LAKIREDDY BALI REDDY COLLEGE OF ENGINEERING

(AUTONOMOUS)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

R PROGRAMMING LAB MANUAL (R20)

Vision of the Department

Vision of the Department

The Computer Science & Engineering aims at providing continuously stimulating educational environment to its students for attaining their professional goals and meet the global challenges.

Mission of the Department

- **DM1:** To develop a strong theoretical and practical background across the computer science discipline with an emphasis on problem solving.
- **DM2:** To inculcate professional behaviour with strong ethical values, leadership qualities, innovative thinking and analytical abilities into the student.
- **DM3:** Expose the students to cutting edge technologies which enhance their employability and knowledge.
- **DM4:** Facilitate the faculty to keep track of latest developments in their research areas and encourage the faculty to foster the healthy interaction with industry.

Program Educational Objectives (PEOs)

- **PEO1:** Pursue higher education, entrepreneurship and research to compete at global level.
- **PEO2:** Design and develop products innovatively in computer science and engineering and in other allied fields.
- **PEO3:** Function effectively as individuals and as members of a team in the conduct of interdisciplinary projects; and even at all the levels with ethics and necessary attitude.
- **PEO4:** Serve ever-changing needs of society with a pragmatic perception.

PROGRAMME OUTCOMES (POs):

PO 1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO 2	**Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO 3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO 4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex

	engineering activities with an understanding of the limitations.
PO 6	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO 8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO 9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO 10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO 11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO 12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAMME SPECIFIC OUTCOMES (PSOs):

PSO 1	The ability to apply Software Engineering practices and strategies in software project development using open-source programming environment for the success of organization.
PSO 2	The ability to design and develop computer programs in networking, web applications and IoT as per the society needs.
PSO 3	To inculcate an ability to analyze, design and implement database applications.

Week 1:

1. a) Installing R and RStudio

R is a programming language and software environment for statistical 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 CoreTeam.

This programming language was named **R**, based on the first letter of first name of the two Rauthors (Robert Gentleman and Ross Ihaka)

R is often used for statistical computing and graphical presentation to analize and visualize data.

Why Use R?

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has a large community support
- It has many packages (libraries of functions) that can be used to solve different problems

To Install R and R Packages

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

To Install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Mac version, save the .dmg file on your computer, double-click it to open, and then drag and drop it to your applications folder.

1.b) Basic functionality of R, variable, data types in R

If you type 5 + 5, and press enter, you will see that R outputs 10.

Example
```
5 + 5
```

Output:
[1] 10

R Syntax

Syntax
To output text in R, use single or double quotes:
Example
```
"Hello World!"
```

To output numbers, just type the number (without quotes):
Example
```
5
10
25
```

To do simple calculations, add numbers together:
Example
```
5 + 5
```

R Print
Print
Unlike many other programming languages, you can output code in R without using a print function:
Example
```
"Hello World!"
```

However, R does have a print() function available if you want to use it. This might be useful if you are familiar with other programming languages, such as Python, which often uses the print()function to output code.
Example
```
print("Hello World!")
```

And there are times you must use the print()function to output code, for example whenworking with forloops (which you will learn more about in a later chapter):
Example
```
for (x in 1:10)
  {print(x)
}
```

R Comments

Comments
Comments can be used to explain R code, and to make it more readable. It can also be used to prevent execution when testing alternative code.
Comments starts with a #. When executing the R-code, R will ignore anything that starts with #.

This example uses a comment before a line of code:

Example

```
# This is a comment
"Hello World!"
```

This example uses a comment at the end of a line of code:

Example

```
"Hello World!" # This is a comment
```

Multiline Comments

Unlike other programming languages, such as Java, there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments:

Example

```
# This is a comment
# written in
# more than just one line
"Hello World!"
```

R Variables

Creating Variables in R

Variables are containers for storing data values.

R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the <-sign. To output (or print)the variable value, just type the variable name:

Example

```
name <- "John"
age <- 40
name # output "John"age
# output 40
```

From the example above, nameand ageare **variables**, while "John"and 40are **values**.

In other programming language, it is common to use = as an assignment operator. In R, we can use both = and <- as assignment operators.

However, <- is preferred in most cases because the = operator can be forbidden in some context in R.

Print / Output Variables

Compared to many other programming languages, you do not have to use a function to print/output variables in R. You can just type the name of the variable:

Example

```
name <- "John Doe"
name # auto-print the value of the name variable
```

However, R does have a print() function available if you want to use it. This might be useful if you are familiar with other programming languages, such as Python, which often usea print() function to output variables.

Example

```
name <- "John Doe"
print(name) # print the value of the name variable
```

And there are times you must use the print()function to output code, for example whenworking with forloops (which you will learn more about in a later chapter):

Example

```
for (x in 1:10)
  {print(x)
}
```

Output:

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

[1] 7

[1] 8

[1] 9

[1] 10

Multiple Variables

R allows you to assign the same value to multiple variables in one line:

Example

```
# Assign the same value to multiple variables in one line
var1 <- var2 <- var3 <- "Orange"
# Print variable values
var1
var2
var3
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for R variables are:

- A variable name must start with a letter and can be a combination of letters, digits, period(.)and underscore(_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

```
# Legal variable
names:myvar<- "John"
my_var<- "John"
myVar <- "John"
```

```
MYVAR <- "John"
myvar2 <- "John"
.myvar<- "John"
# Illegal variable names:
2myvar <-
"John" my-var <-
"John" my var <-
"John"
_my_var<- "John"
my_v@ar<-
"John"TRUE <-
"John"
```

Remember that variable names are case-sensitive!

Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

In R, variables do not need to be declared with any particular type, and can even change type after they have been set:

Example

```
my_var<- 30 # my_var is type of

numericmy_var

Output:

[1] 30

my_var<- "Sally"  # my_var is now of type character (aka string)

my_var

Output

:

[1] "Sally"
```

R has a variety of data types and object classes.

Basic Data Types

Basic data types in R can be divided into the following types:

- numeric- (10.5, 55, 787)
- integer- (1L, 55L, 100L, where the letter "L" declares this as an integer)
- complex- (9 + 3i, where "i" is the imaginary part)
- character(a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")
- logical(a.k.a. boolean) - (TRUE or FALSE)

Use the **class()**function to check the data type of a variable:

Example

```
# numeric
x <- 10.5
class(x)
Output:

[1] "numeric"
```

```
# integer
x <-
1000L
class(x)
```

Output:

```
[1] "integer"
# complex
x <- 9i + 3
class(x)
```

Output:

[1] "complex"

```
# character/string
x <- "R is
exciting"class(x)
```

Output:

```
[1] "Character"

# logical/boolean
x <- TRUE
class(x)
```

Output:

```
[1] "logical"
```

R Numbers

Numbers

There are three number types in R:

- numeric
- integer
- complex

Variables of number types are created when you assign a value to them:

Example

```
x <- 10.5  # numeric
y <- 10L   # integer
z <- 1i    # complex
```

Output:

```
> x

[1] 10.5

> y

[1] 10

> z

[1] 0+1i
```

Numeric

A numericdata type is the most common type in R, and contains any number with or withouta decimal, like: 10.5, 55, 787:

Example

```
x <- 10.5
y <- 55
# Print values of x and y
x
y
```

Output:

```
> x

[1] 10.5

> y

[1] 55
```

```
# Print the class name of x and y
class(x)
class(y)
```

Output:

```
> class(x)

[1] "numeric"

> class(y)

[1] "numeric"
```

Integer

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an integervariable, you mustuse the letter Lafter the integer value:

Example

```
x <-
1000Ly
<- 55L
# Print values of x and y
x
y
```

Output:

```
> x

[1] 1000

> y

[1] 55
```

```
# Print the class name of x and y
class(x)
class(y)
```

Output:

> class(x)

[1] "integer"

> class(y)

[1] "integer"

Complex

A complexnumber is written with an "i" as the imaginary part:

Example

```
x <- 3+5i
y <- 5i
# Print values of x and y
x
y
```

Output:

> x

[1] 3+5i

> y

[1] 0+5i

```
# Print the class name of x and y
class(x)
class(y)
```

Output:

>class(x)

[1] "complex"

> class(y)

[1] "complex"

Type Conversion

You can convert from one type to another with the following functions:

- as.numeric()
- as.integer()
- as.complex()

Example

```
x <- 1L #
integery <- 2 #
numeric
# convert from integer to numeric:
a <- as.numeric(x)

# convert from numeric to integer:
b <- as.integer(y)
# print values of x and y
x
```

```
y
# print the class name of a and b
class(a)
class(b)
```

Output:

```
print values of x and y

> x

[1] 1

> y

[1] 2

 # print the class name of a and b

> class(a)

[1] "numeric"

> class(b)

[1] "integer"
```

Week 2:

2(a) Implement R script to show the usage of various operators available in R language.

R Script:

```
a=40
b=20
print("Arthimetic Operators")
print(paste("addition=",(a+b)))
print(paste("subtraction =",a-b))
print(paste("multiplication=",a*b))
print(paste("division of numbers",a/b))
print(paste("modulo of numbers",a%%b))
print(paste("Quotient of number",a%/%b))
print(paste("power of number=",a^b))
print("Relational Operators")
print(paste("Checks Greater:",a>b))
print(paste("Checks lessthan:",a<b))
print(paste("Checks equal to:",a==b))
print(paste("Checks Greater or equal to:",a>=b))
print(paste("Checks less than or equal to:",a<=b))
print(paste("Checks not equal or not:",a!=b))
print("Logical operators")
print(paste("And operation",a&b))
print(paste("OR operation",a|b))
print(paste("NOT operation of a",!a))
print(paste("NOT operation of b",!b))
print(paste("Logical And
operation",a&&b))print(paste("Logical OR
operation",a||b)) print("Miscellaneous
Operators") print("Colon operator")
print(2:8)
```

Output:

```
[1] "Arthimetic Operators"
 [1] "addition= 60"
 [1] "subtraction = 20"
 [1] "multiplication= 800"
 [1] "division of numbers 2"
 [1] "modulo of numbers 0"
 [1] "Quotient of number 2"
 [1] "power of number= 1.099511627776e+32"
[1] "Relational Operators"
 [1] "Checks Greater: TRUE"
 [1] "Checks lessthan: FALSE"
 [1] "Checks equal to: FALSE"
 [1] "Checks Greater or equal to: TRUE"
 [1] "Checks less than or equal to: FALSE"
 [1] "Checks not equal or not: TRUE"
 [1] "Logical operators"
 [1] "And operation TRUE"
 [1] "OR operation TRUE"
 [1] "NOT operation of a FALSE"
 [1] "NOT operation of b FALSE"
 [1] "Logical And operation TRUE"
 [1] "Logical OR operation TRUE"
 [1] "Miscellaneous Operators"
 [1] "Colon operator"
 [1] 2 3 4 5 6 7 8
```

2(b) Implement R script to read person‘s age from keyboard and display whether he is eligible for voting or not.

```
age = readline(prompt="Enter the Age: ")
age = as.integer(age)
```

```
if(age>=18){
  print(paste("Eligible to vote", age))
}else{
  print(paste("Not Eligible to vote", age))
}
```

Output:

```
Enter the Age: 21
[1] "Eligible to vote 21"
Enter the Age: 17
[1] "Not Eligible to vote 17"
```

2(c) Implement R script to find biggest number between two numbers.

```
# To Implement R script to find biggest between two numbers
a = as.integer(readline(prompt = "Enter the Number 1: "))
b = as.integer(readline(prompt = "Enter the Number 2: "))
if(a>b)
{
  sprintf("a value %d is big", a)
}else
{
  sprintf(" value %d is big", b)
}
```

Output:

```
Enter the Number 1: 10
Enter the Number 2: 5
[1] "a value 10 is big"
```

2(d) Implement R script to check the given year is leap year or not.

ALGORITHM

STEP 1: prompting appropriate messages to the user
STEP 2: take user input using `readline()` into variables **year**
STEP 3: check if year is exactly divisible by **4,100,400** gives a remainder of **0**
STEP 4: if remainder is a non-zero print year is not a leap year.
STEP 5: if remainder is zero print year is a leap year.

```
# Program to check if the input year is a leap year or not
year = as.integer(readline(prompt ="Enter a year: "))
if((year %% 4) == 0) {
  if((year %% 100) == 0) {
    if((year %% 400) == 0) {
      print(paste(year," is a Leap Year"))
    } else {
      print(paste(year," is not a Leap Year"))
    }
    } else  {
      print(paste(year," is a Leap Year"))
    }
    } else  {
      print(paste(year," is  not a Leap Year"))
    }
```

 # Program to check if the input year is a leap year or not

```
year = as.integer(readline(prompt ="Enter a year: "))
if((year %% 4) == 0 & (year %% 100) == 0 & (year %% 400) == 0 ) {
       print(paste(year," is a Leap Year"))
    } else {
      print(paste(year," is not a Leap Year"))
    }
```

Output:

```
Enter a year: 1900
[1] "1900 is not a leap year"


Enter a year: 2000
[1] "2000 is a leap year"
```

Week-3

3(a) Implement R Script to create a list.

Lists are the R objects which contain elements of different types like − numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

Creating a List

Following is an example to create a list containing strings, numbers, vectors and a logical values.

```
# Create a list containing strings, numbers, vectors and logical values
list_data <- list("Red","Green",c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

Output:

```
print(list_data)

[[1]]

[1]

"Red"

[[2]]

[1] "Green"

[[3]]

[1] 21 32 11

[[4]]

[1] TRU

E[[5]]

[1] 51.23

[[6]]

[1] 119.1
```

3(b) Implement R Script to access elements in the list.

Giving a name to list elements

There are only three steps to print the list data corresponding to the name:

1. Creating a list.
2. Assign a name to the list elements with the help of names() function.
3. Print the list data.

Example: 1

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),list("green",12.3))
# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
# Show the list.
print(list_data)
```

Output:

```
print(list_data)
$`1st Quarter`
[1] "Jan" "Feb" "Mar"
$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3
```

Accessing List Elements

- Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

Example:2

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list.
print(list_data[1])

# Access the thrid element. As it is also a list, all its elements will be printed.
print(list_data[3])

# Access the list element using the name of the element.
print(list_data$A_Matrix)
```

Output:

```
print(list_data[1])

$`1st Quarter`

[1] "Jan" "Feb" "Mar"

# Access the third element. As it is also a list, all its elements will be printed.

print(list_data[3])

$`A Inner list`

$`A Inner list`[[1]]

[1] "green"
```

$`A Inner

list`[[2]][1] 12.3

Access the list element using the name of the element.

print(list_data$A_Matrix)

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

[1,] 3 5 -2

[2,] 9 1 8

3(c) Implement R Script to merge two or more lists. Implement R Script to perform matrix operation.

Implement R Script to merge two or more lists.

Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")
# Merge the two lists.
merged.list <- c(list1,list2)

# Print the merged list.
print(merged.list)
```

Output:

print(merged.list)

[[1]]

[1] 1

[[2]]

[1] 2

[[3]]

[1] 3

[[4]]

[1]

"Sun"

[[5]]

[1] "Mon"

[[6]]

[1] "Tue"

Implement R Script to perform matrix operation.

R Matrix

In R, a two-dimensional rectangular data set is known as a matrix. A matrix is created with the help of the vector input to the matrix function. On R matrices, we can perform addition, subtraction, multiplication, and division operation.

In the R matrix, elements are arranged in a fixed number of rows and columns. The matrix elements are the real numbers.

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

Syntax

```
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

```
#Arranging elements sequentially by row.

P <- matrix(c(5:16), nrow = 4, byrow =

TRUE)print(P)

# Arranging elements sequentially by

column.Q <- matrix(c(3:14), nrow = 4, byrow

= FALSE) print(Q)

# Defining the column and row names.

row_names = c("row1", "row2", "row3", "row4")

col_names = c("col1", "col2", "col3")

R <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))

print(R)
```

Output:

```
print(P)

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

[1,]    5    6    7

[2,]    8    9  10

[3,]  11   12   13

[4,]  14   15   16
```

print(Q)

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

print(R)

```
     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.#

Define the column and row names.

```
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")
```

 # Create the matrix. Arranging elements sequentially by row.

```
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))
print(P)
```

 # 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])
```

Output:

print(P)

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

Matrix operations

In R, we can perform the mathematical operations on a matrix such as addition, subtraction, multiplication, etc.

```
R <- matrix(c(5:16), nrow =
4,ncol=3)S <- matrix(c(1:12), nrow
= 4,ncol=3)# Display two matrices
R and S print(R)
print(S)
#Addition
sum<-R+S
print(sum)
#Subtraction
sub<-R-S
print(sub)
#Multiplicatio
nmul<-R*S
print(mul)
```

#Divisio

ndiv<-

R/S

print(div)

Output:

print(R)

```
     [,1] [,2] [,3]
[1,]    5    9   13
[2,]    6   10   14
[3,]    7   11   15
[4,]    8   12   16
```

print(S)

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

sum<-

R+S

print(sum

)

```
     [,1] [,2] [,3]
[1,]    6   14   22
[2,]    8   16   24
[3,]   10   18   26
[4,]   12   20   28
```

sub<-R-S

print(sub)

```
     [,1] [,2] [,3]
[1,]    4    4    4
[2,]    4    4    4
[3,]    4    4    4
```

[4,] 4 4 4

mul<-R*S

print(mul)

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

[1,] 5 45 117

[2,] 12 60 140

[3,] 21 77 165

[4,] 32 96 192

div<-R/S

print(div)

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

[1,] 5.000000 1.800000 1.444444

[2,] 3.000000 1.666667 1.400000

[3,] 2.333333 1.571429 1.363636

[4,] 2.000000 1.500000 1.333333

Week-4

4(a) Implement R script to perform various operations on vectors.

- In R, a sequence of elements which share the same data type is known as vector.
- A vector supports logical, integer, double, character, complex, or raw data type.
- The elements which are contained in vector known as **components** of the vector.
- We can check the type of vector with the help of the **typeof()** function.

The length is an important property of a vector. A vector length is basically the number of elements in the vector, and it is calculated with the help of the length() function.

Vector is classified into two parts, i.e., **Atomic vectors** and **Lists**. They have three common properties, i.e., **function type, function length**, and **attribute function**.

How to create a vector in R?

- In R, we use c() function to create a vector. This function returns a one-dimensional array or simply vector.
- The c() function is a generic function which combines its argument. All arguments are restricted with a common data type which is the type of the returned value.

There are various other ways to create a vector in R, which are as follows:

1) Using the colon(:) operator

We can create a vector with the help of the colon operator. There is the following syntax to use colon operator:

1. **z<-x:y**

This operator creates a vector with elements from x to y and assigns it to z.

Example:

```
A <- 4: -10
A
```

Output

```
[1]  4 3  2  1  0  -1  -2  -3  -4  -5  -6  -7  -8  -9  -10
```

2) Using the seq() function

In R, we can create a vector with the help of the seq() function. A sequence function creates a sequence of elements as a vector. The seq() function is used in two ways, i.e., by setting step size with ?by' parameter or specifying the length of the vector with the 'length.out' feature.

Example

```
numbers <- seq(from = 0, to = 100, by =
20)numbers
```

Output:

```
[1]  0  20  40  60  80 100
```

Note: The seq() function has three parameters: from is where the sequence starts, to is where the sequence stops, and by is the interval of the sequence.

Atomic vectors in R

In R, there are four types of atomic vectors. Atomic vectors play an important role in Data Science. Atomic vectors are created with the help of **c()** function. These atomic vectors are as follows:

Example

```
# Vector of strings
fruits <- c("banana", "apple", "orange")
# Print fruits
fruits
```

Output:

```
[1] "banana" "apple" "orange"
```

In this example, we create a vector that combines numerical values:

Example

```
# Vector of numerical values
numbers <- c(1, 2, 3)
# Print numbers
numbers
```

Output:

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

To create a vector with numerical values in a sequence, use the :operator:

Example

```
# Vector with numerical values in a sequence
numbers <- 1:10
numbers
```

Output:**[1] 1 2 3 4 5 6 7 8 9 10**

Atomic vectors in R

In R, there are four types of atomic vectors. Atomic vectors play an important role in Data Science. Atomic vectors are created with the help of **c()** function. These atomic vectors are as follows:

1. Numeric vector

The decimal values are known as numeric data types in R. If we assign a decimal value to any variable d, then this d variable will become a numeric type. A vector which contains numeric elements is known as a numeric vector.

Example:

```
d<-45.5
num_vec<-c(10.1, 10.2, 33.2)

d
num_vec

class(d)
class(num_vec)
```

Output:

```
[1] 10.1 10.2 33.2

[1] "numeric"

[1] "numeric"
```

2. Integer vector

A non-fraction numeric value is known as integer data. This integer data is represented by "Int." The Int size is 2 bytes and long Int size of 4 bytes. There is two way to assign an integer value to a variable, i.e., by using as.integer() function and appending of L to the value.

A vector which contains integer elements is known as an integer vector.

Example:

```
d<-as.integer(5)
e<-5L

int_vec<-c(1,2,3,4,5)
int_vec<-as.integer(int_vec)

int_vec1<-c(1L,2L,3L,4L,5L)
class(d)

class(e)
class(int_vec)

class(int_vec1)
```

Output:

```
[1] "integer"

[1] "integer"

[1] "integer"

[1] "integer"
```

3. Character vector

A character is held as a one-byte integer in memory. In R, there are two different ways to create a character data type value, i.e., using as.character() function and by typing string between double quotes("") or single quotes(').

A vector which contains character elements is known as an integer vector.

Example:

```
d<-'shubham'
e<-"Arpita"

f<-65
f<-as.character(f)

d
e

f
char_vec<-c(1,2,3,4,5)

char_vec<-as.character(char_vec)
char_vec1<-c("shubham","arpita","nishka","vaishali")

char_vec
class(d)

class(e)
class(f)

class(char_vec)
class(char_vec1)
```

Output:

```
> d

[1] "shubham"

> e

[1] "Arpita"

> f

[1] "65"

> char_vec

[1] "1" "2" "3" "4" "5"

> class(d)

[1] "character"

> class(e)

[1] "character"

> class(f)
```

```
[1] "character"

> class(char_vec)

[1] "character"

> class(char_vec1)

[1] "character"
```

Accessing elements of vectors

We can access the elements of a vector with the help of vector indexing. Indexing denotes the position where the value in a vector is stored. Indexing will be performed with the help of integer, character, or logic.

1) Indexing with integer vector

On integer vector, indexing is performed in the same way as we have applied in C, C++, and java. There is only one difference, i.e., in C, C++, and java the indexing starts from 0, but in R, the indexing starts from 1. Like other programming languages, we perform indexing by specifying an integer value in square braces [] next to our vector.

Example:

```
seq_vec<-seq(1,4,length.out=6)
seq_vec

seq_vec[2]
```

Output

```
[1] 1.0 1.6 2.2 2.8 3.4 4.0
[1] 1.6
```

2) Indexing with a character vector

In character vector indexing, we assign a unique key to each element of the vector. These keys are uniquely defined as each element and can be accessed very easily. Let's see an example to understand how it is performed.

Example:

```
char_vec<-c("shubham"=22,"arpita"=23,"vaishali"=25)
char_vec

char_vec["arpita"]
```

Output

```
shubham  arpita vaishali
```

```
  22    23    25
arpita
    23
```

3) Indexing with a logical vector

In logical indexing, it returns the values of those positions whose corresponding position has a logical vector TRUE. Let see an example to understand how it is performed on vectors.

Example:

```
a<-c(1,2,3,4,5,6)
a[c(TRUE, FALSE,TRUE,TRUE,FALSE,TRUE)]
```

Output

```
[1] 1 3 4 6
```

Vector Operation

In R, there are various operation which is performed on the vector. We can add, subtract, multiply or divide two or more vectors from each other.

1) Combining vectors

The c() function is not only used to create a vector, but also it is also used to combine two vectors. By combining one or more vectors, it forms a new vector which contains all the elements of each vector. Let see an example to see how c() function combines the vectors.

Example:

```
p <- c(1,2,3,5,7,8)
q <- c("subbu","raju","raju","sankar","rajesh","ramesh")
r <- c(p,q)
r
```

Output:

```
[1] "1"     "2"     "3"     "5"     "7"     "8"     "subbu"  "raju"  "raju"   "sankar" "rajesh" "ramesh"
```

2) Arithmetic operations

We can perform all the arithmetic operation on vectors. The arithmetic operations are performed member-by-member on vectors. We can add, subtract, multiply, or divide two vectors. Let see an example to understand how arithmetic operations are performed on vectors.

Example:

```
a<-c(1,3,5,7)
```

```
b<-c(2,4,6,8)
print("Addition of
a+b")a+b
print("Subtraction of a-b")
a-b
print("Division of
a/b")a/b
print("Modolus of
a%%b")a%%b
```

Output:

```
[1] "Addition of a+b"
> a+b
[1]  3  7 11 15
[1] "Subtraction of a-b"
> a-b
[1] -1 -1 -1 -1
[1] "Division of a/b"
> a/b
[1] 0.5000000 0.7500000 0.8333333 0.8750000
[1] "Modolus of a%%b"
> a%%b
[1] 1 3 5
7
```

Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11)
# V2 becomes c(4,11,4,11,4,11)
add.result <- v1+v2
print(add.result)
sub.result <- v1-v2
print(sub.result)
```

Output:

```
[1]  7 19 8 16  4 22
```

```
[1] -1 -3 0 -6 -4  0
```

Vector Element 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)
```

Output:

```
print(sort.result)
[1]  -9  0   3   4   5   8  11 304

print(revsort.result)
[1] 304  11  8   5   4   3   0  -9

print(sort.result)
[1] "Blue"   "Red"    "violet" "yellow"

print(revsort.result)
[1] "yellow" "violet" "Red"    "Blue"
```

4(b) Implement R script for finding the sum and average of given numbers using arrays.

```
thisarray <- c(1:24)
multiarray <- array(thisarray,dim = c(4,3,2))
print(multiarray)

print("sum of array elements:")
print(sum(multiarray))

print("Length of the array:")
len <-length(multiarray)
len

print("Average of array elements:")
```

```
print(sum(multiarray)/len)
```

Output:

```
[1] "sum of array elements:"
[1] 300
[1] "Length of the array:"
[1] 24
[1] "Average of array elements:"
[1] 12.5
```

4(c) Implement R script to display elements of list in reverse order.

```
list1 <-c(1:24)
print(list1)
print(Elements in Reverse Order")
rev.default(list1)
```

Output:

```
[1]  1 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
[1] "Elements in Reverse Order"
[1] 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1
```

4(d) Implement R script to find the minimum and maximum elements in the array.

```
nums = c(10, 20, 30, 40, 50, 60)
array1 <-array(nums)
print("The elements in the Array:")
array1
print(paste("Maximum value :",max(nums)))
print(paste("Minimum value :",min(nums)))
```

Output:

```
[1] "The elements in the Array:"
[1] 10 20 30 40 50 60
[1] "Maximum value : 60"
[1] "Minimum value : 10"
```

Week-5:

5(a) a) Implement R script to perform various operations on matrices.

Matrix operations

In R, we can perform the mathematical operations on a matrix such as addition, subtraction, multiplication, etc.

```
R <- matrix(c(5:16), nrow =

4,ncol=3)S <- matrix(c(1:12), nrow

= 4,ncol=3)# Display two matrices

R and S print(R)

print(S)

#Addition

sum<-R+S

print(sum)

#Subtraction

sub<-R-S

print(sub)

#Multiplicatio

nmul<-R*S

print(mul)

#Division

div<-R/S

print(div)
```

Output:

print(R)

```
     [,1] [,2] [,3]
[1,]    5    9   13
[2,]    6   10   14
[3,]    7   11   15
[4,]    8   12   16
```

print(S)

```
     [,1] [,2] [,3]
[1,]    1    5    9
```

[2,] 2 6 10

[3,] 3 7 11

[4,] 4 8 12

sum<-

R+S

print(sum

)

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

[1,] 6 14 22

[2,] 8 16 24

[3,] 10 18 26

[4,] 12 20 28

sub<-R-S

print(sub)

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

[1,] 4 4 4

[2,] 4 4 4

[3,] 4 4 4

[4,] 4 4 4

mul<-R*S

print(mul)

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

[1,] 5 45 117

[2,] 12 60 140

[3,] 21 77 165

[4,] 32 96 192

div<-R/S

print(div)

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

[1,] 5.000000 1.800000 1.444444

[2,] 3.000000 1.666667 1.400000

[3,] 2.333333 1.571429 1.363636

[4,] 2.000000 1.500000 1.333333

5(b) Implement R script to extract the data from dataframes.

```
exam_data = data.frame(

name = c('Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin',
'Jonas'),

score = c(12.5, 9, 16.5, 12, 9, 20, 14.5, 13.5, 8, 19),

attempts = c(1, 3, 2, 3, 2, 3, 1, 1, 2, 1),

qualify = c('yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes')

print("Original dataframe:")

print(exam_data)

print("Extract 3rd and 5th rows with 1st and 3rd columns :")

result =  exam_data[c(3,5),c(1,3)]

print(result)
```

Output:

```
print(exam_data)
     name         score    attempts     qualify
1  Anastasia    12.5       1            yes
2 Dima           9.0        3            no
3  Katherine    16.5       2            yes
4 James          12.0       3            no
5 Emily          9.0        2            no
6 Michael       20.0        3           yes
7 Matthew       14.5       1            yes
8 Laura          13.5       1           no
9  Kevin          8.0       2           no
10Jonas         19.0       1            yes
[1] "Extract 3rd and 5th rows with 1st and 3rd columns:"
   name          attempts
3 Katherine       2
5 Emily           2
```

5(c) Write R script to display file contents.

- **Reading a text file**

One of the important formats to store a file is in a text file. R provides various methods that one can read data from a text file.

- **read.delim()**: This method is used for reading "tab-separated value" files (".txt"). By default, point (".") is used as decimal points.

  read.delim(file, header = TRUE, sep = "\t", dec = ".", ...)

file: the path to the file containing the data to be read into R.
header: a logical value. If TRUE, read.delim()assumes that your file has a header row, so row 1 is the name of each column. If that's not the case, you can add the argument header

= FALSE.
sep: the field separator character. "\t" is used for a tab-delimited file.
dec: the character used in the file for decimal points.

- **read.delim2()**: This method is used for reading "tab-separated value" files (".txt"). By default, point (",") is used as decimal points.

  read.delim2(file, header = TRUE, sep = "\t", dec = ",", ...)

- **file.choose()**: In R it's also possible to choose a file interactively using the functio n **file.choose()**, and if you're a beginner in R programming then this method is very useful for you.

- **read_tsv()**: This method is also used for to read a tab separated ("\t") values by using the help of **readr** package.

  read_tsv(file, col_names = TRUE)
  file: the path to the file containing the data to be read into R.
  col_names: Either TRUE, FALSE, or a character vector specifying column names. If TRUE, the first row of the input will be used as the column names.

Example:1
```
# R program reading a text file
# Read a text file using read.delim()
myData= read.delim("C:/Users/rajen/OneDrive/Documents/sample.txt", header = FALSE)
print(myData)
```

Output:
1 Welcome to R Programming Lab

Example:2
```
# R program reading a text file
# Read a text file using read.delim2
myData = read.delim2("C:/Users/rajen/OneDrive/Documents/sample.txt", header = FALSE)
print(myData)
```
Output:
1 Welcome to R Programming Lab

Example:3
```
# R program reading a text file using file.choose()
myFile = read.delim(file.choose(), header = FALSE)
# If you use the code above in RStudio
# you will be asked to choose a file
print(myFile)
```
Output:
1 Welcome to R Programming Lab

Example:4
```
# R program to read text file
# using readr package
```

```
# Import the readr library
install.packages("readr")
library(readr)
# Use read_tsv() to read text file
myData = read_tsv("C:/Users/rajen/OneDrive/Documents/sample.txt",
col_names = FALSE)
print(myData)
```

Output:

```
A tibble: 1 x 1
  X1
  <chr>
1 Welcome to R Programming Lab
```

5(d) Write R script to copy file contents from one to another.

```
library(readr)
file1=read_file("C:/Users/rajen/OneDrive/Documents/sample.txt")
print(file1)
write_file(file1,"file2.txt")
d=read_file("file2.txt")
print(d)
```

Output:

```
[1] "Welcome to R Programming Lab"
```

Week-6

6(a) Write an R script to find basic descriptive statistics using summary, str, quartile function on mtcars & cars datasets.

What is Descriptive Statistics?

- Descriptive statistics is the branch of statistics that focuses on **describing** and gaining more insight into the data in its present state.
- It deals with what the data in its **current state** means. It makes the data easier to understand and also gives us knowledge about the data which is necessary to perform further analysis.
- Average measures like mean, median, mode, etc. are a good example of descriptive statistics.

Descriptive Statistics in R

R programming language provides us with lots of simple yet effective functions to perform descriptive statistics and gain more knowledge about our data. Summarizing the data, calculating average measures, finding out cumulative measures, summarizing rows/columns of data structures, etc. everything is possible with trivial commands. Let's start simple with the summarizing functions **str()** and **summary()**.

Summarizing your Data

R provides two very simple functions that can instantly summarize our data for us. These are the **str()** and the **summary()** functions.

str() function

The **str()** function takes a single object as an argument and compactly shows us the **structure** of the input object. It shows us details like length, data type, names and other specifics about the components of the object.

```
# shows us the structure of the input object.
str(mtcars)
```

Output:

```
str(mtcars)

'data.frame':    32 obs. of  11 variables:
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num 6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num 16.5 17 18.6 19.4 17 ...
 $ vs : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
str(cars)
```
Output:
```
str(cars)

 $ dist : num   2 10 4 22 16 10 18 26 34 17 ...
```

summary() function:

The **summary()**function also takes a single object as an argument. It then returns the **averages measures** like mean, median, minimum, maximum, 1st quantile, 3rd quantile, etc. for each component or variable in the object. Here is an example of the summary function in action.

```
summary(mtcars)
```
Output:

summary(mtcars)

```
      mpg             cyl             disp             hp             drat             wt             qsec
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0   Min.   :2.760   Min.   :1.513   Min.   :14.50
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89
 Median :19.20   Median :6.000   Median :196.3   Median :123.0   Median :3.695   Median :3.325   Median :17.71
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7   Mean   :3.597   Mean   :3.217   Mean   :17.85
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0   Max.   :4.930   Max.   :5.424   Max.   :22.90
       vs               am              gear            carb
 Min.   :0.0000   Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4375   Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

```
summary(cars)
```
Output:

summary(cars)

```
     speed           dist
 Min.   : 4.0    Min.   :  2.00
 1st Qu.:12.0    1st Qu.: 26.00
 Median :15.0    Median : 36.00
 Mean   :15.4    Mean   : 42.98
 3rd Qu.:19.0    3rd Qu.: 56.00
 Max.   :25.0    Max.   :120.00
```

Getting the Average Measures

R provides a number of functions that give us different average measures for given data. These average measures include:

- **Mean:** The mean of a given set of numeric or logical values(it may be a vector or a row or column of any other data structure) can be easily found using the **mean()** function.
- **Median:** Finding the median of a set of numeric or logical values is also very easy by using the **median()** function.
- **Standard deviation:** The standard deviation of a set of numerical values can be found using the **sd()** function.
- **Variance:** the **var()** function gives us the variance of a set of numeric or logical values.
- **Median Absolute Variance:** The median absolute variance of a set of numeric or logical values can be found by using the **mad()** function.
- **Maximum:** In a given set of numeric or logical values, we can use the **max()** function to find the maximum or the largest value in the set.

Note: NA is considered to be the largest by the **max()** function unless its na.rm argument is set to TRUE.

- **Minimum:** The **min()** function is a very handy way to find out the smallest value in a set of numeric values.

Note: Like the **max()** function, the min() function considers NA to be the smallest unless na.rm is set to TRUE.

- **Sum:** The sum of a set of numerical values can be found by simply using the **sum()** function.
- **Length:** The length or the number of values in a set is given by the **length()** function.

Example:

```
mean(mtcars$mpg)
median(mtcars$mpg)
sd(mtcars$mpg)
var(mtcars$mpg)
mad(mtcars$mpg)
max(mtcars$mpg, na.rm = TRUE)
min(mtcars$mpg, na.rm = TRUE)
sum(mtcars$mpg)
length(mtcars$mpg)
```

Output:

```
mean(mtcars$mpg)
[1] 20.09062
> median(mtcars$mpg)
[1] 19.2
> sd(mtcars$mpg)
[1] 6.026948
> var(mtcars$mpg)
[1] 36.3241
> mad(mtcars$mpg)
[1] 5.41149
> max(mtcars$mpg, na.rm = TRUE)
```

```
[1] 33.9
> min(mtcars$mpg, na.rm = TRUE)
[1] 10.4
> sum(mtcars$mpg)
[1] 642.9
> length(mtcars$mpg)
[1] 32
```

Quantile Function:

- A quantile is nothing but a sample that is divided into equal groups or sizes. Due to this nature, the quantiles are also called as Fractiles. In the quantiles, the 25th percentile is called as lower quartile, 50th percentile is called as Median and the 75th Percentile is called as the upper quartile.
- This is particularly useful when you're doing exploratory analysis and reporting, especially if you're analyzing data which may not be normally distributed.
- We're going to use the r quantile function; this utility is part of base R (so you don't need to import any libraries) and can be adapted to generate a variety of "rank based" statistics about your sample.

Quantile() function syntax

The syntax of the Quantile() function in R is,

```
quantile(x, probs = , na.rm = FALSE)
```

Where,

- **X** = the input vector or the values
- **Probs** = probabilities of values between 0 and 1.
- **na.rm** = removes the NA values.

We're going to use the r quantile function; this utility is part of base R (so you don't need to import any libraries) and can be adapted to generate a variety of "rank based" statistics about your sample.

Example:

```
# quartile in R example
test = c(9,9,8,9,10,9,3,5,6,8,9,10,11,12,13,11,10)
# get quartile in r code (single line)
quantile(test, prob=c(.25,.5,.75))
```

Output:

```
quantile(test, prob=c(.25,.5,.75))
25% 50% 75%
  8   9   10
```

You can also use the summary function to generate the same information.

```
# quartile in R example - summary function
```

```
test = c(9,9,8,9,10,9,3,5,6,8,9,10,11,12,13,11,10)
summary(test)
```

Output:

```
summary(test)
   Min.   1st Qu.   Median       Mean       3rd Qu.    Max.
  3.000   8.000     9.000        8.941       10.000  13.000
```

Quantile function using mtcars and cars dataset.

```
quantile(mtcars$wt)
```

Output:

```
     0%        25%     50%          75%       100%
1.51300  2.58125   3.32500     3.61000    5.42400
```

```
quantile(mtcars$mpg)
```

Output:

```
   0%    25%    50%    75%    100%
10.400 15.425 19.200 22.800 33.900
```

```
quantile(cars$speed)
```

Output:

```
  0%  25%   50%   75%   100%
   4    12     15      19      25
quantile(cars$speed,c(.2, .4, .8))
```

Output:

```
20% 40% 80%
11    14     20
```

6(b) Write an R script to find subset of dataset by using subset(), aggregate() functions on iris
dataset.

Sub setting Datasets in R

- R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations.
- Whether you're comparing how different demographics respond to marketing campaigns, zooming in on a specific time frame, or pulling information about a select few products from the inventory, subsetting datasets enables you to extract useful observations in your dataset.
- R is a great tool that makes subsetting data easy and intuitive.
- The **subset( )** function is the easiest way to select variables and observations.
- Subsetting your data does not change the content of your data, but simply selects the portion most relevant to the goal you have in mind. In general, there are three ways to subset the rows and columns of your dataset—by index, by name, and by value.

Iris dataset

- Iris dataset gives the measurements in centimetres of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

- In this picture You can see what are we dealing with and how exactly looks the variables (sepal length and width and petal length and width) we are measuring and the object itself:

Iris Versicolor **Iris Setosa** **Iris Virginica**

Format

iris is a data frame with 150 cases (rows) and 5 variables (columns) named Sepaal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species.

Here's a little summary of what you can basically see in dataset iris:

summary(iris)
Output:

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

names(iris)
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

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

Let's take a look at the data itself. Let's see the first 5 rows of data for each class:

```
# Get first 5 rows of each subset
subset(iris, Species == "setosa")[1:5,]
```

Output:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa

```
subset(iris, Species == "versicolor")[1:5,]
```

Output:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
53	6.9	3.1	4.9	1.5	versicolor
54	5.5	2.3	4.0	1.3	versicolor
55	6.5	2.8	4.6	1.5	versicolor

```
subset(iris, Species == "virginica")[1:5,]
```

Output:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
101	6.3	3.3	6.0	2.5	virginica
102	5.8	2.7	5.1	1.9	virginica
103	7.1	3.0	5.9	2.1	virginica
104	6.3	2.9	5.6	1.8	virginica
105	6.5	3.0	5.8	2.2	virginica

AGGREGATE() FUNCTION IN R

- **aggregate() Function in R Splits the data into subsets, computes summary statistics for each subsets and returns the result in a group by form.**
  - aggregate() function is useful in performing all the aggregate operations like sum, count, mean, minimum and Maximum.

Use aggregate() function to find summary statistics by group.

Syntax for Aggregate() Function in R:

aggregate(x, by, FUN, …, simplify = TRUE, drop = TRUE)

X	an R object, mostly a data frame
by	a list of grouping elements, by which the subsets are grouped by
FUN	a function to compute the summary statistics
simplify	a logical indicating whether results should be simplified to a vector or matrix if possible
drop	a logical indicating whether to drop unused combinations of grouping values.

Example:

```
agg_mean = aggregate(iris[,1:4], by = list(iris$Species),FUN=mean,na.rm=TRUE)
agg_mean
```

the above code takes first 4 columns of iris data set and groups by "species" by computing the mean for each group, so the output will be

Output:

	Group.1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	5.006	3.428	1.462	0.246
2	versicolor	5.936	2.770	4.260	1.326
3	virginica	6.588	2.974	5.552	2.026

Example for aggregate() function in R with sum:

Let's use the aggregate() function in R to create the sum of all the metrics across species and group by species.

```
# Aggregate function in R with sum summary statistics
agg_sum = aggregate(iris[,1:4],by=list(iris$Species),FUN=sum, na.rm=TRUE)
agg_sum
```

Output:

	Group.1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	250.3	171.4	73.1	12.3
2	versicolor	296.8	138.5	213.0	66.3
3	virginica	329.4	148.7	277.6	101.3

Example for aggregate() function in R with sum:

Let's use the aggregate() function in R to create the sum of all the metrics across species and group by species.

# Aggregate function in R with sum summary statistics
agg_sum = aggregate(iris[,1:4],by=list(iris$Species),FUN=sum, na.rm=TRUE)
agg_sum

Output:

	Group.1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	250.3	171.4	73.1	12.3
2	versicolor	296.8	138.5	213.0	66.3
3	virginica	329.4	148.7	277.6	101.3

Example for aggregate() function in R with count:

```
# Aggregate function in R with count
agg_count = aggregate(iris[,1:4],by=list(iris$Species),FUN=length)
```

agg_count

the above code takes first 4 columns of iris data set and groups by “species” by computing the count for each group, so the output will be

Output:

	Group.1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	50	50	50	50
2	versicolor	50	50	50	50
3	virginica	50	50	50	50

Example for aggregate() function in R with maximum:
Let’s use the aggregate() function to create the maximum of all the metrics across species and group by species.

```
# Aggregate function in R with maximum

agg_max = aggregate(iris[,1:4],by=list(iris$Species),FUN=max,

na.rm=TRUE)agg_max
```

the above code takes first 4 columns of iris data set and groups by “species” by computing the max for each group, so the output will be

Output:

	Group.1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	5.8	4.4	1.9	0.6
2	versicolor	7.0	3.4	5.1	1.8
3	virginica	7.9	3.8	6.9	2.5

Week-7

7(a) Reading different types of data sets (.txt, .csv) from Web or disk and writing in file in specific disk location.

Reading data from txt or csv files

The R base function read.table() is a general function that can be used to read a file in table format. The data will be imported as a data frame.
Note that, depending on the format of your file, several variants of read.table() are available, including read.csv, read.csv2(), read.delim and read.delim2().

- **read.csv**(): for reading **"comma separated value"** files (".csv").
- **read.csv2**(): variant used in countries that use a comma "," as decimal point and a semicolon ";" as field separators.
- **read.delim**(): for reading *"tab-separated value"* files (".txt"). By default, point (".") is used asdecimal points.
- **read.delim2**(): for reading *"tab-separated value"* files (".txt"). By default, comma (",") is usedas decimal points.

The simplified format of these functions are, as follows:

 # Read tabular data into R
read.table(file, header = FALSE, sep = "", dec = ".")
 # Read "comma separated value" files (".csv")
read.csv(file, header = TRUE, sep = ",", dec = ".", **...**)
 # Or use read.csv2: variant used in countries that # use a comma as decimal point and a semicolon as field separator.
read.csv2(file, header = TRUE, sep = ";", dec = ",", **...**)
 # Read TAB delimited files
read.delim(file, header = TRUE, sep = "\t", dec = ".", **...**) read.delim2(file, header = TRUE, sep = "\t", dec = ",", **...**)

- **file**: the path to the file containing the data to be imported into R.
- **sep**: the field separator character. "\t" is used for tab-delimited file.
- **header**: logical value. If TRUE, **read.table()** assumes that your file has a header row, so row 1is the name of each column. If that's not the case, you can add the argument **header = FALSE**.
- **dec**: the character used in the file for decimal points.

Reading a local file

To import a local .txt or a .csv file, the syntax would be:

```
# Read a txt file, named "mtcars.txt"
my_data <- read.delim("mtcars.txt")
# Read a csv file, named "mtcars.csv"
my_data <- read.csv("mtcars.csv")
```

Note:

The above R code, assumes that the file “mtcars.txt” or “mtcars.csv” is in your current working directory. To know your current working directory, type the function **getwd**() in R console.

- It’s also possible to choose a file interactively using the function **file.choose**(), which I recommend if you’re a beginner in R programming:

```
# Read a txt file
my_data <- read.delim(file.choose())
# Read a csv file
my_data <- read.csv(file.choose())
```

If you use the R code above in RStudio, you will be asked to choose a file.

Reading a file from internet

It’s possible to use the functions **read.delim**(), **read.csv**() and **read.table**() to import files from the web.

```
my_data <- read.delim("http://www.sthda.com/upload/boxplot_format.txt")
head(my_data)
```

- Here I am using weather data.

Example-1: R program reading a .text file

```
# Read a text file using read.delim()
Data1 = read.delim("weather.txt", header = TRUE)
print(Data1)
```

Output:

```
my_data
   outlook temperature humidity windy play
1  overcast        hot     high FALSE yes
2  overcast       cool   normal  TRUE  yes
3  overcast       mild     high TRUE  yes
4  overcast        hot   normal FALSE  yes
5     rainy       mild     high FALSE  yes
6     rainy       cool   normal FALSE  yes
7     rainy       cool   normal  TRUE   no
8     rainy       mild   normal FALSE  yes
9     rainy       mild     high  TRUE  no
10    sunny        hot     high FALSE  no
11    sunny        hot     high  TRUE   no
12    sunny       mild     high FALSE   no
13    sunny       cool   normal FALSE  yes
14    sunny       mild   normal  TRUE  yes
```

```
Data2 <-read.table(“weather.txt”, header=TRUE, sep = "\t")
Data2
```

Output:

```
Data2
   outlook temperature humidity windy play
1   overcast        hot       FALSE yes
```

```
2  overcast       cool  normal  TRUE  yes
3  overcast       mild     high TRUE  yes
4  overcast        hot  normal FALSE  yes
5     rainy       mild     high FALSE  yes
6     rainy       cool  normal FALSE  yes
7     rainy       cool  normal  TRUE   no
8     rainy       mild  normal FALSE  yes
9     rainy       mild     high  TRUE  no
10    sunny        hot     high FALSE  no
11    sunny        hot     high  TRUE   no
12    sunny       mild     high FALSE   no
13    sunny       cool   normal FALSE  yes
14    sunny       mild   normal  TRUE  yes
```

Example-2: R program reading a .csv file

```
Data3 <- read.csv("weather.csv", header=TRUE)
Data3
outlook temperature humidity windy play
1  overcast        hot     high FALSE yes
2  overcast       cool  normal  TRUE  yes
3  overcast       mild     high TRUE  yes
4  overcast        hot  normal FALSE  yes
5     rainy       mild     high FALSE  yes
6     rainy       cool  normal FALSE  yes
7     rainy       cool  normal  TRUE   no
8     rainy       mild  normal FALSE  yes
9     rainy       mild     high  TRUE  no
10    sunny        hot     high FALSE  no
11    sunny        hot     high  TRUE   no
12    sunny       mild     high FALSE   no
13    sunny       cool   normal FALSE  yes
14    sunny       mild   normal  TRUE  yes

Data4 <-read.table("weather.csv", header=TRUE,sep=",")

Data4
outlook temperature humidity windy play
1  overcast        hot     high FALSE yes
2  overcast       cool  normal  TRUE  yes
3  overcast       mild     high TRUE  yes
4  overcast       hot  normal FALSE  yes
5     rainy       mild     high FALSE  yes
6     rainy       cool   normal FALSE  yes
7     rainy       cool   normal  TRUE  no
8     rainy       mild   normal FALSE  yes
9     rainy       mild     high  TRUE  no
10    sunny        hot     high FALSE  no
11    sunny        hot     high  TRUE  no
12    sunny       mild     high FALSE  no
```

13 sunny cool normal FALSE yes
14 sunny mild normal TRUE yes

It's also possible to choose a file interactively using the function **file.choose**()

 To read .txt file

```
data3 <-read.delim(file.choose(), header=TRUE)
data3

data4 <-read.table(file.choose(),header=TRUE, sep="\t")
data4
```

 To read .csv file

```
data1 <- read.csv(file.choose(), header=TRUE)
data1

data2 <-read.table(file.choose(), header=TRUE,sep=",")
data2
```

Reading a file from internet
It's possible to use the functions **read.delim**(), **read.csv**() and **read.table**() to import files from the web.

```
my_data <- read.delim("http://www.sthda.com/upload/boxplot_format.txt")
head(my_data)
```

Output:

	Nom	variable	Group
1	IND1	10	A
2	IND2	7	A
3	IND3	20	A
4	IND4	14	A
5	IND5	14	A
6	IND6	12	A
7	IND7	10	A
8	IND8	23	A
9	IND9	17	A
10	IND10	20	A
11	IND11	14	A
12	IND12	13	A
13	IND13	11	B
14	IND14	17	B
15	IND15	21	B
16	IND16	11	B
17	IND17	16	B
18	IND18	14	B
19	IND19	17	B
20	IND20	17	B

21	IND21	19	B
22	IND22	21	B
23	IND23	7	B
24	IND24	13	B
25	IND25	0	C
26	IND26	1	C
27	IND27	7	C
28	IND28	2	C
29	IND29	3	C
30	IND30	1	C
31	IND31	2	C
32	IND32	1	C
33	IND33	3	C
34	IND34	0	C
35	IND35	1	C
36	IND36	4	C
37	IND37	3	D
38	IND38	5	D
39	IND39	12	D
40	IND40	6	D
41	IND41	4	D
42	IND42	3	D
43	IND43	5	D
44	IND44	5	D
45	IND45	5	D
46	IND46	5	D
47	IND47	2	D
48	IND48	4	D
49	IND49	3	E
50	IND50	5	E
51	IND51	3	E
52	IND52	5	E
53	IND53	3	E
54	IND54	6	E
55	IND55	1	E
56	IND56	1	E
57	IND57	3	E
58	IND58	2	E
59	IND59	6	E
60	IND60	4	E
61	IND61	11	F
62	IND62	9	F
63	IND63	15	F
64	IND64	22	F
65	IND65	15	F
66	IND66	16	F
67	IND67	13	F
68	IND68	10	F
69	IND69	26	F
70	IND70	26	F

71 IND71 24 F
72 IND72 13 F

Import dataset in R programming

R is a programming language designed for data analysis. Therefore, loading data is one of the core features of R.

R contains a set of functions that can be used to load data sets into memory. You can also load datainto memory using R Studio - via the menu items and toolbars.

Data Formats

R can load data in two different formats:

- CSV files
- Text files

CSV means Comma Separated Values. You can export CSV files from many data carrying applications. For instance, you can export CSV files from data in an Excel spreadsheet. Here is an example of how a CSV file looks like inside:

```
name,id,salary
"John Doe",1,99999.00
"Joe Blocks",2,120000.00
"Cindy Loo",3,150000.00
```

As you can see, the values on each line are separated by commas. The first line contains a list of column names. These column names tell what the data in the following lines mean. These names only make sense to you. R does not care about these names. R just uses these name to identify data from the different columns.

A text file is typically similar to a CSV file, but instead of using commas as separators between values, text files often use other characters, like e.g. a Tab character. Here is an example of how a text file could look inside:

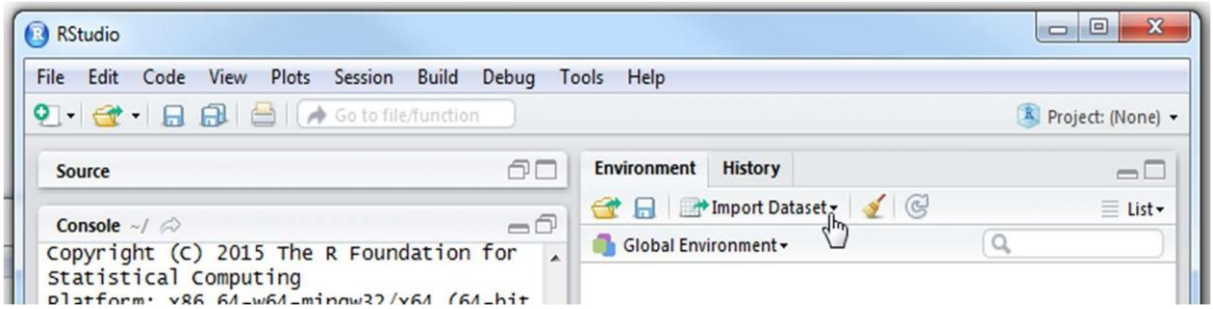
```
name            id     salary

"John Doe"      1      99999.00
"Joe Blocks"    2      120000.00
"Cindy Loo"     3      150000.00
```

As you can see, the data might be easier to read in text format - if you look at the data directly in the data file that is. Once the data is loaded into R / R studio, there is no difference. You can look at the data in R Studio's tabular data set viewer, and then you cannot see the difference between CSV files and text files.

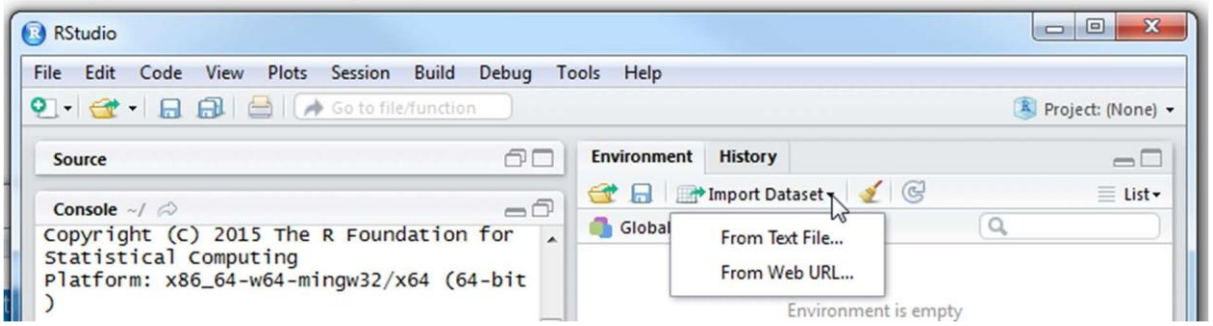
Actually, the name "text files" is a bit confusing. Both CSV files and text files contains data in textual form (as characters). One just uses commas as separator between the values, whereas the others use a tab character.

Load Data Via R Studio Menu Items

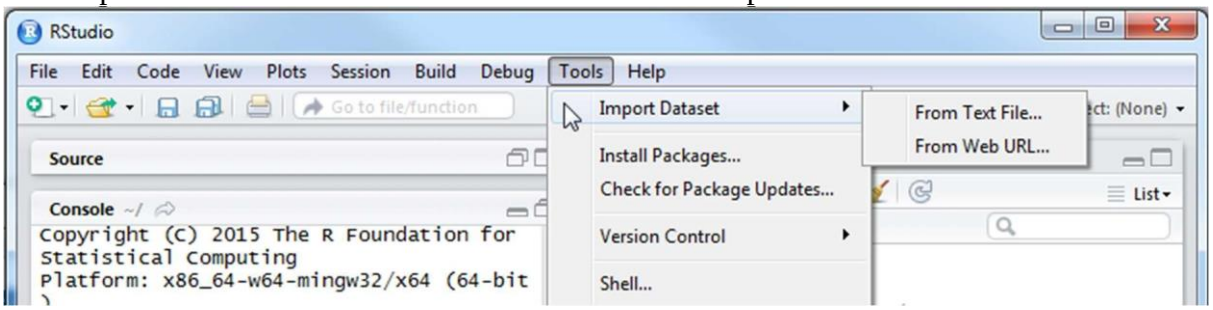
The easiest way to load data into memory in R is by using the R Studio menu items. R Studio has menu items for loading data in two different places. The first is in the toolbar of the upper right section of R Studio. This screenshot shows where the "Import Dataset" button is (look for the little mouse pointer "hand") :



When you click the button you get this little menu:



You can also import data from the top menu of R Studio. The next screenshot shows where the "Import Dataset" menu item is located in R Studio's top menu:



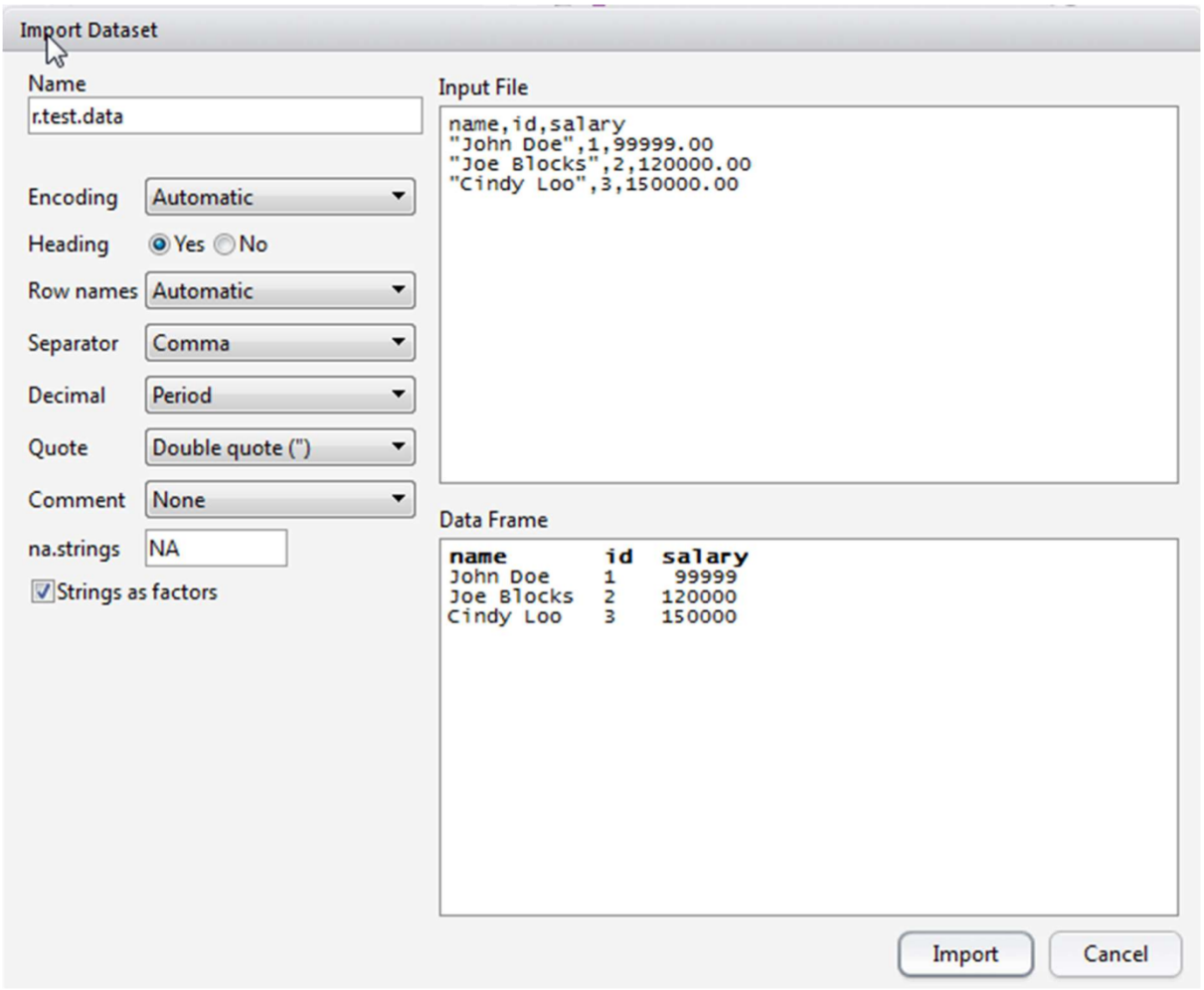
Text File or Web URL

As you can see in both the "Import Dataset" menu items, you can import a data set "From Text File" or "From Web URL". These two options refer to where you load the data from. "From Text File" means from a text file on your local computer. "From Web URL" means that you load the data from a web server somewhere on the internet.

Regardless of whether you choose "From Text File" or "From Web URL", R can load the file as either a CSV or text file. The location of the file has nothing to do with the data format used inside the file. Don't get confused by that. The menu item "From Text file" does not mean "text file format" (tab characters as separators). It just means "a file on your local computer". "From Local File" would probably have been a more informative text for this menu item.

Selecting Data Format

After you have chosen the location to load the file from, you will be shown a dialog like this:



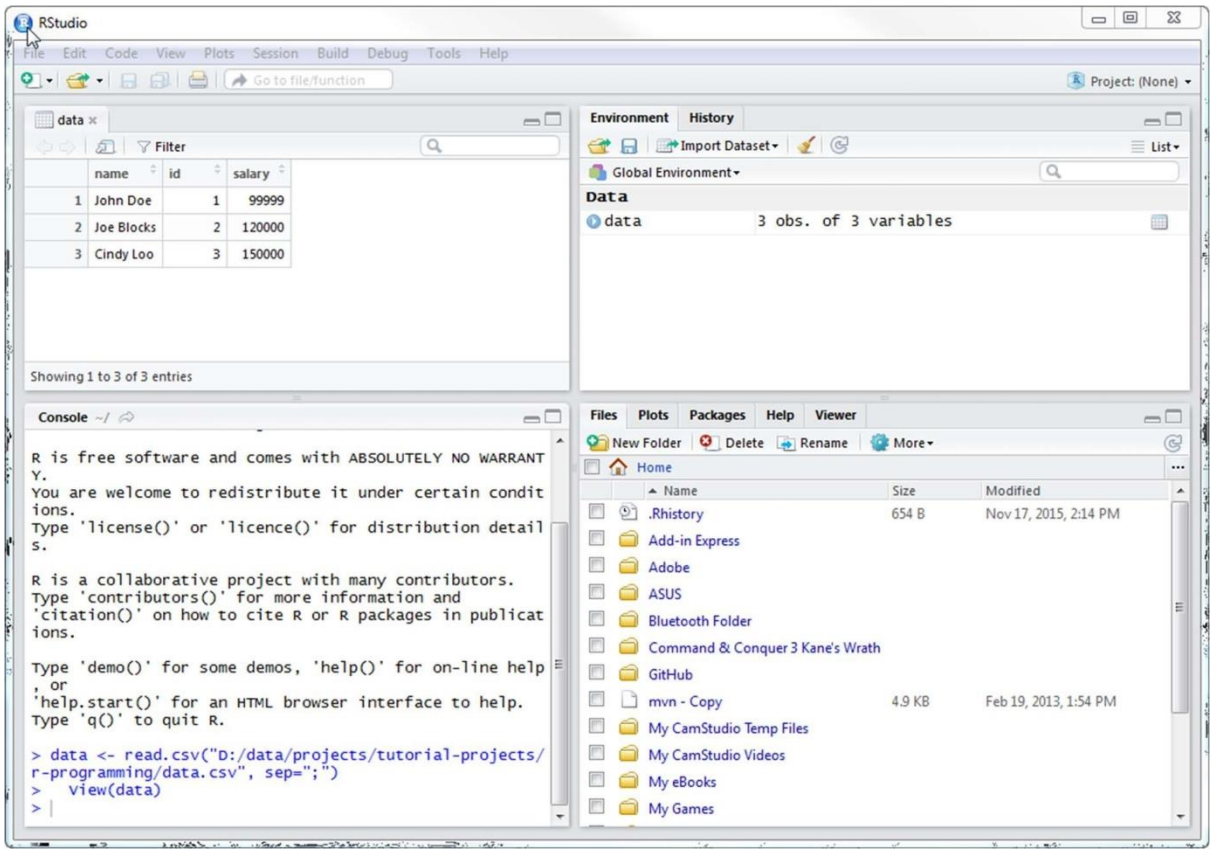
The select boxes (drop down boxes) allows you to specify different configurations about the data format of the file you are about to import. In the boxes on the right you can see two boxes. The top box shows you what the data file looks like. The bottom box shows you how R Studio interprets the data in the file based on the configurations chosen in the select boxes in the left side of the dialog. If you change the choices in the select boxes you will see that the bottom right box changes.

When you have selected all the configurations you need in the select boxes on the left, click the "Import" button. The data will now be loaded into R Studio.

Note that R Studio prints the R commands needed to load the data into the R console in the left side of R studio. You can copy these functions and use them to load data into R via R code.

After the Data is Loaded

After you have loaded the data into R Studio it will look similar to the screenshot below:



7(b) Reading Excel data sheet in R.
Steps to Import an Excel file into R

Step 1: Install the readxl package
In the R Console, type the following command to install the **readxl** package:
install.packages("readxl")

Step 2: Prepare your Excel File
Let's suppose that you have an Excel file with some data about products:

Product	Price
Refrigerator	1200
Oven	750
Dishwasher	900
Coffee Maker	300

And let's say that the Excel file name is **product_list**, and your goal is to import that file into R.
Step 3: Import the Excel file into R

In order to import your file, you'll need to apply the following template in the R Editor:

library("readxl")
read.excel("Path where your Excel file is stored\\FileName.xlsx")

Example:

```
my_data <- read_excel("product_list.xlsx")
my_data
```

(OR)

```
my_data <- read_excel(file.choose())
my_data
```

Note:

If you use the R code above in RStudio, you will be asked to choose a file.

Output:

```
# A tibble: 4 x 2
    Product         Price
     <chr>          <dbl>
1  Refrigerator    1200
2  Oven             750
3  Dishwasher       900
4   Coffee Maker   300
```

Importing Excel files using xlsx package

The **xlsx** package, a java-based solution, is one of the powerful R packages to **read**, **write** and **format Excel files**.

Installing and loading xlsx package

- Install

```
install.packages("xlsx")
```

- Load

```
library("xlsx)
```

Using xlsx package

There are two main functions in **xlsx** package for reading both xls and xlsx Excel files: **read.xlsx**()and **read.xlsx2**() [faster on big files compared to read.xlsx function].

The simplified formats are:

```
read.xlsx(file, sheetIndex, header=TRUE)
read.xlsx2(file, sheetIndex, header=TRUE)
```

- file: file path
- sheetIndex: the index of the sheet to be read
- header: a logical value. If TRUE, the first row is used as column names.

Example:

```
library("xlsx")
my_data1 <- read.xlsv(file.choose(), 1) # read first sheet
```

7(c) Reading XML dataset in R.

In R, we can read the xml files by installing "XML" package into the R environment. This package will be installed with the help of the familiar command i.e., install. packages.

install.packages("XML")

Creating XML File

Save the following data with the .xml file extension to create an xml file. XML tags describe the meaning of data, so that data contained in such tags can easily tell or explain about the data.

Example: xml_data.xml

Reading XML File

In R, we can easily read an xml file with the help of xmlParse() function. This function is stored as a list in R. To use this function, we first need to load the xml package with the help of the library() function. Apart from the xml package, we also need to load one additional package named methods.

 # To download file to the current working directory

download.file("https://www.w3schools.com/xml/simple.xml", "breakfast.xml")

 # Install XML package

install.packages("XML")

 # To load library

library(XML)

 # Giving the input file name to the function.

doc <- xmlParse("breakfast.xml")
print(doc)

#Converting the data into list

xml_data <-xmlToList(doc)
print(xml_data)
xmldataframe <- xmlToDataFrame("breakfast.xml")
xmldataframe

Output:

```
library(XML)
> doc <- xmlParse("breakfast.xml")
> print(doc)
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>Two of our famous Belgian Waffles with plenty of real maple
syrup</description>
    <calories>650</calories>
  </food>
  <food>
```

```
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>Light Belgian waffles covered with strawberries and whipped
cream</description>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>Light Belgian waffles covered with an assortment of fresh berries and
whipped cream</description>
    <calories>900</calories>
  </food>
  <food>
    <name>French Toast</name>
    <price>$4.50</price>
    <description>Thick slices made from our homemade sourdough bread</description>
    <calories>600</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <price>$6.95</price>
    <description>Two eggs, bacon or sausage, toast, and our ever-popular hash
browns</description>
    <calories>950</calories>
  </food>
</breakfast_menu>

> xml_data <-xmlToList(doc)
> print(xml_data)
$food
$food$name
[1] "Belgian Waffles"
$food$price
[1] "$5.95"
$food$description
[1] "Two of our famous Belgian Waffles with plenty of real maple syrup"
$food$calories
[1] "650"
$food
$food$name
[1] "Strawberry Belgian Waffles"
$food$price
[1] "$7.95"
$food$description
[1] "Light Belgian waffles covered with strawberries and whipped cream"
$food$calories
[1] "900"
$food
$food$name
```

[1] "Berry-Berry Belgian Waffles"
$food$price
[1] "$8.95"
$food$description
[1] "Light Belgian waffles covered with an assortment of fresh berries and whipped cream"
$food$calories
[1] "900"
$food
$food$name
[1] "French Toast"
$food$price
[1] "$4.50"
$food$description
[1] "Thick slices made from our homemade sourdough bread"
$food$calories
[1] "600"
$food
$food$name
[1] "Homestyle Breakfast"
$food$price
[1] "$6.95"
$food$description
[1] "Two eggs, bacon or sausage, toast, and our ever-popular hash browns"
$food$calories
[1] "950"

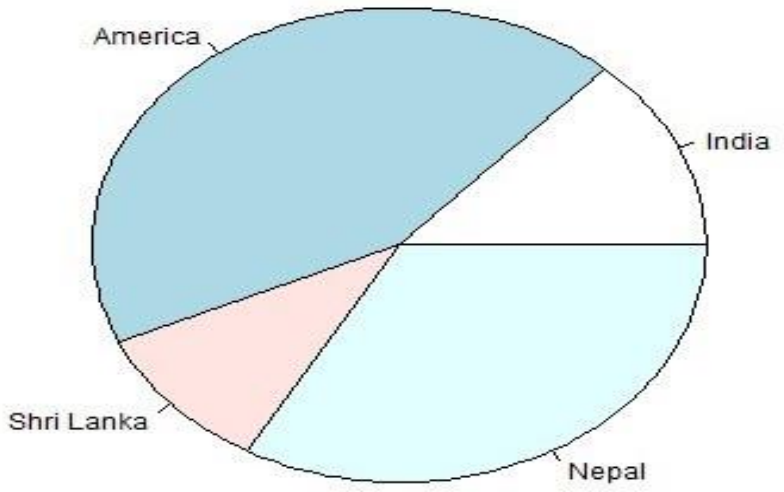
Week-8

8(a) Implement R script to create a Pie chart, Bar chart, scatter plot and Histogram. (Introduction to ggplot2 graphics)

Program:

```
# Creating data for the graph.
x <- c(20, 65, 15, 50)
print(x)
labels <- c("India", "America", "Shri Lanka", "Nepal")
print(labels)
# Giving the chart file a name.
png(file = "Country.jpg")
# Plotting the chart.
pie(x,labels)
# Saving the file.
dev.off()
```

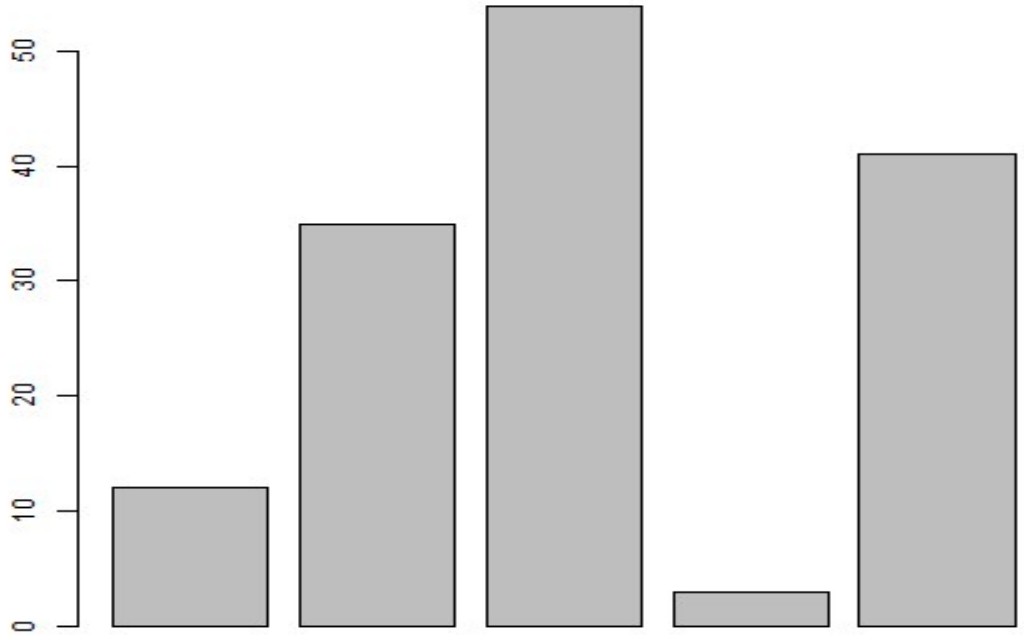
Output:



Program:

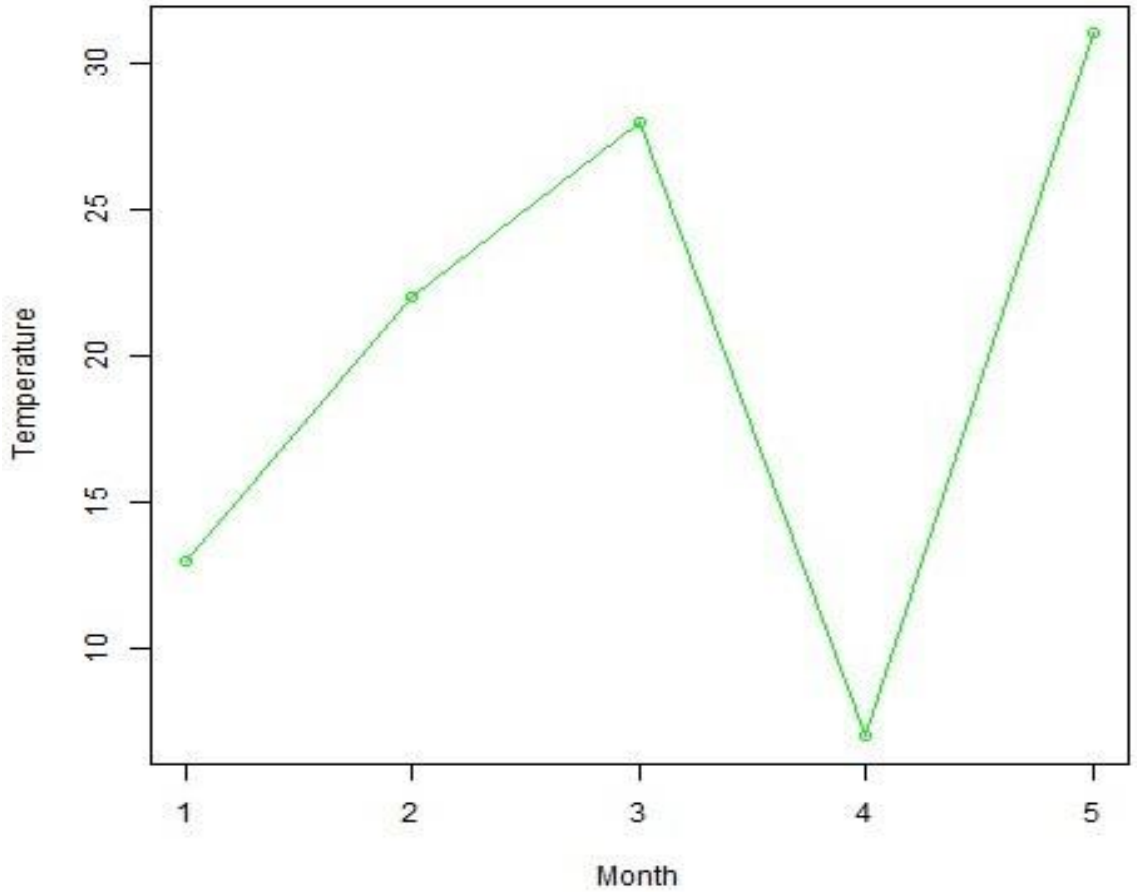
```
# Creating the data for Bar chart
H<- c(12,35,54,3,41)
# Giving the chart file a name
png(file = "bar_chart.png")
# Plotting the bar chart
barplot(H)
# Saving the file
dev.off()
```

Output:




```
# Creating the data for the chart.
v <- c(13,22,28,7,31)
# Giving a name to the chart file.
png(file = "line_graph_feature.jpg")
# Plotting the bar chart.
plot(v,type = "o",col="green",xlab="Month",ylab="Temperature")
# Saving the file.
dev.off()
```

Output:

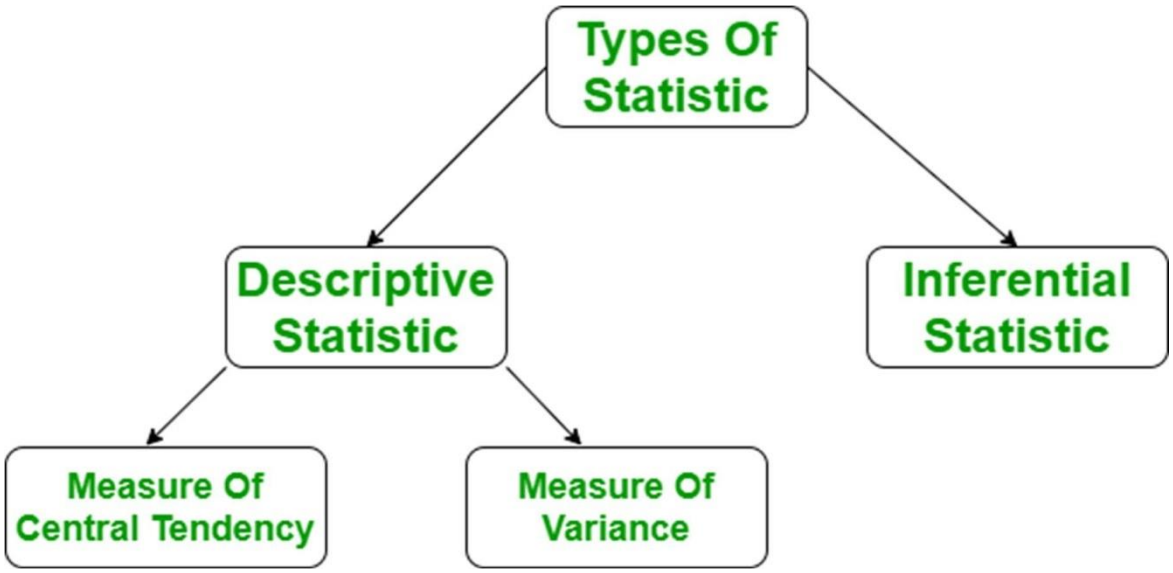


8(b) Implement R Script to perform mean, median, mode, range, summary, variance, standard deviation operations.

In the descriptive analysis, we describe our data in some manner and present it in a meaningful way so that it can be easily understood. Most of the time it is performed on small data sets and this analysis helps us a lot to predict some future trends based on the current findings. Some measures that are used to describe a data set are measures of central tendency and measures of variability or dispersion.

Process of Descriptive Analysis

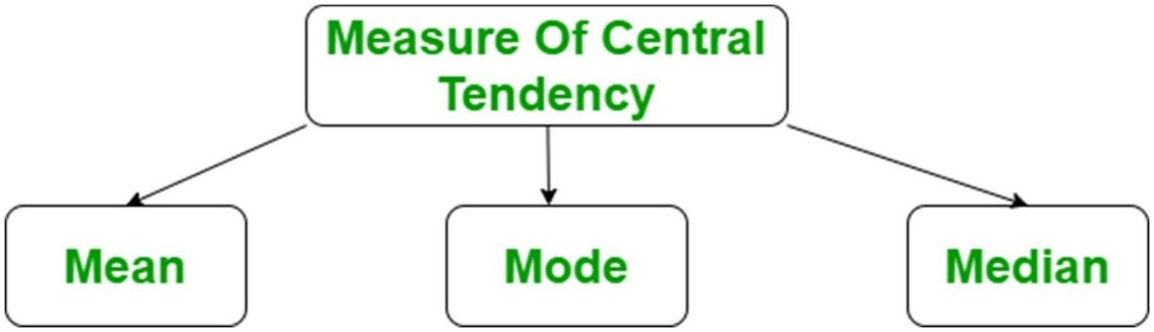
- Measure of central tendency
- Measure of variability



Measure of central tendency

It represents the whole set of data by single value.It gives us the location of central points. There are three main measures of central tendency:

- Mean
- Mode
- Median



Measure of variability

Measure of variability is known as the spread of data or how well is our data is distributed. The most common variability measures are:

Mean

It is calculated by taking the sum of the values and dividing with the number of values in a data series.

The function **mean()** is used to calculate this in R.

Syntax

The basic syntax for calculating mean in R is −

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

Following is the description of the parameters used −

- **x** is the input vector.
- **trim** is used to drop some observations from both end of the sorted vector.
- **na.rm** is used to remove the missing values from the input vector.

Example:

```
# Create a vector.
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)
# Find Mean.
```

```
result.mean <- mean(x,trim = 0.3)
print(result.mean)
```
Output:
[1] 5.55

Median

The middle most value in a data series is called the median. The **median()** function is used in R to calculate this value.

Syntax

The basic syntax for calculating median in R is −

```
median(x, na.rm = FALSE)
```

Following is the description of the parameters used −

- **x** is the input vector.
- **na.rm** is used to remove the missing values from the input vector.

Example
```
# Create the vector.
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# Find the median.
median.result <- median(x)
print(median.result)
```
Output:
[1] 5.6

Mode

The mode is the value that has highest number of occurrences in a set of data. Unike mean and median, mode can have both numeric and character data.

R does not have a standard in-built function to calculate mode. So we create a user function to calculate mode of a data set in R. This function takes the vector as input and gives the mode value as output.

Example

```
# Create the function.
getmode <- function(v) {
uniqv <- unique(v)
  uniqv[which.max(tabulate(match(v, uniqv)))]
}
# Create the vector with numbers.
v <- c(2,1,2,3,1,2,3,4,1,5,5,3,2,3)

# Calculate the mode using the user function.
result <- getmode(v)
print(result)
```

```
# Create the vector with characters.
charv <- c("o","it","the","it","it")

# Calculate the mode using the user function.
result <- getmode(charv)
print(result)
```

Output:

```
[1] 2
[1] "it"
```

Measures of Variability

Following are some of the measures of variability that R offers to differentiate between data sets:

- Variance
- Standard Deviation
- Range
- Mean Deviation
- Interquartile Range

Population vs sample variance

Different formulas are used for calculating variance depending on whether you have data from a whole population or a sample.

Population variance

When you have collected data from every member of the population that you're interested in, you can get an exact value for population variance.

The **population variance** formula looks like this:

Formula	Explanation
$\sigma^2 = \frac{\Sigma (X - \mu)^2}{N}$	❖ σ^2 = population variance ❖ Σ = sum of… ❖ X = each value ❖ μ = population mean ❖ N = number of values in the population

Sample variance

When you collect data from a sample, the sample variance is used to make estimates or inferences about the population variance.

The **sample variance** formula looks like this:

Formula	Explanation
$s^2 = \frac{\Sigma (X - \bar{x})^2}{n - 1}$	❖ s^2 = sample variance ❖ Σ = sum of… ❖ X = each value ❖ $\bar{x}$ = sample mean ❖ n = number of values in the sample

With samples, we use $n - 1$ in the formula because using n would give us a biased estimate that consistently underestimates variability. The sample variance would tend to be lower than the real variance of the population.

Reducing the sample n to $n - 1$ makes the variance artificially large, giving you an unbiased estimate of variability: it is better to overestimate rather than underestimate variability in samples.

Variance

- The **variance** is a measure of variability. It is calculated by taking the average of squared deviations from the mean.
- Variance tells you the degree of spread in your data set. The more spread the data, the larger the variance is in relation to the mean.

Steps for calculating the variance

The variance is usually calculated automatically by whichever software you use for your statistical analysis. But you can also calculate it by hand to better understand how the formula works.

There are five main steps for finding the variance by hand. We'll use a small data set of 6 scores to walk through the steps.

Data set					
46	69	32	60	52	41

Step 1: Find the mean

To find the mean, add up all the scores, then divide them by the number of scores.

Mean ($\bar{x}$)
$\bar{x}$ = (46 + 69 + 32 + 60 + 52 + 41) ÷ 6 = **50**

Step 2: Find each score's deviation from the mean
Subtract the mean from each score to get the deviations from the mean.
Since $\bar{x}$ = 50, take away 50 from each score.

Score	**Deviation from the mean**
46	46 – 50 = **-4**
69	69 – 50 = **19**
32	32 – 50 = **-18**

Score	Deviation from the mean
60	60 – 50 = **10**
52	52 – 50 = **2**
41	41 – 50 = **-9**

Step 3: Square each deviation from the mean

Multiply each deviation from the mean by itself. This will result in positive numbers.

Squared deviations from the mean
$(-4)^2 = 4 \times 4 =$ **16**
$19^2 = 19 \times 19 =$ **361**
$(-18)^2 = -18 \times -18 =$ **324**
$10^2 = 10 \times 10 =$ **100**
$2^2 = 2 \times 2 =$ **4**
$(-9)^2 = -9 \times -9 =$ **81**

Step 4: Find the sum of squares

Add up all of the squared deviations. This is called the sum of squares.

Sum of squares
16 + 361 + 324 + 100 + 4 + 81 = **886**

Step 5: Divide the sum of squares by $n - 1$ or N

Divide the sum of the squares by $n - 1$ (for a sample) or N (for a population).
Since we're working with a sample, we'll use n – 1, where $n = 6$.

Variance
886 ÷ (6 – 1) = 886 ÷ 5 = **177.2**

Example:

```
# Defining vector

x <- c(46, 69, 32, 60, 52, 41)

# Print variance of xprint(var(x))
```

Output:
[1] 177.2

Range:
In statistics, the **range** is the spread of your data from the lowest to the highest value in the distribution. It is a commonly used measure of variability.

The range is calculated by subtracting the lowest value from the highest value. While a large range means high variability, a small range means low variability in a distribution.

Calculate the range
The formula to calculate the range is:

$$R = H - L$$

- *R = range*
- *H = highest value*
- *L = lowest value*

The range is the easiest measure of variability to calculate.
To find the range, follow these steps:

1. Order all values in your data set from low to high.
2. Subtract the lowest value from the highest value.

This process is the same regardless of whether your values are positive or negative, or whole numbers or fractions.

Range example

Your data set is the ages of 8 participants.

Participant	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Age	37	19	31	29	21	26	33	36

First, order the values from low to high to identify the lowest value (*L*) and the highest value (*H*).

Age	19	21	26	29	31	33	36	37

Then subtract the lowest from the highest value.

$$\boldsymbol{R = H - L}$$

$$R = 37 - 19 = 18$$

The range of our data set is **18 years**.

How useful is the range?

The range generally gives you a good indicator of variability when you have a distribution without extreme values. When paired with measures of central tendency, the range can tell you about the span of the distribution.

Example:

```
# Defining vector
x <- c(19, 21, 26, 29, 31, 33, 36, 37)

# range() function output
print(range(x))

# Using max() and min() function
# to calculate the range of data set
print(max(x)-min(x))
```

Output:

```
print(range(x))
[1] 19 37

# Using max() and min() function
# To calculate the range of data set
print(max(x)-min(x))
[1] 18
```

Standard Deviation

- The **standard deviation** is the average amount of variability in your dataset. It tells you, on average, how far each value lies from the mean.
- A high standard deviation means that values are generally far from the mean, while a low standard deviation indicates that values are clustered close to the mean.

What does standard deviation tell you?

Standard deviation is a useful measure of spread for **normal distributions**.

In normal distributions, data is symmetrically distributed with no skew. Most values cluster around a central region, with values tapering off as they go further away from the center. The standard deviation tells you how spread out from the center of the distribution your data is on average.

Many scientific variables follow normal distributions, including height, standardized test scores, or job satisfaction ratings. When you have the standard deviations of different samples, you can compare their distributions using statistical tests to make inferences about the larger populations they came from.

Standard deviation formulas for populations and samples

Different formulas are used for calculating standard deviations depending on whether you have data from a whole population or a sample.

Population standard deviation

When you have collected data from every member of the population that you're interested in, you can get an exact value for population standard deviation.

The **population standard deviation** formula looks like this:

Formula	Explanation
$\sigma = \sqrt{\frac{\Sigma (X-\mu)^2}{N}}$	❖ σ = population standard deviation ❖ ∑ = sum of… ❖ *X* = each value ❖ μ = population mean ❖ *N* = number of values in the population

Sample standard deviation

When you collect data from a sample, the sample standard deviation is used to make estimates or inferences about the population standard deviation.

The **sample standard deviation** formula looks like this:

Formula	Explanation
$s = \sqrt{\frac{\Sigma (X-\bar{x})^2}{n-1}}$	❖ *s* = sample standard deviation ❖ ∑ = sum of… ❖ *X* = each value ❖ $\bar{x}$ = sample mean ❖ *n* = number of values in the sample

With samples, we use $n - 1$ in the formula because using n would give us a biased estimate that consistently underestimates variability.

The sample standard deviation would tend to be lower than the real standard deviation of the population.

Reducing the sample n to $n - 1$ makes the standard deviation artificially large, giving you a conservative estimate of variability.

Steps for calculating the standard deviation

The standard deviation is usually calculated automatically by whichever software you use for your statistical analysis. But you can also calculate it by hand to better understand how the formula works.

There are six main steps for finding the standard deviation by hand.

We'll use a small data set of 6 scores to walk through the steps.

Data set					
46	69	32	60	52	41

Step 1: Find the mean

To find the mean, add up all the scores, then divide them by the number of scores.

Mean ($\bar{x}$)
$\bar{x}$ = (46 + 69 + 32 + 60 + 52 + 41) ÷ 6 = **50**

Step 2: Find each score's deviation from the mean
Subtract the mean from each score to get the deviations from the mean.
Since $\bar{x}$ = 50, here we take away 50 from each score.

Score	Deviation from the mean
46	46 – 50 = **-4**
69	69 – 50 = **19**
32	32 – 50 = **-18**
60	60 – 50 = **10**
52	52 – 50 = **2**
41	41 – 50 = **-9**

Step 3: Square each deviation from the mean
Multiply each deviation from the mean by itself. This will result in positive numbers.

Squared deviations from the mean
$(-4)^2$ = 4 × 4 = **16**
19^2 = 19 × 19 = **361**
$(-18)^2$ = -18 × -18 = **324**
10^2 = 10 × 10 = **100**
2^2 = 2 × 2 = **4**
$(-9)^2$ = -9 × -9 = **81**

Step 4: Find the sum of squares
Add up all of the squared deviations. This is called the sum of squares.

Sum of squares
16 + 361 + 324 + 100 + 4 + 81 = **886**

Step 5: Find the variance
Divide the sum of the squares by *n* – 1 (for a sample) or *N* (for a population) – this is the variance.

Since we're working with a sample size of 6, we will use $n - 1$, where $n = 6$.

Variance
886 ÷ (6 – 1) = 886 ÷ 5 = **177.2**

Step 6: Find the square root of the variance
To find the standard deviation, we take the square root of the variance.

Standard deviation
√177.2 = **13.31**

From learning that *SD* = 13.31, we can say that each score deviates from the mean by 13.31 points on average.

You can calculate standard deviation in R using the sd() function. This standard deviation function is a part of standard R, and needs no extra packages to be calculated.

Example:

```
# Defining vector
x <- c(46, 69, 32, 60, 52, 41)

# Standard deviation
d <- sqrt(var(x))

# Print standard deviation of x
print(d)

# Use sd() function to calculate Standard Deviation
print(sd(x))
```

Output:

```
# Print standard deviation of x
  print(d)
[1] 13.31165

# Use sd() function to calculate Standard Deviation
   print(sd(x))
[1] 13.31165
```

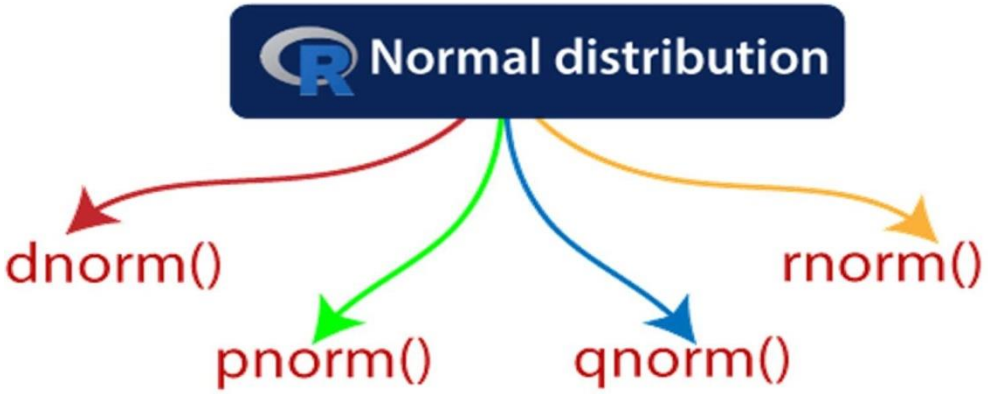
Week 9

9 (a) Implement R Script to perform Normal, Binomial distributions.

R Normal Distribution

In random collections of data from independent sources, it is commonly seen that the distribution of data is normal. It means that if we plot a graph with the value of the variable in the horizontal axis and counting the values in the vertical axis, then we get a bell shape curve. The curve center represents the mean of the data set. In the graph, fifty percent of the value is located to the left of the mean. Andthe other fifty percent to the right of the graph. This is referred to as the normal distribution.

R allows us to generate normal distribution by providing the following functions:



This function can have the following parameters:

S.No	Parameter	Description
1.	x	It is a vector of numbers.
2.	p	It is a vector of probabilities.
3.	n	It is a vector of observations.
4.	mean	It is the mean value of the sample data whose default value is zero.
5.	sd	It is the standard deviation whose default value is 1.

Let's start understanding how these functions are used with the help of the examples.

dnorm():Density

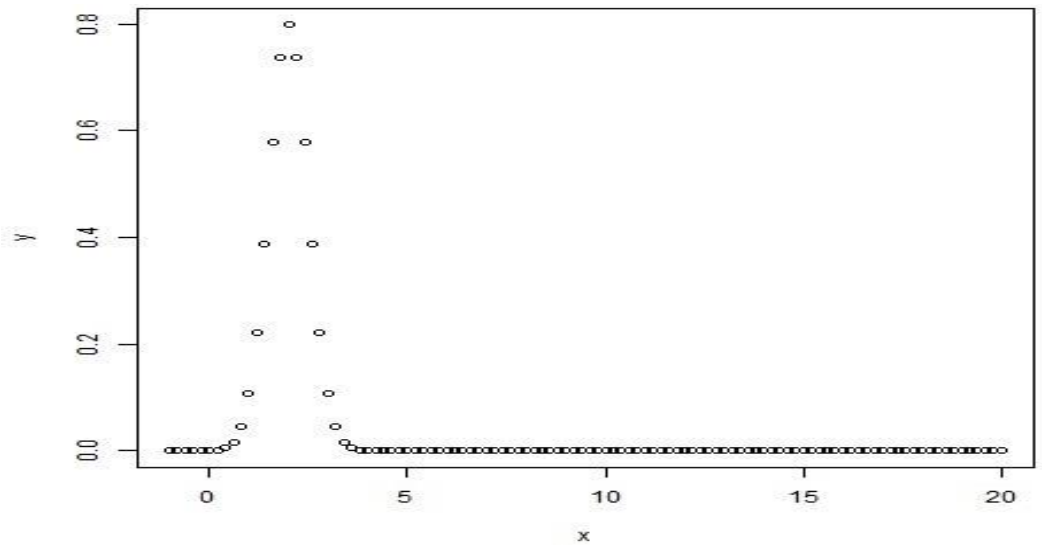
The dnorm() function of R calculates the height of the probability distribution at each point for a given mean and standard deviation. The probability density of the normal distribution is:

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

Example

```
# Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
x <- seq(-1, 20, by = .2)
# Choosing the mean as 2.0 and standard deviation as 0.5.
y <- dnorm(x, mean = 2.0, sd = 0.5)
# Giving a name to the chart file.
png(file = "dnorm.png")
#Plotting the graph
plot(x,y)
# Saving the file.
dev.off()
```

Output:



pnorm():Direct Look-Up

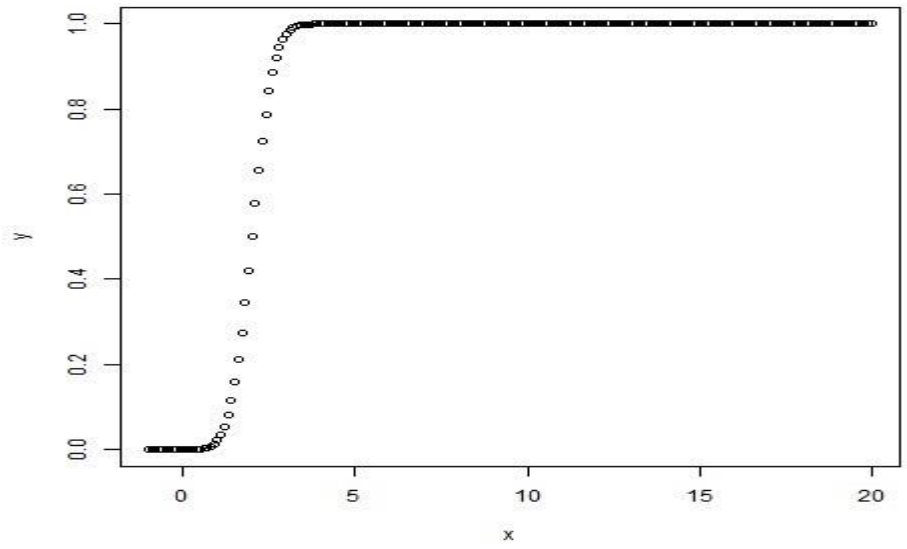
The dnorm() function is also known as "Cumulative Distribution Function". This function calculates the probability of a normally distributed random numbers, which is less than the value of a given number. The cumulative distribution is as follows:

$$f(x)=P(X\leq x)$$

Example:

```
# Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
x <- seq(-1, 20, by = .1)
# Choosing the mean as 2.0 and standard deviation as 0.5.
y <- pnorm(x, mean = 2.0, sd = 0.5)
# Giving a name to the chart file.
png(file = "pnorm.png")
#Plotting the graph
plot(x,y)
# Saving the file.
dev.off()
```

Output:



qnorm():Inverse Look-Up

The qnorm() function takes the probability value as an input and calculates a number whose cumulative value matches with the probability value. The cumulative distribution function and the inverse cumulative distribution function are related by

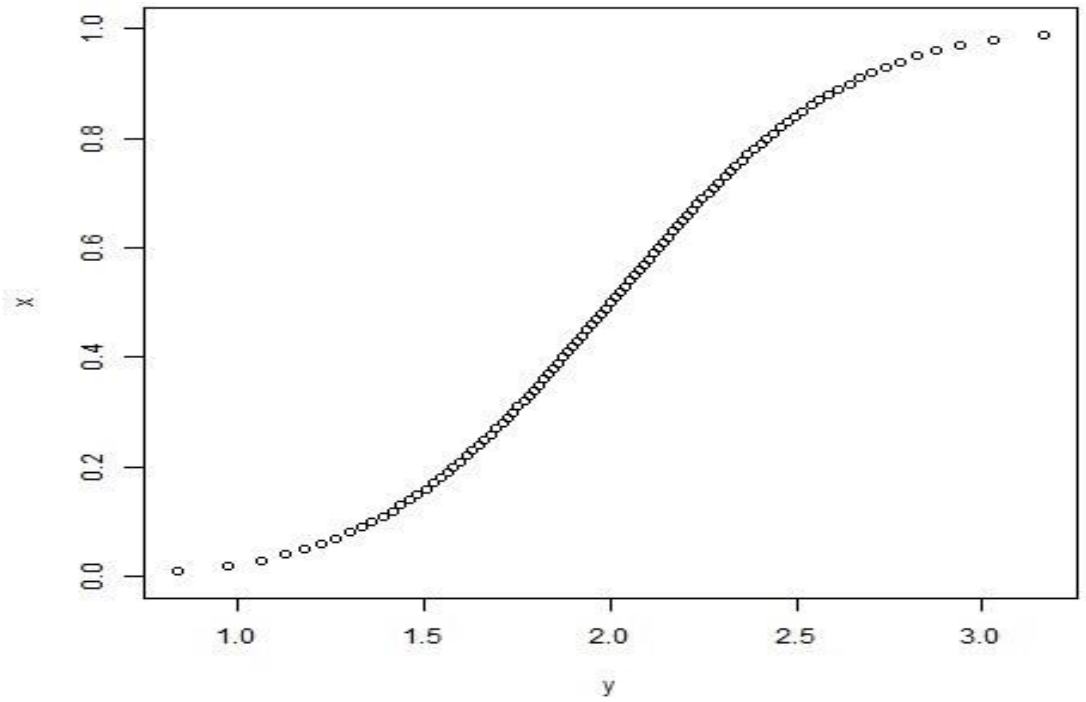
$$p=f(x)$$

$$x=f^{-1}(p)$$

Example:

```
# Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
x <- seq(0, 1, by = .01)
# Choosing the mean as 2.0 and standard deviation as 0.5.
y <- qnorm(x, mean = 2.0, sd = 0.5)
# Giving a name to the chart file.
png(file = "qnorm.png")
#Plotting the graph
plot(y,x)
# Saving the file.
dev.off()
```

Output:



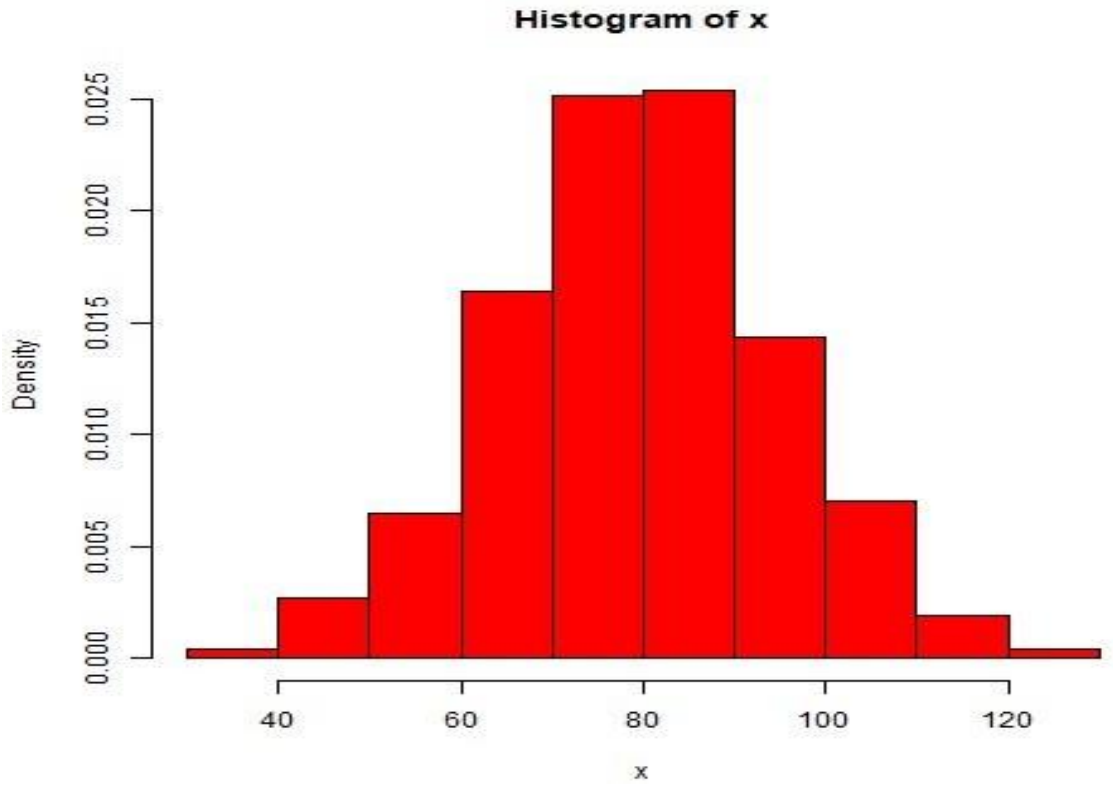
rnorm():Random variates

The rnorm() function is used for generating normally distributed random numbers. This function generates random numbers by taking the sample size as an input. Let's see an example in which we draw a histogram for showing the distribution of the generated numbers.

Example:

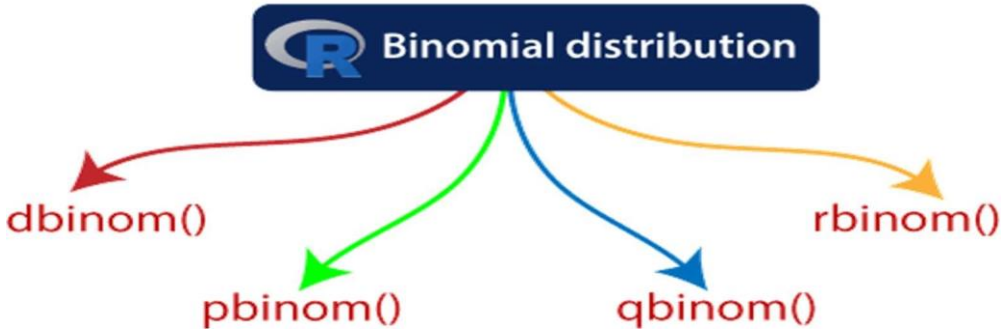
```
# Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
x <- rnorm(1500, mean=80,  sd=15 )
# Giving a name to the chart file.
png(file = "rnorm.png")
#Creating histogram
hist(x,probability =TRUE,col="red",border="black")
# Saving the file.
dev.off()
```

Output:



Binomial Distribution

The binomial distribution is also known as **discrete probability distribution**, which is used to find the probability of success of an event. The event has only two possible outcomes in a series of experiments. The tossing of the coin is the best example of the binomial distribution. When a coin is tossed, it gives either a head or a tail. The probability of finding exactly three heads in repeatedly tossing the coin ten times is approximate during the binomial distribution. R allows us to create binomial distribution by providing the following function:



This function can have the following parameters:

S.No	Parameter	Description
1.	x	It is a vector of numbers.
2.	p	It is a vector of probabilities.
3.	n	It is a vector of observations.
4.	size	It is the number of trials.
5.	prob	It is the probability of the success of each trial.

Let's start understanding how these functions are used with the help of the examples

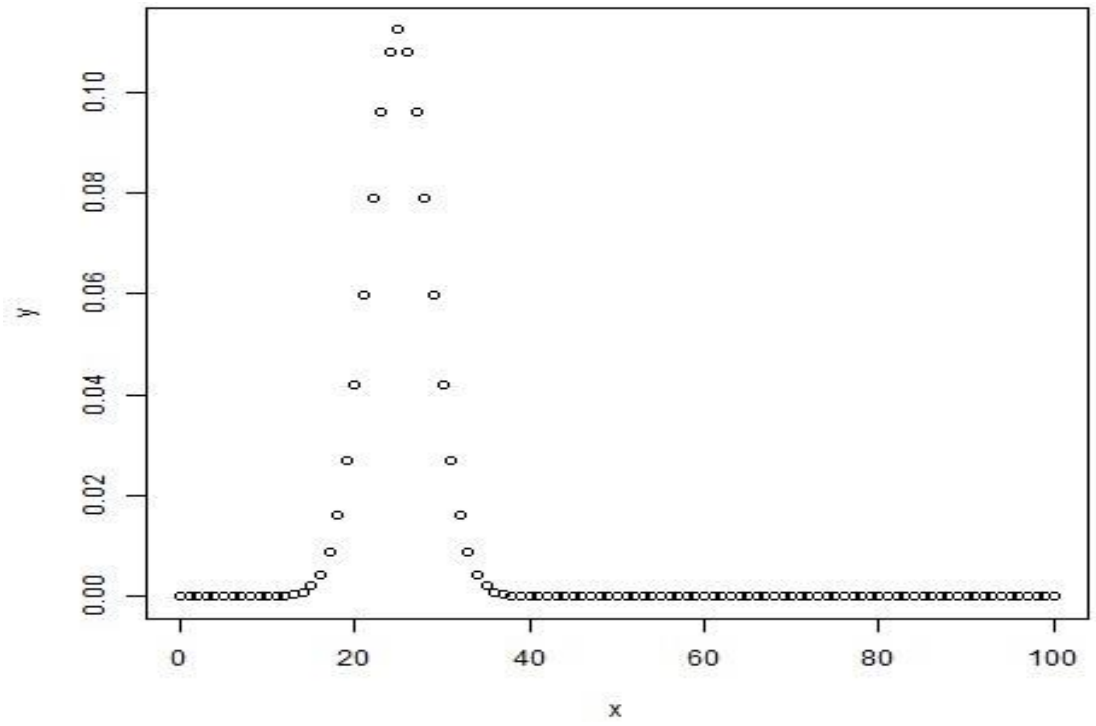
dbinom(): Direct Look-Up, Points

The dbinom() function of R calculates the probability density distribution at each point. In simple words, it calculates the density function of the particular binomial distribution.

Example

```
# Creating a sample of 100 numbers which are incremented by 1.5.
x <- seq(0,100,by = 1)
# Creating the binomial distribution.
y <- dbinom(x,50,0.5)
# Giving a name to the chart file.
png(file = "dbinom.png")
# Plotting the graph.
plot(x,y)
# Saving the file.
dev.off()
```

Output:



pbinom():Direct Look-Up, Intervals

The dbinom() function of R calculates the cumulative probability(a single value representing the probability) of an event. In simple words, it calculates the cumulative distribution function of the particular binomial distribution.

Example

```
# Probability of getting 20 or fewer heads from 48 tosses of a coin.
x <- pbinom(20,48,0.5)
#Showing output
print(x)
```

Output:
[1] 0.1561634

qbinom(): Inverse Look-Up
The qbinom() function of R takes the probability value and generates a number whose cumulative value matches with the probability value. In simple words, it calculates the inverse cumulative distribution function of the binomial distribution.
Let's find the number of heads that have a probability of 0.45 when a coin is tossed 51 times.
Example
```
# Finding number of heads with the help of qbinom() function
x <- qbinom(0.45,48,0.5)
#Showing output
print(x)
```

Output:
[1] 24

rbinom()
The rbinom() function of R is used to generate required number of random values for given probability from a given sample.
Let's see an example in which we find nine random values from a sample of 160 with a probability of 0.5.
Example:
```
# Finding random values
x <- rbinom(9,160,0.5)
#Showing  output
print(x)
```

Output:
[1] 94 74 83 81 78 84 77 78 81

9(b) Implement R Script to perform correlation, Linear and multiple regression.
A step-by-step guide to linear regression in R
Linear regression is a regression model that uses a straight line to describe the relationship between variables. It finds the line of best fit through your data by searching for the value of the regression coefficient(s) that minimizes the total error of the model.
There are two main types of linear regression:

- **Simple linear regression** uses only one independent variable
- **Multiple linear regression** uses two or more independent variables

In this step-by-step guide, we will walk you through linear regression in R using two sample datasets.
Simple linear regression:
The first dataset contains observations about income (in a range of $15k to $75k) and happiness (rated on a scale of 1 to 10) in an imaginary sample of 500 people. The income values are divided by 10,000 to make the income data match the scale of the happiness scores (so a value of $2 represents $20,000, $3 is $30,000, etc.)
Multiple linear regression:

The second dataset contains observations on the percentage of people biking to work each day, the percentage of people smoking, and the percentage of people with heart disease in an imaginary sample of 500 towns.

To install the packages, you need for the analysis, run this code:

```
install.packages("ggplot2")
install.packages("dplyr")
install.packages("broom")
install.packages("ggpubr")
```

Next, load the packages into your R environment by running this code:

```
library(ggplot2)
library(dplyr)
library(broom)
library(ggpubr)
```

Step 1: Load the data into R

Follow these four steps for each dataset:

1. In RStudio, go to **File > Import dataset > From Text (base)**.
2. Choose the data file you have downloaded (income.data or heart.data), and an **Import Dataset** window pops up.
3. In the **Data Frame** window, you should see an **X** (index) column and columns listing the datafor each of the variables (income and happiness or biking, smoking, and heart.disease).
4. Click on the **Import** button and the file should appear in your **Environment** tab on the upper right side of the RStudio screen.

After you've loaded the data, check that it has been read in correctly using summary().

Simple regression

```
summary(income.data)
```

- Because both our variables are quantitative, when we run this function, we see a table in our console with a numeric summary of the data. This tells us the minimum, median, mean, and maximum values of the independent variable (income) and dependent variable (happiness):

```
       X                income          happiness    
 Min.   :  1.0   Min.   :1.506   Min.   :0.266  
 1st Qu.:125.2   1st Qu.:3.006   1st Qu.:2.266  
 Median :249.5   Median :4.424   Median :3.473  
 Mean   :249.5   Mean   :4.467   Mean   :3.393  
 3rd Qu.:373.8   3rd Qu.:5.992   3rd Qu.:4.503  
 Max.   :498.0   Max.   :7.482   Max.   :6.863  
```

Multiple regression

```
summary(heart.data)
```

- Again, because the variables are quantitative, running the code produces a numeric summary of the data for the independent variables (smoking and biking) and the dependent variable (heart disease):

```
       X                biking          smoking      heart.disease
 Min.   :  1.0   Min.   : 1.119   Min.   : 0.5259   Min.   : 0.5519
 1st Qu.:125.2   1st Qu.:20.205   1st Qu.: 8.2798   1st Qu.: 6.5137
 Median :249.5   Median :35.824   Median :15.8146   Median :10.3853
 Mean   :249.5   Mean   :37.788   Mean   :15.4350   Mean   :10.1745
 3rd Qu.:373.8   3rd Qu.:57.853   3rd Qu.:22.5689   3rd Qu.:13.7240
 Max.   :498.0   Max.   :74.907   Max.   :29.9467   Max.   :20.4535
```

Step 2: Make sure your data meet the assumptions

We can use R to check that our data meet the four main assumptions for linear regression.

Simple regression

1. **Independence of observations** (aka no autocorrelation)

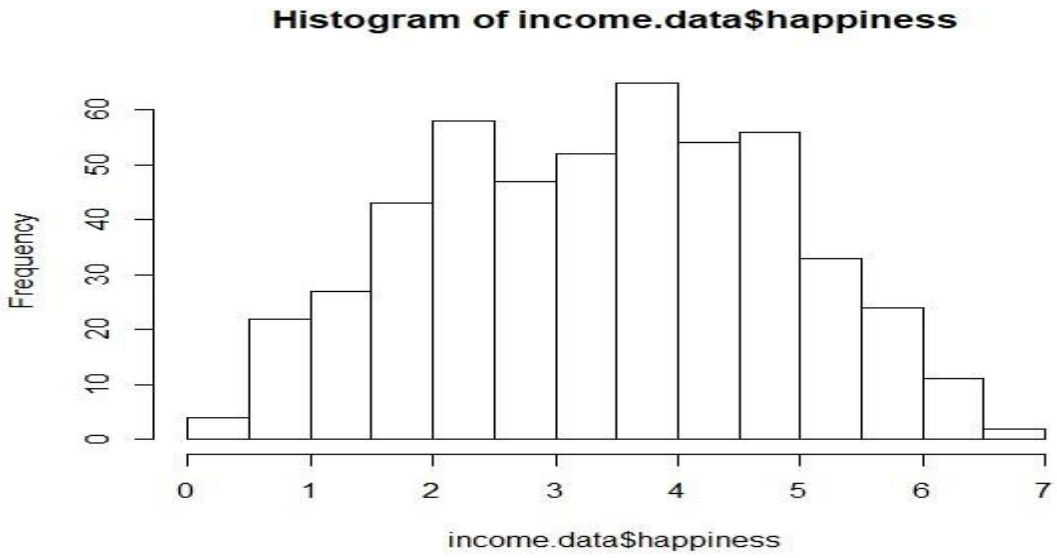
Because we only have one independent variable and one dependent variable, we don't need to test for any hidden relationships among variables.

If you know that you have autocorrelation within variables (i.e. multiple observations of the same test subject), then do not proceed with a simple linear regression! Use a structured model, like a linear mixed-effects model, instead.

2. **Normality**

To check whether the dependent variable follows a normal distribution, use the hist()function.

```
hist(income.data$happiness)
```

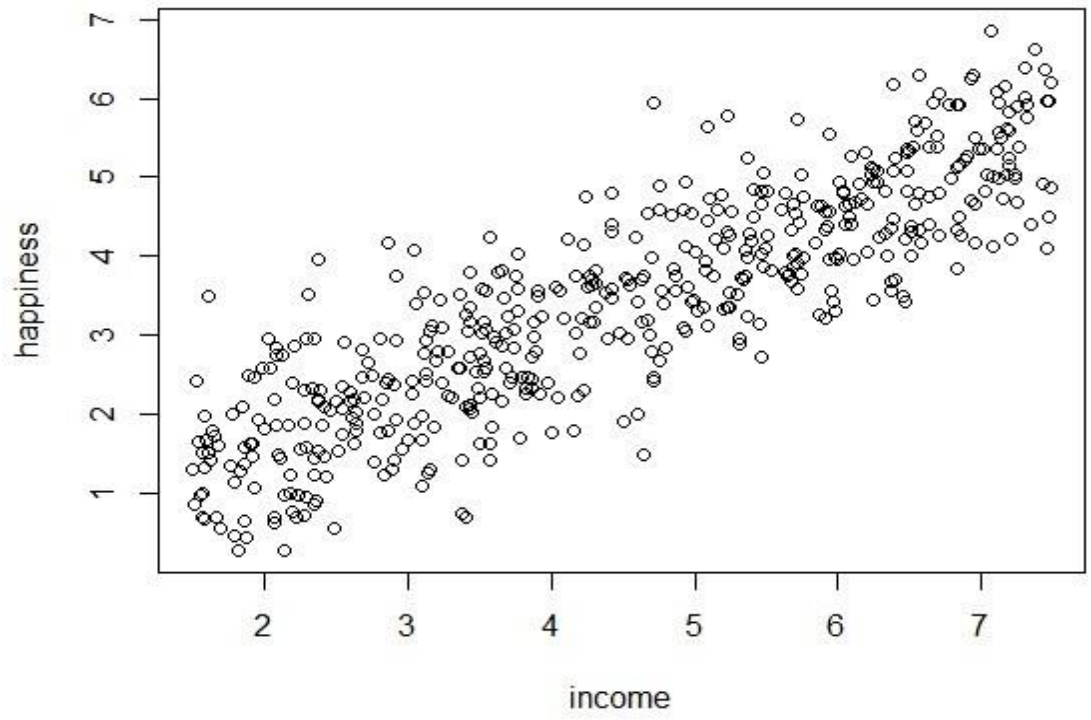


The observations are roughly bell-shaped (more observations in the middle of the distribution, fewer on the tails), so we can proceed with the linear regression.

3. **Linearity**

The relationship between the independent and dependent variable must be linear. We can test this visually with a scatter plot to see if the distribution of data points could be described with a straight line.

```
plot(happiness ~ income, data = income.data)
```



The relationship looks roughly linear, so we can proceed with the linear model.

4. **Homoscedasticity** (aka homogeneity of variance)

This means that the prediction error doesn't change significantly over the range of prediction of the model. We can test this assumption later, after fitting the linear model.

Multiple regression

1. **Independence of observations** (aka no autocorrelation)

Use the cor()function to test the relationship between your independent variables and make sure they aren't too highly correlated.

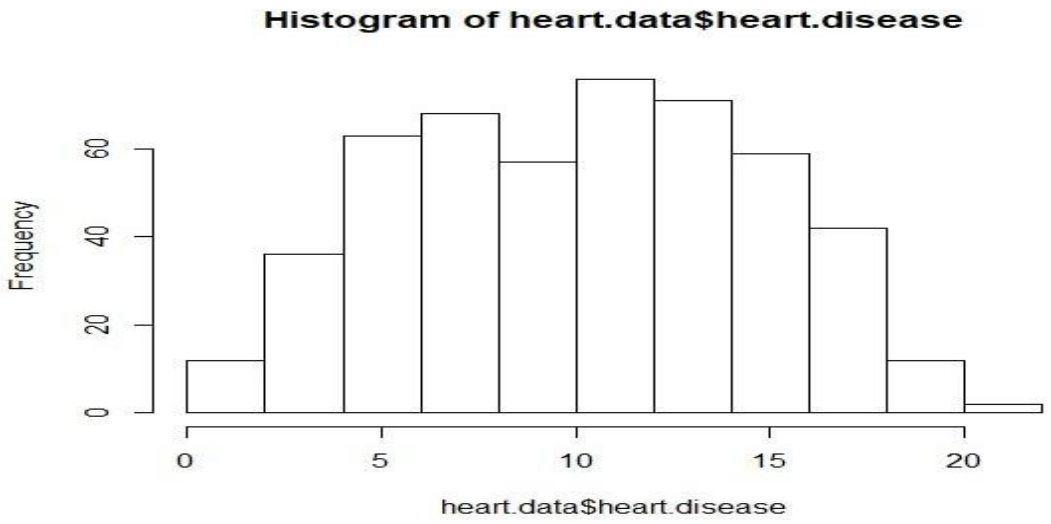
```
cor(heart.data$biking, heart.data$smoking)
```

When we run this code, the output is 0.015. The correlation between biking and smoking is small (0.015 is only a 1.5% correlation), so we can include both parameters in our model.

2. **Normality**

Use the hist()function to test whether your dependent variable follows a normal distribution.

```
hist(heart.data$heart.disease)
```

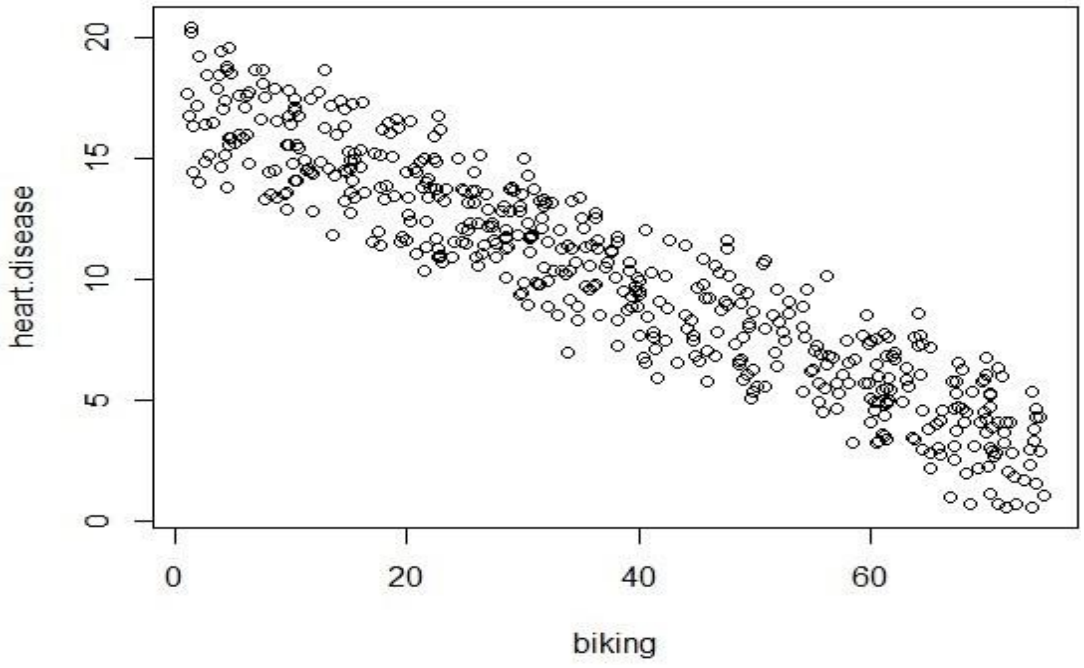


The distribution of observations is roughly bell-shaped, so we can proceed with the linear regression.

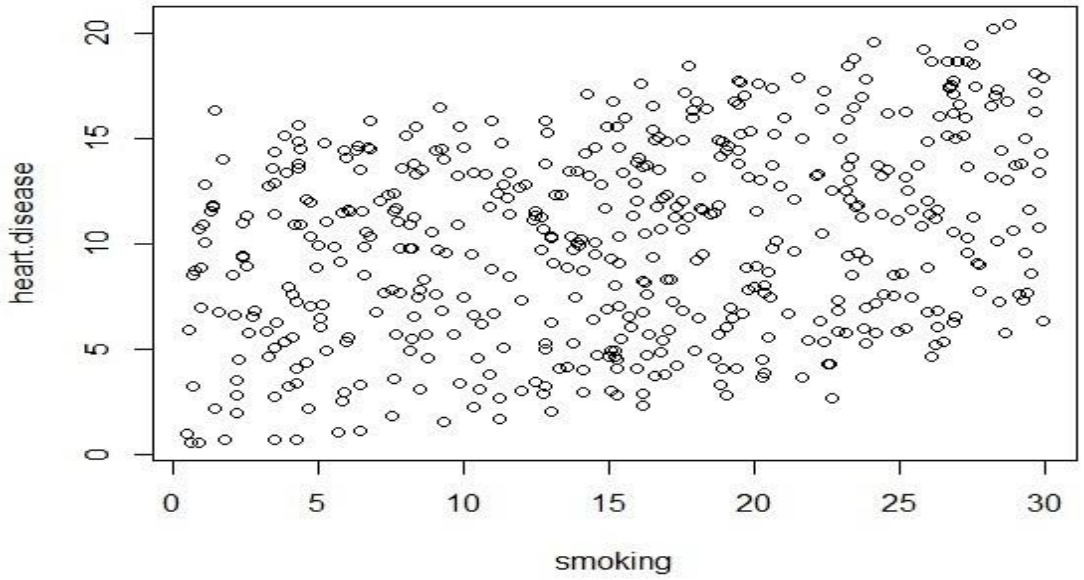
3. **Linearity**

We can check this using two scatterplots: one for biking and heart disease, and one for smoking and heart disease.

plot(heart.disease ~ biking, data=heart.data)



plot(heart.disease ~ smoking, data=heart.data)



Although the relationship between smoking and heart disease is a bit less clear, it still appears linear. We can proceed with linear regression.

4. **Homoscedasticity**

We will check this after we make the model.

Step 3: Perform the linear regression analysis

Now that you've determined your data meet the assumptions, you can perform a linear regression analysis to evaluate the relationship between the independent and dependent variables.

Simple regression: income and happiness

Let's see if there's a linear relationship between income and happiness in our survey of 500 people with incomes ranging from $15k to $75k, where happiness is measured on a scale of 1 to 10.

To perform a simple linear regression analysis and check the results, you need to run two lines of code. The first line of code makes the linear model, and the second line prints out the summary of the model:

```
income.happiness.lm <- lm(happiness ~ income, data = income.data)
summary(income.happiness.lm)
```

The output looks like this:

```
Call:
lm(formula = happiness ~ income, data = income.data)

Residuals:
     Min       1Q   Median       3Q      Max
-2.02479 -0.48526  0.04078  0.45898  2.37805

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.20427    0.08884   2.299   0.0219 *
income       0.71383    0.01854  38.505   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7181 on 496 degrees of freedom
Multiple R-squared:  0.7493,    Adjusted R-squared:  0.7488
F-statistic:  1483 on 1 and 496 DF,  p-value: < 2.2e-16
```

This output table first presents the model equation, then summarizes the model residuals (see step 4).

The **Coefficients** section shows:

1. The estimates (**Estimate**) for the model parameters – the value of the y-intercept (in this case 0.204) and the estimated effect of income on happiness (0.713).
2. The standard error of the estimated values (**Std. Error**).
3. The test statistic (**t value**, in this case the *t*-statistic).
4. The *p*-value (**Pr(>| t |)**), aka the probability of finding the given t-statistic if the null hypothesis of no relationship were true.

The final three lines are model diagnostics – the most important thing to note is the ***p*-value** (here it is 2.2e-16, or almost zero), which will indicate whether the model fits the data well.

From these results, we can say that there is a **significant positive relationship** between income and happiness (*p*-value < 0.001), with a 0.713-unit (+/- 0.01) increase in happiness for every unit increase in income.

Multiple regression: biking, smoking, and heart disease

Let's see if there's a linear relationship between biking to work, smoking, and heart disease in our imaginary survey of 500 towns. The rates of biking to work range between 1 and 75%, rates of smoking between 0.5 and 30%, and rates of heart disease between 0.5% and 20.5%.

To test the relationship, we first fit a linear model with heart disease as the dependent variable and biking and smoking as the independent variables. Run these two lines of code:

heart.disease.lm<-lm(heart.disease ~ biking + smoking, data = heart.data)

summary(heart.disease.lm)

The output looks like this:

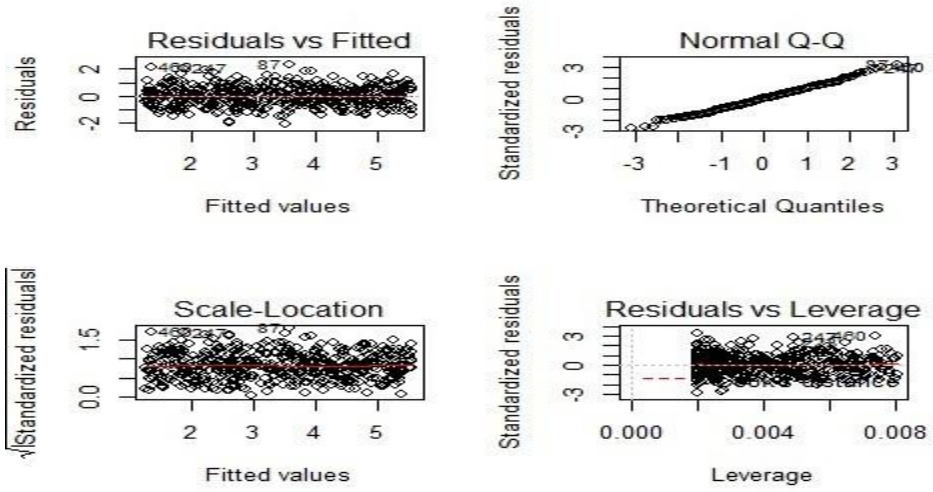
```
Call:
lm(formula = heart.disease ~ biking + smoking, data = heart.data)

Residuals:
    Min      1Q  Median      3Q     Max
-2.1789 -0.4463  0.0362  0.4422  1.9331

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 14.984658   0.080137  186.99   <2e-16 ***
biking      -0.200133   0.001366 -146.53   <2e-16 ***
smoking      0.178334   0.003539   50.39   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.654 on 495 degrees of freedom
Multiple R-squared:  0.9796,    Adjusted R-squared:  0.9795
F-statistic: 1.19e+04 on 2 and 495 DF,  p-value: < 2.2e-16
```

The estimated effect of biking on heart disease is -0.2, while the estimated effect of smoking is 0.178.
This means that for every 1% increase in biking to work, there is a correlated 0.2% decrease in the incidence of heart disease. Meanwhile, for every 1% increase in smoking, there is a 0.178% increase in the rate of heart disease.
The standard errors for these regression coefficients are very small, and the t-statistics are very large (-147 and 50.4, respectively). The *p*-values reflect these small errors and large t-statistics. For both parameters, there is almost zero probability that this effect is due to chance.

Step 4: Check for homoscedasticity
Before proceeding with data visualization, we should make sure that our models fit the homoscedasticity assumption of the linear model.
Simple regression
We can run plot(income.happiness.lm)to check whether the observed data meets ourmodel assumptions:

```
par(mfrow=c(2,2))
plot(income.happiness.lm)
par(mfrow=c(1,1))
```

Note that the par(mfrow())command will divide the **Plots** window into the number of rowsand columns specified in the brackets. So par(mfrow=c(2,2)) divides it up into two rows and two columns. To go back to plotting one graph in the entire window, set the parameters again and replace the (2,2) with (1,1).
These are the residual plots produced by the code:



Residuals are the unexplained variance. They are not exactly the same as model error, but they are calculated from it, so seeing a bias in the residuals would also indicate a bias in the error.

The most important thing to look for is that the red lines representing the mean of the residuals are all basically horizontal and centered around zero. This means there are no outliers or biases in the data that would make a linear regression invalid.

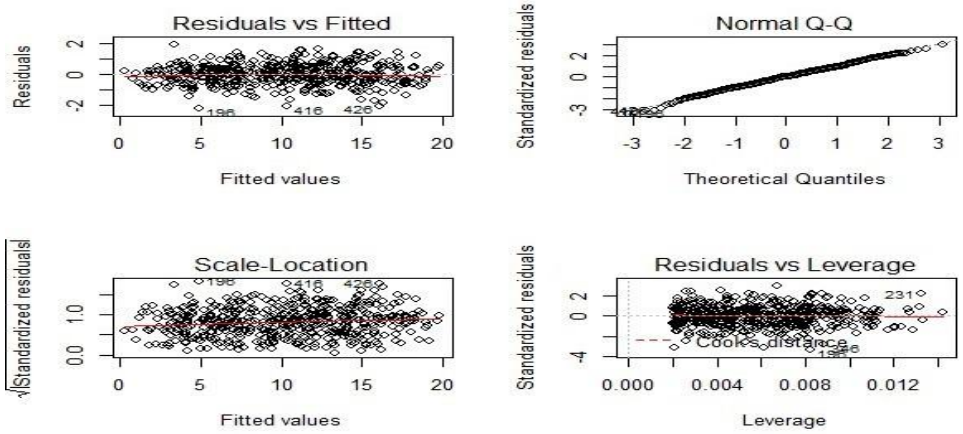
In the **Normal Q-Qplot** in the top right, we can see that the real residuals from our model form an almost perfectly one-to-one line with the theoretical residuals from a perfect model. Based on these residuals, we can say that our model meets the assumption of homoscedasticity.

Multiple regression

Again, we should check that our model is actually a good fit for the data, and that we don't have large variation in the model error, by running this code:

```
par(mfrow=c(2,2))
plot(heart.disease.lm)
par(mfrow=c(1,1))
```

The output looks like this:



As with our simple regression, the residuals show no bias, so we can say our model fits the assumption of homoscedasticity.

Step 5: Visualize the results with a graph

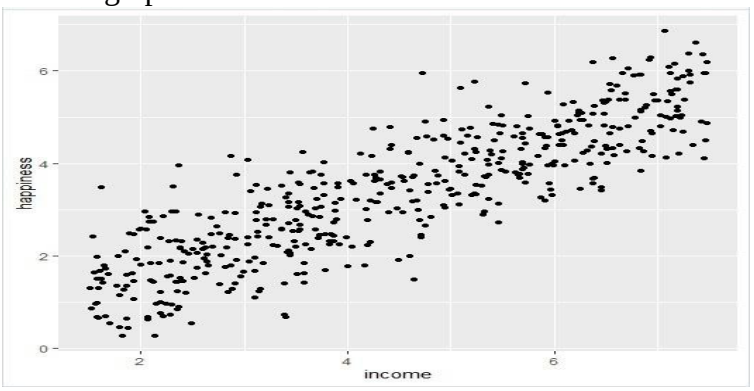
Next, we can plot the data and the regression line from our linear regression model so that the results can be shared.

Simple regression

Follow 4 steps to visualize the results of your simple linear regression.

1. **Plot the data points on a graph**

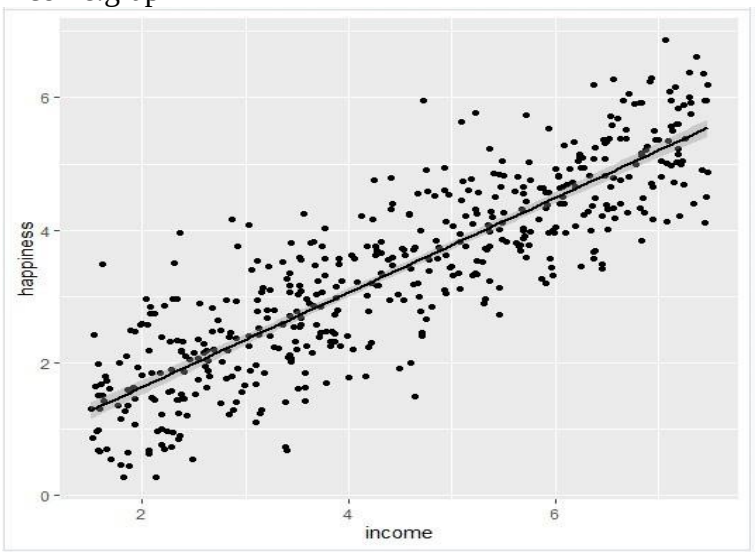
```
income.graph<-ggplot(income.data, aes(x=income, y=happiness))+
              geom_point()
income.graph
```



2. **Add the linear regression line to the plotted data**

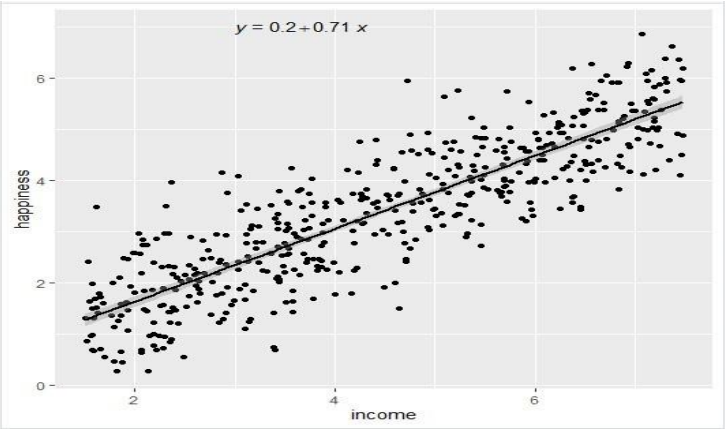
Add the regression line using geom_smooth()and typing in lmas your method for creatingthe line. This will add the line of the linear regression as well as the standard error of the estimate (in this case +/- 0.01) as a light grey stripe surrounding the line:

```
income.graph <- income.graph + geom_smooth(method="lm", col="black")
income.graph
```



3. **Add the equation for the regression line.**

```
income.graph <- income.graph + stat_regline_equation(label.x = 3, label.y = 7)
income.graph
```

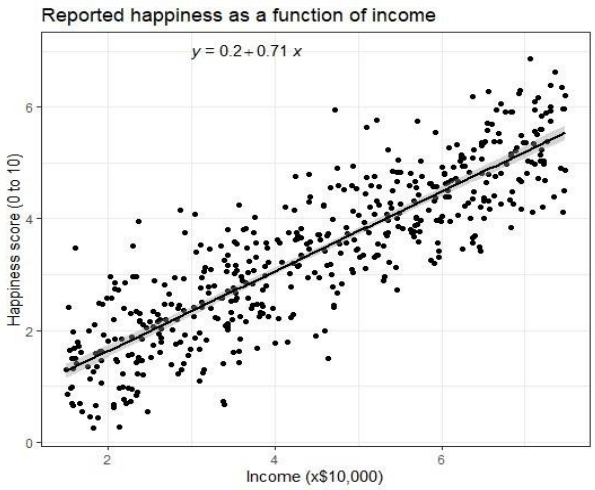


4. **Make the graph ready for publication**

We can add some style parameters using theme_bw()and making custom labels using labs().

```
income.graph + theme_bw() + labs(title = "Reported happiness as a function of income",
    x = "Income (x$10,000)",
    y = "Happiness score (0 to 10)")
```

This produces the finished graph that you can include in your papers:



Multiple regression

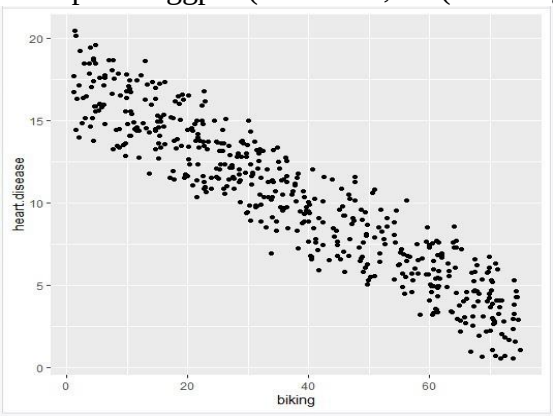
4. **Change the 'smoking' variable into a factor**

This allows us to plot the interaction between biking and heart disease at each of the three levels of smoking we chose.

```
plotting.data$smoking <- as.factor(plotting.data$smoking)
```

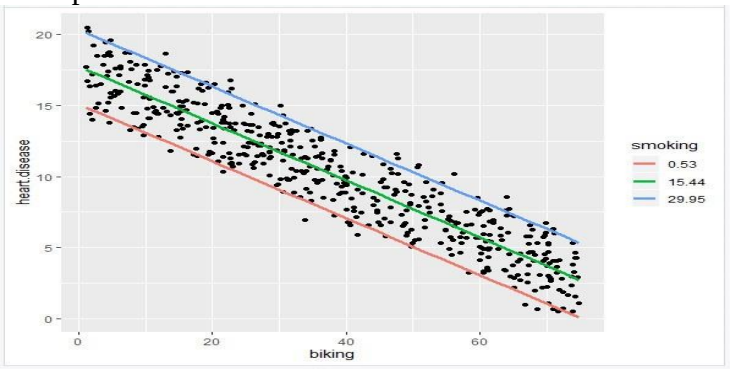
5. **Plot the original data**

heart.plot <- ggplot(heart.data, aes(x=biking, y=heart.disease)) + geom_point



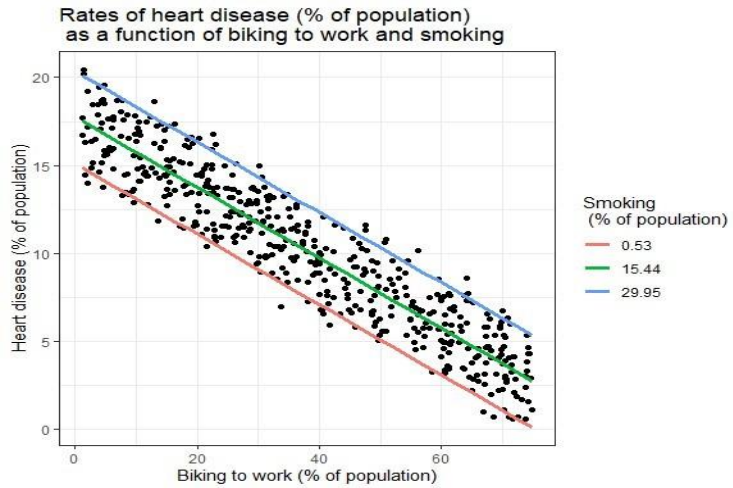
6. Add the regression lines

heart.plot <- heart.plot +
geom_line(data=plotting.data, aes(x=biking, y=predicted.y, color=smoking), size=1.25)
heart.plot



7. Make the graph ready for publication

heart.plot <- heart.plot + theme_bw() +
labs(title = "Rates of heart disease (% of population) \n as a function of biking to work and smoking", x = "Biking to work (% of population)", y = "Heart disease (% of population)", color = "Smoking \n (% of population)")
heart.plot



Because this graph has two regression coefficients, the stat_regline_equation()functionwon't work here. But if we want to add our regression model to the graph, we can do so like this:

heart.plot + annotate(geom="text", x=30, y=1.75, label=" = 15 + (-0.2*biking) + 0.178*smoking)")
This is the finished graph that you can include in your papers!

Step 6: Report your results
In addition to the graph, include a brief statement explaining the results of the regression model.
Specifically, we found a 0.2% decrease (± 0.0014) in the frequency of heart disease for every 1% increase in biking, and a 0.178% increase (± 0.0035) in the frequency of heart disease for every 1% increase in smoking.

What is the use of Linear regression analysis?
Linear regression analysis is **used to predict the value of a variable based on the value of another variable**. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

Week 10

10(a) Working with Non-Tabular Data Types: Time series, spatial data, Network data.

R - Time Series Analysis

Time series is a series of data points in which each data point is associated with a timestamp. A simple example is the price of a stock in the stock market at different points of time on a given day. Another example is the amount of rainfall in a region at different months of the year. R language uses many functions to create, manipulate and plot the time series data. The data for the time series is stored inan R object called **time-series object**. It is also a R data object like a vector or data frame.

The time series object is created by using the **ts()** function.

Syntax

The basic syntax for **ts()** function in time series analysis is −

```
timeseries.object.name <-  ts(data, start, end, frequency)
```

Following is the description of the parameters used −

- **data** is a vector or matrix containing the values used in the time series.
- **start** specifies the start time for the first observation in time series.
- **end** specifies the end time for the last observation in time series.
- **frequency** specifies the number of observations per unit time.

Except the parameter "data" all other parameters are optional.

Example

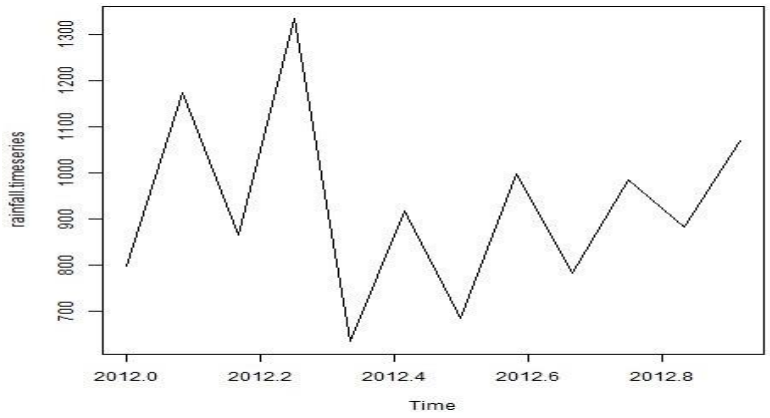
Consider the annual rainfall details at a place starting from January 2012. We create an R time series object for a period of 12 months and plot it.

```
# Get the data points in form of a R vector.
rainfall <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)

# Convert it to a time series object.
rainfall.timeseries <- ts(rainfall,start = c(2012,1),frequency = 12)

# Print the timeseries data.
print(rainfall.timeseries)

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

# Plot a graph of the time series.
plot(rainfall.timeseries)

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

Output:

```
print(rainfall.timeseries)
      Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
2012 799.0 1174.8  865.1 1334.6  635.4  918.5  685.5  998.6  784.2  985.0  882.8 1071.0
```

The Time series chart –



Multiple Time Series

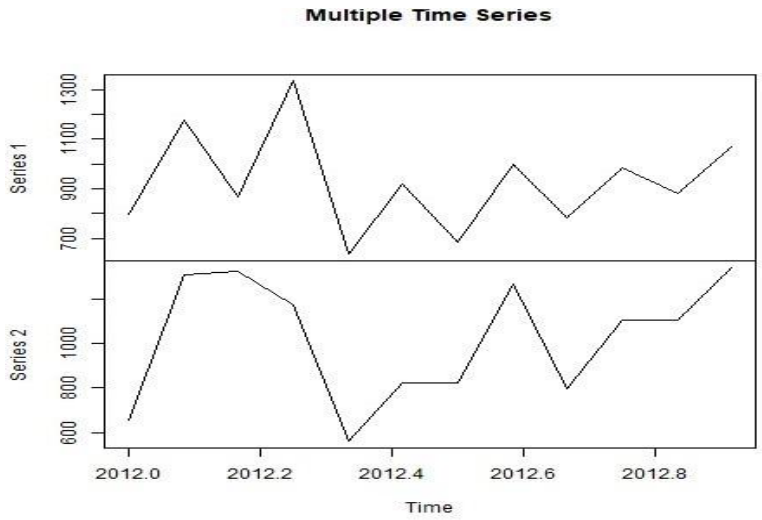
We can plot multiple time series in one chart by combining both the series into a matrix.

```
# Get the data points in form of a R vector.
rainfall1 <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)
rainfall2 <-
c(655,1306.9,1323.4,1172.2,562.2,824,822.4,1265.5,799.6,1105.6,1106.7,1337.8)
# Convert them to a matrix.
combined.rainfall <-  matrix(c(rainfall1,rainfall2),nrow = 12)
# Convert it to a time series object.
rainfall.timeseries <- ts(combined.rainfall,start = c(2012,1),frequency = 12)
# Print the timeseries data.
print(rainfall.timeseries)
# Give the chart file a name.
png(file = "rainfall_combined.png")
# Plot a graph of the time series.
plot(rainfall.timeseries, main = "Multiple Time Series")
# Save the file.
dev.off()
```

Output:

```
         Series 1 Series 2
Jan 2012    799.0    655.0
Feb 2012   1174.8   1306.9
Mar 2012    865.1   1323.4
Apr 2012   1334.6   1172.2
May 2012    635.4    562.2
Jun 2012    918.5    824.0
Jul 2012    685.5    822.4
Aug 2012    998.6   1265.5
Sep 2012    784.2    799.6
Oct 2012    985.0   1105.6
Nov 2012    882.8   1106.7
Dec 2012   1071.0   1337.8
```

The Multiple Time series chart −



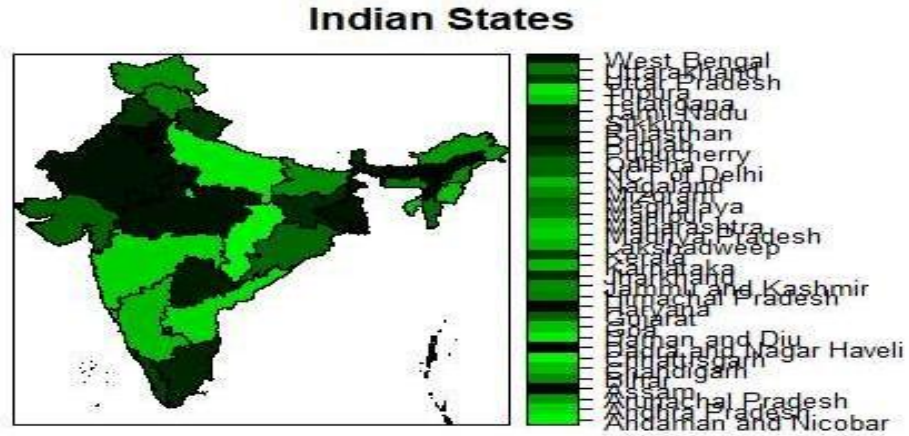
Spatial Data Analysis using R

Download the data set IND_adm1.rds from github

Make this data set available in current working directory. (IND_adm1.rds)

```
library(rjson)
library(ggmap)
library(RgoogleMaps)
library(png)
library(sp)
library(RColorBrewer)
gadm <- readRDS("IND_adm1.rds", refhook = NULL)
ind1 = gadm
ind1
spplot(ind1, "NAME_1", scales=list(draw=T), colorkey=F, main="India")
ind1$NAME_1 = as.factor(ind1$NAME_1)
ind1$fake.data = runif(length(ind1$NAME_1))
spplot(ind1,"NAME_1", col.regions=rgb(0,ind1$fake.data,0), colorkey=T, main="Indian
States")
```

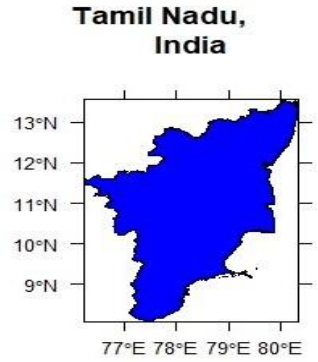
Output:




```
#Tamil Nadu
TN=ind1[ind1$NAME_1=="Tamil Nadu",]
```

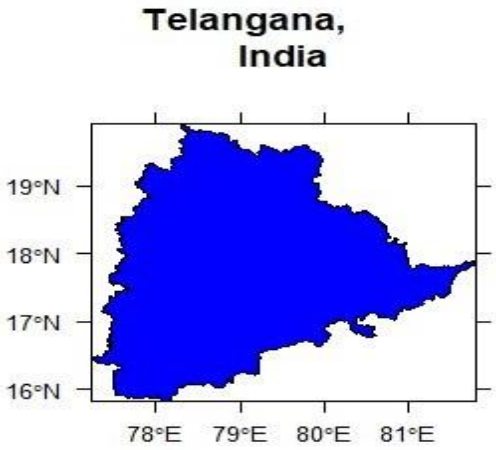
```
spplot(TN,"NAME_1", col.regions=rgb(0,0,1), main = "Tamil Nadu,
    India",scales=list(draw=T), colorkey =F)
```

Output:



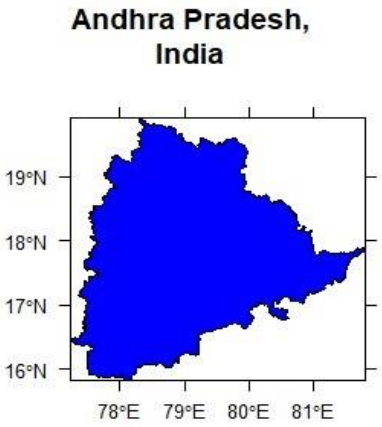

```
#Telangana
TS=ind1[ind1$NAME_1=="Telangana",]
spplot(TS,"NAME_1", col.regions=rgb(0,0,1), main = "Telangana,
    India",scales=list(draw=T), colorkey =F)
```

Output:




```
#Adhra Pradesh
AP=ind1[ind1$NAME_1=="Andhra Pradesh",]
spplot(TS,"NAME_1", col.regions=rgb(0,0,1), main = "Andhra Pradesh,
India",scales=list(draw=T), colorkey =F)
```

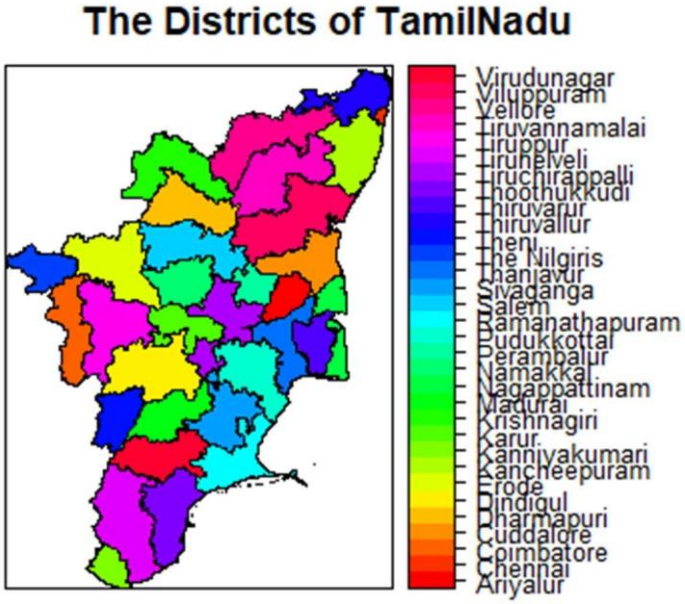
Output:



#Districts of TamilNadu

```
ind2=readRDS("IND_adm2.rds")
TN_districts = (ind2[ind2$NAME_1=="Tamil Nadu",])
TN_districts$NAME_2=as.factor(TN_districts$NAME_2)
col = rainbow(length(levels(TN_districts$NAME_2)))
spplot(TN_districts,"NAME_2", main="The Districts of TamilNadu",col.regions=col,
colorkey=T)
```

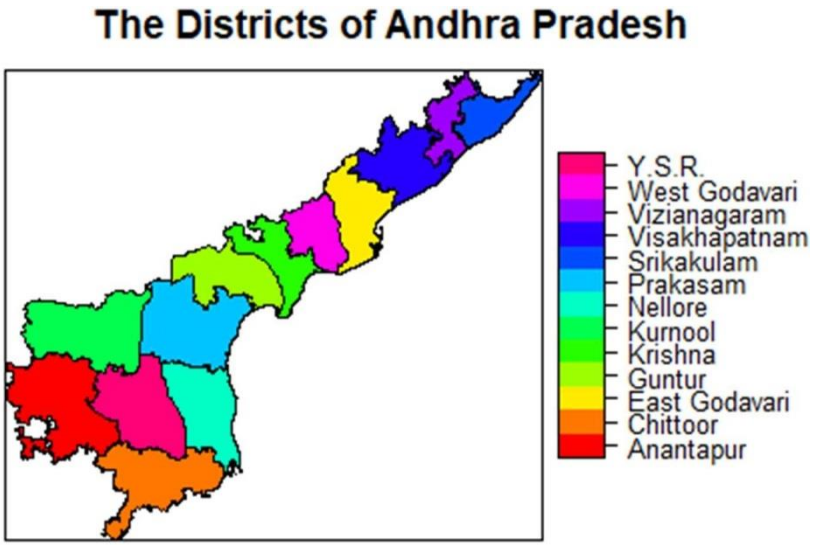
Output:



#Districts of Andhra Pradesh

```
AP_districts = (ind2[ind2$NAME_1=="Andhra Pradesh",])
AP_districts$NAME_2=as.factor(AP_districts$NAME_2)
col = rainbow(length(levels(AP_districts$NAME_2)))
spplot(AP_districts,"NAME_2",main="The Districts of Andhra Pradesh",
col.regions=col, colorkey=T)
```

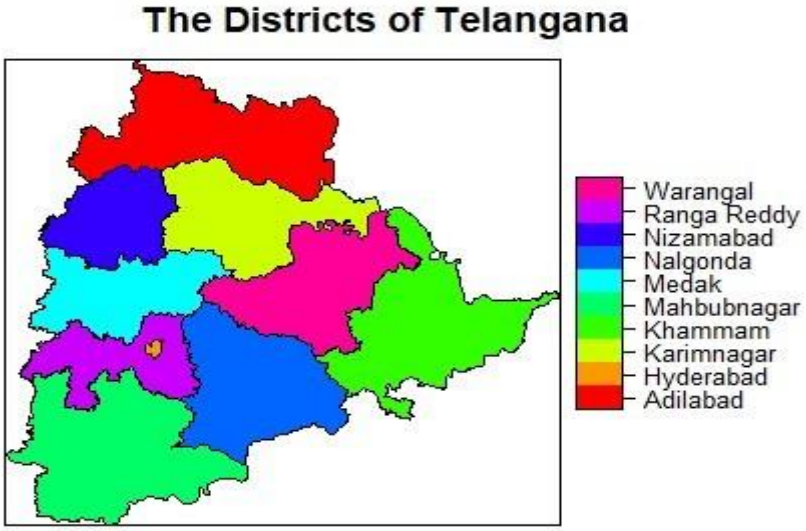
Output:



#Districts of Telangana

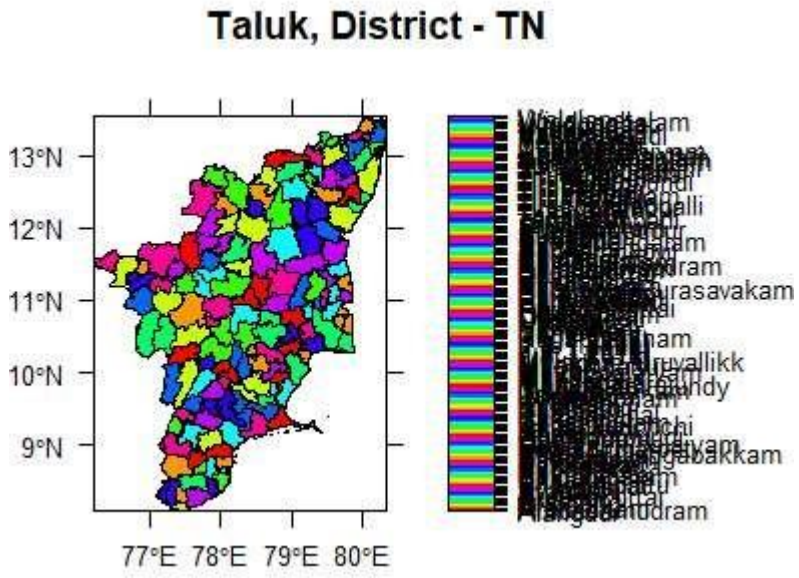
```
TS_districts = (ind2[ind2$NAME_1=="Telangana",])
TS_districts$NAME_2=as.factor(TS_districts$NAME_2)
col = rainbow(length(levels(TS_districts$NAME_2)))
spplot(TS_districts,"NAME_2", main="The Districts of Telangana", col.regions=col,
colorkey=T)
```

Output:



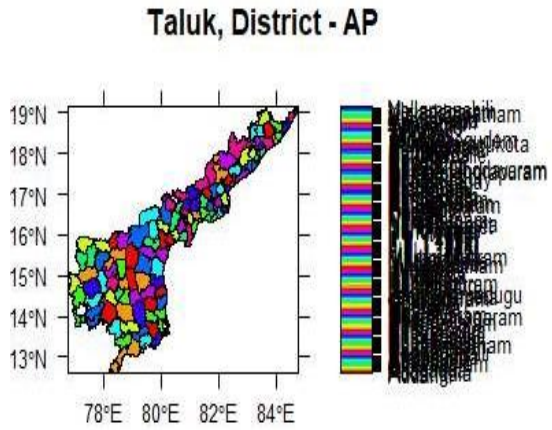

```
#Taluk,District-TamilNadu
ind3=readRDS("IND_adm3.rds")
TN_TALUKS=ind3[ind3$NAME_1=='Tamil Nadu',]
TN_TALUKS$NAME_3<-as.factor(TN_TALUKS$NAME_3)
col1=rainbow(length(levels(TN_TALUKS$NAME_3)))
spplot(TN_TALUKS,"NAME_3",main = "Taluk, District - TN",
      colorkey=T,col.regions=col,scales=list(draw=T))
```

Output:




```
#Taluk,District-Andhra Pradesh
AP_TALUKS=ind3[ind3$NAME_1=='Andhra Pradesh',]
AP_TALUKS$NAME_3<-as.factor(AP_TALUKS$NAME_3)
col1=rainbow(length(levels(AP_TALUKS$NAME_3)))
spplot(AP_TALUKS,"NAME_3",main = "Taluk, District - AP",
      colorkey=T,col.regions=col,scales=list(draw=T))
```

Output:



Network Data Analysis using R

```
#Social Network Analysis
#Load the library igraph
library(igraph)
#Create a simple graph
g <- graph(c(1,2))
#Plot the graph
plot(g)
```

Output:

```
#Social Network Analysis
#Load the library igraph
library(igraph)
#Create a simple graph
g <- graph(c(1,2))
#Plot the graph
#plot(g)
#For node or vertex,if you want different color rather than default
#Modify the size
#we can choose different color for edge
plot(g,vertex.color="green",vertex.size=40,edge.color='red')
#After nodes 1 to 2, we can add other nodes also

library(igraph)
g <- graph(c(1,2,2,3,3,4,4,1),directed=F,n=7)
plot(g,vertex.color="green",vertex.size=40,edge.color='red')
# we can see the connection between edges
g[]
# Now we can have 4 nodes
# if the arrow is growing from one node to another it is called directed graph
#we can add the number of nodes also in the graph
# Here we will take another graph with string objects
# If we make directed = false
library(igraph)
g1 <- graph(c("Amy","Ram","Ram","Li","Li","Amy","Amy","Li","Kate","Li"),directed=F)
#then use the plot
plot(g1,vertex.color="green",vertex.size=40,edge.color='red')
g1

#Network Measures
#one such measure is degree
#degree means number of connections
#we can also get the information by setting mode=all
library(igraph)
g1 <- graph(c("Amy","Ram","Ram","Li","Li","Amy","Amy","Li","Kate","Li"),directed=T)
#then use the plot
plot(g1,vertex.color="green",vertex.size=40,edge.color='red')
g1
degree(g1,mode='all')
degree(g1,mode='in')
degree(g1,mode='out')
# we can get the diameter of the netwrok
diameter(g1,directed=F, weights=NA)
# we can calculate the density
edge_density(g1,loops=F)
ecount(g1)/(vcount(g1)*(vcount(g1)-1))
# we have 5 edges and 4 vertexs
# reciprocity and closeness
reciprocity(g1)
```

```
closeness(g1,mode='all', weights=NA)
# we can calculate betweenness
betweenness(g1,directed=T,weights=NA)
edge_betweenness(g1,directed=T,weight=NA)
```

10(b) Data Transformations: Converting Numeric Variables into Factors, Date Operations, String Parsing, Geocoding.

Converting Numeric Variables into Factors

Factors are used to represent categorical data. Factors can be ordered or unordered and are an important class for statistical analysis and for plotting.

Factors are stored as integers, and have labels associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set value, known as *levels*. By default, R always sorts *levels* in alphabetical order. For instance, if you have a factor with 2 levels:

The factor() Command

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

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

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

```
levels(sex)
[1] "female" "male"
nlevels(sex)
[1] 2
```

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

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

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

levels(food)

[1] "low" "medium" "high"

min(food) # works!

[1] low
```

```
Levels: low < medium < high
```

Note:
In R's memory, these factors are represented by numbers (1, 2, 3). They are better than using simple integer labels because factors are self-describing: `"low"`, `"medium"`, and `"high"`" is more descriptive than `1`, `2`, `3`. Which is low? You wouldn't be able to tell with just integer data. Factors have this information built in. It is particularly helpful when there are many levels (like the subjects in our example data set).

Categorical Variables

Categorical variables in R are stored into a factor. Let's check the code below to convert a character variable into a factor variable in R. Characters are not supported in machine learning algorithm, and the only way is to convert a string to an integer.

Syntax
factor(x = character(), levels, labels = levels, ordered = is.ordered(x))
Arguments:

- **x**: A vector of categorical data in R. Need to be a string or integer, not decimal.
- **Levels**: A vector of possible values taken by x. This argument is optional. The default value isthe unique list of items of the vector x.
- **Labels**: Add a label to the x categorical data in R. For example, 1 can take the label `male`while 0, the label `female`.
- **ordered**: Determine if the levels should be ordered in categorical data in R.

```
# Create gender vector

gender_vector <- c("Male", "Female", "Female", "Male",

"Male")class(gender_vector)

# Convert gender_vector to a factor

factor_gender_vector <-factor(gender_vector)

class(factor_gender_vector)
```

Output:

```
## [1] "character"

## [1] "factor"
```

A categorical variable in R can be divided into **nominal categorical variable** and **ordinal categorical variable**.

Nominal Categorical Variable

A categorical variable has several values but the order does not matter. For instance, male or female. Categorical variables in R does not have ordering.

```
# Create a color vector
color_vector <- c('blue', 'red', 'green', 'white', 'black', 'yellow')
# Convert the vector to factor
factor_color <- factor(color_vector)
factor_color
```

Output:

```
## [1] blue  red green white black yellow
## Levels: black blue green red white yellow
```

From the factor_color, we can't tell any order.

Ordinal Categorical Variable

Ordinal categorical variables do have a natural ordering. We can specify the order, from the lowest to the highest with order = TRUE and highest to lowest with order = FALSE.

Example:

We can use summary to count the values for each factor variable in R.

```
# Create Ordinal categorical vector
day_vector <- c('evening', 'morning', 'afternoon', 'midday', 'midnight', 'evening')
# Convert `day_vector` to a factor with ordered level
factor_day <- factor(day_vector, order = TRUE, levels =c('morning', 'midday', 'afternoon',
'evening', 'midnight'))
# Print the new variable
factor_day
```

Output:

```
## [1] evening  morning   afternoon midday midnight  evening
```

Example:

```
## Levels: morning < midday < afternoon < evening <
midnight# Append the line to above code
# Count the number of occurence of each level
summary(factor_day)
```

Output:

```
##  morning  midday afternoon  evening  midnight##
        1       1       1       2       1
```

R ordered the level from 'morning' to 'midnight' as specified in the levels parenthesis.
See an example below for the **as.Date()** function

```
#as.Date()function in R
dv <- as.Date("2012-05-28")
#standard date format for as.Date() is "YYYY-MM-DD"
print(dv)
```
Output:
[1] "2012-05-28"

Now, when we don't have input value in a standard date format, we still can use the as.Date() function to create a dates value. See an example below:

```
dv1 <- as.Date("01/22/2015",format='%m/%d/%y')
print(dv1)
```
Output:
[1] "2020-01-22"

In this example, if you could see, the input date value is "01/22/2015", which is not the standard date format. However, we have **format =** argument under the function, which allows it to arrange thedate values in a standard form and present it to us.

%d - means a day of the month in number format

%m - stands for the month in number format

%Y - stands for the year in the "YYYY" format. If we have the year value in two digits, we will usethe "%y" instead of "%Y." See an example below:

```
dv1 <- as.Date("01/22/15",format='%m/%d/%y')
print(dv1)
```
Output:
[1] "2015-01-22"

When we have a month name instead of month number under the input value, we can use the %B operator under the format = argument while using the as.Date() function.

```
#example-3
dv2<-as.Date("15 April,2020",format='%d %B,%Y')
print(dv2)
```
Output:
[1] "2020-04-15"

Getting the Current Date and Time for System
Using the Sys.Date(), Sys.time() Function

- In R programming, if you use Sys.Date() function, it will give you the system date. You don't need to add an argument inside the parentheses to this function.
- There is again a function named Sys.timezone() that allows us to get the timezone based on the location at which the user is running the code on the system.
- And finally, we have the Sys.time() function. Which, if used, will return the current date as well as the time of the system with the timezone details.

```
#To get the Current Date and Time
```

Sys.Date()#Current system Date

Output:
[1] "2022-01-30"

Sys.timezone() #Timezone of the system

Output:
[1] "Asia/Calcutta"

Sys.time() #Current System Time

Output:
[1] "2022-01-30 17:04:11 IST"

Using the lubridate Package
Well, there is a package named lubridate, which has a function named now() that can give us the current date, current time, and the current timezone details in a single call (same as the Sys.time() function).

#The lubridate package
install.packages("lubridate")
library(lubridate)
now()
Output:
[1] "2022-01-30 17:11:37 IST"