Class : 0

> print(cm$overall['Accuracy'])
 Accuracy
0.7741935
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 22  4
         1  3  2

Accuracy : 0.7742
```

FIG. A.7 Performance evaluation of a classification model (Decision Tree)

> predictions <- predict(model, test_ds, na.action = na.pass)

> cm <- confusionMatrix(predictions, test_ds$class)

> print(cm)

> print(cm$overall[‘Accuracy’])

A.4.5.2 Supervised learning - regression

The *summary* function, when applied to a regression model, displays the different performance measures such as residual standard error, multiple R-squared, etc., both for simple and multiple linear regression.

A.4.5.3 Unsupervised learning - clustering

As we have seen in Chapter 3, there are two popular measures of cluster quality: purity and silhouette width. Purity can be calculated only when class label is known for the data set subjected to clustering. On the other hand, silhouette width can be calculated for any data set.

Purity

We will use a Lower Back Pain Symptoms data set released by Kaggle (https://www.kaggle.com/sammy123/lower-back-pain-symptoms-dataset). The data set *spine.csv* consists of 310 observations and 13 attributes (12 numeric predictors, 1 binary class attribute).

```
> library(fpc)

> data <- read.csv("spine.csv") #Loading the Kaggle data set

> data_wo_class <- data[,-length(data)] #Stripping off the class attribute from the data set before clustering

> class <- data[,length(data)] #Storing the class attribute in a separate variable for later use

> dis = dist(data_wo_class)^2

> res = kmeans(data_wo_class,2) #Can use other clustering algorithms too

#Let us define a custom function to compare cluster value with the original class value and calculate the percentage match

ClusterPurity <- function(clusters, classes) {

sum(apply(table(classes, clusters), 2, max)) / length(clusters)

}

> ClusterPurity(res$cluster, class)
```

Output

```
[1] 0.6774194
```

Silhouette width

Use the R library **cluster** to find out/plot the silhouette width of the clusters formed. The piece of code below clusters the records in the data set *spinem.csv* (the same data set as spine.csv with the target variable removed) using the *k-means* algorithm and then calculates the silhouette width of the clusters formed.

```
> library (cluster)

> data <- read.csv("spinem.csv")

> dis = dist(data)^2

> res = kmeans(data,2) #Can use other clustering algorithms too

> sil_width <- silhouette (res$cluster, dis)

> sil_summ <- summary(sil_width)

> sil_summ$clus.avg.widths # Returns silhouette width of each cluster

> sil_summ$avg.width # Returns silhouette width of the overall data set
```

Output

```
Silhouette width of each cluster:

1         2

0.7473583 0.1921082

Silhouette width of the overall data set:

[1] 0.5413785
```

A.5 FEATURE ENGINEERING

A.5.1 Feature Construction

For performing feature construction, we can use *mutate* function of the **dplyr** package. Following is a small code for feature construction using the iris data set.

A.5.1.1 Dummy coding categorical (nominal) variables

As in the above case, we can use the *dummy.code* function of the **psych** package to encode categorical variables. Following is a small code for the same.

```
> library(psych)

> age <- c(18,20,23,19,18,22)

> city <- c('City A','City B','City A','City C','City B')

> data <- data.frame(age, city)

> data
```

A.5.1.2 Encoding categorical (ordinal) variables

A.5.1.3 Transforming numeric (continuous) features to categorical features

A.5.2 Feature Extraction

A.5.2.1 Principal Component Analysis (PCA)

For performing principal component analysis (PCA), we can use the *prcomp* function of the **stats** package. Following is the code using the iris data set. PCA should be applied to the predictors. The class variable can be used to visualize the principal components.

The output of the *biplot* function is given in Figure A.8

FIG. A.8 Principal components of the iris data set

A.5.2.2 Singular Value Decomposition (SVD)

For performing singular value decomposition, we can use the *svd* function of the **stats** package. Following is the code using the iris data set.

```
> sing_val_decomp <- svd(iris[,1:4])

> print(sing_val_decomp$d)
```

Output:

```
[1] 95.959914 17.761034 3.460931 1.884826
```

A.5.2.3 Linear Discriminant Analysis (LDA)

For performing linear discriminant analysis, we can use the *lda* function of the **MASS** package. Following is the code using the UCI data set *btissue*.

The output of the above function is given in Figure A.9

FIG. A.9 LDA of the btissue data set

A.5.3 Feature Subset Selection

Feature subset selection is a topic of intense research. There are many approaches to select a subset of features which can improve model performance. It is not possible to cover all such approaches as a part of this text. However, only for basic feature selection functionalities, the **FSelector** package of R can be used.

```
library(FSelector)

data <- iris
```

```
feat_subset <- cfs(Species ~ ., data) # Selects feature subset using correlation and entropy measures for continuous and discrete data
```

Below is a programme to perform feature subset selection before applying the same for training a model.

```
library(caret)

library(FSelector)

inputdata <- read.csv(“apndcts.csv”)

split = 0.7

set.seed(123)

trainIndex <- createDataPartition (y = inputdata$class, p = split, list = FALSE)

train_ds <- inputdata [trainIndex,]

test_ds <- inputdata [-trainIndex,]

feat_subset <- cfs(class ~ ., train_ds) #Feature selection done class <- train_ds$class

train_ds <- cbind(train_ds[feat_subset], class) # Subset training data created

class <- test_ds$class

test_ds <- cbind(test_ds[feat_subset], class) # Subset test data created

train_ds$class <- as.factor(train_ds$class)

test_ds$class <- as.factor(test_ds$class)
```

Applying Decision Tree classifier here. Any other model can also be applied **...**

model <- train(class ~ ., method = “rpart”, data = train_ds) predictions <- predict(model, test_ds, na.action = na.pass)

cm <- confusionMatrix(predictions, test_ds$class)

cm$overall[‘Accuracy’]

Note:

The **e1071** package is an important R package which contains many statistical functions along with some critical classification algorithms such as Naïve Bayes and support vector machine (SVM). It is created by David Meyer and team and maintained by David Meyer.

A.6 MACHINE LEARNING MODELS

A.6.1 Supervised Learning – Classification

In Chapters 6, 7, and 8, conceptual overview of different supervised learning algorithms has been presented. Now, you will understand how to implement them using R. For the sake of simplicity, the code for implementing each of the algorithms is kept as consistent as possible. Also, we have used benchmark data sets from UCI repository (these data sets will also be available online, refer to the URL https://archive.ics.uci.edu/ml).

A.6.1.1 Naïve Bayes classifier

To implement this classifier, the *naiveBayes* function of the **e1071** package has been used. The full code for the implementation is given below.

> library(caret)

> library (e1071)

```
> inputdata <- read.csv("apndcts.csv")

> split = 0.7

> set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list =
FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> train_ds$class <- as.factor(train_ds$class) #Pre-processing step

> test_ds$class <- as.factor(test_ds$class) #Pre-processing step

> model <- naiveBayes (class ~ ., data = train_ds)

> predictions <- predict(model, test_ds, na.action = na.pass)

> cm <- confusionMatrix(predictions, test_ds $class)

> cm$overall['Accuracy']
```

Output Accuracy:

```
0.8387097
```

A.6.1.2 kNN classifier

To implement this classifier, the *knn* function of the **class** package has been used. The full code for the implementation is given below.

```
> library(caret)

> library (class)
> inputdata <- read.csv("apndcts.csv")

> split = 0.7

> set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list =
FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> train_ds$class <- as.factor(train_ds$class) #Pre-processing step

> test_ds$class <- as.factor(test_ds$class) #Pre-processing step

> model <- knn(train_ds, test_ds, train_ds$class, k = 3)

> cm <- confusionMatrix(model, test_ds$class)

> cm$overall['Accuracy']
```

Output Accuracy:

```
1
```

A.6.1.3 Decision tree classifier

To implement this classifier implementation, the *train* function of the **caret** package can be used with a parameter *method* = *"rpart"* to indicate that the train function will use the decision tree classifier. Optionally, the *rpart* function of the **rpart** package can also be used. The full code for the implementation is given below.

```
> library(caret)
> library(rpart) # Optional

> inputdata <- read.csv("apndcts.csv")

> split = 0.7

> set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list =
FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> train_ds$class <- as.factor(train_ds$class) #Pre-processing step

> test_ds$class <- as.factor(test_ds$class) #Pre-processing step

> model <- train(class ~ ., method = "rpart", data = train_ds, prox = TRUE)

# May also use the rpart function as shown below …

#> model <- rpart(formula = class ~ ., data = train_ds) # Optional

> predictions <- predict(model, test_ds, na.action = na.pass)

> cm <- confusionMatrix(predictions, test_ds $class)

> cm$overall['Accuracy']
```

Output Accuracy:

```
0.8387097
```

A.6.1.4 Random forest classifier

To implement this classifier, the *train* function of the **caret** package can be used with the parameter *method* = *"rf"* to indicate that the train function will use the decision tree classifier. Optionally, the *randomForest* function of the **randomForest** package can also be used. The full code for the implementation is given below.

```
> library(caret)

> library(randomForest) # Optional

> inputdata <- read.csv("apndcts.csv")

> split = 0.7

> set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list = FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> train_ds$class <- as.factor(train_ds$class) #Pre-processing step

> test_ds$class <- as.factor(test_ds$class) #Pre-processing step

> model <- train(class ~ ., method = "rf", data = train_ds, prox = TRUE)

# May also use the randomForest function as shown below …

#> model <- randomForest(class ~ . , data = train_ds, ntree=400)

> predictions <- predict(model, test_ds, na.action = na.pass)
> cm <- confusionMatrix(predictions, test_ds $class)

> cm$overall[‘Accuracy’]
```

Output Accuracy:

```
0.8387097
```

A.6.1.5 SVM classifier

To implement this classifier, the *svm* function of the **e1071** package has been used. The full code for the implementation is given below.

```
> library(caret)

> library (e1071)

> inputdata <- read.csv(“apndcts.csv”)

> split = 0.7

> set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list =
FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> train_ds$class <- as.factor(train_ds$class) #Pre-processing step

> test_ds$class <- as.factor(test_ds$class) #Pre-processing step

> model <- svm(class ~ ., data = train_ds)

> predictions <- predict(model, test_ds, na.action = na.pass)
> cm <- confusionMatrix(predictions, test_ds $class)

> cm$overall['Accuracy']
```

Output Accuracy:

```
0.8709677
```

A.6.2 Supervised Learning – Regression

As discussed in Chapter 9, two main algorithms used for regression are simple linear regression and multiple linear regression. Implementation of both the algorithms in R code is shown below.

```
> data <- read.csv("auto-mpg.csv")

> attach(data) # Data set is attached to the R search path, which means that while evaluating a variable, objects in the data set can be accessed by simply giving their names. So, instead of "data$mpg", simply giving "mpg" will suffice.

> data <- data[!is.na(as.numeric(as.character(horsepower))),]

> data <- mutate(data, horsepower = as.numeric(horsepower))

> outliers_mpg <- boxplot.stats(mpg)$out

> data <- data[-which(mpg == outliers_mpg),]

> outliers_cylinders <- boxplot.stats(cylinders)$out

> data <- data[-which(cylinders == outliers_cylinders),]

> outliers_displacement <- boxplot.stats(displacement)$out

> data <- data[-which(displacement == outliers_displacement),]
> outliers_weight <- boxplot.stats(weight)$out

> data <- data[-which(weight == outliers_weight),]

> outliers_acceleration <- boxplot.stats(acceleration)$out

> data <- data[-which(acceleration == outliers_acceleration),]
```

A.6.2.1 Simple linear regression

```
> reg_pred <- lm(mpg ~ cylinders)

> summary(reg_pred)
```

A.6.2.2 Multiple linear regression

```
> reg_pred <- lm(mpg ~ cylinders + displacement)

> reg_pred <- lm(mpg ~ cylinders + weight + acceleration)

> reg_pred <- lm(mpg ~ cylinders + displacement + horsepower + weight +
acceleration)
```

A.6.3 Unsupervised Learning

To implement the *k-means* algorithm, the *k-means* function of the **cluster** package has been used. Also, because silhouette width is a more generic performance measure, it has been used here. The complete R code for the implementation is given below.

```
> library (cluster)

> data <- read.csv("spinem.csv")

> dis = dist(data)^2

> res = kmeans(data,2)
> sil_width <- silhouette (res$cluster, dis)

> sil_summ <- summary(sil_width)

> sil_summ$avg.width # Returns silhouette width of the overall data set
```

Output Accuracy:

```
[1] 0.5508955
```

A.6.4 Neural Network

To implement this classifier, the *neuralnet* function of the **neuralnet** package has been used. The full code for the implementation is given below.

A.6.4.1 Single-layer feedforward neural network

```
> library(caret)

> library(neuralnet)

> inputdata <- read.csv("apndcts.csv")

> split = 0.7

> set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list = FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> model <- neuralnet(class ~ At1 + At2 + At3 + At4 + At5 + At6 + At7, train_ds)
```

> plot(model)

Output is presented in Figure A.10.

A.6.4.2 Multi-layer feedforward neural network

> library(caret)

> library(neuralnet)

> inputdata <- read.csv("apndcts.csv")

> split = 0.7 > set.seed(123)

> trainIndex <- createDataPartition (y = inputdata$class, p = split, list = FALSE)

> train_ds <- inputdata [trainIndex,]

> test_ds <- inputdata [-trainIndex,]

> model <- neuralnet(class ~ At1 + At2 + At3 + At4 + At5 + At6 + At7, train_ds, hidden = 3) # Multi-layer NN

> plot(model)

FIG. A.10 Single-layer NN

Output is presented in Figure A.11.

FIG. A.11 Multi-layer NN

A.7 MACHINE LEARNING L AB SCHEDULE