III/IV B.Tech (Regular) DEGREE EXAMINATION

April, 2018 **Computer Science & Engineering**

Sixth Semester **Introduction to Data Analytics**

SEHEMA of EVALUATION

Answer Question No.1 compulsorily. (1X12 = 12 Marks)

Answer ONE question from each unit. (4X12=48 Marks)

1 Answer all questions (1X12=12 Marks)

a) What is the significance of R?

ANS: We perform a hypothesis test of the **significance** of the correlation coefficient to decide whether the linear relationship in the sample data is strong enough to use to model the relationship in the population. The sample data are used to compute **r**, the correlation coefficient for the sample.

b) How do we call a function in R?

ANS: 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
}
```

c) What is the need of a frame?

ANS: A **data frame** is used for storing data tables. It is a list of vectors of equal length

Following are the characteristics of a data frame.

The column names should be non-empty.

- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

d) Illustrate rbind () function.

ANS: To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function.

rbind(x1,x2,...)

x1,x2:vector, matrix, data frames

Read in the data from the file:

```
>x <- read.csv("data1.csv",header=T,sep=",")
>x2 <- read.csv("data2.csv",header=T,sep=",")
>x3 <- rbind(x,x2)
```

The column of the two datasets must be same, otherwise the combination will be meaningless

e) Which package has the rename() function.

ANS: Reshape package

f) What is the use of merge() function.

ANS: To merge two data frames (datasets) horizontally, you use the merge() function. In most cases, two data frames are joined by one or more common key variables

```
total <- merge(dataframeA, dataframeB, by="ID")
```

g) What is the linear regression?

ANS: **Linear regression** is a basic and commonly used type of predictive analysis. ... These regression estimates are used to explain the relationship between one dependent variable and one or more independent variables.

h) List the applications of ANOVA.

ANS: ANOVA models can realistically be used in numerous industries and applications:

1. Comparing the gas mileage of different vehicles, or the same vehicle under different fuel types, or road types.
2. Understanding the impact of temperature, pressure or chemical concentration on some chemical reaction (power reactors, chemical plants, etc)
3. Understanding the impact of different catalysts on chemical reaction rates
4. Studying whether advertisements of different kinds solicit different numbers of customer responses
5. Understanding the performance, quality or speed of manufacturing processes based on number of cells

i) Why is T-test needed?

ANS: A **t-test** is an analysis of two population means through the use of statistical examination; a **t-test** with two samples is commonly used with small sample sizes, **testing** the difference between the samples when the variances of two normal distributions are not known.

j) What is classification?

ANS: A classification is an ordered set of related categories used to group data according to its similarities. It consists of codes and descriptors and allows survey responses to be put into meaningful categories in order to produce useful data.

k) Write any one formulae for calculating distances in clustering.

ANS: Euclidean distance computes the root of square difference between co-ordinates of pair of objects.

$$Dist_{XY} = \sqrt{\sum_{k=1}^{m} (X_{ik} - X_{jk})^2}$$

l) Why do we need logistic regression?

ANS: **Logistic regression** is the appropriate **regression** analysis to conduct when the dependent variable is binary. **Logistic regression** is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

UNIT I

2 a) Discuss with an example how a function is defined and called in R. 6 M

ANS: A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

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.

Function Definition

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

}
```

Function Components

The different parts of a function are −

- **Function Name** − This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** − The function body contains a collection of statements that defines what the function does.
- **Return Value** − The return value of a function is the last expression in the function body to be evaluated.

Calling a Function

```
# 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 without an Argument

```
# Create a function without an argument.

new.function <- function() {

  for(i in 1:5) {

    print(i^2)

  }

}



# Call the function without supplying an argument.

new.function()
```

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

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# 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)
```

2 b) Write a program for matrix multiplication in R. 6 M

ANS: Multiplies two matrices, if they are conformable. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors of the same length, it will return the inner product (as a matrix).

Usage

```
x %*% y
```

Arguments

x, y   Numeric or complex matrices or vectors.

Details

When a vector is promoted to a matrix, its names are not promoted to row or column names, unlike as.matrix.

Promotion of a vector to a 1-row or 1-column matrix happens when one of the two choices allows x and y to get conformable dimensions.

Example:

```
v1 <- c(1,2,3)
> v2 <- matrix(c(3,1,2,2,1,3,3,2,1), ncol = 3, byrow = TRUE)
> v1 %*% t(v2)
     [,1] [,2] [,3]
[1,]   11   13   10
```

(OR)

3 a) Explain how to read data from CSV files in R. 6 M

ANS: **Reading CSV**

The best way to read data from a CSV file is to use read.table. It might be tempting to use read.csv but that is more trouble than it is worth, and all it does is call read.table with some arguments preset. The result of using read.table is a data.frame

```
#Importing and exporting a csv file
getwd()#to know present working directory
setwd("h:\\class hours")#to set the directory
read.csv("salesjan2009.csv")->a#Importing a csv file

#Exporting csv file by using write.csv function

write.table(dt, file="mydata.csv",sep=",",row.names=F)

x<-10:1
x
y<--4:5
y
q<-c("Hockey","Football","Baseball","Curling","rugby","Lacrosse","Basketball","Tennis",
   + "Cricket","Soccer")
thedf<-data.frame(x,y,q)
thedf
write.csv(thedf,"thedf.csv")
```

Text file;

Reading text files into R;

You can import data from a text file (often CSV) using read.table(), read.csv() or read.csv2(). The option header = TRUE indicates that the first line of the CSV file should be interpreted as variables names and the option sep = gives the separator (generally "," or ";").

Syntax is mydata<-read.table("data.txt",header=**TRUE**)

Writing text document

R users prefer to export their data from R to TXT file, because the TXT file is easy imported on other applications. However, most used statical applications

```
write.table(dt, "mydata.txt", sep=",")
```

3 b) Write a R program to read & write a Excel file 6 M

ANS:

Excel data:

Reading excel data:

One of the best ways to read an Excel file is to export it to a comma delimited file and import it using the method above. Alternatively you can use the **xlsx** package to access Excel files. The first row should contain variable/column names.

```
# read in the first worksheet from the workbook myexcel.xlsx
# first row contains variable names
library(xlsx)
mydata<- read.xlsx("c:/myexcel.xlsx", 1)
# read in the worksheet named mysheet
mydata<- read.xlsx("c:/myexcel.xlsx", sheetName = "mysheet")
```

writing excel data:

thexlsx package can do a lot more than just reading data: it can also be used to write data frames to Excel workbooks and to manipulate the data further into those files. If you would also like to write a data frame to an Excel workbook, you can just use write.xlsx() and write.xlsx2().

```
library(xlsx)
write.xlsx(mydata, "c:/mydata.xlsx")
```

UNIT II

4 a) List and write the descriptions of the character functions. 6 M

ANS: **CHARACTER FUNCTIONS:**

R that can be used as the basic building blocks for manipulating data. They can be divided into numerical (mathematical, statistical,probability) and character functions.

In R, strings are stored in a character vector. You can create strings with a single quote / double quote.

For example, x = "I love R Programming"

1. Convert object into character type

2. Check the character type

3. Concatenate Strings

The **paste function** is used to join two strings. It is one of the most important string manipulation task. Every analyst performs it almost daily to structure data.

Paste Function Syntax

paste (objects, sep = " ", collapse = NULL)

The **sep=** keyword denotes a separator or delimiter. The default separator is a single space. The **collapse=** keyword is used to separate the results.

4. String Formatting

Suppose the value is stored in fraction and you need to convert it to percent. The **sprintf** is used to perform C-style string formatting.

5. Extract or replace substrings

substr Syntax - substr(x, starting position, end position)

6. String Length

The **nchar** function is used to compute the length of a character value.

7. Replace the first match of the string

sub Syntax - sub(sub-string, replacement, x, ignore.case = FALSE)

8. Extract Word from a String

Suppose you need to pull a first or last word from a character string.

Word Function Syntax (Library : stringr)

word(string, position of word to extract, separator)

9. Convert Character to Uppercase / Lowercase

tolower(x)

toupper(x)

Output : "I LOVE R PROGRAMMING"

10. Repeat the character N times

strrep("x",3)

Output : "xxx"

11. Find String in a Character Variable

The str_detect() function helps to check whether a sub-string exists in a string. It is equivalent to 'contain' function of SAS. It returns TRUE/FALSE against each value.

x = c("Aon Hewitt", "Aon Risk", "Hewitt", "Google")
library(stringr)
str_detect(x,"Aon")

Output : TRUE TRUE FALSE FALSE

12. Splitting a Character Vector

In case of text mining. it is required to split a string to calculate the most frequently used keywords in the list. There is a function called 'strsplit()' in base R to perform this operation.

x = c("I love R Programming")
strsplit(x, " ")

Output : "I"   "love"   "R"   "Programming"

13. Pattern Matching

Most of the times, string manipulation becomes a daunting task as we need to match the pattern in strings. In these cases, Regex is a popular language to check the pattern. In R, it is implemented with **grepl** function.

Example -

x = c("Deepanshu", "Dave", "Sandy", "drahim", "Jades")

Keeping characters starts with the letter 'D'

x[grepl("^D",x)]

Output : "Deepanshu" "Dave"

Keeping characters do not start with the letter 'D'

x[!grepl("(?i)^d",x)]

Output : "Sandy" "Jades"

Keeping characters end with 'S'

x[grepl("s$",x)]

Output : "Jades"

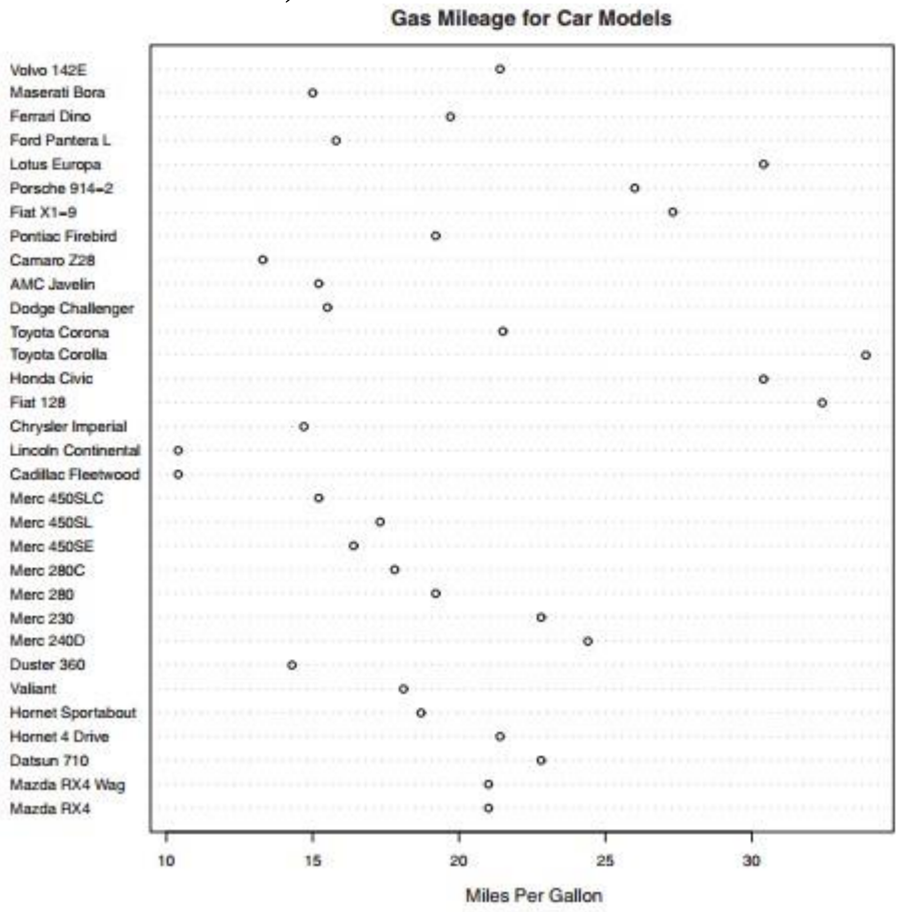
4 b) Illustrate dot plots and Kernal density plots. 6 M

ANS: Dot Plot

Dot plots provide a method of plotting a large number of labeled values on a simple horizontal scale. You create them with the dotchart() function, using the format dotchart(x,labels=)

Where x is a numeric vector and labels specifies a vector that labels each point. You can add a groups option to designate a factor specifying how the elements of x are grouped. If so, the option gcolor controls the color of the groups label, and cex controls the size of the labels. Here's an example with the mtcars dataset:

dotchart(mtcars$mpg,labels=row.names(mtcars),cex=.7,main="Gas Mileage for Car Models", xlab="Miles Per Gallon")



This graph allows you to see the mpg for each make of car on the same horizontal axis. Dot plots typically become most interesting when they're sorted and grouping factors are distinguished by symbol and color.

Kernel density plots

kernel density estimation is a nonparametric method for estimating the probability density function of a random variable. Although the mathematics are beyond the scope of this text, in general, kernel density plots can be an effective way to view the distribution of a continuous variable. The format for a density plot is

plot(density(*x*))

where *x* is a numeric vector. Because the plot() function begins a new graph, use the lines() function when superimposing a density curve on an existing graph

```
par(mfrow=c(2,1))
d <- density(mtcars$mpg)

plot(d)

d <- density(mtcars$mpg)
plot(d, main="Kernel Density of Miles Per Gallon")
polygon(d, col="red", border="blue")
rug(mtcars$mpg, col="brown")
```

Creates the minimal graph with all the defaults in place

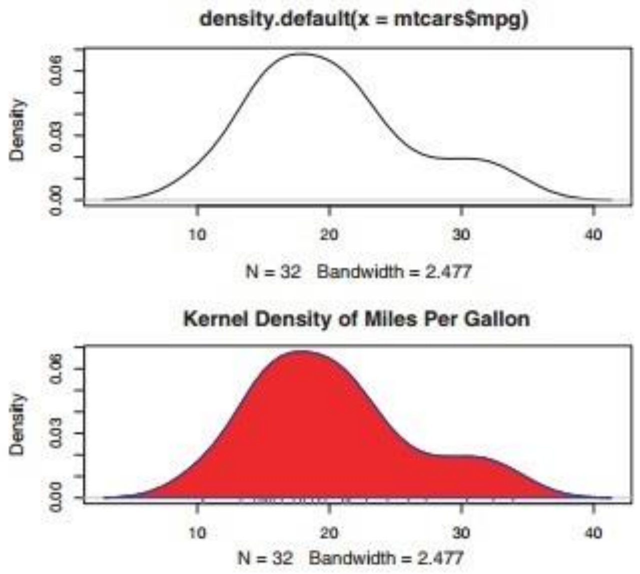
Adds a title

Adds a brown rug

Colors the curve blue and fills the area under the curve with solid red

The polygon() function draws a polygon whose vertices are given by x and y. These values are provided by the density() function in this case.

Kernel density plots can be used to compare groups. This is a highly underutilized approach, probably due to a general lack of easily accessible software. Fortunately, the sm package fills this gap nicely.



Kernel density plots

The sm.density.compare() function in the sm package allows you to superimpose the kernel density plots of two or more groups. The format is sm.density.compare(*x*, *factor*)
where *x* is a numeric vector and *factor* is a grouping variable. Be sure to install the sm package before first use.

(OR)

5 a) Discuss aggregating and reconstructing 6 M

ANS: **Aggregating data**

It's relatively easy to collapse data in R using one or more by variables and a defined function. The format is
aggregate(*x*, *by*, *FUN*)
where *x* is the data object to be collapsed, *by* is a list of variables that will be crossed to form the new observations, and *FUN* is the scalar function used to calculate summary statistics that will make up the new observation values
Example

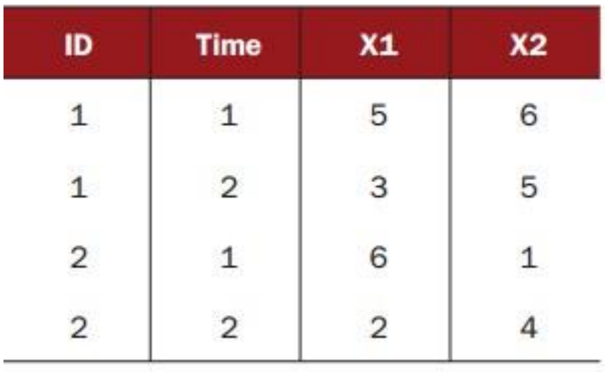
```
options(digits=3)
> attach(mtcars)
> aggdata <-aggregate(mtcars, by=list(cyl,gear), FUN=mean, na.rm=TRUE)
> aggdata
Group.1 Group.2 mpg cyl disp hp drat wt qsec vs am gear carb
1 4 3 21.5 4 120 97 3.70 2.46 20.0 1.0 0.00 3 1.00
2 6 3 19.8 6 242 108 2.92 3.34 19.8 1.0 0.00 3 1.00
3 8 3 15.1 8 358 194 3.12 4.10 17.1 0.0 0.00 3 3.08
4 4 4 26.9 4 103 76 4.11 2.38 19.6 1.0 0.75 4 1.50
5 6 4 19.8 6 164 116 3.91 3.09 17.7 0.5 0.50 4 4.00
6 4 5 28.2 4 108 102 4.10 1.83 16.8 0.5 1.00 5 2.00
7 6 5 19.7 6 145 175 3.62 2.77 15.5 0.0 1.00 5 6.00
8 8 5 15.4 8 326 300 3.88 3.37 14.6 0.0 1.00 5 6.00
```

In these results, Group.1 represents the number of cylinders (4, 6, or 8) and Group.2 represents the number of gears (3, 4, or 5).
the aggregate() function , the by variables must be in a list (even if there's only one). You can declare a custom name for the groups from within the list, for instance, using by=list(Group.cyl=cyl, Group.gears=gear).

The reshape package

The reshape package is a tremendously versatile approach to both restructuring and aggregating datasets. Because of this versatility, it can be a bit challenging to learn.
We'll go through the process slowly and use a small dataset so that it's clear what's happening. Because reshape isn't included in the standard installation of R, you'll need to install it one time, using install.packages("reshape").
Basically, you'll "melt" data so that each row is a unique ID-variable combination. Then you'll "cast" the melted data into any shape you desire. During the cast, you can aggregate the data with any function you wish.

The dataset **(mydata)**

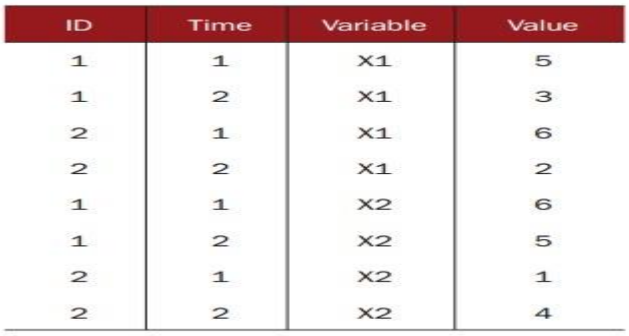


ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

In this dataset, the *measurements* are the values in the last two columns (5, 6, 3, 5, 6, 1, 2, and 4).
Each measurement is uniquely identified by a combination of ID variables
For example,
the measured value 5 in the first row is uniquely identified by knowing that it's from observation (ID) 1, at Time 1, and on variable X1

MELTING

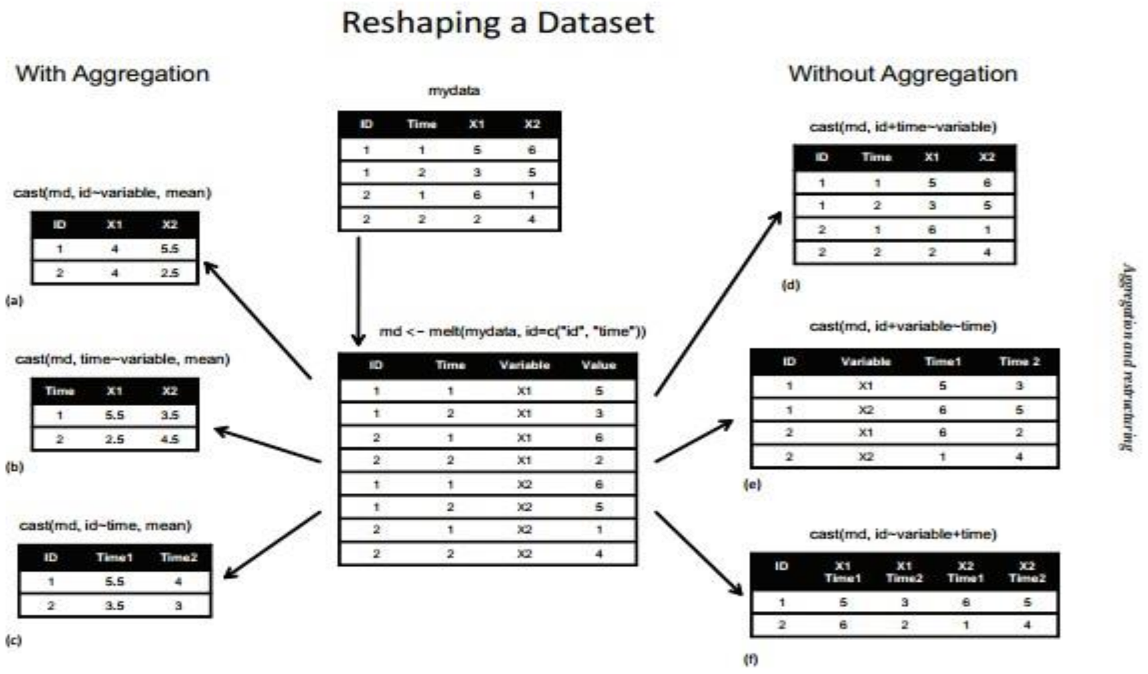
When you melt a dataset, you restructure it into a format where each measured variable is in its own row, along with the ID variables needed to uniquely identify it. If you melt the data from table 5.8, using the following code
library(reshape)
md <- melt(mydata, id=(c("id", "time")))
The melted dataset is



ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

Data is melted form, you can recast it into any shape, using the cast() function.

CASTING
The cast() function starts with melted data and reshapes it using a formula that you provide and an (optional) function used to aggregate the data. The format is
newdata <- cast(*md*, *formula*, *FUN*)
where *md* is the melted data, *formula* describes the desired end result, and *FUN* is the (optional) aggregating function. The formula takes the form
rowvar1 + rowvar2 + … ~ colvar1 + colvar2 + …
In this formula, *rowvar1 + rowvar2 + …* define the set of crossed variables that define the rows, and *colvar1 + colvar2 + …* define the set of crossed variables that define the columns.


Reshaping a Dataset
With Aggregation
Without Aggregation
mydata
md <- melt(mydata, id=c("id", "time"))
cast(md, id~variable, mean)
(a)
cast(md, time~variable, mean)
(b)
cast(md, id~time, mean)
(c)
cast(md, id+time~variable)
(d)
cast(md, id+variable~time)
(e)
cast(md, id~variable+time)
(f)
Aggregation and restructuring

mydata

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

md <- melt(mydata, id=c("id", "time"))

ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

cast(md, id~variable, mean)

ID	X1	X2
1	4	5.5
2	4	2.5

(a)

cast(md, time~variable, mean)

Time	X1	X2
1	5.5	3.5
2	2.5	4.5

(b)

cast(md, id~time, mean)

ID	Time1	Time2
1	5.5	4
2	3.5	3

(c)

cast(md, id+time~variable)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

(d)

cast(md, id+variable~time)

ID	Variable	Time1	Time 2
1	X1	5	3
1	X2	6	5
2	X1	6	2
2	X2	1	4

(e)

cast(md, id~variable+time)

ID	X1 Time1	X1 Time2	X2 Time1	X2 Time2
1	5	3	6	5
2	6	2	1	4

(f)

5 b) Exemplify the subset() function in detail. 6 M

ANS: Subset function:
The subset function is probably the easiest way to select variables and observations.
Here are two examples:
newdata <- subset(leadership, age >= 35 | age < 24,select=c(q1, q2, q3, q4))
newdata <- subset(leadership, gender=="M" & age > 25,select=gender:q4)
In the first example, you select all rows that have a value of age greater than or equal to 35 *or* age less than 24. You keep the variables q1 through q4.
In the second example,you select all men over the age of 25 and you keep variables gender through q4

UNIT III

6 a) Explain binomial distribution. 6 M
ANS:

Binomial Distribution:

The binomial distribution model deals with finding the probability of success of an event which has only two possible outcomes in a series of experiments. For example, tossing of a coin always gives a head or a tail. The probability of finding exactly 3 heads in tossing a coin repeatedly for 10 times is estimated during the binomial distribution.
R has four in-built functions to generate binomial distribution. They are described below.

```
dbinom(x, size, prob)
```

```
pbinom(x, size, prob)
qbinom(p, size, prob)
rbinom(n, size, prob)
```

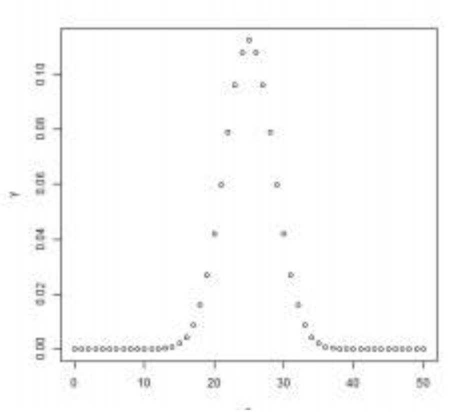
Following is the description of the parameters used −

- **x** is a vector of numbers.
- **p** is a vector of probabilities.
- **n** is number of observations.
- **size** is the number of trials.
- **prob** is the probability of success of each trial.

dbinom()

This function gives the probability density distribution at each point.

```
# Create a sample of 50 numbers which are incremented by 1.
x <-seq(0,50,by=1)

# Create the binomial distribution.
y <-dbinom(x,50,0.5)

# Give the chart file a name.
png(file ="dbinom.png")

# Plot the graph for this sample.
plot(x,y)
```



pbinom()

This function gives the cumulative probability of an event. It is a single value representing the probability.

```
# Probability of getting 26 or less heads from a 51 tosses of a coin.
x <-pbinom(26,51,0.5)

print(x)
```

When we execute the above code, it produces the following result −

```
[1] 0.610116
```

qbinom()

This function takes the probability value and gives a number whose cumulative value matches the probability value.

```
x <- qbinom(0.25,51,1/2)

print(x)

[1] 23
```

rbinom()

This function generates required number of random values of given probability from a given simple.

# Find 8 random values from a sample of 150 with probability of 0.4.

```
x <-rbinom(8,150,.4)

print(x)
```

When we execute the above code, it produces the following result −

```
[1] 58 61 59 66 55 60 61 67
```

6 b) Illustrate correlation and covariance with an example. 6 M

ANS: Correlation:-a mutual relationship or connection between two or more things.
When dealing with more than one variable, we need to test their relationships with each other. Two simple, straightforward methods are **correlation and covariance**.
To examine these concepts we look at the economics data from ggplot2.

```
>require(ggplot2)
>head(economics)
date pce pop psavertuempmedunemploy year month
1 1967-06-30 507.8 198712 9.8 4.5 2944 1967 Jun
2 1967-07-31 510.9 198911 9.8 4.7 2945 1967 Jul
3 1967-08-31 516.7 199113 9.0 4.6 2958 1967 Aug
4 1967-09-30 513.3 199311 9.8 4.9 3143 1967 Sep
5 1967-10-31 518.5 199498 9.7 4.7 3066 1967 Oct
6 1967-11-30 526.2 199657 9.4 4.8 3018 1967 Nov
```

In the economics dataset, pce is personal consumption expenditures and psavert is the personal savings rate. Wecalculate their correlation using cor.

```
>cor(economics$pce, economics$psavert)
[1] -0.9271222
```

This very low correlation makes sense because spending and saving are opposites of each other. Correlation is defined as

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y}$$

Where the means of *x* and *y*, and *sx* and *sy* are the standard deviations of *x* and *y*. It can range between -1 and 1,with higher positive numbers meaning a closer relationship between the two variables, lower negative numbers meaning an inverse relationship and numbers near zero meaning no relationship.

```
> # use cor to calculate correlation
>cor(economics$pce, economics$psavert)
```

```
[1] 0.9271222
>>
## calculate each part of correlation
>xPart<- economics$pce - mean(economics$pce)
>yPart<- economics$psavert - mean(economics$psavert)
>nMinusOne<- (nrow(economics) - 1)
>xSD<- sd(economics$pce)
>ySD<- sd(economics$psavert)
> # use correlation formula
>sum(xPart * yPart) / (nMinusOne * xSD * ySD)
[1] -0.9271222
```

To compare multiple variables at once, use cor on a matrix (only for numeric variables).

```
>cor(economics[, c(2, 4:6)])
pcepsavertuempmedunemploy
pce 1.0000000 -0.92712221 0.5145862 0.32441514
psavert -0.9271222 1.00000000 -0.3615301 -0.07641651
uempmed 0.5145862 -0.36153012 1.0000000 0.78427918
unemploy 0.3244151 -0.07641651 0.7842792 1.00000000
```

covariance, which is like a variance between variables

$$cov(X, Y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})$$

The cov function works similarly to the cor function, with the same arguments for dealing with missing data. In fact, ?cor and ?cov pull up the same help menu.

```
>cov(economics$pce, economics$psavert)
[1] -8412.231
>cov(economics[, c(2, 4:6)])
pcepsavertuempmedunemploy
pce 6810308.380 -8412.230823 2202.786256 1573882.2016
psavert -8412.231 12.088756 -2.061893 -493.9304
uempmed 2202.786 -2.061893 2.690678 2391.6039
unemploy 1573882.202 -493.930390 2391.603889 3456013.5176
> # check that cov and cor*sd*sd are the same
>identical(cov(economics$pce, economics$psavert),
+ cor(economics$pce, economics$psavert) *
+ sd(economics$pce) * sd(economics$psavert))
[1] TRUE
```

(OR)

7 a) Demonstrate paste and sprintf with example. 6 M

ANS: **paste**

The first function new R users reach for when putting together strings is paste. This function takes a series of strings, or expressions that evaluate to strings, and puts them together into one string. We start by putting together three simple strings.

```
> paste("Hello", "Jared", "and others")
```

[1] "Hello Jared and others" Notice that spaces were put between the strings. This is because paste has a third argument, sep, that determines what to put in between entries. This can be any valid text, including empty text ("").

```
> paste("Hello", "Jared", "and others", sep = "/")
```

[1] "Hello/Jared/and others" Like many functions in R, paste is vectorized. This means each element can be a vector of data to be put together.

> **paste**(**c**("Hello", "Hey", "Howdy"), **c**("Jared", "Bob", "David"))
[1] "Hello Jared" "Hey Bob" "Howdy David" In this case each vector had the same number of entries so they paired one-to-one. When the vectors do not have the same length they are recycled.

> **paste**("Hello", **c**("Jared", "Bob", "David"))
[1] "Hello Jared" "Hello Bob" "Hello David"
> **paste**("Hello", **c**("Jared", "Bob", "David"), **c**("Goodbye", "Seeya"))
[1] "Hello Jared Goodbye" "Hello Bob Seeya" "Hello David Goodbye" paste also has the ability to collapse a vector of text into one vector containing all the elements with any arbitrary separator, using the collapse argument.
> vectorOfText <- **c**("Hello", "Everyone", "out there", ".")
> **paste**(vectorOfText, collapse = " ")
[1] "Hello Everyone out there ." Chapter 13. Manipulating Strings Page 2 of 13
> **paste**(vectorOfText, collapse = "*")
[1] "Hello*Everyone*out there*."

Sprintf

While paste is convenient for putting together short bits of text, it can become unwieldy when piecing together long pieces of text, such as when inserting a number of variables into a long piece of text. For instance, we might have a lengthy sentence that has a few spots that require the insertion of special variables. An example is “Hello Jared, your party of eight will be seated in 25 minutes” where “Jared,” “eight” and “25” could be replaced with other information. Reforming this with paste can make reading the line in code difficult. To start, we make some variables to hold the information.
> person <- "Jared"
> partySize <- "eight"
> waitTime <- 25 Now we build the paste expression.

> **paste**("Hello ", person, ", your party of ", partySize,
+ " will be seated in ", waitTime, " minutes.", sep="")
[1] "Hello Jared, your party of eight will be seated in 25 minutes." Making even a small change to this sentence would require putting the commas in just the right places. A good alternative is the sprintf function. With this function we build one long string with special markers indicating where to insert values.

> **sprintf**("Hello %s, your party of %s will be seated in %s minutes",
+ person, partySize, waitTime)
[1] "Hello Jared, your party of eight will be seated in 25 minutes" Here, each %s was replaced with its corresponding variable. While the long sentence is easier to read in code, we must maintain the order of %s’s and variables. sprintf is also vectorized. Note that the vector lengths must be multiples of each other.

> **sprintf**("Hello %s, your party of %s will be seated in %s minutes",
+ **c**("Jared", "Bob"), **c**("eight", 16, "four", 10), waitTime)
[1] "Hello Jared, your party of eight will be seated in 25 minutes"
[2] "Hello Bob, your party of 16 will be seated in 25 minutes"
[3] "Hello Jared, your party of four will be seated in 25 minutes"
[4] "Hello Bob, your party of 10 will be seated in 25 minutes"

7 b) Discuss linear regression in detail. 6 M

ANS: **Linear Regression** analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called predictor variable whose value is gathered through experiments.

The general mathematical equation for a linear regression is −

```
y = ax + b
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are constants which are called the coefficients.

Steps to Establish a Regression

A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height and corresponding weight.
- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation using these
- Get a summary of the relationship model to know the average error in prediction. Also called **residuals**.
- To predict the weight of new persons, use the **predict()** function in R.

Input Data

Below is the sample data representing the observations −

```
# Values of height
151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight.
63, 81, 56, 91, 47, 57, 76, 72, 62, 48
```

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in linear regression is −

```
lm(formula,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create Relationship Model & get the Coefficients

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)



# Apply the lm() function.

relation <- lm(y~x)



print(relation)
```

When we execute the above code, it produces the following result −

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
   -38.4551       0.6746
```

Get the Summary of the Relationship

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)



# Apply the lm() function.

relation <- lm(y~x)



print(summary(relation))
```

When we execute the above code, it produces the following result −

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q     Median      3Q     Max
-6.3002    -1.6629  0.0412    1.8944  3.9775

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -38.45509    8.04901  -4.778  0.00139 **
x             0.67461    0.05191  12.997 1.16e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.253 on 8 degrees of freedom
Multiple R-squared:  0.9548,    Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06
```

predict() Function

Syntax

The basic syntax for predict() in linear regression is −

```
predict(object, newdata)
```

Following is the description of the parameters used −

- **object** is the formula which is already created using the lm() function.
- **newdata** is the vector containing the new value for predictor variable.

Predict the weight of new persons

```
# The predictor vector.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

# The resposne vector.

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

# Find weight of a person with height 170.

a <- data.frame(x = 170)

result <-  predict(relation,a)

print(result)
```

When we execute the above code, it produces the following result −

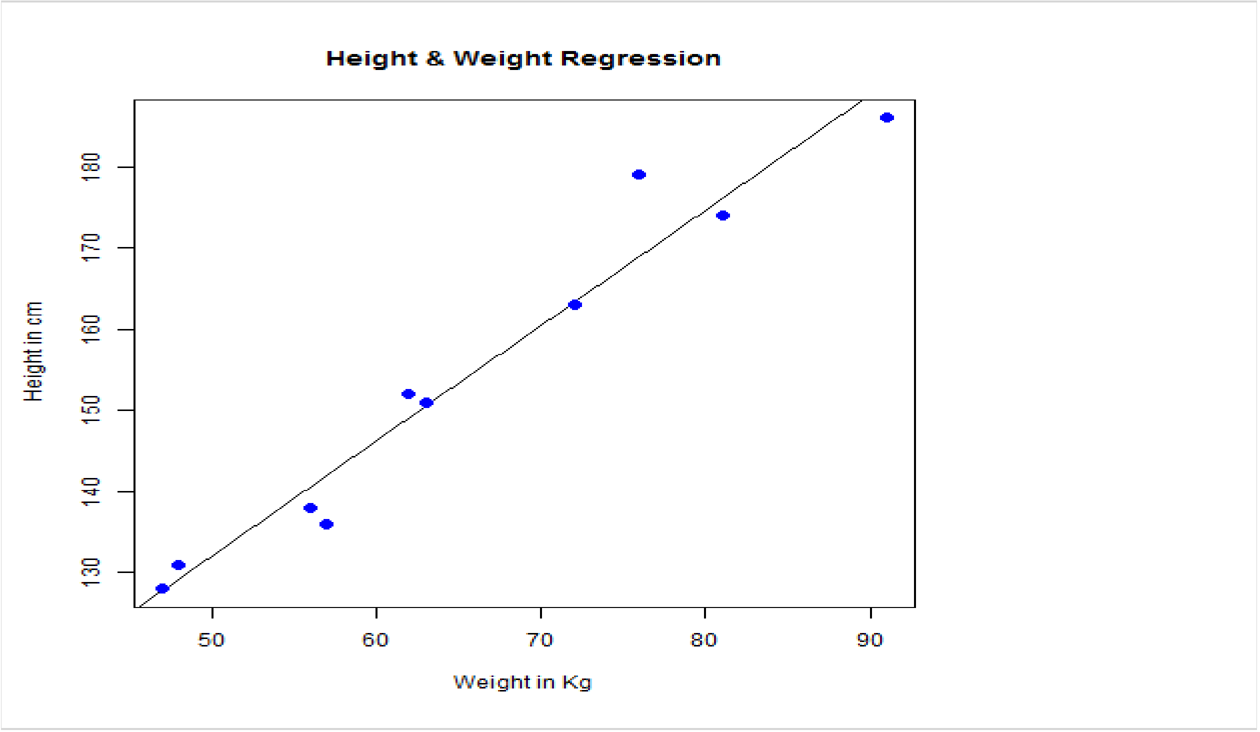
```
       1
76.22869
```

Visualize the Regression Graphically

```
# Create the predictor and response variable.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
```

```
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
relation <- lm(y~x)


# Give the chart file a name.
png(file = "linearregression.png")


# Plot the chart.
plot(y,x,col = "blue",main = "Height & Weight Regression",
abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")
```

it produces the following result −



UNIT IV

8 a) Demonstrate partitioning cluster analysis. 6 M

ANS: *Partitioning cluster analysis*

In the partitioning approach, observations are divided into *K* groups and reshuffled to form the most cohesive clusters possible according to a given criterion.

two methods: k-means and partitioning around medoids (PAM).

K-means clustering

The most common partitioning method is the k-means cluster analysis. Conceptually, the k-means algorithm is as follows:

1 Select *K* centroids (*K* rows chosen at random).
2 Assign each data point to its closest centroid.
3 Recalculate the centroids as the average of all data points in a cluster (that is, the centroids are *p*-length mean vectors, where *p* is the number of variables).

4 Assign data points to their closest centroids.
5 Continue steps 3 and 4 until the observations aren't reassigned or the maximum number of iterations (R uses 10 as a default) is reached.
R uses an efficient algorithm by Hartigan and Wong (1979) that partitions the observations into *k* groups such that the sum of squares of the observations to their assigned cluster centers is a minimum. This means, in steps 2 and 4, each observation is assigned to the cluster with the smallest value of

$$ss(k) = \sum_{i=1}^{n}\sum_{j=0}^{p}(x_{ij} - \bar{x}_{kj})^2$$

where *k* is the cluster, *xij* is the value of the *j* th variable for the *i* th observation, *x* ¯*kj* is the mean of the *j* th variable for the *k* th cluster, and *p* is the number of variables.
K-means clustering can handle larger datasets than hierarchical cluster approaches.
The format of the k-means function in R is kmeans(*x*, *centers*),
where *x* is abnumeric dataset (matrix or data frame) and *centers* is the number of clusters tobextract. The function returns the cluster memberships, centroids, sums of squares (within, between, total), and cluster sizes.
Partitioning around medoids
Because it's based on means, the k-means clustering approach can be sensitive to outliers. A more robust solution is provided by partitioning around medoids (PAM).
Rather than representing each cluster using a centroid (a vector of variable means), each cluster is identified by its most representative observation (called a *medoid*).
Whereas k-means uses Euclidean distances, PAM can be based on any distance measure. It can therefore accommodate mixed data types and isn't limited to continuous variables.
The PAM algorithm is as follows:
1 Randomly select *K* observations (call each a medoid).
2 Calculate the distance/dissimilarity of every observation to each medoid.
3 Assign each observation to its closest medoid.
4 Calculate the sum of the distances of each observation from its medoid (total cost).
5 Select a point that isn't a medoid, and swap it with its medoid.
6 Reassign every point to its closest medoid.
7 Calculate the total cost.
8 If this total cost is smaller, keep the new point as a medoid.
9 Repeat steps 5–8 until the medoids don't change
You can use the pam() function in the cluster package to partition around medoids.
The format is pam(*x*, *k*, metric="euclidean", stand=FALSE)
where *x* is a data matrix or data frame, *k* is the number of clusters, metric is the type of distance/dissimilarity measure to use, and stand is a logical value indicating whether the variables should be standardized before calculating this metric

8 b) Illustrate the methods involved in preparing the data. 6 M

ANS: The Wisconsin Breast Cancer dataset is available as a comma-delimited text file. The dataset contains
699 fine-needle aspirate samples, where 458 (65.5%) are benign and 241 (34.5%) are malignant. The dataset contains a total of 11 variables and doesn't include the variable names in the file. Sixteen samples have missing data and are coded in the text file with a question mark (?).
The variables are as follows:
ID
- Clump thickness
- Uniformity of cell size
- Uniformity of cell shape
- Marginal adhesion

Single epithelial cell size
■ Bare nuclei
■ Bland chromatin
■ Normal nucleoli
■ Mitoses
■ Class

The first variable is an ID variable (which you'll drop), and the last variable (class) contains the outcome (coded 2=benign, 4=malignant).

For each sample, nine cytological characteristics previously found to correlate with malignancy are also recorded. These variables are each scored from 1 (closest to benign) to 10 (most anaplastic). But no one predictor alone can distinguish between benign and malignant samples. The challenge is to find a set of classification rules that can be used to accurately predict malignancy from some combination of these nine cell characteristics.

```
breast <- read.table(url, sep=",", header=FALSE, na.strings="?")
names(breast) <- c("ID",
"clumpThickness","sizeUniformity","shapeUniformity","maginalAdhesion",
"singleEpithelialCellSize", "bareNuclei","blandChromatin", "normalNucleoli", "mitosis",
"class")
df <- breast[-1]
df$class <- factor(df$class, levels=c(2,4),
labels=c("benign", "malignant"))
set.seed(1234)
train <- sample(nrow(df), 0.7*nrow(df))
df.train <- df[train,]
df.validate <- df[-train,]
table(df.train$class)
table(df.validate$class)
```

The training sample has 499 cases (329 benign, 160 malignant), and the validation sample has 210 cases (129 benign, 81 malignant).

The training sample will be used to create classification schemes using logistic regression, a decision tree, a conditional decision tree, a random forest, and a support vector machine

The validation sample will be used to evaluate the effectiveness of these schemes.

(OR)

9 a) Explain decision tree approach for classification 6 M

ANS:

Decision tree is a graph to represent choices and their results in form of a tree. The nodes in the graph represent an event or choice and the edges of the graph represent the decision rules or conditions. It is mostly used in Machine Learning and Data Mining applications using R.

The algorithm is as follows:

1 Choose the predictor variable that best splits the data into two groups such that the purity (homogeneity) of the outcome in the two groups is maximized (that is, as many benign cases in one group and malignant cases in the other as possible). If the predictor is continuous, choose a cut-point that maximizes purity for the two groups created. If the predictor variable is categorical (not applicable in this case), combine the categories to obtain two groups with maximum purity.

2 Separate the data into these two groups, and continue the process for each subgroup.

3 Repeat steps 1 and 2 until a subgroup contains less than a minimum number of observations or no splits decrease the impurity beyond a specified threshold.

The subgroups in the final set are called *terminal nodes*. Each terminal node is classified as one category of the outcome or the other based on the most frequent value of the outcome for the sample in that node.
4 To classify a case, run it down the tree to a terminal node, and assign it the
modal outcome value assigned in step 3.

The R package **"party"** is used to create decision trees.

Install R Package

Use the below command in R console to install the package. You also have to install the dependent packages if any.

```
install.packages("party")
```

The package "party" has the function **ctree()** which is used to create and analyze decison tree.

Syntax

The basic syntax for creating a decision tree in R is −

```
ctree(formula, data)
```

Following is the description of the parameters used −

- **formula** is a formula describing the predictor and response variables.
- **data** is the name of the data set used.

Input Data

We will use the R in-built data set named **readingSkills** to create a decision tree. It describes the score of someone's readingSkills if we know the variables "age","shoesize","score" and whether the person is a native speaker or not.

Here is the sample data.

```
# Load the party package. It will automatically load other dependent packages.
library(party)
# Print some records from data set readingSkills.
print(head(readingSkills))
```

When we execute the above code, it produces the following result and chart −

```
nativeSpeaker   age   shoeSize      score
1           yes     5   24.83189   32.29385
2           yes     6   25.95238   36.63105
3            no    11   30.42170   49.60593
4           yes     7   28.66450   40.28456
```

```
5          yes    11   31.88207   55.46085
6          yes    10   30.07843   52.83124
Loading required package: methods
Loading required package: grid
...............................
...............................
```

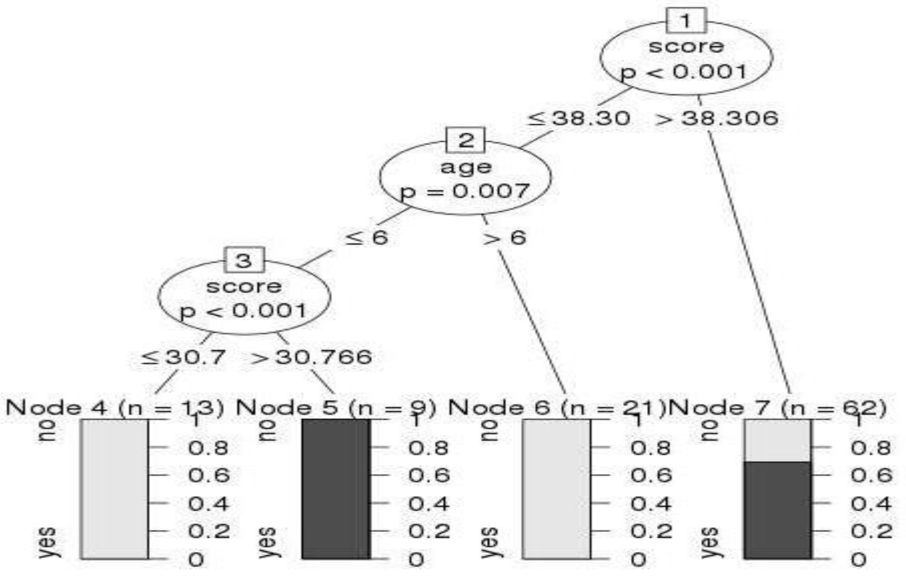
Example

We will use the **ctree()** function to create the decision tree and see its graph.

```
# Load the party package. It will automatically load other dependent packages.

library(party)

# Create the input data frame.

input.dat <- readingSkills[c(1:105),]

# Give the chart file a name.

png(file = "decision_tree.png")

# Create the tree.

output.tree<- ctree(

nativeSpeaker ~ age + shoeSize + score,   data = input.dat)

# Plot the tree.

plot(output.tree)
```

When we execute the above code, it produces the following result −

```
null device
        1
Loading required package: methods
Loading required package: grid
Loading required package: mvtnorm
Loading required package: modeltools
Loading required package: stats4
Loading required package: strucchange
Loading required package: zoo

Attaching package: ‘zoo’

The following objects are masked from ‘package:base’:

as.Date, as.Date.numeric

Loading required package: sandwich
```



9 b) Discuss SVM in detail. 6 M

ANS: *Support vector machines* (SVMs) are a group of supervised machine-learning models that can be used for classification and regression. They're popular at present, in part because of their success in developing accurate prediction models, and in part because of the elegant mathematics that underlie the approach

SVMs seek an optimal hyperplane for separating two classes in a multidimensional space.

The hyperplane is chosen to maximize the *margin* between the two classes' closest points. The points on the boundary of the margin are called *support vectors* (they help define the margin), and the middle of the margin is the separating hyperplane.

For an *N*-dimensional space (that is, with *N* predictor variables), the optimal hyperplane (also called a *linear decision surface*) has *N* – 1 dimensions. If there are two variables, the surface is a line. For three variables, the surface is a plane. For 10 variables, the surface is a 9-dimensional hyperplane. Trying to picture it will give you headache.

Consider the two-dimensional example. Circles and triangles represent the two groups. The *margin* is the gap, represented by the distance between the two dashed lines. The points on the dashed lines (filled circles and triangles) are the support vectors. In the two-dimensional case, the optimal hyperplane is the black line in the middle of the gap. In this idealized example, the two groups are linearly separable—the line can completely separate the two groups without errors.

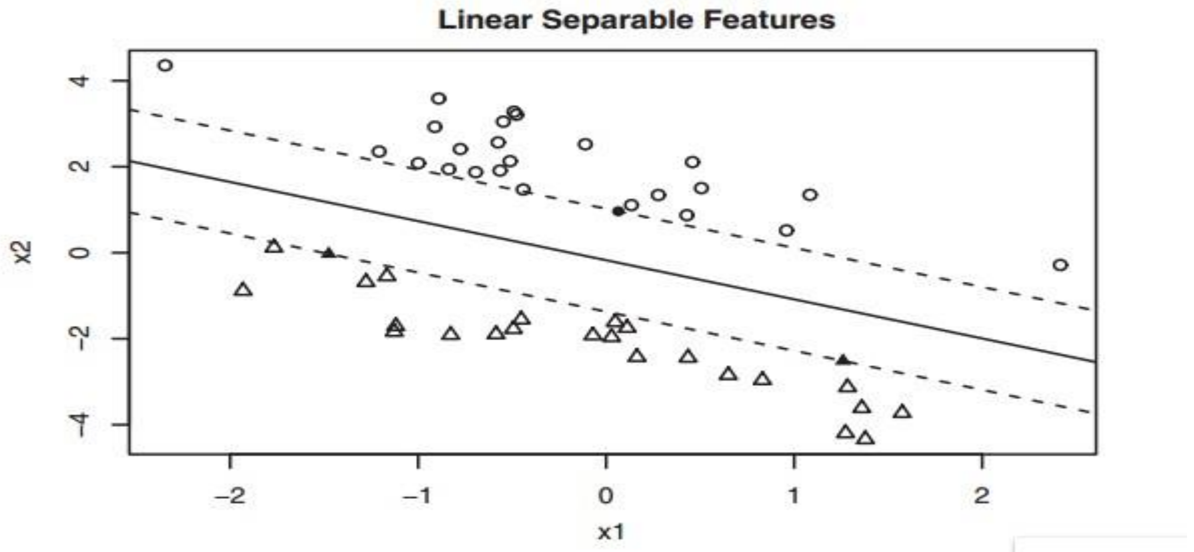


Fig: Two-group classification problem

The optimal hyperplane is identified using quadratic programming to optimize the margin under the constraint that the data points on one side have an outcome value of +1 and the data on the other side has an outcome value of -1. If the data points are “almost” separable (not all the points are on one side or the other), a penalizing term is added to the optimization in order to account for errors, and “soft” margins are produced.\