UNIT-1: STATISTICAL PROGRAMMING WITH R

Topics: Introduction, How to run R, R Sessions and Functions, Basic Math, Variables, Data Types, Vectors, Conclusion, Advanced Data Structures, Data Frames, Lists, Matrices, Arrays, Classes.

1 Introduction:

R is an interpreted programming language (hence also called as scripting language) for statistical data manipulation and analysis, Graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R development Core Team.

R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

This programming language was named **R**, based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs Language **S**.

1.1 Features of R:

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R −

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.
- R is free, open source, powerful and highly extensible.

As a conclusion, R is world's most widely used statistics programming language. It's the # 1 choice of data scientists and supported by a vibrant and talented community of contributors.

Local Environment Setup:

If you are still willing to set up your environment for R, you can follow the steps given below.

Windows Installation

You can download the Windows installer version of R from R-3.2.2 for Windows (32/64 bit) and save it in a local directory.

As it is a Windows installer (.exe) with a name "R-version-win.exe". You can just double click and run the installer accepting the default settings. If your Windows is 32-bit version, it installs the 32-bit version. But if your windows is 64-bit, then it installs both the 32-bit and 64-bit versions.

After installation you can locate the icon to run the Program in a directory structure "R\R3.2.2\bin\i386\Rgui.exe" under the Windows Program Files. Clicking this icon brings up the R-GUI which is the R console to do R Programming.

Linux Installation

R is available as a binary for many versions of Linux at the location R Binaries.

The instruction to install Linux varies from flavor to flavor. These steps are mentioned under each type of Linux version in the mentioned link. However, if you are in a hurry, then you can use yum command to install R as follows −

1.2 How to run R:

R programming runs in two modes:

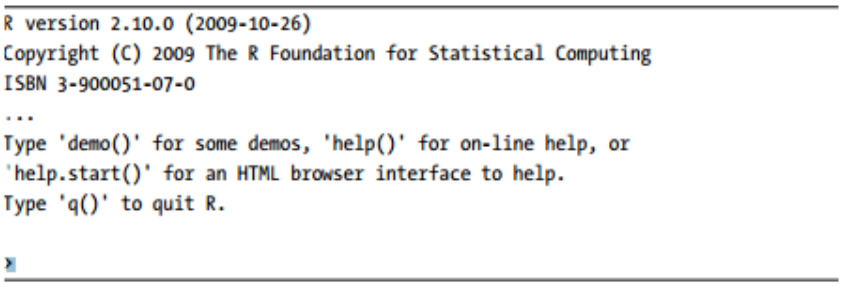
1. Interactive mode
2. Batch mode

The typically used one is Interactive Mode. In this mode we type in commands, R displays results, type in more commands and so on. On the other hand batch mode does not require interaction with the user. It's useful for production jobs, such as when a program must be run periodically, i.e once per day, we can automate the process.

Running R in Interactive Mode:

Start an R session by typing "R" on the command line in Linux or on a Mac. On a windows machine, start R by clicking the R icon.

This result in greeting and the R prompt, which is the > sign. The screen will be as follows.



Then execute R commands. The window in which all this appears is called is called the R console.

Example, consider a standard normal distribution—that is, with mean 0 and variance 1. If a random variable X has that distribution, then its values are centered around 0, some negative, some positive, averaging in the end to 0. Now form a new random variable Y = |X|. Since we've taken the absolute value, the values of Y will not be centered around 0, and the mean of Y will be positive.

Let's find the mean of Y. Our approach is based on a simulated example of N (0,1) variates.

```
> mean(abs(rnorm(100)))
```

```
[1] 0.7194236
```

This code generates the 100 random variates, finds their absolute values, and then finds the mean of the absolute values.

The [1] you see means that the first item in this line of output is item 1. In this case, our output consists of only one line (and one item).

For example, if there were two rows of output with six items per row, the second row would be labeled [7].

```
> rnorm(10)
```

```
[1] -0.6427784  -1.0416696  -1.4020476  -0.6718250  -0.9590894  -0.8684650

[7] -0.5974668   0.6877001  1.3577618  -2.2794378
```

We can store R commands in a file. By convention, R code files have the suffix .R or .r. If you create a code file called z.R, you can execute the contents of that file by issuing the following command:

```
> source("z.R")
```

Running R in Batch Mode:

Sometimes it's preferable to automate the process of running R. For example, you may wish to run an R script that generates a graph without needing to bother with manually launching R and executing the script yourself. Here you would run R in batch mode. Consider the file z.r, which produces the histogram and saves it to a PDF file:

```
Pdf("xh.pdf")  # set graphical output file

hist(rnorm(100)) # generate 100 N(0,1) variates and plot their histogram

dev.off( )  # close the graphical output file
```

The information in the comments are marked with #.

We could run it automatically by typing

```
R CMD BATCH –vanilla < z.r
```

The –vanilla option tells R not to load any startup file information, and not to save any.

1.3 R Sessions and Functions:

We start R from our shell command line, and get the greeting message and the > prompt:

R : Copyright 2005, The R Foundation for Statistical Computing Version 2.1.1 (2005-06-20), ISBN 3-900051-07-0 ... Type 'q()' to quit R.

>

Now let's make a simple data set, a vector in R parlance, consisting of the numbers 1, 2 and 4, and name it x:

```
> x <- c(1,2,4)
```

The standard assignment operator in R is <-. However, there are also ->, = and even the assign()function. The “c” stands for “concatenate.” Here we are concatenating the numbers 1, 2 and 4. Or more precisely, we are concatenating three one-element vectors consisting of those numbers. This is because any object is considered a one-element vector. Thus we can also do, for instance,

```
> q <- c(x,x,8)
```

Which sets q to (1,2,4,1,2,4,8).

For example:

To print the vector on the screen, simply type its name.

```
> x
```

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

X consists of the numbers 1, 2 and 4.

The [1] here means in this row of output, the first item is item 1 of that output. If two rows of output with six items per row, the second row would be labeled [7].

In interactive mode, one can always print an object in R by simply typing its name. For example, print the third element of x.

```
> x[3]
```

```
[1] 4
```

The selector here is 3 is called the index or subscript. The elements of R vectors are indexed starting from 1 not from 0. Subsetting is a very important operation on vectors.

Here’s an example:

```
> x <- c(1,2,4)
```

```
> x[2:3]
```

```
[1] 2 4
```

The expression x[2:3] refers to the subvector of x consisting of elements 2 through 3, which are 2 and 4 here

Example: Find the mean and standard deviation for the above given data.

```
> mean(x)

[1] 2.333333

> sd(x)

[1] 1.527525
```

R Functions:

The heart of R programming consists of writing functions. A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result. 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.

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.

R has many in-built functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as user defined functions.

Built-in Function

Simple examples of in-built functions are seq(), mean(), max(), sum(x) and paste(...) etc. They are directly called by user written programs.

1. Create a sequence of numbers from 32 to 44.

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

[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
```

2. Find mean of numbers from 25 to 82.

```
> print(mean(25:82))

[1] 53.5
```

3. Find sum of numbers from 41 to 68.

```
> print(sum(41:68))

[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

Calling a Function

```
# counts the number of odd integers in x
> oddcount <- function(x)  {
+    k <- 0  # assign 0 to k
+    for (n in x)  {
+       if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
+    }
+    return(k)
+ }
> oddcount(c(1,3,5))
[1] 3
> oddcount(c(1,2,3,7,9))
[1] 4
```

First, we told R that we wanted to define a function named oddcount with one argument, x. The left brace demarcates the start of the body of the function. We wrote one R statement per line. Until the body of the function is finished, R reminds you that you're still in the definition by using + as its prompt, instead of the usual >.

Calling a Function without an Argument

Create a function without an argument.

> new.function <- function() {

+ for(i in 1:5) {

+ print(i^2)

+ }

+ }

> new.function() # Call the function without supplying an argument.

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

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

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 <- fun

ction(a,b,c) {

 + result <- a * b + c

  + print(result)

  + }

>  new.function(5,3,11)     # Call the function by position of arguments.

> new.function(a = 11, b = 5, c = 3)      # Call the function by names of the arguments.
```

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

[1] 26

[1] 58

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.

> new.function <- function(a = 3, b = 6) {

 + result <- a * b
```

+ print(result)

+ }

> new.function() # Call the function without giving any argument.

> new.function(9,5) # Call the function with giving new values of the argument

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

[1] 18

[1] 45

1.4 Basic Math:

We begin with “Hello, World!” of basic Math: 1+1. In the console there is a right angle bracket (>) where code should be entered. Simply test R by running

> 1+1

[1] 2

Some more complicated expressions:

> 1+2+3

[1] 6

> 3*7*2

[1] 42

> 4/3

[1] 1.333

These follow the basic order of operations: Parenthesis, Exponents, Multiplication, Division, Addition and Subtraction. This means operations inside parenthesis take priority over operations. Next priority list is exponentiation. After that multiplication and division are performed, followed by addition and subtraction.

This is why the first two lines in the following code have the same result while the third is different.

```
>  4*6+5

[1] 29

>  (4*6)+5

[1] 29

> 4*(6+5)

[1] 44
```

We have put white space in between each operator such as * and / but it is not necessary.

1.5 Variables:

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many Robjects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name , var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.

_var_name	invalid	Starts with _ which is not valid

Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using print() or cat()function. The cat() function combines multiple items into a continuous print output.

1. Assignment using equal operator.

```
var.1 = c(0,1,2,3)
```

2. Assignment using leftward operator.

```
var.2 <- c("learn","R")
```

3. Assignment using rightward operator.

```
c(TRUE,1) -> var.3

print(var.1)

cat ("var.1 is ", var.1 ,"\n")

cat ("var.2 is ", var.2 ,"\n")

cat ("var.3 is ", var.3 ,"\n")
```

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

```
[1] 0 1 2 3

var.1 is  0 1 2 3

var.2 is  learn R

var.3 is  1 1
```

Note − The vector c(TRUE,1) has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x <- "Hello"

cat("The class of var_x is ",class(var_x),"\n")

var_x <- 34.5

cat("  Now the class of var_x is ",class(var_x),"\n")

var_x <- 27L

cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

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

```
The class of var_x is  character

Now the class of var_x is  numeric

Next the class of var_x becomes  integer
```

Finding Variables

To know all the variables currently available in the workspace we use the ls() function. Also the ls() function can use patterns to match the variable names.

```
print(ls())
```

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

```
[1] "my var"     "my_new_var" "my_var"     "var.1"

[5] "var.2"      "var.3"      "var.name"   "var_name2."

[9] "var_x"      "varname"
```

Note − It is a sample output depending on what variables are declared in your environment.

The ls() function can use patterns to match the variable names.

```
# List the variables starting with the pattern "var".

print(ls(pattern = "var"))
```

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

```
[1] "my var"     "my_new_var" "my_var"     "var.1"

[5] "var.2"      "var.3"      "var.name"   "var_name2."

[9] "var_x"      "varname"
```

The variables starting with dot(.) are hidden, they can be listed using "all.names = TRUE" argument to ls() function.

```
print(ls(all.name = TRUE))
```

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

```
[1] ".cars"       ".Random.seed" ".var_name"    ".varname"     ".varname2"

[6] "my var"      "my_new_var"   "my_var"      "var.1"        "var.2"

[11]"var.3"       "var.name"     "var_name2."   "var_x"
```

Deleting Variables

Variables can be deleted by using the rm() function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

```
rm(var.3)

print(var.3)
```

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

```
[1] "var.3"

Error in print(var.3) : object 'var.3' not found
```

All the variables can be deleted by using the rm() and ls() function together.

```
rm(list = ls())

print(ls())
```

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

```
character(0)
```

1.6 R- Data Types

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.

In contrast to other programming languages like C and java in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are −

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

The simplest of these objects is the vector object and there are six data types of these atomic vectors, also termed as six classes of vectors.

Data Type	Example	Verify
Logical	TRUE, FALSE	v <- TRUE

		print(class(v)) it produces the following result − [1] "logical"
Numeric	12.3, 5, 999	v <- 23.5 print(class(v)) it produces the following result − [1] "numeric"
Integer	2L, 34L, 0L	v <- 2L print(class(v)) it produces the following result − [1] "integer"
Complex	3 + 2i	v <- 2+5i print(class(v)) it produces the following result − [1] "complex"
Character	'a' , '"good", "TRUE", '23.4'	v <- "TRUE" print(class(v)) it produces the following result − [1] "character"

Raw	"Hello" is stored as 48 65 6c 6c 6f	v <- charToRaw("Hello") print(class(v)) it produces the following result − [1] "raw"

In R programming, the very basic data types are the R-objects called vectors which hold elements of different classes as shown above.

1.7 Vectors

When you want to create vector with more than one element, you should use c() function which means to combine the elements into a vector.

```
# Create a vector.

apple <- c('red','green',"yellow")

print(apple)

# Get the class of the vector.

print(class(apple))
```

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

```
[1] "red"    "green"  "yellow"

[1] "character"
```

Conclusion:

Data comes in many types, and R is well equipped to handle them. In addition to basic calculations, R can handle numeric, character and time based data. One important thing of working with R requires vectorization. This allows operating on multiple elements in a vector simultaneously, which leads to faster and more mathematical code.

1.8 Advanced Data Structures:

Sometimes data requires more complex storage than simple vectors. R provides a host of data structures. The most common are data. frame, matrix and list followed by array.

Data Frames:

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the data.frame() function.

```
# Create the data frame.

BMI <-          data.frame(

gender = c("Male", "Male","Female"),

height = c(152, 171.5, 165),

weight = c(81,93, 78),

Age = c(42,38,26)

)

print(BMI)
```

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

```
 Gender height weight Age

1   Male  152.0     81  42

2   Male  171.5     93  38

3   Female  165.0     78  26
```

Lists: A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.

list1 <- list(c(2,5,3),21.3,sin)

# Print the list.

print(list1)
```

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

```
[[1]]

[1] 2 5 3

[[2]]

[1] 21.3

[[3]]

function (x)  .Primitive("sin")
```

Matrices

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types.. We use matrices containing numeric elements to be used in mathematical calculations.

A Matrix is created using the **matrix()** function.

Syntax

The basic syntax for creating a matrix in R is −

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Following is the description of the parameters used −

- **data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.

- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
- **dimname** is the names assigned to the rows and columns.

Example

Create a matrix taking a vector of numbers as input

```
# Elements are arranged sequentially by row.
M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
print(M)
# Elements are arranged sequentially by column.
N <- matrix(c(3:14), nrow = 4, byrow = FALSE)
print(N)
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))
print(P)
```

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

```
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
```

```
     [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
     col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14
```

Accessing Elements of a Matrix

Elements of a matrix can be accessed by using the column and row index of the element. We consider the matrix P above to find the specific elements below.

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")
# Create the matrix.
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))
# Access the element at 3rd column and 1st row.
print(P[1,3])
# Access the element at 2nd column and 4th row.
print(P[4,2])
```

 # Access only the 2nd row.

```
print(P[2,])
```

 # Access only the 3rd column.

```
print(P[,3])
```

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

```
[1] 5

[1] 13

col1 col2 col3

  6    7    8

row1 row2 row3 row4

  5    8   11   14
```

Matrix Computations

The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

Matrix Addition & Subtraction

 # Create two 2x3 matrices.

```
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)

print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)

print(matrix2)
```

 # Add the matrices.

```
result <- matrix1 + matrix2
```

```
cat("Result of addition","\n")

print(result)

# Subtract the matrices

result <- matrix1 - matrix2

cat("Result of subtraction","\n")

print(result)
```

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

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

[1,]    3   -1    2

[2,]    9    4    6

     [,1] [,2] [,3]

[1,]    5    0    3

[2,]    2    9    4

Result of addition

     [,1] [,2] [,3]

[1,]    8   -1    5

[2,]   11   13   10

Result of subtraction

     [,1] [,2] [,3]

[1,]   -2   -1   -1

[2,]    7   -5    2
```

Matrix Multiplication & Division

 # Create two 2x3 matrices.

```
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)

print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)

print(matrix2)
```

 # Multiply the matrices.

```
result <- matrix1 * matrix2

cat("Result of multiplication","\n")

print(result)
```

 # Divide the matrices

```
result <- matrix1 / matrix2

cat("Result of division","\n")

print(result)
```

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

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of multiplication
     [,1] [,2] [,3]
```

[1,] 15 0 6

[2,] 18 36 24

Result of division

[,1] [,2] [,3]

[1,] 0.6 -Inf 0.6666667

[2,] 4.5 0.4444444 1.5000000

Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.

a <- array(c('green','yellow'),dim = c(3,3,2))

print(a)
```

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

```
, , 1

     [,1]     [,2]     [,3]

[1,] "green"  "yellow" "green"

[2,] "yellow" "green"  "yellow"

[3,] "green"  "yellow" "green"

, , 2

     [,1]     [,2]     [,3]

[1,] "yellow" "green"  "yellow"
```

[2,] "green" "yellow" "green"

[3,] "yellow" "green" "yellow"

R Objects and Classes:

An object is a data structure having some attributes and methods which act on its attributes.

Class is a blueprint for the object. We can think of class like a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. While most programming languages have a single class system, R has three class systems. Namely, S3, S4 and more recently Reference class systems.

S3 Class

S3 class is somewhat primitive in nature. It lacks a formal definition and object of this class can be created simply by adding a class attribute to it.

This simplicity accounts for the fact that it is widely used in R programming language. In fact most of the R built-in classes are of this type.

Example 1: S3 class

> # create a list with required components

> s <- list(name = "John", age = 21, GPA = 3.5)

> # name the class appropriately

> class(s) <- "student"

S4 Class

S4 class is an improvement over the S3 class. They have a formally defined structure which helps in making object of the same class look more or less similar.

Class components are properly defined using the setClass() function and objects are created using the new() function.

Example 2: S4 class

< setClass("student", slots=list(name="character", age="numeric", GPA="numeric"))

Reference Class

It is more similar to the object oriented programming we are used to seeing in other major programming languages.

Reference classes are basically S4 classed with an environment added to it.

Example 3: Reference class

< setRefClass("student")

Comparision between S3 vs S4 vs Reference Class

S3 Class	S4 Class	Referene Class
Lacks formal definition	Class defined using setClass()	Class defined using setRefClass()
Objects are created by setting the class attribute	Objects are created using new()	Objects are created using generator functions
Attributes are accessed using $	Attributes are accessed using @	Attributes are accessed using $
Methods belong to generic function	Methods belong to generic function	Methods belong to the class
Follows copy-on-modify semantics	Follows copy-on-modify semantics	Does not follow copy-on-modify semantics

UNIT-2: R PROGRAMMING STRUCTURES

Topics: R Programming structures, Control Statements, Loops, - Looping over nonvector, sets -If-Else, Arithmetic and Boolean Operators and values, Default Values for Argument , Return Values, Deciding whether to explicitly call return- Returning complex objects, Functions are Objective, No Pointers in R, Recursion, A Quicksort Implementation- Extended Example: A Binary Search Tree.

2. R Programming Structures

R is a full programming language, similar to scripting languages such as Perl and Python. One can define functions, use constructs such as loops and conditionals, etc.

R is also block-structured, in a manner similar to those of the above languages, as well as C. Blocks are delineated by braces, though they are optional if the block consists of just a single statement. Statements are separated by new line characters or, optionally by semicolon.

2.1 Control Statements

R has the standard *control structures* you would expect. These are the basic control-flow constructs of the *R* language. They function in much the same way as *control statements* in any Algol-like language.

2.1.1 Loops

R has three statements that provide explicit looping. They are for, while and repeat. The two built-in constructs, next and break, provide additional control over the evaluation.

For Loop:

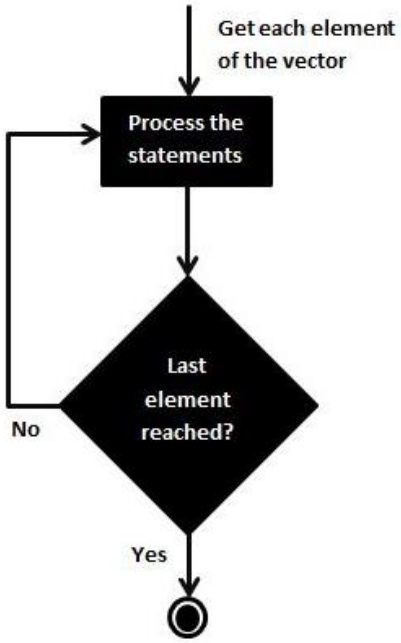
A For loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The basic syntax for creating a **for** loop statement in R is −

```
for (value in vector) {
   statements
}
```

Flow diagram



for (n in x) {

It means that there will be one iteration of the loop for each component of the vector x, with n taking on the values of those components. In other words, in the first iteration, **n = x[1],** in the second iteration **n = x[2],** etc.

For example: The following code uses this structure to output the square of every Element in a vector:

```
> x <- c(5,12,13)
> for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

While loop:

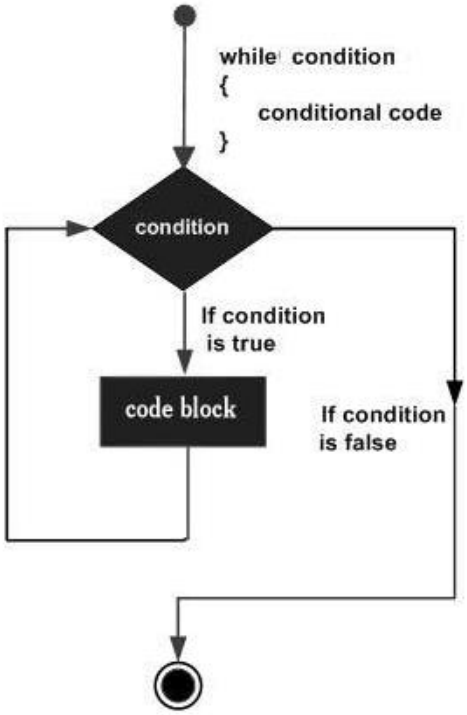
The While loop executes the same code again and again until a stop condition is met.

Syntax
The basic syntax for creating a while loop in R is –

```
while (test_expression) {

statement
```

}

Flow Diagram



Here key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
> i <- 1
> while (i<6) {
> print(i)
> i=i+1
> }

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

Repeat Loop:

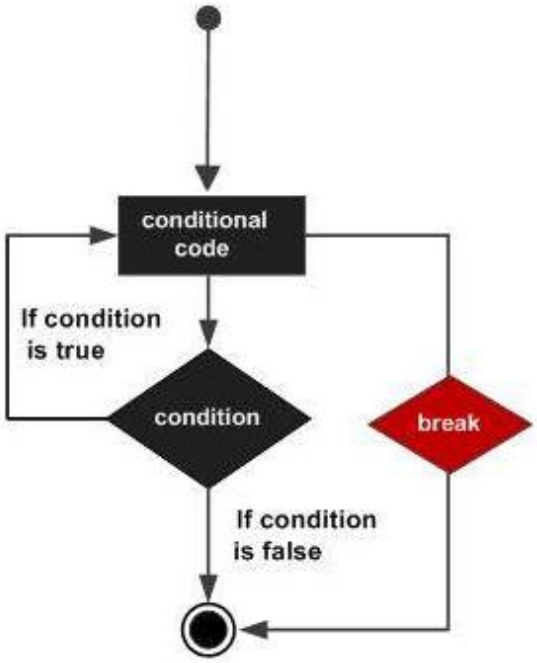
The **Repeat loop** executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a repeat loop in R is –

```
repeat {

 commands

   if(condition) {

      break
   }
}
```

Flow Diagram



Example

```
> x <- 1
> repeat {
> print(x)
> x=x+1
> if (x==6) {
> break
```

```
> }
>}

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

2.1.1.1 Looping Over Nonvector Sets

R does not directly support iteration over nonvector sets, but there are a couple of indirect yet easy ways to accomplish it:

- Use lapply(), assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order.
- Use get(). As its name implies, this function takes as an argument a character string representing the name of some object and returns the object of that name. It sounds simple, but get() is a very powerful function.

The **for** construct works on any vector, regardless of mode. One can loop over a vector of file names, for instance. Say we have files x and y with contents

```
1
2
3
4
5
6
```

and

```
5
12
13
```

Then this loop prints each of them:

```
> for (fn in c("x","y")) print(scan(fn))
Read 6 items
[1] 1 2 3 4 5 6
Read 3 items
```

[1] 5 12 13

2.2 If-Else

R *if* statement. *If* the test_expression is TRUE, the statement gets executed. But *if* it's FALSE, nothing happens. Here, test_expression can be a logical or numeric vector, but only the first element is taken into consideration.

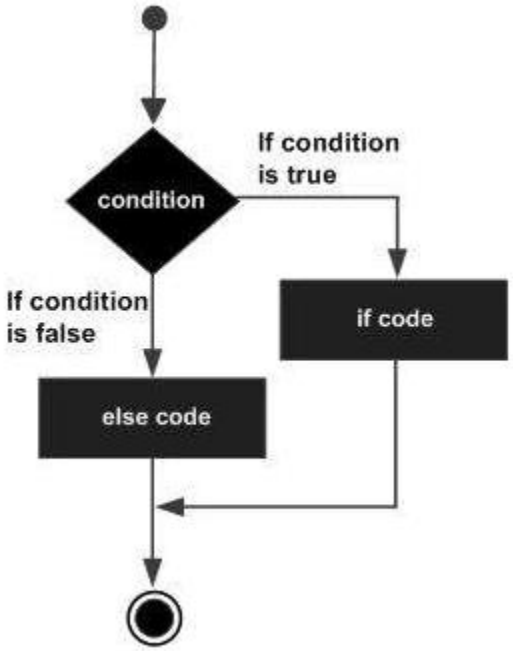
Syntax

The basic syntax for creating an **if...else** statement in R is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
} else {
   // statement(s) will execute if the boolean expression is false.
}
```

If the Boolean expression evaluates to be **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow diagram



Example

```
x <- c("what","is","truth")
if("Truth" %in% x) {
   print("Truth is found")
} else {
   print("Truth is not found")
}
 [1] "Truth is not found"
```

Here "Truth" and "truth" are two different strings.

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using **if**, **else if**, **else** statements there are few points to keep in mind.

- An **if** can have zero or one **else** and it must come after any **else if**'s.
- An **if** can have zero to many **else if's** and they must come before the else.
- Once an **else if** succeeds, none of the remaining **else if**'s or **else**'s will be tested.

Syntax

The basic syntax for creating an **if...else if...else** statement in R is −

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true.
} else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true.
} else {
   // executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")
```

```
if("Truth" %in% x) {
  print("Truth is found the first time")
} else if ("truth" %in% x) {
  print("truth is found the second time")
} else {
  print("No truth found")
}
[1] "truth is found the second time"
```

2.3 Arithmetic and Boolean Operators and Values

R has many operators to carry out different mathematical and logical operations.

x + y	addition
x - y	subtraction
x * y	multiplication
x / y	division
x ^ y	exponentiation
x %% y	modular arithmetic
x %/% y	integer division
x == y	test for equality
x <= y	test for less-than-or-equal
x >= y	test for greater-than-or-equal
x && y	boolean and for scalars
x || y	boolean or for scalars
x & y	boolean and for vectors (vector x,y,result)
x | y	boolean or for vectors (vector x,y,result)
!x	boolean negation

The boolean values are **TRUE** and **FALSE**. They can be abbreviated to **T** and **F**, but must be capitalized.

These values change to 1 and 0 in arithmetic expressions

For example:

```
> 1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
> (1 < 2) == TRUE
[1] TRUE
> (1 < 2) == 1
[1] TRUE
```

You can invent your own operators! Just write a function whose name begins and ends with %. An example is given below.

There are set operations,

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
> 2 %in% x # note that plain "in" doesn’t work
[1] TRUE
> 2 %in% y
[1] FALSE
```

2.4 Default values for Arguments

R functions often specify **default values** for function **arguments**. Invocation of the function may override **defaults** for some **arguments** and accept **defaults** for others. Inside of a

function with **default argument values**, **arguments** always have a value even if it is NA or NULL — they are never 'missing'.

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

Example:

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

```
[1] 18
[1] 45
```

2.5 Return Value

The return value of a function can be any **R** object. We require functions to do some processing and return back the result. This is accomplished with return() function in R.

The syntax of using return in function is

return(expression)

Example:

A function which will return whether a given number is positive, negative or zero.

```
> check <- function(x)
> if (x>0) {
> result <- “positive”
> }
> else if (x<0) {
result <- “negative”
> }
> else {
> result <- “zero”
```

```
}
check(0)
check(4)
check(-2)
```

Multiple Returns:

The return() function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it.

In the following example, we create a list my_list with multiple elements and return this single list.

```
> multi_return <- function() {
> my_list <- list ("color"="red", "size"=20, "shape"="round")
> return(my_list)
> }
> a <- multi_return()
> a$color
[1] "red"
> a$size
[1] 20
> a$shape
[1] "round"
```

2.5.1 Decide whether to explicitly call return

The prevailing R idiom is to avoid explicit calls to **return()**. One of the reasons cited for this approach is that calling that function lengthens execution time. However, unless the function is very short, the time saved is negligible, so this might not be the most compelling reason to refrain from using **return()**. But it usually isn't needed nonetheless.

Consider our second example from the preceding section:

```
oddcount <- function(x) {
k <- 0
for (n in x) {
if (n %% 2 == 1) k <- k+1
  }
```

```
k
}
```

Here, we simply ended with a statement listing the expression to be returned—in this case, k. A call to **return()** wasn't necessary. Good software design, however, should be mean that you can glance through a function's code and immediately spot the various points at which control is returned to the caller. The easiest way to accomplish this is to use an explicit return() call in all lines in the middle of the code that cause a return. (You can still omit a return() call at the end of the function if you wish.)

2.5.2 Returning Complex Objects

Since the return value can be any R object, you can return complex objects.

Here is an example of a function being returned:

```
> g
function() {
t <- function(x) return(x^2)
return(t)
}
> g()
function(x) return(x^2)
<environment: 0x8aafbc0>
```

If your function has multiple return values, place them in a list or other container.

2.6 Functions are Objects

The functions are first-class objects, of the class **function**. That they can be used for the most part just like, say, a vector. In terms of syntax and operation, **R** functions are similar to those of C, Python and so on.

This is seen in the syntax of function creation:

```
> g <- function(x) {
+ return(x+1)
+ }
```

Here **function()** is a built-in **R** function whose job is to create functions! On the right-hand side above, there are really two arguments to **function()**, the first of which is x and the second is "return(x+1)". These will be used by **function()** to create the desired function, which it then assigns to g.

Recall that when using **R** in interactive mode, simply typing the name of an object results in printing that object to the screen.

```
> g
function(x) {
return(x+1)
}
```

This is handy if you're using a function that you've written but have forgotten what its arguments are, for instance. It's also useful if you are not quite sure what an R library function does; by looking at the code you may understand it better.

Similarly, we can assign functions, use them as arguments to other functions, and so on:

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1
> f(3,2)
[1] 5
> f <- f2
> f(3,2)
[1] 1
> g <- function(h,a,b) h(a,b)
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

The return value of **function()** is a function, even if you don't assign it to a variable. Thus you can create anonymous functions, familiar to Python programmers. Modifying the above example, for instance, we have:

```
> g <- function(h,a,b) h(a,b)
> g(function(x,y) return(x*y),2,3)
[1] 6
```

Here, the expression

```
function(x,y) return(x*y)
```

created the specified function, which then played the role of g()'s formal argument **h** in the call to **g().**

2.7 No Pointers in R

R does not have variables corresponding to ***pointers*** or ***references*** like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature called ***reference classes***, which may reduce the difficulty.)

Here, the value of x, the argument to sort(), changed. By contrast, here's how it works in R:

```
> x <- c(13,5,12)
> sort(x)
[1] 5 12 13
> x
[1] 13 5 12
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

```
> x <- sort(x)
> x
[1] 5 12 13
```

2.8 Recursion

A Function that calls itself is called a recursive function and this technique is known as recursion. This special programming technique can be used to solve problems by breaking them into smaller and simpler sub problems.

Example: Finding the factorial of a number.

```
> recursive.factorial <- function(x) {
> if (x==0) return (1)
> else return (x * recursive.factorial (x-1))
> }
> recursive.factorial(5)
[1] 120
> recursive.factorial(2)
[1] 2
```

2.8.1 A Quick Sort Implementation

A classic example is **Quicksort**, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88). We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired.

R's vector-filtering capability and its c() function make implementation of **Quicksort** quite easy.

This example is for the purpose of demonstrating recursion. R's own sort function, sort(), is much faster, as it is written in C.

```
> quicksort <- function(vect) {
> if (length (vect) <=1) {
> return(vect)
> }
> element <- vect[1]
> partition <- vect[-1]
> v1 <- partition [partition < element]
> v2 <- partition [partition >= element]
> v1 <- quicksort(v1)
> v2 <- quicksort(v2)
> return (c(v1, element, v2))
> }
> quicksort(c(4, 65, 2, -31, 0, 99, 83, 782, 1))
[1] -31 0 1 2 4 65 83 99 782
```

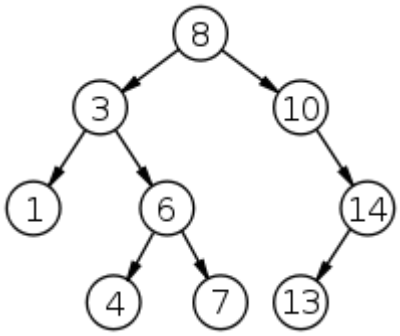
2.8.2 Extended Example: A Binary Search Tree

Treelike data structures are common in both computer science and statistics. In R, for example, the **rpart** library for a recursive partioning approach to regression and classification is very popular.

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.

- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes

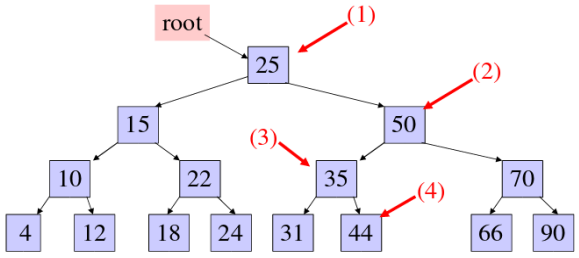


The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Example: search for 45 in the tree

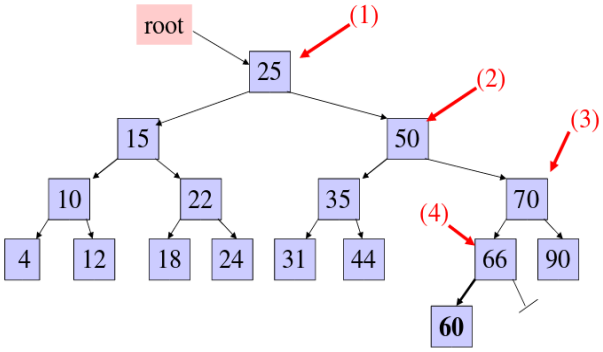
(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



UNIT-3: DOING MATH AND SIMULATION IN R

Topics: Doing Math and Simulation in R, Math Function, Extended Example Calculating Probability-Cumulative Sums and Products-Minima and Maxima-Calculus, Functions for Statistical Distribution, Sorting, Linear Algebra Operation on Vectors and Matrices, Extended Example: Vector Product- Extended Example: Finding stationary distribution on Markov chains, Set operation, Input/Output, Accessing the Keyboard and Monitor, Reading and Writing files

3.1Doing Math and Simulation in R:

R contains built in functions for math operations and statistical distributions.

3.2 Math functions:

R includes an extensive set of built-in math functions.

- exp(): exponential function, base e
- log(): natural algorithm
- log10(): logarithm base 10
- sqrt(): square root
- abs(): absolute value
- sin(), cos() and so on: trigonometry functions
- min() and max(): minimum value and maximum value within a vector
- which.min() and which.max(): index of the minimal element and maximal element of a vector
- pmin() and pmax(): element-wise minima and maxima of several vectors
- sum() and prod(): sum and product of the elements of vector
- cumsum() and cumprod(): cumulative sum and product of the elements of a vector
- rounding off value to the nearest integer: round( )
- rounding off value to the nearest integer below: floor( )
- rounding off value to the nearest integer above: ceiling( )
- factorial(): factorial function

3.3 Extended Example Calculating Probability:

Calculating probability using prod() function. Suppose we have n independent events, and the ith event has the probability pi of occurring. What is the probability of exactly one of these events occurring?

Consider an example where the value of n=3. The events being named A, B, and C. Then the calculation will be modularized as:

Probability(exact occurrence of one event)= Probability(A and not B and not C) + Probability(not A and B and not C) + Probability(not A and not B and C)

$$\sum_{i=1}^{n} p_i (1-p_1)\dots(1-p_{i-1})(1-p_{i+1})\dots(1-p_n)$$

(The i^{th} term inside the sum is the probability that event i occurs and all the others do not occur.) Here's code to compute this, with our probabilities pi contained in the vector p:

```
>exactlyone <- function(p) {
+ notp <-1-p
+ tot <- 0.0
+ for (i in 1:length(p))
+ tot <- tot + p[i] * prod(notp[-i])
+ return(tot)
}
```

notp <-1-p

creates a vector of all the "not occur" probabilities 1− pj, using recycling. The expression notp[-i] computes the product of all the elements of notp, except the ith.

3.4 Cumulative Sums and Products

The cumulative sum and product are returned by the functions cumsum( ) and cumprod( ).

```
> x <- c(10, 15, 20)
> cumsum(x)
[1] 10 25 45
```

> cumprod(x)

[1] 10 150 3000

In x, the sum of the first element is 10, the sum of the first two elements is 25, and the sum of the first three elements is 45. The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

3.5 MinimaandMaxima

The two functions used to computer minimum value is min( ) and pmin(). The first function min( ) returns a single value from a vector, whereas pmin( ) is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.

Here's an example:

> x <- c(10, 25, 20)

> min(x)

[1] 10

> min (x[1], x[2])

[1] 10

> min (x[3], x[2])

[1] 20

Pairwise min( )

> x <- c(10,25,20)

> y <- c(11,13,15)

> pmin(x, y)

[1] 10 13 15

The minimization/maximization can be done via nlm() and optim().

For example, let's find the smallest value of f(x)=x2 −sin(x).

```
> x <- c(10,25,20)

> nlm(function(x) return(x^2-sin(x)),8)

 $minimum [1] -0.2324656

$estimate [1] 0.4501831

$gradient [1] 4.024558e-09

$code [1] 1

$iterations [1] 5
```

Here, the minimum value was found to be approximately−0.23, occur-ring at x = 0.45. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case. The second argument specifies the initial guess, which we set to be 8.

3.6 Calculus

The R Programming language contains the capability of performing calculus including symbolic differentiation and numerical integration, as you can see in the following **example.**

```
> D(expression(exp(x^2)),"x") # derivative

 exp(x^2) * (2 * x)

> integrate(function(x) x^2,0,1)

0.3333333 with absolute error < 3.7e-15
```

Here, R reported

d /dx e^{x2} =2xe^{x2}

and

integrate of 0 and 1 x^2dx ≈0.3333333

R has packages for differential equations (odesolve), for interfacing R with the Yacas symbolic math system (ryacas), and for other calculus operations. These packages, and thousands of others, are available from the Comprehensive R Archive Network (CRAN);

3.7 Functions for Statistical Distributions

R has functions available for most of the famous statistical distributions. Prefix the name as follows:

- With d for the density or probability mass function (pmf)
- With p for the cumulative distribution function (cdf)
- With q for quantiles
- With r for random number generation.

The rest of the name indicates the distribution. Table 8-1 lists some common statistical distribution functions.

Table 8-1: Common R Statistical Distribution Functions

Distribution	Density/pmf	Cdf	Quantities	Random Numbers
Normal	dnorm()	pnorm()	qnorm()	rnorm()
Chi square	dchisq()	pchisq()	qchisq()	rchisq()
Binomial	dbinom()	pbinom()	qbinom()	rbinom()

As an example, let's simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

```
> mean(rchisq(1000,df=2))

[1] 1.938179
```

'chisq' represents the chi square distribution used and 'r' represents the random numbers which will be generated in this distribution. The first argument depicts the total random variates to be produced. The 'df' used represents the degree of freedom to be specified.

Example compute the 95th percentile of the chi-square distribution with two degrees of freedom:

```
> qchisq(0.95,2)

[1] 5.991465
```

Here, we used q to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile. The first argument in the d, p, and q series is actually a vector so that we can evaluate the density/pmf, cdf, or quantile function at multiple points.

Let's find both the 50th and 95th percentiles of the chi-square distribution with 2 degrees of freedom.

```
qchisq(c(0.5,0.95),df=2)

[1] 1.386294 5.991465
```

3.8 Sorting

Ordinary numerical sorting of a vector can be done with the sort() function, as in this example:

```
> x <- c(13,5,12,5)

> sort(x)

[1] 5 5 12 13

->x

[1] 13 5 12 5
```

If you want the indices of the sorted values in the original vector, use the order() function. Here's an example:

```
> x <- c(13,5,12,5)

> order(x)

[1] 2 4 3 1
```

This means that x[2] is the smallest value in x, x[4] is the second smallest, x[3] is the third smallest, and so on.

Another function which tells the rank of every single element present in a vector is called rank( )

```
> x <- c(13,5,12,5)

> rank(x)

[1] 4.0 1.5 3.0 1.5
```

The above console demonstrates that the value 13 lies at 4^{th} rank, which means that the fourth smallest element in x is 13. Now 5, number appears two times in the vector x, so the rank 1.5 is allotted to both the numbers.

3.9 Sorting:

Elements in a vector can be sorted using the sort() function.

```
v <- c(3,8,4,5,0,11, -9, 304)

# Sort the elements of the vector.

sort.result <- sort(v)

print(sort.result)

# Sort the elements in the reverse order.

revsort.result <- sort(v, decreasing = TRUE)

print(revsort.result)

# Sorting character vectors.

v <- c("Red","Blue","yellow","violet")

sort.result <- sort(v)

print(sort.result)

# Sorting character vectors in reverse order.

revsort.result <- sort(v, decreasing = TRUE)
```

print(revsort.result)

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

[1] -9 0 3 4 5 8 11 304

[1] 304 11 8 5 4 3 0 -9

[1] "Blue" "Red" "violet" "yellow"

[1] "yellow" "violet" "Red" "Blue"

3.1.8 Linear Algebra Operations on Vectors and Matrices

The vector quantity can be multiplied to a scalar quantity as below.

> x <- c(13,5,12,5)

> y <- 2*x

> y

[1] 26 10 24 10

If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:

> crossprod(1:3,c(5,12,13))

[,1]

[1,] 68

The function computed 1·5+2·12 + 3·13 = 68. Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product. For matrix multiplication in the mathematical sense, the operator to use is %*%, not *. For instance, here we compute the matrix product:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 1 \end{pmatrix}$$

>a

[,1] [,2]

[1,] 1 2

[2,] 3 4

>b

[,1] [,2]

[1,] 1 -1

[2,] 0 1

>a% *%b

[,1] [,2]

[1,] 1 1

[2,] 3 1

The function solve() will solve systems of linear equations and even find matrix inverses. For example:

x1 + x2 =2

−x1 + x2 =4

Its matrix form is as follows:

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

Here's the code:

> a <- matrix(c(1,1,-1,1),nrow=2,ncol=2)

> b <- c(2,4)

> solve(a,b)

[1] 3 1

> solve(a)

[,1] [,2]

[1,] 0.5 0.5

[2,] -0.5 0.5

3.10Extended Vector Cross Product

Example:

Suppose we have two vectors u=(2,0,-8) and v=(0,0,1).The vector which is perpendicular/orthogonal (90 degrees) to both vectors u and v is the vector cross product. The cross product vector can be determined by using the pracma package in R.

Finding the vector cross product of the two vectors

u = (2,0,-8) and v = (0,0,1)

u <- c(2,0,-8)

v <- c(0,0,1)

crossprod1 <- cross(u,v)

crossprod1

[1] 0 -2 0

3.11 Finding Stationary distribution of Markov Chains

A markov chain is a collection of random variables $\{X_t\}$ (where the index $_t$ runs through 0, 1, ..) having the property that, given the present, the future is conditionally independent of the past.

Let pij denote the transition probability of moving from state i to state j during a time step. We are interested in calculating the vector π =(π1,...,πs), where πi is the long-run proportion of time spent at state i, over all states i. Let P denote the transition probability matrix whose ith row, jth column element is pij. Then it can be shown that π must satisfy Equation 8.4,

π = πP----------→ (8.4)

which is equivalent to Equation 8.5:

$(I - P^T)\pi = 0$ ----------→(8.5)

Here I is the identity matrix and P^T denotes the transpose of P.

Any single one of the equations in the system of Equation 8.5 is redundant. We thus eliminate one of them, by removing the last row of I−P in Equation 8.5. That also means removing the last 0 in the 0 vector on the right-hand side of Equation 8.5.

But note that there is also the constraint shown in Equation 8.6.

$$\sum_i \pi_i = 1$$ -------→(8.6)

In matrix terms, this is as follows:

$$1_n^T \pi = 1$$

where 1n is a vector of n 1s. So, in the modified version of Equation 8.5, we replace the removed row with a row of all 1s an d, on the right-hand side, replace the removed 0 with a 1. We can then solve the system.

All this can be computed with R's solve() function, as follows:

```
findpi1 <- function(p) {
 n <- nrow(p)
imp <- diag(n) - t(p)
imp[n,] <- rep(1,n)
 rhs <- c(rep(0,n-1),1)
 pivec <- solve(imp,rhs)
 return(pivec)
 }
```

Here are the main steps:

1. Calculate $I - P^T$ in line 3. Note again that diag(), when called with a scalar argument, returns the identity matrix of the size given by that argument.

2. Replace the last row of P with 1 values in line 4.

3. Set up the right-hand side vector in line 5.

4. Solve for π in line 6.

3.12 Set Operation

R includes some set operations, including these:

• union(x,y): Union of the sets x and y

• intersect(x,y): Intersection of the sets x and y

• setdiff(x,y): Set difference between x and y, consisting of all elements of x that are not in y

• setequal(x,y): Test for equality between x and y

• c %in% y: Membership, testing whether c is an element of the set y

• choose(n,k): Number of possible subsets of size k chosen from a set of size n

Some simple examples of using these functions:

```
> x <- c(1,2,5)

> y <- c(5,1,8,9)

> union(x,y) [1] 1 2 5 8 9

> intersect(x,y)

[1] 1 5

> setdiff(x,y)

[1] 2
```

```
> setdiff(y,x)

[1] 8 9

> setequal(x,y)

[1] FALSE

> setequal(x,c(1,2,5))

[1] TRUE

> 2 %in% x

[1] TRUE

> 2 %in% y

[1] FALSE

> choose(5,2)

[1] 10
```

3.13 Input/Output

I/O plays a central role in most real-world applications of computers. Example: Consider an ATM machine, which uses multiple I/O Operations for both input-reading your card and reading your typed-in cash request and output-printing instructions on the screen, printing your receipt, and most important, controlling the machine to output your money.

R is not the tool you would choose for running an ATM, but it features a highly versatile array of I/O capabilities, start with the basics of access to the keyboard and monitor, and then go into considerable detail on reading and writing files. Finally R facilities for accessing the internet.

3.13.1 Accessing the Keyboard and Monitor

R provides several functions for accessing the keyboard and monitor. Here, we'll look at the scan(), readline(), print(), and cat() functions.

3.13.2 Using the scan()Function

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list.

Suppose we have files named z1.txt, z2.txt, z3.txt, and z4.txt. The z1.txt file contains the following:

123

4 5

6

The z2.txt file contents are as follows:

123

4.2 5

6

The z3.txt file contains this:

abc

de f

g

And finally, the z4.txt file has these contents:

abc

123 6

y

Let’s see what we can do with these files using the scan() function.

```
> scan("z1.txt")
```

Read 4 items

[1] 123 4 5 6

> scan("z2.txt")

Read 4 items

[1] 123.0 4.2 5.0 6.0

> scan("z3.txt")

Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, : scan() expected 'a real', got 'abc'

> scan("z3.txt",what="")

Read 4 items

[1] "abc" "de" "f" "g"

> scan("z4.txt",what="")

Read 4 items

[1] "abc" "123" "6" "y"

In the first call, we got a vector of four integers (though the mode is numeric). The second time, since one number was nonintegral, the others were shown as floating-point numbers, too.

In the third case, we got an error. The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So, the nonnumeric contents of the file z3 produced an error. But we then tried again, with what="". This assigns a character string to what, indicating that we want character mode.

If you do not wish scan() to announce the number of items it has read, include the argument quiet=TRUE.

3.13.3 Using the readline()Function

If you want to read in a single line from the keyboard, readline() is very handy.

> w <- readline()

abc de f

>w

[1] "abc de f"

Typically, readline() is called with its optional prompt, as follows:

> inits <- readline("type your initials: ")

type your initials: NM

> inits

[1] "NM"

3.13.4 Printing to the Screen

At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression. This won't work if you need to print from within the body of a function. In that case, you can use the print() function, like this:

> x <- 1:3

> print(x^2)

[1] 1 4 9

Recall that print() is a generic function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class "table", then the print.table() function will be called.

It's a little better to use cat() instead of print(), as the latter can print only one expression and its output is numbered, which may be a nuisance. Compare the results of the functions:

> print("abc")

[1] "abc"

> cat("abc\n")

abc

```
>x
[1] 1 2 3
> cat(x,"abc","de\n")
1 2 3 abc de
```

If you don’t want the spaces, set sep to the empty string "", as follows:

```
> cat(x,"abc","de\n",sep="")
123abcde
```

Any string can be used for sep. Here, we use the newline character:

```
> cat(x,"abc","de\n",sep="\n")
1
 2
3
abc
de
```

You can even set sep to be a vector of strings, like this:

```
> x <- c(5,12,13,8,88)
> cat(x,sep=c(".",".",".","\n","\n"))
5.12.13.8
88
```

3.13.5 Reading and Writing files

In R, we can read data from files stored outside the R environment. We can also write data into files which will be stored and accessed by the operating system. R can read and write into various file formats like csv, excel, xml etc.

To read data from a csv file and then write data into a csv file. The file should be present in current working directory so that R can read it. We can also set our own directory and read files from there.

Getting and Setting the Working Directory:

You can check which directory the R workspace is pointing to using the getwd() function. You can also set a new working directory using setwd()function.

 # Get and print current working directory.

print(getwd())

 # Set current working directory.

setwd("/web/com")

 # Get and print current working directory.

print(getwd())

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

[1] "/web/com/1441086124_2016"

[1] "/web/com"

This result depends on your OS and your current directory where you are working.

Reading a Data Frame or Matrix from a file

For illustration purpose the following data file is considered:

Name Age

Mohan 34

Pary 32

Tim 16

The following code reads the file:

```
> z <- read.table (“z”, header=TRUE)

> z

Name  Age

1 Mohan 34

2 Pary    32

3 Tim    16
```

Scan ( ) function does not work when the read file is an amalgam of numeric and character data. This amalgam can be read in matrix in the way:

```
> x <- matrix(scan(“x”), nrow=5, byrow=TRUE)
```

For generic purpose, read.table( ) is used. It yields a data frame, which is converted to a matrix( ). The technique is:

```
read.matrix <- function(filename) {

as.matrix(read.table(filename))

}
```

Reading Text Files:

There are two types of files existing in literature: text files and binary files. The binary file as the name suggests consists only of zeros and ones. Whereas the text files contain ASCII text and other human-understandable language. Also the new line characters are present in the text files. The executable files and the image files are generally the binary files. The text files can be read using the readLines( ) function.

Example:

The file S1 has the following text

Abc 35

Def 38

Ghi 29

To read the file the following code is constructed:

> S1 <- readLines("S1")

> S1

[1] "Abc 35" "Def 38" "Ghi 29"

Accessing files on Remote Machines via URLs:

Web URLs can be used with few I/O functions, namely read.table( ) and scan( ). These functions accept web URLs as arguments.

Writing to a file:

write.table( ) is a function used to write a data frame. The following example illustrates the same:

```
> kids <- c("Abn", "klj)

> ages <- c(12, 10)

> d <- data.frame(kids, ages, stringsAsFactors=FALSE)

>d

Kids  ages

1 Abn 12

2 klj 10

> write.table(d, "kds")
```

The new file kds contents will be:

"kids" "ages"

"1" "Abn" 12

“2” “klj” 10

When a matrix is written in a file but row and column name need not be stated:

```
> write.table(xc, “xcnew”, row.names=FALSE, col.names=FALSE)
```

cat( ) is another function which writes to a file, part by part. The following code:

```
> cat(“abc\n, file=”u”)
```

```
> cat(“de\n”, file= “u”, append=TRUE)
```

Multiple fields can be written as:

```
> cat (file=”v”,1,2,”xyz\n”)
```

It produces a single line as

```
1 2 xyz
```

writeLines( ) function can also be used. A connection needs to be created with write privileges:

```
> c <- file(“www”, “w”)
```

```
> writeLines(c (“abc”, “de”, “f”),c)
```

```
>close(c)
```

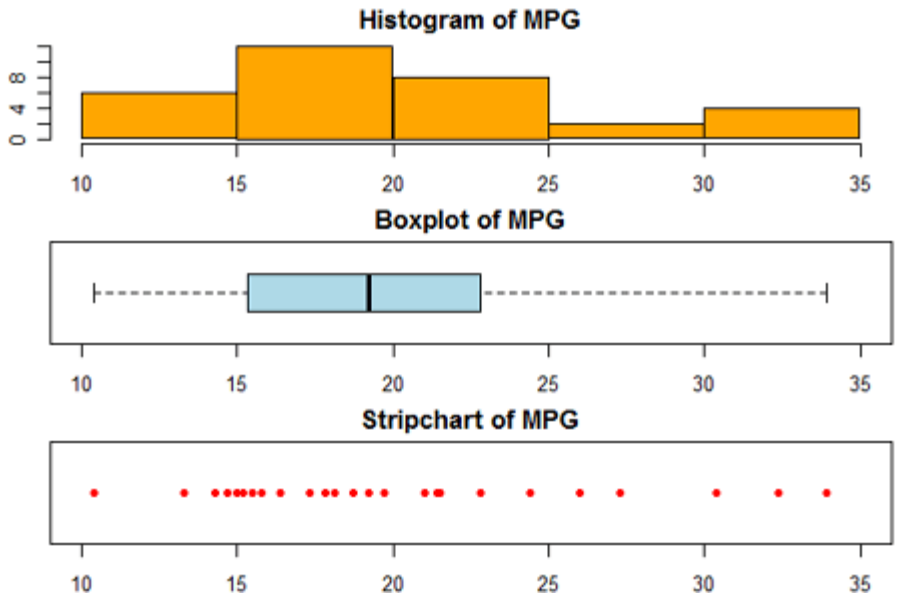
A new file www is created with the matter shown below:

```
abc
de
f
```

The file opened for writing needs to be closed explicitly.

UNIT-4: GRAPHICS

4.1 Introduction:



One of the main reasons data analysts turn to R is for its strong graphic capabilities.

Creating a Graph provides an overview of creating and saving graphs in R.

The remainder of the section describes how to create basic graph types. These include density plots (histograms and kernel density plots), dot plots, bar charts (simple, stacked, grouped), line charts, pie charts (simple, annotated, 3D), boxplots (simple, notched, violin plots, bagplots) and Scatterplots (simple, with fit lines, scatterplot matrices, high density plots, and 3D plots).

The Advanced Graphs section describes how to customize and annotate graphs, and covers more statistically complex types of graphs.

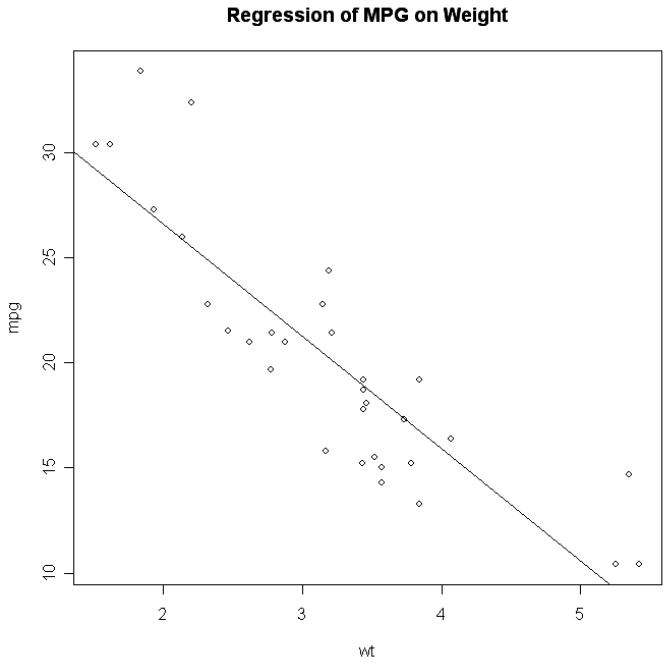
4.2 Creating Graphs:

In R, graphs are typically created interactively

```
# Creating a Graph
attach(mtcars)
plot(wt, mpg)
abline(lm(mpg~wt))
title("Regression of MPG on Weight")
```

The plot( ) function opens a graph window and plots weight vs. miles per gallon.

The next line of code adds a regression line to this graph. The final line adds a title.



Saving Graphs

You can save the graph in a variety of formats from the menu **File -> Save As**.

You can also save the graph via code using one of the following functions.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Viewing Several Graphs

Creating a new graph by issuing a high level plotting command (plot, hist, boxplot, etc.) will typically **overwrite** a previous graph. To avoid this, open a new graph window before creating a new graph. To open a new graph window use one of the functions below.

Function	Platform
windows()	Windows
X11()	Unix
quartz()	Mac

Graphical Parameters

You can specify fonts, colors, line styles, axes, reference lines, etc. by specifying graphical parameters. This allows a wide degree of customization. Graphical parameters, are covered in the Advanced Graphs section. The Advanced Graphs section also includes a more detailed coverage of axis and text customization.

Creating various types of Graphs:

Pie Chart:

R Programming language has numerous libraries to create charts and graphs. A pie-chart is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart.

In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input. The additional parameters are used to control labels, color, title etc.

Syntax

The basic syntax for creating a pie-chart using the R is −

```
pie(x, labels, radius, main, col, clockwise)
```

Following is the description of the parameters used −

- **x** is a vector containing the numeric values used in the pie chart.

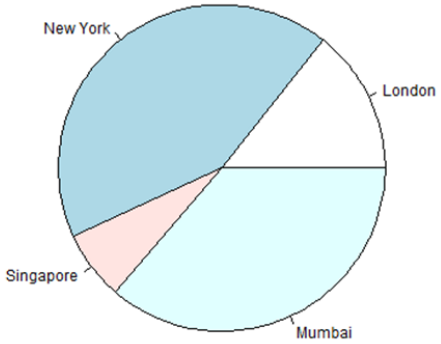
- **labels** is used to give description to the slices.
- **radius** indicates the radius of the circle of the pie chart.(value between −1 and +1).
- **main** indicates the title of the chart.
- **col** indicates the color palette.
- **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

Example

A very simple pie-chart is created using just the input vector and labels. The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels <- c("London", "New York", "Singapore", "Mumbai")

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

# Plot the chart.
pie(x,labels)

# Save the file.
dev.off()
```

Output:



Pie Chart Title and Colors

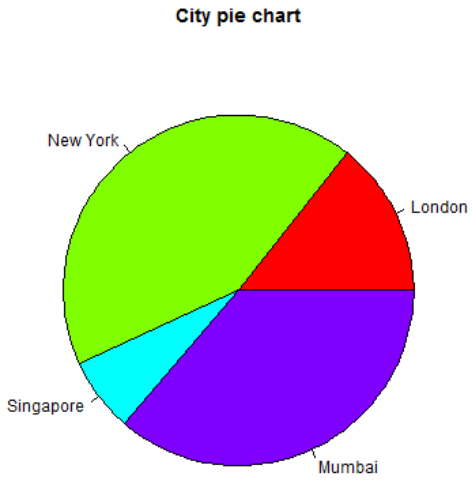
We can expand the features of the chart by adding more parameters to the function. We will use parameter **main** to add a title to the chart and another parameter is **col** which will make use of rainbow colour pallet while drawing the chart. The length of the pallet should be same as the number of values we have for the chart. Hence we use length(x).

Example

The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels <- c("London", "New York", "Singapore", "Mumbai")

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

# Plot the chart with title and rainbow color pallet.
pie(x, labels, main = "City pie chart", col = rainbow(length(x)))

# Save the file.
dev.off()
```

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



Bar Chart:

A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable. R uses the function **barplot()** to create bar charts. R can draw both vertical and horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Syntax

The basic syntax to create a bar-chart in R is −

```
barplot(H, xlab, ylab, main, names.arg, col)
```

Following is the description of the parameters used −

- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.

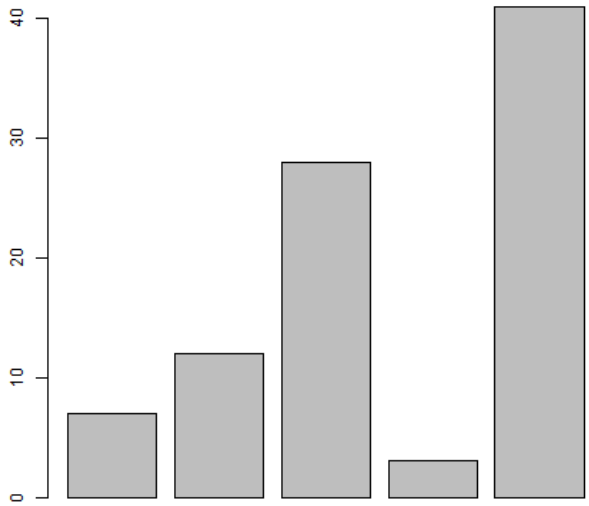
Example

A simple bar chart is created using just the input vector and the name of each bar.

The below script will create and save the bar chart in the current R working directory.

```
# Create the data for the chart.
H <- c(7,12,28,3,41)

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

# Plot the bar chart.
barplot(H)

# Save the file.
dev.off()
```

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



Bar Chart Labels, Title and Colors

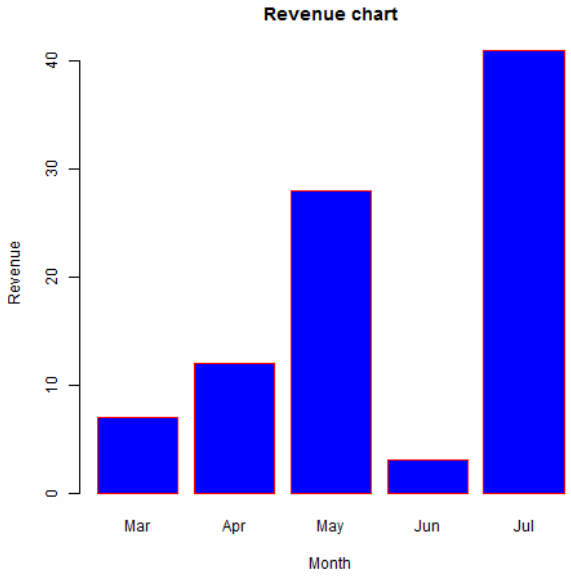
The features of the bar chart can be expanded by adding more parameters. The **main** parameter is used to add **title**. The **col** parameter is used to add colors to the bars. The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.

Example

The following script will create and save the bar chart in the current R working directory.

```
# Create the data for the chart.
H <- c(7,12,28,3,41)
M <- c("Mar","Apr","May","Jun","Jul")
# Give the chart file a name.
png(file = "barchart_months_revenue.png")
# Plot the bar chart.
barplot(H,names.arg = M,xlab = "Month",ylab = "Revenue",col = "blue",
main = "Revenue chart",border = "red")
# Save the file.
dev.off()
```

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



Box Plot:

Boxplots are a measure of how well distributed is the data in a data set. It divides the data set into three quartiles. This graph represents the minimum, maximum, median, first quartile and third quartile in the data set. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.

Boxplots are created in R by using the **boxplot()** function.

Syntax

The basic syntax to create a boxplot in R is −

```
boxplot(x, data, notch, varwidth, names, main)
```

Following is the description of the parameters used −

- **x** is a vector or a formula.
- **data** is the data frame.
- **notch** is a logical value. Set as TRUE to draw a notch.
- **varwidth** is a logical value. Set as true to draw width of the box proportionate to the sample size.
- **names** are the group labels which will be printed under each boxplot.
- **main** is used to give a title to the graph.

Example

We use the data set "mtcars" available in the R environment to create a basic boxplot. Let's look at the columns "mpg" and "cyl" in mtcars.

```
input <- mtcars[,c('mpg','cyl')]
print(head(input))
```

When we execute above code, it produces following result −

```
                mpg  cyl
Mazda RX4       21.0   6
Mazda RX4 Wag   21.0   6
```

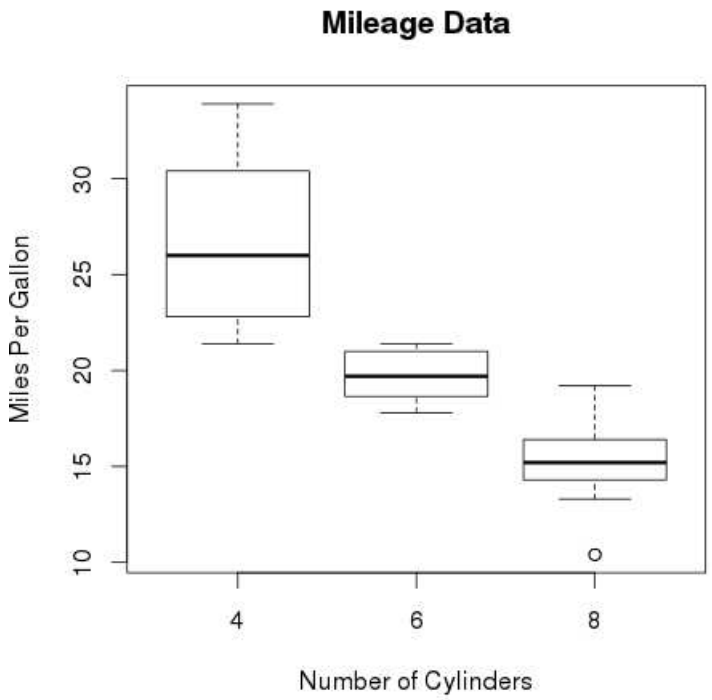
```
Datsun 710        22.8   4
Hornet 4 Drive    21.4   6
Hornet Sportabout 18.7   8
Valiant           18.1   6
```

Creating the Boxplot

The below script will create a boxplot graph for the relation between mpg (miles per gallon) and cyl (number of cylinders).

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

# Plot the chart.
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders",
   ylab = "Miles Per Gallon", main = "Mileage Data")

# Save the file.
dev.off()
```

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



Histogram:

A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chat but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

R creates histogram using **hist()** function. This function takes a vector as an input and uses some more parameters to plot histograms.

Syntax

The basic syntax for creating a histogram using R is −

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

Following is the description of the parameters used −

- **v** is a vector containing numeric values used in histogram.
- **main** indicates title of the chart.
- **col** is used to set color of the bars.
- **border** is used to set border color of each bar.
- **xlab** is used to give description of x-axis.
- **xlim** is used to specify the range of values on the x-axis.
- **ylim** is used to specify the range of values on the y-axis.
- **breaks** is used to mention the width of each bar.

Example

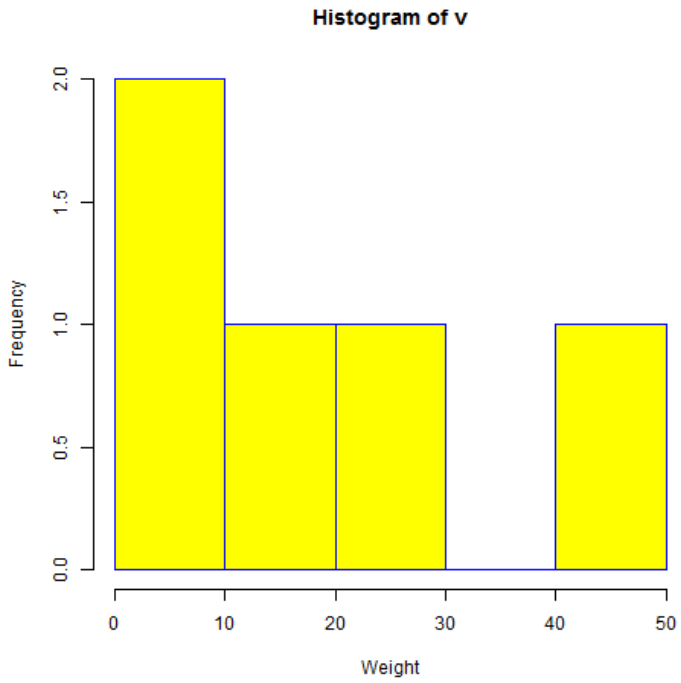
A simple histogram is created using input vector, label, col and border parameters.

The script given below will create and save the histogram in the current R working directory.

```
# Create data for the graph.
v <-  c(9,13,21,8,36,22,12,41,31,33,19)
```

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

# Create the histogram.
hist(v,xlab = "Weight",col = "yellow",border = "blue")

# Save the file.
dev.off()
```

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



Range of X and Y values

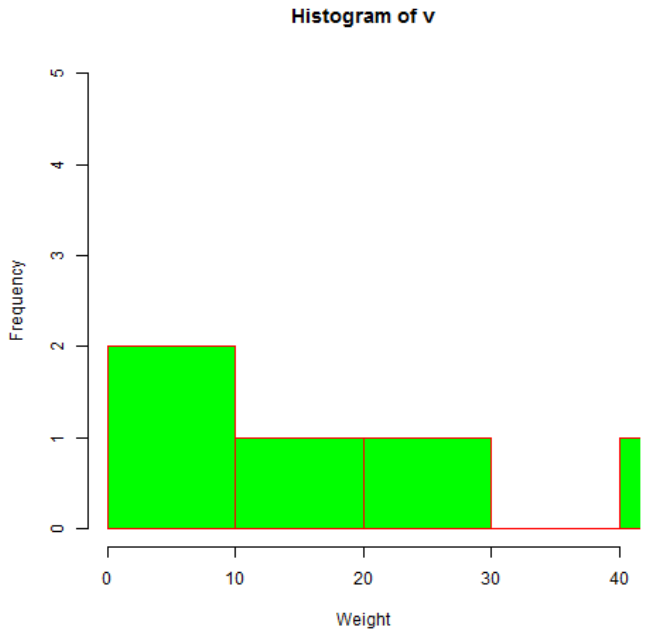
To specify the range of values allowed in X axis and Y axis, we can use the xlim and ylim parameters.

The width of each of the bar can be decided by using breaks.

```
# Create data for the graph.
v <- c(9,13,21,8,36,22,12,41,31,33,19)
```

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

# Create the histogram.
hist(v,xlab = "Weight",col = "green",border = "red", xlim = c(0,40), ylim = c(0,5),
   breaks = 5)

# Save the file.
dev.off()
```

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



Line chart:

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

Syntax

The basic syntax to create a line chart in R is −

```
plot(v,type,col,xlab,ylab)
```

Following is the description of the parameters used −

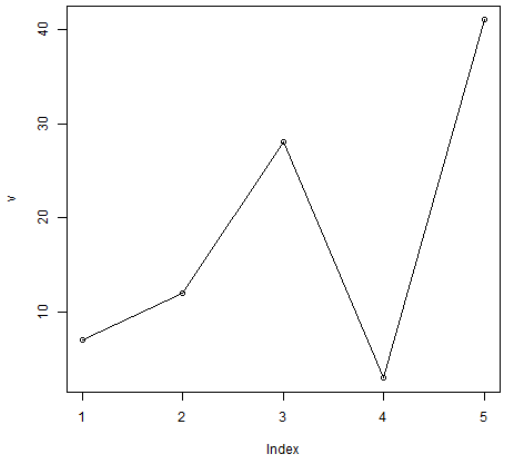
- **v** is a vector containing the numeric values.
- **type** takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the Title of the chart.
- **col** is used to give colors to both the points and lines.

Example

A simple line chart is created using the input vector and the type parameter as "O". The below script will create and save a line chart in the current R working directory.

```
# Create the data for the chart.
v <- c(7,12,28,3,41)

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

# Plot the bar chart.
plot(v,type = "o")

# Save the file.
dev.off()
```

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



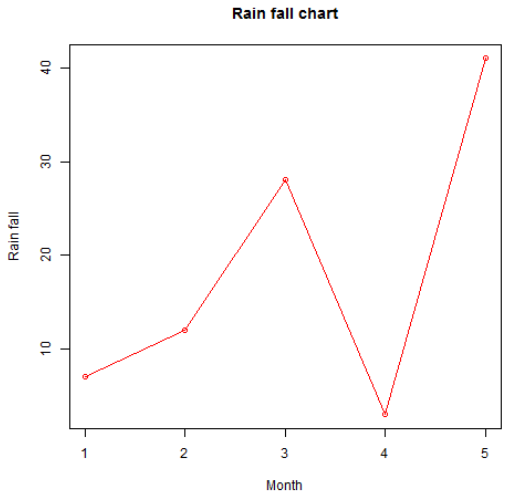
Line Chart Title, Color and Labels

The features of the line chart can be expanded by using additional parameters. We add color to the points and lines, give a title to the chart and add labels to the axes.

Example

```
# Create the data for the chart.
v <- c(7,12,28,3,41)

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

# Plot the bar chart.
plot(v,type = "o", col = "red", xlab = "Month", ylab = "Rain fall",
   main = "Rain fall chart")

# Save the file.
dev.off()
```

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



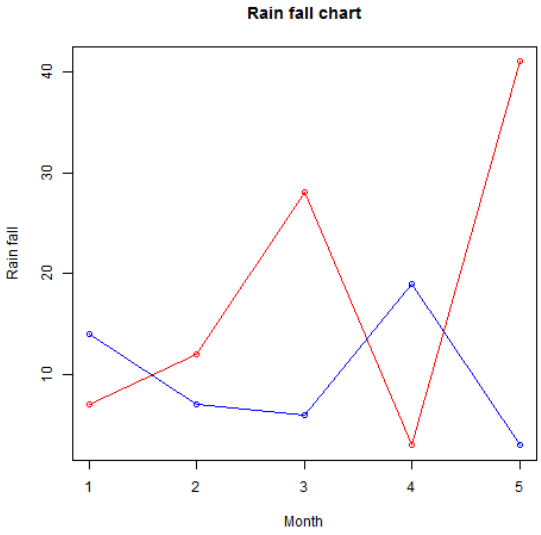
Multiple Lines in a Line Chart

More than one line can be drawn on the same chart by using the **lines()**function.

After the first line is plotted, the lines() function can use an additional vector as input to draw the second line in the chart,

```
# Create the data for the chart.
v <- c(7,12,28,3,41)
t <- c(14,7,6,19,3)

# Give the chart file a name.
png(file = "line_chart_2_lines.jpg")

# Plot the bar chart.
plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",
   main = "Rain fall chart")

lines(t, type = "o", col = "blue")

# Save the file.
dev.off()
```

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



Scatterplot:

Scatterplots show many points plotted in the Cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and another in the vertical axis.

The simple scatterplot is created using the **plot()** function.

Syntax

The basic syntax for creating scatterplot in R is −

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

Following is the description of the parameters used −

- **x** is the data set whose values are the horizontal coordinates.
- **y** is the data set whose values are the vertical coordinates.
- **main** is the tile of the graph.
- **xlab** is the label in the horizontal axis.
- **ylab** is the label in the vertical axis.
- **xlim** is the limits of the values of x used for plotting.

- **ylim** is the limits of the values of y used for plotting.
- **axes** indicates whether both axes should be drawn on the plot.

Example

We use the data set **"mtcars"** available in the R environment to create a basic scatterplot. Let's use the columns "wt" and "mpg" in mtcars.

```
input <- mtcars[,c('wt','mpg')]
print(head(input))
```

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

```
                wt  mpg
Mazda RX4       2.620  21.0
Mazda RX4 Wag   2.875  21.0
Datsun 710      2.320  22.8
Hornet 4 Drive  3.215  21.4
Hornet Sportabout  3.440  18.7
Valiant         3.460  18.1
```

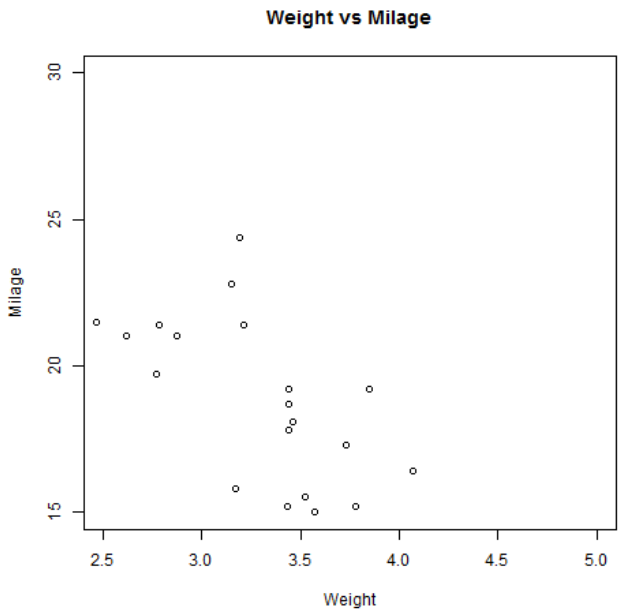
Creating the Scatterplot

The below script will create a scatterplot graph for the relation between wt(weight) and mpg(miles per gallon).

```
# Get the input values.
input <- mtcars[,c('wt','mpg')]

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

# Plot the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.
plot(x = input$wt,y = input$mpg,
  xlab = "Weight",
  ylab = "Milage",
  xlim = c(2.5,5),
  ylim = c(15,30),
```

```
  main = "Weight vs Milage"
)

# Save the file.
dev.off()
```

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



4.3 The Workhorse of R Base Graphics: The plot() function

The plot() function forms the foundation for much of R's base graphing operations, serving as the vehicle for producing many different kinds of graphs.

R is polymorphic, in the sense that the same function can lead to different operations for different classes. You can apply plot(), for example, to many different types of objects, getting a different type of plot for each.

polymorphism may allow you to write your program without worrying about the various types of objects that might be plotted. The functions that work with polymorphism, such as plot() and print(), are known as generic functions. When a generic function is called, R will then dispatch the call to the proper class method, meaning that it will reroute the call to a function defined for the object's class.

Let’s see what happens when we call plot() with an X vector and a Y vector, which are interpreted as a set of pairs in the (x,y) plane.

```
> plot(c(1,2,3), c(1,2,4))
```

This will cause a window to pop up, plotting the points (1,1), (2,2), and (3,4), as shown in Figure 12-1. As you can see, this is a very plain-Jane graph.

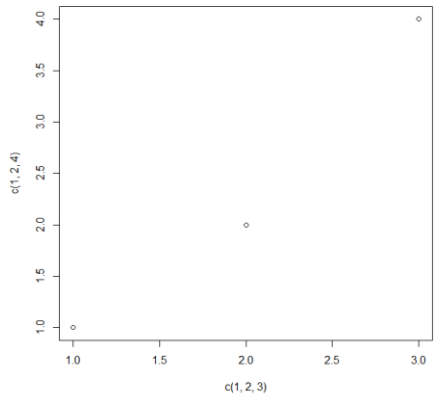


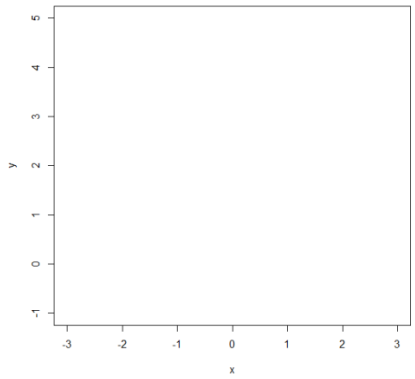
Fig 4.1.1 Simple Point Plot

NOTE: The points in the graph in Figure 12-1 are denoted by empty circles. If you want to use a different character type, specify a value for the named argument pch (for point character).

The plot() function works in stages, which means you can build up a graph in stages by issuing a series of commands. For example, as a base, we might first draw an empty graph, with only axes, like this:

```
> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")
```

This draws axes labeled x and y. The horizontal (x) axis ranges from−3 to 3. The vertical (y) axis ranges from−1 to 5. The argument type="n" means that there is nothing in the graph itself.



4.4 Customizing Graphs

The graphs can be customized using many options that R provides.

4.4.1 Changing character sizes: The cex Option

The cex (for character expand) function allows you to expand or shrink characters within a graph which can be very useful. You can use it as a named parameter in various graphing functions. For instance the text “abc” at some point say (2,5,4), in your graph but with a larger font, in order to call attention to this particular text. You could do this by typing the following:

```
text(2.5,4,"abc",cex = 1.5)
```

This prints the same text as in our earlier example but with characters 1.5 times the normal size.

4.4.2 Changing the Range of Axes: The xlim and ylim Options

The ranges on x and y-axes of your plot be broader or narrower than the default. This is especially useful if you will be displaying several curves in the same graph.

You can adjust the axes by specifying the xlim and/or ylim parameters in your call to plot() or points(). For example, ylim=c(0,90000) specifies a range on the y-axis of 0 to 90,000.

Example:

```
> plot(c(0, 100), c(0, 0.03), type = "n", xlab="score", ylab="density")
```

4.4.3 Graphing Explicit Functions

Say you want to plot the function g(t)=(t2 + 1) 0.5 for t between 0 and 5. You could use the following R code:

```
g <- function(t) { return (t^2+1)^0.5 } # define g()

x <- seq(0,5,length=10000) # x = [0.0004, 0.0008, 0.0012,..., 5]

y <- g(x) # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]

plot(x,y,type="l")
```

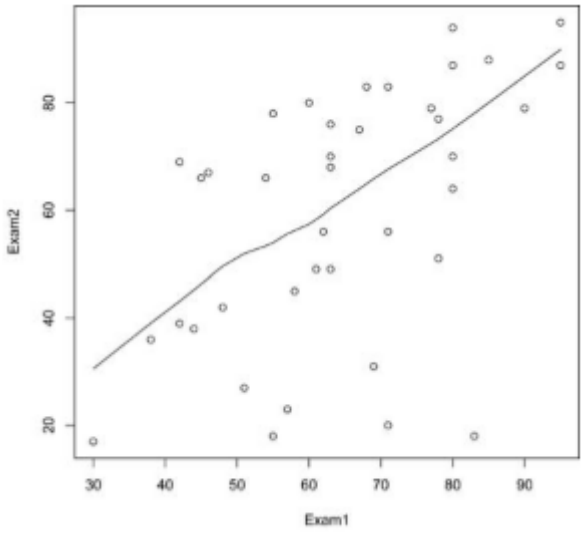


Fig: Smoothing the exam score relation

But you could avoid some work by using the curve() function, which basically uses the same method:

```
> curve((x^2+1)^0.5,0,5)
```

If you are adding this curve to an existing plot, use the add argument:

```
> curve((x^2+1)^0.5,0,5,add=T)
```

4.5 The plot( ) Function in R:

The prevalent function of R programming is the plot( ) function. It's a common function having multiple methods to be called depending upon the object passed to the function plot( ). For example, two vectors are passed and the plot is constructed based upon these two vectors as follows:

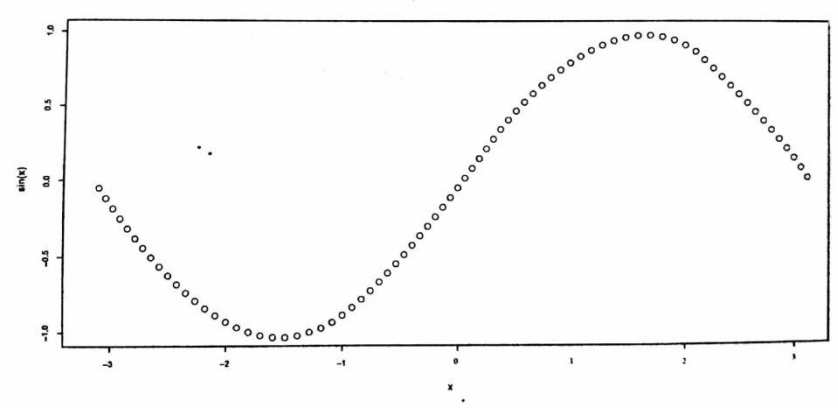
Plot(c(2,3), c(4,5))

This command will yield a plot of the points (2,4) and (3,5).

Example: To demonstrate the plot of a sine function with range –pi to pi:

x <- seq (-pi, pi, 0.1)

plot(x, sin(x))



4.5.1.Titles and Labeling Axes:

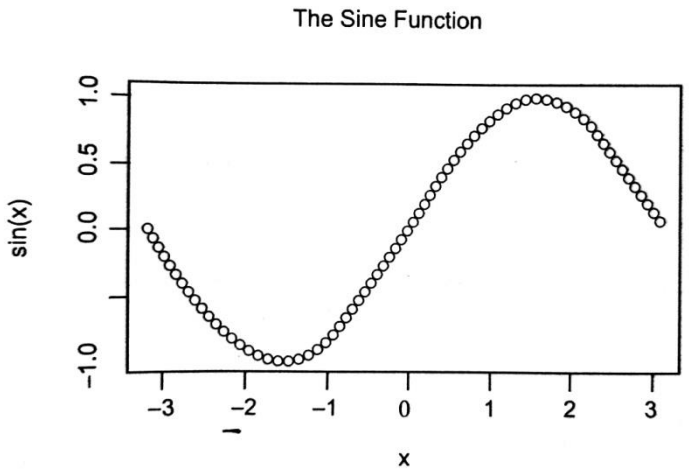
The argument required to provide title to the plot is main. And for labeling the axis we use xlab and ylab for x axis and y axis respectively.

Example:

plot(x, sin(x),

main="The sine function",

ylab = "sin(x)")



4.5.2.Changing color and plot type:

The plot type can be modified by using the attribute type. It has the following effect and accept the strings given below:

“b” – both points and lines

“c” – empty points joined by lines

“h” – histogram like vertical lines

“l”- lines

“n”- does not produce any points or lines

“o” – overplotted points and lines

“p” – points

“s” and “S” – stair steps

Also ‘col’ attribute defines the color on the other hand

Example:

x<- seq(-pi, pi, 0.1)

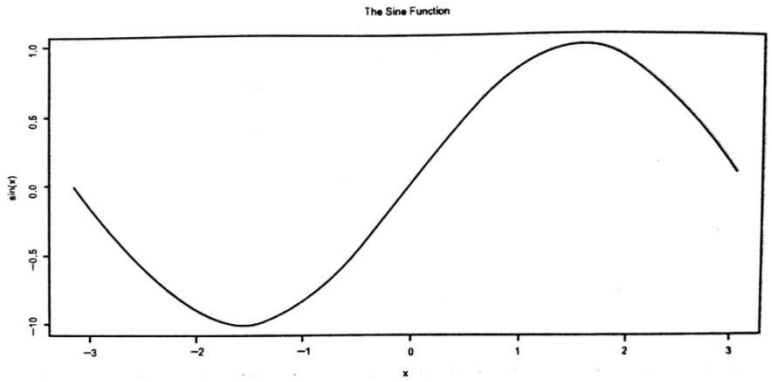
plot(x, sin(x),

main=”The sine function”,

ylab=”sin(x)”,

type=”l”,

col=”blue”)



4.5.3.Overlaying plots:

If the plot ( ) function is called many times, the current graph will be plotted in the same window and the previous existing graph will be replaced by the same. It is done by using the lines( ) and points( ) function which add lines and points to the respective existing plot.

Example:

x<-seq(-pi, pi, 0.1)

plot(x, sin(x),

main=”Overlaying graphs”,

ylab=” “,

type=”l”,

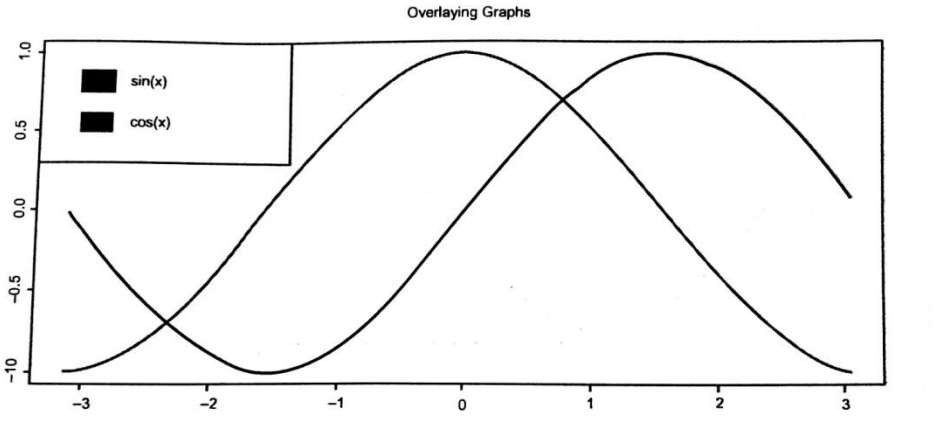
col=”blue”)

lines(x, cos(x), col=”red”)

legend(“topleft”, c(“sin(x)”, “cos(x)”),

fill=c(“blue”, “red”)

)



4.6. Saving Graphs to files

To save a plot to an image file, three things are required in sequence:

1. Open a graphics device.

The default graphics device in R is your computer screen. To save a plot to an image file, you need to tell R to open a new type of device — in this case, a graphics file of a specific type, such as PNG, PDF, or JPG.

The R function to create a PNG device is png(). Similarly, you create a PDF device with pdf() and a JPG device with jpg().

2. Create the plot.

3. Close the graphics device, with the dev.off() function.

Example:

First set your working directory to your home folder (or to any other folder you prefer). If you use Linux, you’ll be familiar with using “~/” as the shortcut to your home folder, but this also works on Windows and Mac:

```
> setwd("~/")
> getwd()
[1] "C:/Users/Andrie"
```

Next, write the three lines of code to save a plot to file:

```
> png(filename="faithful.png")
> plot(faithful)
> dev.off()
```

Now you can check your file system to see whether the file faithful.png exists.

UNIT-5: PROBABILITY DISTRIBUTION

Probability Distributions, Normal Distributions- Binomial Distributions- Poisons Distributions, Other distributions, Basic Statistics, Correlation and Covariance, T-Tests-ANOVA

5.1 Probability Distribution:

A probability distribution describes how the values of a random variable are distributed. For example, the collection of all possible outcomes of a sequence of coin tossing is known to follow the binomial distribution. Whereas the means of sufficiently large samples of a data population are known to resemble the normal distribution. Since the characteristics of these theoretical distributions are well understood, they can be used to make statistical inferences on the entire data population as a whole.

In the following, we demonstrate how to compute a few well-known probability distributions that occurs frequently in statistical study.

5.1.1 Normal Distribution:

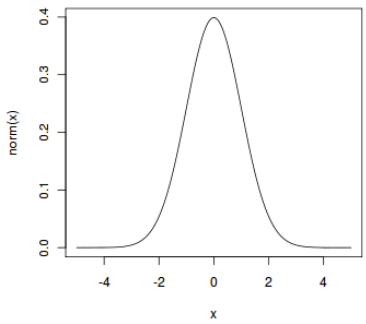
The **normal distribution** is defined by the following probability density function, where μ is the population mean and σ^2 is the variance.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

If a random variable X follows the normal distribution, then we write:

$$X \sim N(\mu, \sigma^2)$$

In particular, the normal distribution with $\mu = 0$ and $\sigma = 1$ is called the standard normal distribution, and is denoted as N(0,1). It can be graphed as follows.



The normal distribution is important because of the Central Limit Theorem, which states that the population of all possible samples of size n from a population with mean μ and variance σ^2 approaches a normal distribution with mean μ and σ^2/n when n approaches infinity.

R has four in built functions to generate normal distribution. They are described below.

```
dnorm(x, mean, sd)
pnorm(x, mean, sd)
qnorm(p, mean, sd)
rnorm(n, mean, sd)
```

Following is the description of the parameters used in above functions −

- **x** is a vector of numbers.
- **p** is a vector of probabilities.
- **n** is number of observations(sample size).
- **mean** is the mean value of the sample data. It's default value is zero.
- **sd** is the standard deviation. It's default value is 1.

Problem

Assume that the test scores of a college entrance exam fits a normal distribution. Furthermore, the mean test score is 72, and the standard deviation is 15.2. What is the percentage of students scoring 84 or more in the exam?

Solution

We apply the function pnorm of the normal distribution with mean 72 and standard deviation 15.2. Since we are looking for the percentage of students scoring higher than 84, we are interested in the upper tail of the normal distribution.

```
> pnorm(84, mean=72, sd=15.2, lower.tail=FALSE)
[1] 0.21492
```

Answer

The percentage of students scoring 84 or more in the college entrance exam is 21.5%.

5.1.2 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.

$$f(x) = \left(\begin{array}{c} n \\ x \end{array} \right) p^x (1-p)^{(n-x)} \quad where\ x = 0, 1, 2, ..., n$$

R has four in-built functions to generate binomial distribution. They are described below.

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

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.

Example:

X is a Binomially Distributed with n=20 trails and p=1/6 probability of success.

X~BIN(n=20, p=1/6)

dbinom()

This function is used to find values for the probability density function of X, f(x)

Suppose we would like to find the P(X=3)

> dbinom(x=3, size=20, prob=1/6)

[1] 0.2378866

Suppose to find the multiple probabilities P(X=0) & P(X=1) &.....& P(X=3)

> dbinom(x=0:3, size=20, prob=1/6)

[1] 0.0260405 0.10433621 0.19823881 0.23788657

Suppose to find the P(X<=3)

> sum(dbinom(x=0:3, size=20, prob=1/6))

[1] 0.5665456

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.

```
# How many heads will have a probability of 0.25 will come out when a coin is tossed 51 times.
x <- qbinom(0.25,51,1/2)

print(x)
```

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

```
[1] 23
```

rbinom()

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

```
# 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
```

5.1.3 Poisson Distribution:

The Poisson distribution is the probability distribution of independent event occurrences in an interval. If λ is the mean occurrence per interval, then the probability of having x occurrences within a given interval is:

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!} \quad where\ x = 0, 1, 2, 3, ...$$

R has four in-built functions to generate poisson distribution. They are described below.

dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)

Arguments:

X	vector of (non-negative integer) quantiles.
Q	vector of quantiles.
P	vector of probabilities.
N	number of random values to return.
lambda	vector of (non-negative) means.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

dpois(x, lambda, log=FALSE):

To find the values for the probability density function of X, f(x).

Example:

To find the probability of p(X=4)

> dpois(x=4, lambda =7)   #Probability of exactly 4 occurrences for a random variable 7

> [1] 0.09122619

For multiple probability occurrences of p(X=0) & p(X=1) &.....& p(X=4)

>dpois(x=0:4, lambda=7)

>[1] 0.000911882 0.006383174 0.022341108 0.052129252 0.091226192

Another way of p(X<=4)

>sum(dpois(x=0:4, lambda=7))

[1] 0.1729916

ppois(q, lambda, lower.tail = TRUE, log.p = FALSE):

It returns the probabilities associated with the probability distribution function, F(x)
Example:
>P(X<=4)

>ppois(q=4, lambda=7, lower.tail=T)

[1] 0.1729916

Another way of P(X>=12)

> ppois(q=12, lambda=7, lower.tail=F)

[1] 0.02699977

qpois(p, lambda, lower.tail = TRUE, log.p = FALSE):

To find the quantiles for the poisson distribution.

rpois(n, lambda):

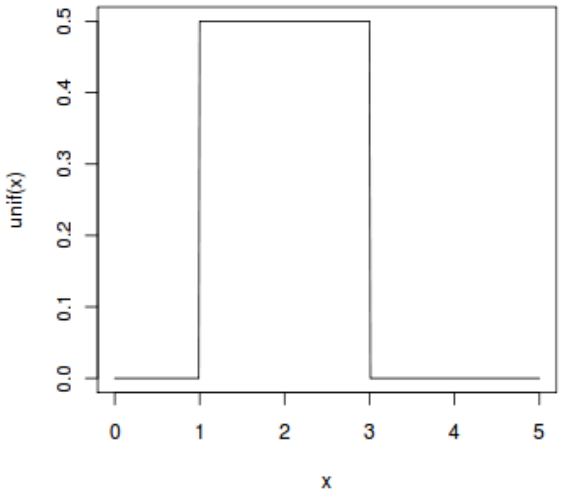
To take a random sample from a poisson distribution.

5.1.4 Continuous Uniform Distribution:

The continuous uniform distribution is the probability distribution of random number selection from the continuous interval between a and b. Its density function is defined by the following.

$$f(x) = \begin{cases} \frac{1}{b-a} & when\ a \leq x \leq b \\ 0 & when\ x < a\ or\ x > b \end{cases}$$

Here is a graph of the continuous uniform distribution with a = 1, b = 3.



These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `punif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

Usage

```
dunif(x, min = 0, max = 1, log = FALSE)
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
runif(n, min = 0, max = 1)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If `length(n) > 1`, the length is taken to be the number required.
min, max	lower and upper limits of the distribution. Must be finite.

log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Problem
Select ten random numbers between one and three.

Solution

We apply the generation function runif of the uniform distribution to generate ten random numbers between one and three.

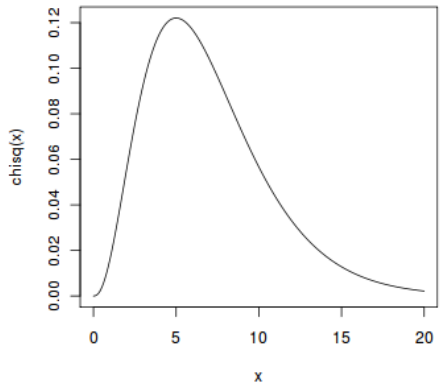
```
> runif(10, min=1, max=3)
[1] 1.6121 1.2028 1.9306 2.4233 1.6874 1.1502 2.7068
[8] 1.4455 2.4122 2.2171
```

5.1.5 Chi-squared Distribution

If X1,X2,…,Xm are m independent random variables having the standard normal distribution, then the following quantity follows a Chi-Squared distribution with m degrees of freedom. Its mean is m, and its variance is 2m.

$V = X21 + X22 + \cdots + X2m \sim \chi 2(m)$

Here is a graph of the Chi-Squared distribution 7 degrees of freedom.



Problem:

Find the 95th percentile of the Chi-Squared distribution with 7 degrees of freedom.

Solution

We apply the quantile function qchisq of the Chi-Squared distribution against the decimal values 0.95.

```
> qchisq(.95, df=7)        # 7 degrees of freedom

[1] 14.067
```

Answer

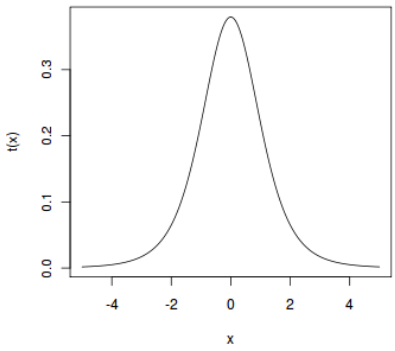
The 95th percentile of the Chi-S.quared distribution with 7 degrees of freedom is 14.067.

5.1.6 Student T Distribution:

Assume that a random variable Z has the standard normal distribution, and another random variable V has the Chi-Squared distribution with m degrees of freedom. Assume further that Z and V are independent, then the following quantity follows a Student t distribution with m degrees of freedom.

$$t = \frac{Z}{\sqrt{V/m}} \sim t_{(m)}$$

Here is a graph of the Student t distribution with 5 degrees of freedom.



R has four in-built functions to generate student T distribution. They are described below

Density, distribution function, quantile function and random generation for the t distribution with df degrees of freedom (and optional non-centrality parameter ncp).

dt(x, df, ncp, log = FALSE)
pt(q, df, ncp, lower.tail = TRUE, log.p = FALSE)

qt(p, df, ncp, lower.tail = TRUE, log.p = FALSE)
rt(n, df, ncp)

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If `length(n) > 1`, the length is taken to be the number required.
df	degrees of freedom (> *0*, maybe non-integer). `df = Inf` is allowed.
ncp	non-centrality parameter *delta*; currently except for `rt()`, only for `abs(ncp) <= 37.62`. If omitted, use the central t distribution.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Example Problem:

Find the 2.5th and 97.5th percentiles of the Student t distribution with 5 degrees of freedom.

Solution

We apply the quantile function qt of the Student t distribution against the decimal values 0.025 and 0.975.

```
> qt(c(.025, .975), df=5)   # 5 degrees of freedom

[1] -2.5706  2.5706
```

Answer

The 2.5th and 97.5th percentiles of the Student t distribution with 5 degrees of freedom are -2.5706 and 2.5706 respectively.

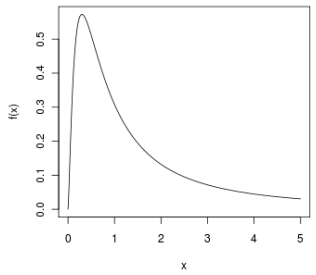
5.1.7 F Distribution:

If V 1 and V 2 are two independent random variables having the Chi-Squared distribution with m1 and m2 degrees of freedom respectively, then the following quantity follows an F

distribution with m1 numerator degrees of freedom and m2 denominator degrees of freedom, i.e., (m1,m2) degrees of freedom.

$$F = \frac{V_1/m_1}{V_2/m_2} \sim F_{(m_1,m_2)}$$

Here is a graph of the F distribution with (5, 2) degrees of freedom.



R has four in-built functions to generate F distribution. They are described below

Density, distribution function, quantile function and random generation for the F distribution with df1 and df2 degrees of freedom (and optional non-centrality parameter ncp).

df(x, df1, df2, ncp, log = FALSE)
pf(q, df1, df2, ncp, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2, ncp, lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2, ncp)

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
df1, df2	degrees of freedom. Inf is allowed.
ncp	non-centrality parameter. If omitted the central F is assumed.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Example Problem

Find the 95th percentile of the F distribution with (5, 2) degrees of freedom.

Solution

We apply the quantile function qf of the F distribution against the decimal value 0.95.

> qf(.95, df1=5, df2=2)

[1] 19.296

Answer

The 95th percentile of the F distribution with (5, 2) degrees of freedom is 19.296.

5.2 Basic Statistics: Correlation and Covariance

5.2.1 Covariance:

Covariance is a statistical term, defined as a systematic relationship between a pair of random variables wherein a change in one variable reciprocated by an equivalent change in another variable.

Covariance can take any value between -∞ to +∞, wherein the negative value is an indicator of negative relationship whereas a positive value represents the positive relationship. Further, it ascertains the linear relationship between variables. Therefore, when the value is zero, it indicates no relationship. In addition to this, when all the observations of the either variable are same, the covariance will be zero.

The covariance of two variables x and y in a data set measures how the two are linearly related. A positive covariance would indicate a positive linear relationship between the variables, and a negative covariance would indicate the opposite.

The sample covariance is defined in terms of the sample means as:

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

Similarly, the population covariance is defined in terms of the population mean μ_x, μ_y as:

$$\sigma_{xy} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y)$$

Problem

Find the covariance of eruption duration and waiting time in the data set faithful. Observe if there is any linear relationship between the two variables.

Solution

We apply the cov function to compute the covariance of eruptions and waiting.

```
> duration = faithful$eruptions   # eruption durations
> waiting = faithful$waiting      # the waiting period
> cov(duration, waiting)          # apply the cov function
[1] 13.978
```

Answer

The covariance of eruption duration and waiting time is about 14. It indicates a positive linear relationship between the two variables.

5.2.2 Correlation Coefficient:

The correlation coefficient of two variables in a data set equals to their covariance divided by the product of their individual standard deviations. It is a normalized measurement of how the two are linearly related.

Formally, the sample correlation coefficient is defined by the following formula, where s_x and s_y are the sample standard deviations, and s_{xy} is the sample covariance.

$$r_{xy} = \frac{s_{xy}}{s_x s_y}$$

Similarly, the population correlation coefficient is defined as follows, where σ_x and σ_y are the population standard deviations, and σ_{xy} is the population covariance.

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

If the correlation coefficient is close to 1, it would indicate that the variables are positively linearly related and the scatter plot falls almost along a straight line with positive slope. For -1, it indicates that the variables are negatively linearly related and the scatter plot almost falls along a straight line with negative slope. And for zero, it would indicate a weak linear relationship between the variables.

Problem

Find the correlation coefficient of eruption duration and waiting time in the data set faithful. Observe if there is any linear relationship between the variables.

Solution

We apply the cor function to compute the correlation coefficient of eruptions and waiting.

```
> duration = faithful$eruptions   # eruption durations
> waiting = faithful$waiting      # the waiting period
```

```
> cor(duration, waiting)          # apply the cor function
[1] 0.90081
```

Answer

The correlation coefficient of eruption duration and waiting time is 0.90081. Since it is rather close to 1, we can conclude that the variables are positively linearly related.

Difference between Covariance & Correlation

BASIS FOR COMPARISON	COVARIANCE	CORRELATION
Meaning	Covariance is a measure indicating the extent to which two random variables change in tandem.	Correlation is a statistical measure that indicates how strongly two variables are related.
What is it?	Measure of correlation	Scaled version of covariance
Values	Lie between $-\infty$ and $+\infty$	Lie between -1 and +1
Change in scale	Affects covariance	Does not affects correlation
Unit free measure	No	Yes

5.3 T-Tests:

The R function t.test() can be used to perform both one and two sample t-tests on vectors of data.

The function contains a variety of options and can be called as follows:

> t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"), mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95)

Arguments:

Here x is a numeric vector of data values and y is an optional numeric vector of data values. If y is excluded, the function performs a one-sample t-test on the data contained in x, if it is included it performs a two-sample t-tests using both x and y.

The option mu provides a number indicating the true value of the mean (or difference in means if you are performing a two sample test) under the null hypothesis. The option alternative is a character string specifying the alternative hypothesis, and must be one of the following: "two.sided" (which is the default), "greater" or "less" depending on whether the alternative hypothesis is that the mean is different than, greater than or less than mu, respectively. For example the following call:

> t.test(x, alternative = "less", mu = 10) performs a one sample t-test on the data contained in x where the null hypothesis is that =10 and the alternative is that <10.

The option paired indicates whether or not you want a paired t-test (TRUE = yes and FALSE = no). If you leave this option out it defaults to FALSE.

The option var.equal is a logical variable indicating whether or not to assume the two variances as being equal when performing a two-sample t-test. If TRUE then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used. If you leave this option out it defaults to FALSE.

Finally, the option conf.level determines the confidence level of the reported confidence interval for in the one-sample case and 1- 2 in the two-sample case.

A. One-sample t-tests

Ex. An outbreak of Salmonella-related illness was attributed to ice cream produced at a certain factory. Scientists measured the level of Salmonella in 9 randomly sampled batches of ice cream. The levels (in MPN/g) were 0.593 0.142 0.329 0.691 0.231 0.793 0.519 0.392 0.418

Is there evidence that the mean level of Salmonella in the ice cream is greater than 0.3 MPN/g?

Answer:

Let be the mean level of Salmonella in all batches of ice cream. Here the hypothesis of interest can be expressed as:

H0: = 0.3

Ha: > 0.3

Hence, we will need to include the options alternative="greater", mu=0.3. Below is the relevant R-code:

> x = c(0.593, 0.142, 0.329, 0.691, 0.231, 0.793, 0.519, 0.392, 0.418)

> t.test(x, alternative="greater", mu=0.3)

One Sample t-test

data: x

t = 2.2051, df = 8, p-value = 0.02927

Alternative hypothesis: true mean is greater than 0.3

From the output we see that the p-value = 0.029. Hence, there is moderately strong evidence that the mean Salmonella level in the ice cream is above 0.3 MPN/g.

B. Two-sample t-tests

Ex. 6 subjects were given a drug (treatment group) and an additional 6 subjects a placebo (control group). Their reaction time to a stimulus was measured (in ms). We want to perform a two-sample t-test for comparing the means of the treatment and control groups.

Let 1 be the mean of the population taking medicine and 2 the mean of the untreated population. Here the hypothesis of interest can be expressed as:

H0: 1- 2=0

Ha: 1- 2<0

Here we will need to include the data for the treatment group in x and the data for the control group in y. We will also need to include the options alternative="less", mu=0. Finally, we need to decide whether or not the standard deviations are the same in both groups.

Below is the relevant R-code when assuming equal standard deviation:

> Control = c(91, 87, 99, 77, 88, 91)

> Treat = c(101, 110, 103, 93, 99, 104)

> t.test(Control,Treat,alternative="less", var.equal=TRUE)

Two Sample t-test

data: Control and Treat

t = -3.4456, df = 10, p-value = 0.003136

alternative hypothesis: true difference in means is less than 0.

5.4 ANOVA - analysis of variance

The analysis of variance is a commonly used method to determine differences between several samples. R provides a function to conduct ANOVA so: aov(model, data)

The first stage is to arrange your data in a .CSV file. Use a column for each variable and give it a meaningful name. Don't forget that variable names in R can contain letters and numbers but the only punctuation allowed is a period. You need to set out your data file so that each column represents a factor in your analysis. Usually the 1st column will be your dependent variable (i.e. what you are actually measuring) and subsequent columns would be the independent factors (e.g. site, treatment).

The second stage is to read your data file into memory and give it a sensible name.

The next stage is to attach your data set so that the individual variables are read into memory.

Finally we need to define the model and run the analysis.

ANOVA models

We set-up our anova in a general way: dependent ~ explanatory1... explanatory2...

The model can take a variety of forms:

Model	**Meaning**
y ~ x1	y is explained by x1 only, a one-way anova
y ~ x1 + x2	y is explained by x1 and x2, a two-way anova
y ~ x1 + x2 + x3	y is explained by x1, x2 and x3, a 3-way anova
y ~ x1 * x2	y is explained by x1, x2 and also by the interaction between them

In reality we would give the variables more meaningful names. However, we can see that is pretty simple to alter our basic model to cope with more complex analyses.

ANOVA Step by Step

First create your data file. Use a spreadsheet and make each column a variable. Each row is a replicate. The first row should contain the variable names. Save this as a .CSV file		
Read your data into R and assign a variable to it. This opens up a window and you select your file.	your.data	= read.csv(file.choose())
Allow R to read the variables within the data file.		attach(your.data)
Decide on the anova model and run the analysis	your.aov	= aov(dependent ~ explanatory)
View the result		summary(your.aov)
Carry out pairwise post-hoc testing using Tukey HSD test		TukeyHSD(your.aov)

UNIT-6: Linear Models in R

Linear Models, Simple Linear Regression, -Multiple Regression Generalized Linear Models, Logistic Regression, - Poisson Regression- other Generalized Linear Models-Survival Analysis, Nonlinear Models, Splines- Decision- Random Forests

6.0. Simple Linear Regression:

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 other variable is called response variable whose value is derived from the predictor variable.

In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

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
```

6.1. Multiple Regression:

Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is −

```
y = a + b1x1 + b2x2 +...bnxn
```

Following is the description of the parameters used −

- **y** is the response variable.
- **a, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

We create the regression model using the **lm()** function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

lm() Function

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

Syntax

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

```
lm(y ~ x1+x2+x3...,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between the response variable and predictor variables.
- **data** is the vector on which the formula will be applied.

Example

Input Data

Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.

The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

```
input <- mtcars[,c("mpg","disp","hp","wt")]
print(head(input))
```

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

```
                   mpg   disp   hp    wt
Mazda RX4         21.0    160    110   2.620
Mazda RX4 Wag     21.0  160    110   2.875
Datsun 710        22.8    108     93  2.320
Hornet 4 Drive     21.4    258    110   3.215
Hornet Sportabout  18.7  360    175   3.440
Valiant             18.1  225    105   3.460
```

Create Relationship Model & get the Coefficients

```
input <- mtcars[,c("mpg","disp","hp","wt")]
# Create the relationship model.
model <- lm(mpg~disp+hp+wt, data = input)
# Show the model.
print(model)
# Get the Intercept and coefficients as vector elements.
cat("# # # # The Coefficient Values # # # ","\n")
a <- coef(model)[1]
print(a)
Xdisp <- coef(model)[2]
Xhp <- coef(model)[3]
Xwt <- coef(model)[4]
print(Xdisp)
print(Xhp)
print(Xwt)
```

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

```
Call:
lm(formula = mpg ~ disp + hp + wt, data = input)

Coefficients:
(Intercept)        disp          hp          wt
```

```
 37.105505      -0.000937        -0.031157    -3.800891

# # # # The Coefficient Values # # #
(Intercept)
  37.10551
       disp
-0.0009370091
       hp
-0.03115655
      wt
-3.800891
```

Create Equation for Regression Model

Based on the above intercept and coefficient values, we create the mathematical equation.

```
Y = a+Xdisp.x1+Xhp.x2+Xwt.x3
or
Y = 37.15+(-0.000937)*x1+(-0.0311)*x2+(-3.8008)*x3
```

Apply Equation for predicting New Values

We can use the regression equation created above to predict the mileage when a new set of values for displacement, horse power and weight is provided.

For a car with disp = 221, hp = 102 and wt = 2.91 the predicted mileage is −

```
Y = 37.15+(-0.000937)*221+(-0.0311)*102+(-3.8008)*2.91 = 22.7104
```

6.2. Generalized Linear Model:

glm() function is used to fit the generalized linear model with the following syntax:

glm(formula, family=familytype(link=linkfunction),data= _)

Family	Default Link Function
Binomial	(link=”logit”)
Gaussian	(link=”identity”)
Gamma	(link=”inverse”)
Inverse.Gaussian	(link=”log”)
quasi	(link=”identity”, variance=”constant”)
quasibinomial	(link=”logit”)
quasipoisson	(link=”log”)

The generalized linear models can be further classified as logistic regression, poisson regression and survival analysis

6.2.1. Logistic Regression:

The Logistic Regression is a regression model in which the response variable (dependent variable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is −

```
y = 1/(1+e^-(a+b1x1+b2x2+b3x3+...))
```

Following is the description of the parameters used −

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

The function used to create the regression model is the **glm()**function.

Syntax

The basic syntax for **glm()** function in logistic regression is −

```
glm(formula,data,family)
```

Following is the description of the parameters used −

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. It's value is binomial for logistic regression.

Example

The in-built data set "mtcars" describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

```
# Select some columns form mtcars.

input <- mtcars[,c("am","cyl","hp","wt")]

print(head(input))
```

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

```
                  am   cyl  hp    wt
Mazda RX4          1   6    110   2.620
Mazda RX4 Wag      1   6    110   2.875
Datsun 710         1   4     93   2.320
Hornet 4 Drive     0   6    110   3.215
Hornet Sportabout  0   8    175   3.440
Valiant            0   6    105   3.460
```

Create Regression Model

We use the **glm()** function to create the regression model and get its summary for analysis.

```
input <- mtcars[,c("am","cyl","hp","wt")]


am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)


print(summary(am.data))
```

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

```
Call:
glm(formula = am ~ cyl + hp + wt, family = binomial, data = input)

Deviance Residuals:
```

```
    Min       1Q   Median       3Q      Max
-2.17272  -0.14907 -0.01464  0.14116  1.27641

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 19.70288    8.11637   2.428   0.0152 *
cyl          0.48760    1.07162   0.455   0.6491
hp           0.03259    0.01886   1.728   0.0840 .
wt          -9.14947    4.15332  -2.203   0.0276 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.2297  on 31  degrees of freedom
Residual deviance:  9.8415  on 28  degrees of freedom
AIC: 17.841

Number of Fisher Scoring iterations: 8
```

Conclusion

In the summary as the p-value in the last column is more than 0.05 for the variables "cyl" and "hp", we consider them to be insignificant in contributing to the value of the variable "am". Only weight (wt) impacts the "am" value in this regression model.

6.2.2. Poisson Regression:

Poisson Regression involves regression models in which the response variable is in the form of counts and not fractional numbers. For example, the count of number of births or number of wins in a football match series. Also the values of the response variables follow a Poisson distribution.

The general mathematical equation for Poisson regression is −

```
log(y) = a + b1x1 + b2x2 + bnxn.....
```

Following is the description of the parameters used −

- **y** is the response variable.
- **a** and **b** are the numeric coefficients.

- **x** is the predictor variable.

The function used to create the Poisson regression model is the **glm()** function.

Syntax

The basic syntax for **glm()** function in Poisson regression is −

```
glm(formula,data,family)
```

Following is the description of the parameters used in above functions −

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. It's value is 'Poisson' for Logistic Regression.

Example

We have the in-built data set "warpbreaks" which describes the effect of wool type (A or B) and tension (low, medium or high) on the number of warp breaks per loom. Let's consider "breaks" as the response variable which is a count of number of breaks. The wool "type" and "tension" are taken as predictor variables.

Input Data

```
input <- warpbreaks
print(head(input))
```

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

```
  breaks  wool  tension
1   26      A     L
2   30      A     L
3   54      A     L
4   25      A     L
5   70      A     L
```

```
6    52      A     L
```

Create Regression Model

```
output <-glm(formula = breaks ~ wool+tension,
                 data = warpbreaks,
                family = poisson)
print(summary(output))
```

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

```
Call:
glm(formula = breaks ~ wool + tension, family = poisson, data = warpbreaks)

Deviance Residuals:
    Min       1Q     Median       3Q      Max
 -3.6871  -1.6503  -0.4269     1.1902   4.2616

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.69196    0.04541  81.302  < 2e-16 ***
woolB       -0.20599    0.05157  -3.994 6.49e-05 ***
tensionM    -0.32132    0.06027  -5.332 9.73e-08 ***
tensionH    -0.51849    0.06396  -8.107 5.21e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 297.37  on 53  degrees of freedom
Residual deviance: 210.39  on 50  degrees of freedom
AIC: 493.06

Number of Fisher Scoring iterations: 4
```

In the summary we look for the p-value in the last column to be less than 0.05 to consider an impact of the predictor variable on the response variable.

6.3. Other Generalized Linear Models:

Survival Analaysis:

Survival analysis deals with predicting the time when a specific event is going to occur. It is also known as failure time analysis or analysis of time to death. For example predicting the number of days a person with cancer will survive or predicting the time when a mechanical system is going to fail.

The R package named **survival** is used to carry out survival analysis. This package contains the function **Surv()** which takes the input data as a R formula and creates a survival object among the chosen variables for analysis. Then we use the function **survfit()** to create a plot for the analysis.

Install Package

```
install.packages("survival")
```

Syntax

The basic syntax for creating survival analysis in R is −

```
Surv(time,event)
survfit(formula)
```

Following is the description of the parameters used −

- **time** is the follow up time until the event occurs.
- **event** indicates the status of occurrence of the expected event.
- **formula** is the relationship between the predictor variables.

Example

We will consider the data set named "pbc" present in the survival packages installed above. It describes the survival data points about people affected with primary biliary cirrhosis (PBC) of

the liver. Among the many columns present in the data set we are primarily concerned with the fields "time" and "status". Time represents the number of days between registration of the patient and earlier of the event between the patient receiving a liver transplant or death of the patient.

```
# Load the library.
library("survival")


# Print first few rows.
print(head(pbc))
```

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

```
  id time status trt      age sex ascites hepato spiders edema bili chol
1  1  400      2   1 58.76523   f       1      1       1   1.0 14.5  261
2  2 4500      0   1 56.44627   f       0      1       1   0.0  1.1  302
3  3 1012      2   1 70.07255   m       0      0       0   0.5  1.4  176
4  4 1925      2   1 54.74059   f       0      1       1   0.5  1.8  244
5  5 1504      1   2 38.10541   f       0      1       1   0.0  3.4  279
6  6 2503      2   2 66.25873   f       0      1       0   0.0  0.8  248
  albumin copper alk.phos    ast trig platelet protime stage
1    2.60    156   1718.0 137.95  172      190    12.2     4
2    4.14     54   7394.8 113.52   88      221    10.6     3
3    3.48    210    516.0  96.10   55      151    12.0     4
4    2.54     64   6121.8  60.63   92      183    10.3     4
5    3.53    143    671.0 113.15   72      136    10.9     3
6    3.98     50    944.0  93.00   63       NA    11.0     3
```

From the above data we are considering time and status for our analysis.

Applying Surv() and survfit() Function

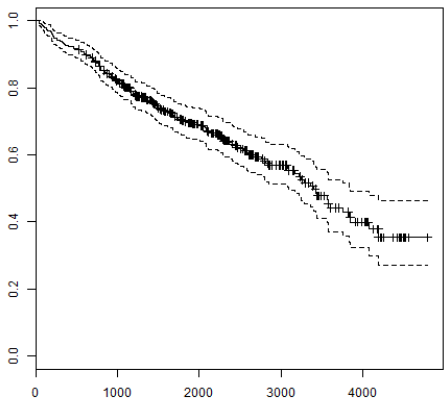
Now we proceed to apply the **Surv()** function to the above data set and create a plot that will show the trend.

```
# Load the library.
library("survival")
```

```
# Create the survival object.
survfit(Surv(pbc$time,pbc$status == 2)~1)

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

# Plot the graph.
plot(survfit(Surv(pbc$time,pbc$status == 2)~1))

# Save the file.
dev.off()
```

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

```
Call: survfit(formula = Surv(pbc$time, pbc$status == 2) ~ 1)

      n  events  median 0.95LCL 0.95UCL
    418     161    3395    3090    3853
```



The trend in the above graph helps us predicting the probability of survival at the end of a certain number of days.

6.4. Non Linear Models in R:

The non linear models include the splines, random forest, decision trees, generalized additive modeling.

6.4.1. Splines:

The data which exhibits non linear behavior tends to fit a smoothing spline.This helps in making important predictions on newly available data.The spline can be represented with a function f which contains N different functions.

The basic sytax of spline yields:

```
smooth.spline(x, y = NULL, w = NULL, df, spar = NULL, lambda = NULL, cv = FALSE,
        all.knots = FALSE, nknots = .nknots.smspl,
        keep.data = TRUE, df.offset = 0, penalty = 1,
        control.spar = list(), tol = 1e-6 * IQR(x), keep.stuff = FALSE)
```

6.4.2. DecisionTrees:

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 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.

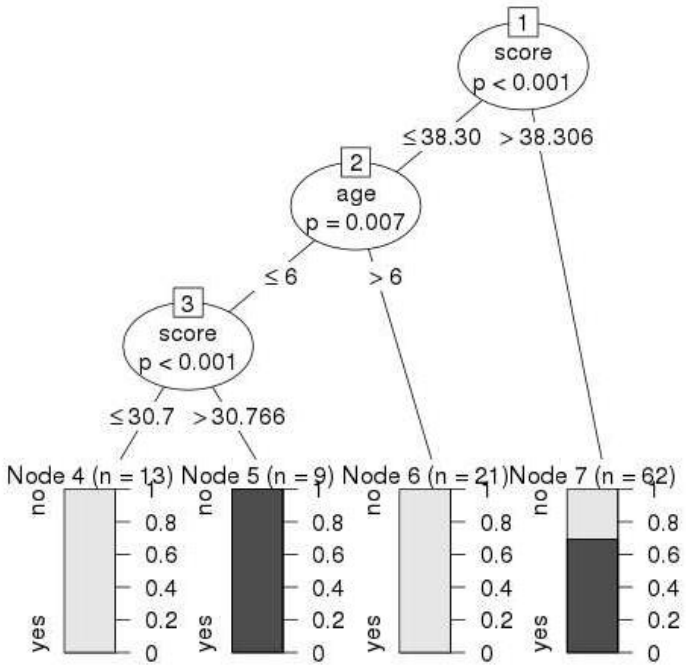
```
# Load the party package. It will automatically load other dependent packages.
```

```
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)


# Save the file.

dev.off()
```

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
```



Conclusion

From the decision tree shown above we can conclude that anyone whose readingSkills score is less than 38.3 and age is more than 6 is not a native Speaker.

6.4.3. Random Forests:

In the random forest approach, a large number of decision trees are created. Every observation is fed into every decision tree. The most common outcome for each observation is used as the final output. A new observation is fed into all the trees and taking a majority vote for each classification model.

An error estimate is made for the cases which were not used while building the tree. That is called an **OOB (Out-of-bag)**error estimate which is mentioned as a percentage.

The R package **"randomForest"** is used to create random forests.

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("randomForest)
```

The package "randomForest" has the function **randomForest()**which is used to create and analyze random forests.

Syntax

The basic syntax for creating a random forest in R is −

```
randomForest(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.

Here is the sample data.

```
# Load the party package. It will automatically load other required 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 **randomForest()** function to create the decision tree and see it's graph.

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

library(party)

library(randomForest)


# Create the forest.

output.forest <- randomForest(nativeSpeaker ~ age + shoeSize + score,

          data = readingSkills)


# View the forest results.

print(output.forest)


# Importance of each predictor.

print(importance(fit,type = 2))
```

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

```
Call:
 randomForest(formula = nativeSpeaker ~ age + shoeSize + score,
               data = readingSkills)
             Type of random forest: classification
```

```
                Number of trees: 500
No. of variables tried at each split: 1

       OOB estimate of  error rate: 1%
Confusion matrix:
    no yes class.error
no  99   1        0.01
yes  1  99        0.01
        MeanDecreaseGini
age              13.95406
shoeSize         18.91006
score            56.73051
```

Conclusion

From the random forest shown above we can conclude that the shoesize and score are the important factors deciding if someone is a native speaker or not. Also the model has only 1% error which means we can predict with 99% accuracy.