**RAJALAKSHMI ENGINEERING COLLEGE**



**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

**AD23431- STATISTICAL ANALYSIS AND COMPUTING LAB**

**II Year – IV Semester**

**LABORATORY MANUAL**

|  |  |  |
| --- | --- | --- |
| **List of Experiments** | | |
| 1 | Implement simple programs in R 3. 4. 5. 6 7. 8. | |
| 2 | Perform data preprocessing in R | |
| 3 | Perform statistical analysis for a given dataset | |
| 4 | Implement decision tree algorithm in R | |
| 5 | Implement K-Nearest Neighbor algorithm in R | |
| 6 | Implement Naive Bayesian classifier in R | |
| 7 | Implement linear regression in R | |
| 8 | Implement K-means clustering algorithm in R | |
| 9 | Implementation of Searching and Sorting algorithms | |
| 10 | Hashing –Linear probing | |
| **Requirements** | | |
| Hardware | | Intel i3, CPU @ 1.20GHz 1.19 GHz, 4 GB RAM, 32 Bit Operating System |
| Software | | R |
| Operating System | | Windows |

### AD23431-STATISTICAL ANALYSIS AND COMPUTING LAB PLAN

|  |  |  |
| --- | --- | --- |
| **Exercise no.** | **Exercise Name** | **Required Hours** |
| 1 | Implement simple programs in R 3. 4. 5. 6 7. 8. | 3 |
| 2 | Perform data preprocessing in R | 3 |
| 3 | Perform statistical analysis for a given dataset | 3 |
| 4 | Implement decision tree algorithm in R | 3 |
| 5 | Implement K-Nearest Neighbor algorithm in R | 3 |
| 6 | Implement Naive Bayesian classifier in R | 3 |
| 7 | Implement linear regression in R | 3 |
| 8 | Implement K-means clustering algorithm in R | 3 |
| 9 | Implementation of Searching and Sorting algorithms | 3 |
| 10 | Hashing –Linear probing | 3 |

|  |  |
| --- | --- |
| **EXP.NO:1 a)** | **Install R programming Environment and**  **Basic Packages using install.package() command** |
| **DATE:** |

### AIM:

To download and install R programming Environment and Install basic packages using

install.packages() command in R.

### PROCEDURE:

**STEP 1:** Download and install R.

1. Go to the [CRAN website] (https://cran.r-project.org/).
2. Click on the link for your operating system (Windows, macOS, or Linux).
3. Follow the instructions to download and install R.

**STEP 2:** Download and install R Studio

1. Go to the [RStudio website] (https://rstudio.com/products/rstudio/download/).
2. Download the appropriate installer for your operating system.

**STEP 3:** Install basic R packages

1. Open RStudio: Start RStudio from your desktop or Start menu.
2. Install Packages: Use the `install.packages()` command to install the basic packages. Here are some essential packages you might want to install:

### SYNTAX:

install.packages(c("tidyverse", "data.table", "ggplot2", "dplyr", "readr", "tidyr", "purrr", "tibble", "stringr", "lubridate", "shiny", "plotly", "knitr"))

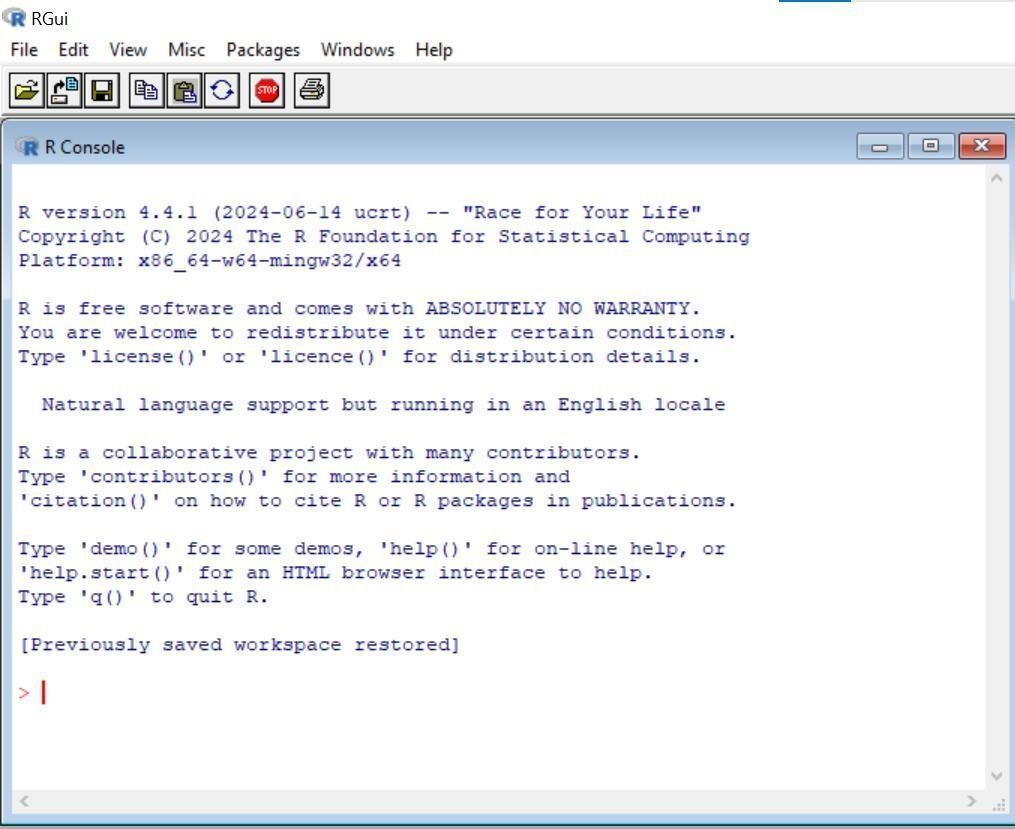
#### EXAMPLE R SCRIPT to install packages:

You can create a script file in RStudio to install these packages. Here is an example:

1. Create a New Script: Click on `File` > `New File` > `R Script`.
2. # List of packages to install packages <- c(

"tidyverse", "data.table", "ggplot2", "dplyr", "readr", "tidyr", "purrr", "tibble", "stringr", "lubridate", "shiny", "plotly", "knitr")

# Install the packages install.packages(packages)



1. Run the Script: Click on `Source` or press `Ctrl+Shift+S` to run the script and install the packages.
2. Verify Installation

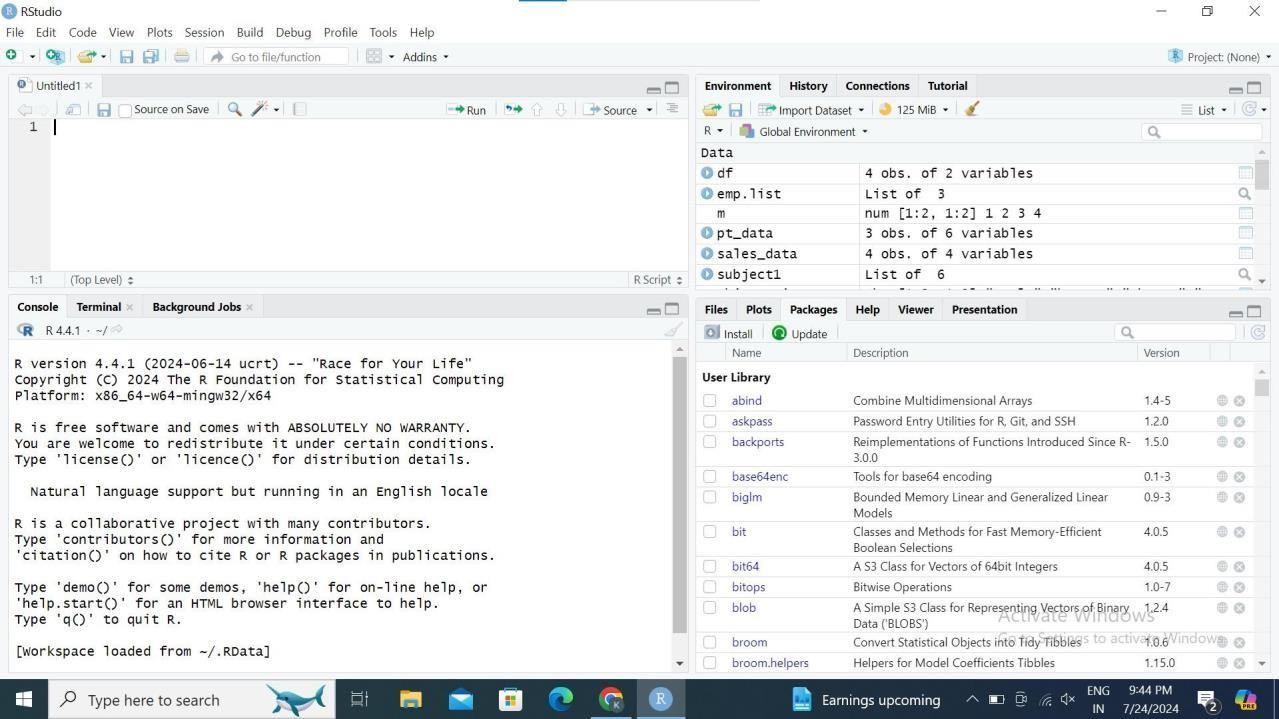
To verify that the packages are installed correctly, you can load a package using the `library()` function. For example:

library(tidyverse)

**OUTPUT:**

**R STUDIO CONSOLE**

**R STUDIO**



### RESULT:

Thus, the R programming Environment and all essential R packages are installed successfully using install.packages() command.

|  |  |
| --- | --- |
| **EXP.NO:1 b)** | **Basics of R Programming**  **(Datatypes, Variables and Operators)** |
| **DATE:** |

### AIM:

To implement and understand the basics of R programming with its datatypes variables and operators.

### ALGORITHM:

**STEP 1:** Start the program

**STEP 2:** Assign values in logical, numerical, character and complex in raw form to a variable v

**STEP 3:** Print the class of v

**STEP 4:** Assign values to variables and print them

**STEP 5:** Perform arithmetic operations, Relational operations and assignment operation and print them

**STEP 6:** Stop

### PROGRAM:

#### Datatypes:

v<-TRUE

print(class(v)) v<-23.5

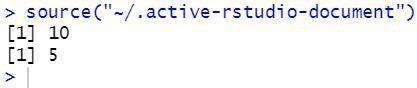
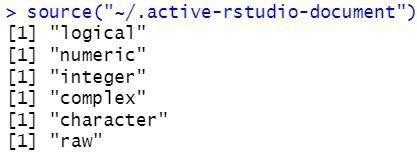
print(class(v)) v<-2L

print(class(v)) v<-2+5i

print(class(v)) v<-"True" print(class(v))

v<-charToRaw("Hello") print(class(v))

#### Output:



**Variables:**

x<-10 y<-5

print(x) print(y)

#### Output:

**Operators: Arithmetic operators:**

x<-10 y<-5

result<-x+y print(result) result<-x-y print(result) result<-x/y print(result) result<-x\*y print(result) result<-x%%y print(result) result<-x%/%y print(result) result<-x^y print(result)

#### Output:



**Relational operators:**

result<-x>y print(result) result<-x<y print(result) result<-x==y print(result) result<-x!=y print(result)

#### Output:

**Assignment operators:**

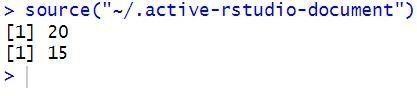
x<-10

y=5 20->x

15->y x<<-20 y<<-15

print(x) print(y)

#### Output:



**RESULT:**

Thus, the basic R programs for datatypes, variables and operators are implemented and executed successfully.

|  |  |
| --- | --- |
| **EXP.NO:1 c)** | **Implementing different data structures in R**  **(Vectors, List and DataFrames)** |
| **DATE:** |

### AIM:

To implement and understand the basics of R programming with its vectors, list and Data frames.

### ALGORITHM:

**STEP 1:** Start the program

**STEP 2:** Assign a vector for subject names, temperature and flu status for 3 patients using c() **STEP 3:** Create factor using factor() with duplicates values and assign level with distinct values **STEP 4:** Display the specific elements and check for certain values in factor

**STEP 5:** Create a list using list() from the patient details and access multiple elements

**STEP 6:** Create a dataframe using data.frame() with multiple vectors as features access the elements

**STEP 7:** Create a matrix using matrix() with different allocations and access the element

**STEP 8:** Stop the program

### PROGRAM:

#### Vectors:

subject\_name<-c("John Doe","Jane Doe","Steven Grant") temperature<-c(98.1,98.6,101.4)

flu\_status<-c(FALSE,FALSE,TRUE) print(temperature[2]) print(temperature[2:3]) print(temperature[-2]) print(flu\_status[3])

print(flu\_status[-2]) print(subject\_name[1]) print(subject\_name[-1])

#### Output:

**Factors:**

gender<-factor(c("MALE","FEMALE","MALE")) print(gender)

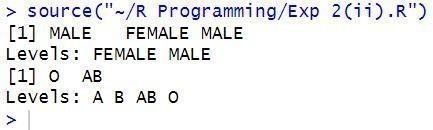
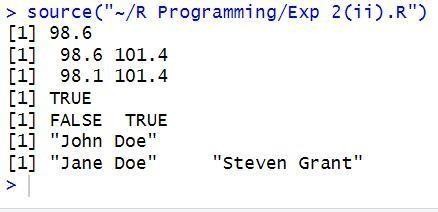
blood<-factor(c("O","AB","A"), levels=c("A","B","AB","O")) print(blood[1:2])

symptoms<-factor(c("SEVERE","MILD","MODERATE"), levels=c("MILD","MODERATE","SEVERE"),

ordered=TRUE) symptoms->"MODERATE"

#### Output:

**Lists:**

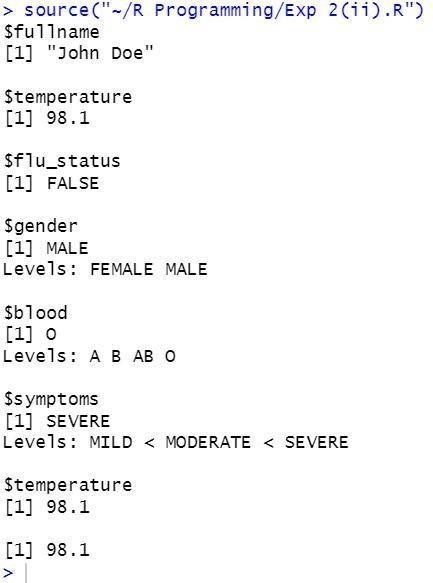


subject1<-list(fullname=subject\_name[1], temperature=temperature[1], flu\_status=flu\_status[1], gender=gender[1], blood=blood[1], symptoms=symptoms[1])

print(subject1) print(subject1[2]) print(subject1[[2]]) subject1$temperature

subject1[c("temperature","flu\_status")]

#### Output:



**Data Frames:**

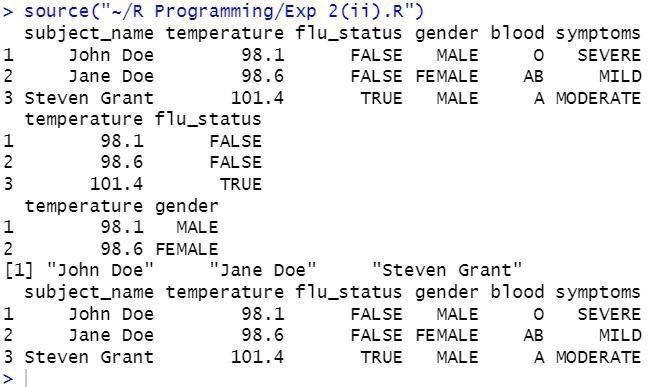
pt\_data<-data.frame(subject\_name, temperature, flu\_status, gender,blood,symptoms)

print(pt\_data) pt\_data$subject\_name

print(pt\_data[c("temperature","flu\_status")]) print(pt\_data[c(1,2),c(2,4)]) print(pt\_data[,1])

print(pt\_data[,])

#### Output:



**Matrices:**

m<-matrix(c(1,2,3,4),ncol=2) print(m)

m<-matrix(c(1,2,3,4,5,6),nrow=3) print(m)

print(m[1,])

print(m[1,])

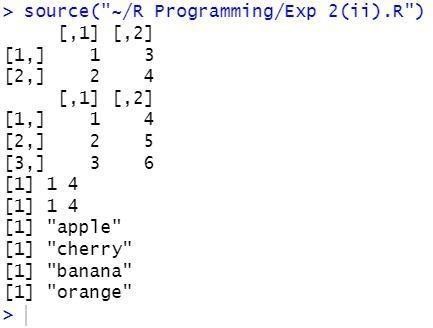
thismatrix <- matrix(c("apple", "banana", "cherry","orange"), nrow = 2, ncol = 2) for (rows in 1:nrow(thismatrix)) {

for (columns in 1:ncol(thismatrix)) { print(thismatrix[rows, columns])

}

}

**Output:**



### RESULT:

Thus, the different data structures like vector, lists, matrix, dataframes and matrices are executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO:1 d)** | **Read a CSV file and analyse the data in the file**  **Using R** |
| **DATE:** |

### AIM:

To read a CSV file and analyse the data in the file using R.

### ALGORITHM:

**STEP 1:**Start the program

**STEP 2:** Install the necessary packages readxl and ggplot2 using install.packages().

**STEP 3:** Load the installed libraries readxl and ggplot2 with the library() function for reading Excel files and plotting data.

**STEP 4:** Read the dataset "Student2.xlsx" located at "C:/221801024/Student2.xlsx" using the read\_excel() function and store it in the variable data.

**STEP 5:** Display the first few rows of the dataset using the head() function to check the structure of the data.

**STEP 6:** Generate summary statistics of the dataset using the summary() function to get an overview of the variables (e.g., mean, median, min, max).

**STEP 7:** Create a frequency table for the CGPA column using the table() function and store the result in bar\_data.

**STEP 8:** Create a bar plot of the CGPA frequencies using the barplot() function. Customize the chart by setting the main title (main), x-axis label (xlab), y-axis label (ylab), bar color (col), and rotate the x-axis labels (las).

**STEP 9:** Create a frequency table for the Year column using the table() function and store the result in pie\_data.

**STEP 10:** Plot a pie chart of the Year distribution using the pie() function, setting the main title (main) and assigning rainbow colors (col = rainbow()) to the slices of the pie chart.

**STEP 11:**Stop the program

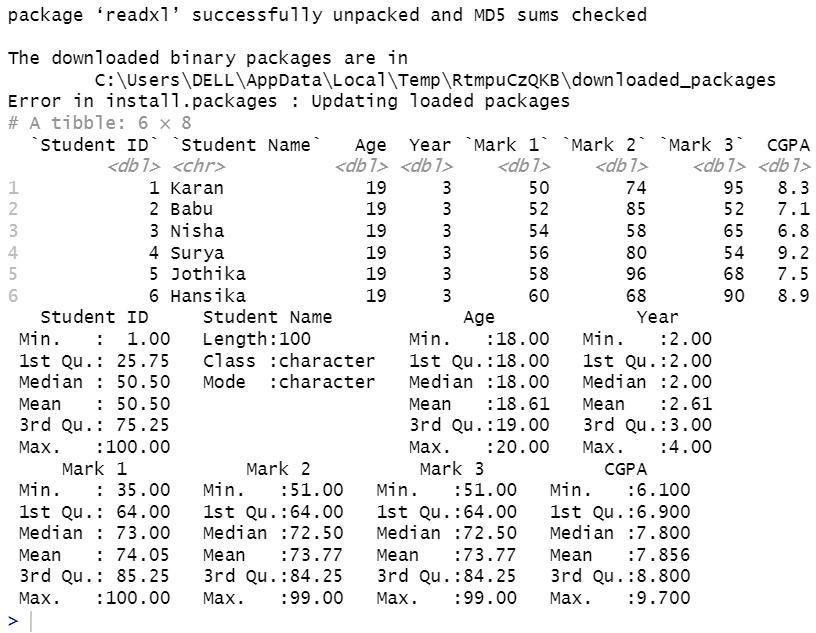
### PROGRAM:

# Install and load necessary packages install.packages("readxl") install.packages("ggplot2") library(readxl)

library(ggplot2)

data <- read\_excel("C:/221801024/Student2.xlsx")

print(head(data))



# Summary statistics of the dataset

print(summary(data)) bar\_data <- table(data$CGPA) barplot(bar\_data,

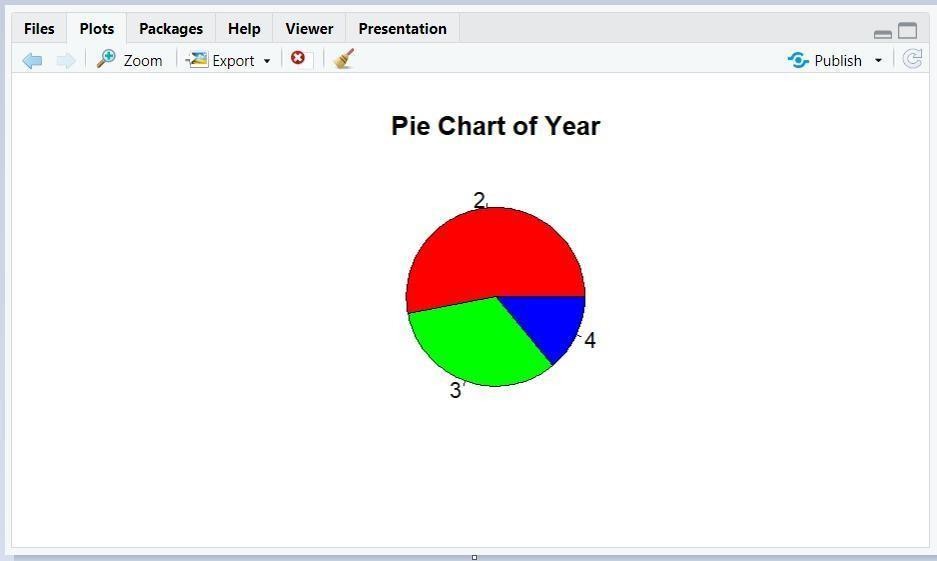
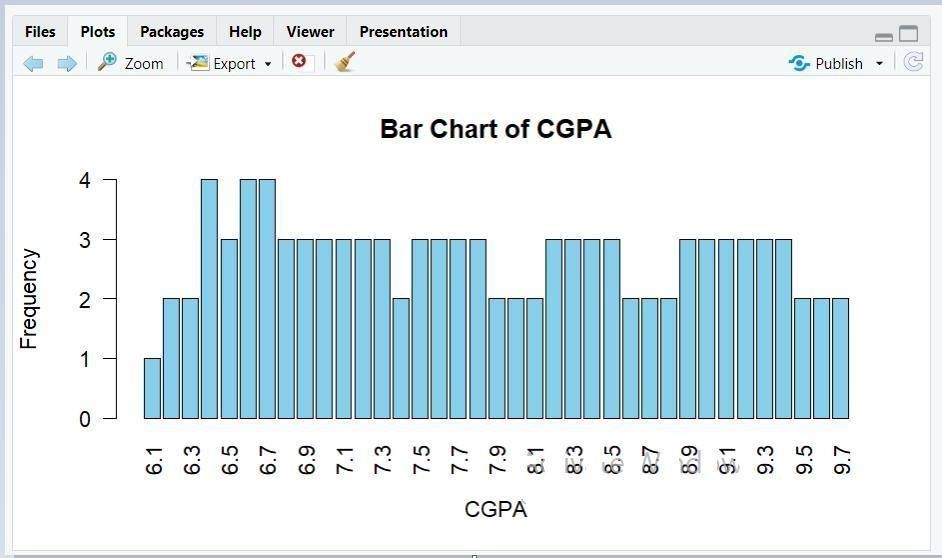
main = "Bar Chart of CGPA", xlab = "CGPA",

ylab = "Frequency", col = "skyblue", las = 2)

pie(pie\_data <- table(data$Year), main = "Pie Chart of Year",

col = rainbow(length(pie\_data)))

### OUTPUT:



**RESULT:**

Thus the CSV file was read and analysed the data in the file using R successfully.

**Ex No: 2**

# PERFORM DATA PREPROCESSING IN R

**Aim***:*

To perform preprocessing of data using R

#### Procedure:

**Step-by-Step Data Preprocessing in R Step 1: Loading Data**

First, we load the dataset and display its structure to understand its columns and initial format.

# Load the dataset (using mtcars dataset for demonstration) data(mtcars)

head(mtcars)

#### Output:

mpg cyl disp hp drat wt qsec vs am gear carb Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

...

#### Step 2: Handling Missing Values

Check for missing values (NA) in the dataset and decide how to handle them. For demonstration purposes, we assume there are no missing values in mtcars.

# Check for missing values any(is.na(mtcars))

#### Output:

[1] FALSE

#### Step 3: Handling Categorical Variables

If the dataset has categorical variables, convert them to factors if needed. In mtcars, there are no explicit categorical variables, but if there were, you would convert them using as.factor().

# Example: Convert a hypothetical categorical variable to factor # mtcars$cyl <- as.factor(mtcars$cyl)

#### Step 4: Scaling Numeric Data (Optional)

Scaling numeric variables is necessary for some algorithms but not always required. Here, we'll scale the numeric columns disp, hp, drat, wt, and qsec using scale() function.

# Scale numeric columns (optional)

num\_cols <- c("disp", "hp", "drat", "wt", "qsec") mtcars\_scaled <- as.data.frame(scale(mtcars[, num\_cols]))

# Combine scaled numeric columns with non-numeric columns mtcars\_processed <- cbind(mtcars\_scaled, mtcars[, -(which(names(mtcars) %in% num\_cols))])

head(mtcars\_processed)

#### Output (scaled numeric data combined with original non-numeric columns):

disp hp drat wt qsec mpg cyl vs am gear carb

|  |  |  |
| --- | --- | --- |
| Mazda RX4 | 0.5795256 0.6641339 0.5765943 -0.6201678 -0.7896003 21.0 6 | 0 1 4 4 |
| Mazda RX4 | Wag 0.5795256 0.6641339 0.5765943 -0.3553820 -0.4712017 21.0 | 6 0 1 4 4 |

Datsun 710 -1.0060260 -0.5110630 0.4815843 -0.9316786 0.4328237 22.8 4 1 1 4 1

...

#### Step 5: Splitting Data into Training and Testing Sets

Split the dataset into training and testing sets for model training and evaluation purposes.

# Split data into 80% training and 20% testing set.seed(123) # For reproducibility

train\_indices <- sample(nrow(mtcars), 0.8 \* nrow(mtcars)) train\_data <- mtcars[train\_indices, ]

test\_data <- mtcars[-train\_indices, ]

# Display dimensions of training and testing sets cat("Training data dimensions:", dim(train\_data), "\n") cat("Testing data dimensions:", dim(test\_data), "\n")

#### Output (dimensions of training and testing sets):

Training data dimensions: 25 11

Testing data dimensions: 7 11

#### Result:

Thus, preprocessing data using R is a cleaned, transformed, and formatted dataset ready for analysis or modeling.

**Ex No: 3**

# PERFORM STATISTICAL ANALYSIS FOR A GIVEN DATASET

**Aim***:*

To perform statistical analysis for given dataset.

#### Procedure:

* **plot() Function:** This function is used to Draw a scatter plot with axes and titles.

**Syntax:**

*plot(x, y = NULL, ylim = NULL, xlim = NULL, type = “b”….)*

* **data() function:** This function is used to load specified data sets.

**Syntax:**

*data(list = character(), lib.loc = NULL, package = NULL…..)*

* **table() Function:** The table function is used to build a contingency table of the counts at each combination of factor levels.

table(x, row.names = NULL, ...)

* **barplot() Function:** It creates a bar plot with vertical/horizontal bars.

**Syntax:**

*barplot(height, width = 1, names.arg = NULL, space = NULL…)*

* **pie() Function:** This function is used to create a pie chart.

**Syntax:**

*pie(x, labels = names(x), radius = 0.6, edges = 100, clockwise = TRUE …)*

* **hist() Function:** The function **hist()** creates a histogram of the given data values.

**Syntax:**

*hist(x, breaks = “Sturges”, probability = !freq, freq = NULL,…)*

**Note:** You can find the information about each function using the “?” symbol before the beginning of each function.

R built-in datasets are very useful to start with and develop skills, So we will be using a few Built-in datasets. Let’s start by creating a simple bar chart by using chickwts dataset and learn how to use datasets and few functions of RStudio for R Statistics.

Bar charts



A Bar chart represents categorical data with rectangular bars where the bars can be plotted vertically or horizontally.

# ? is used before a function # to get help on that function

?plot

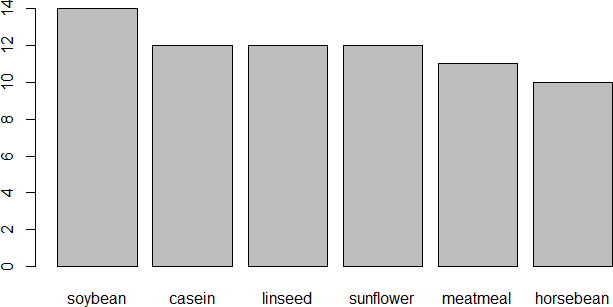
?chickwts

data(chickwts) #loading data into workspace plot(chickwts$feed) # plot feed from chickwts

## Output:

*R – Statistics*

In the above code ‘?’ in front of a particular function means that it gives information about that function with its syntax. In R ‘#’ is used for commenting single line and there is no multiline comment in R. Here we are using **chickwts** as the dataset and feed is the attribute in the dataset.



**Plots graph in decreasing order**

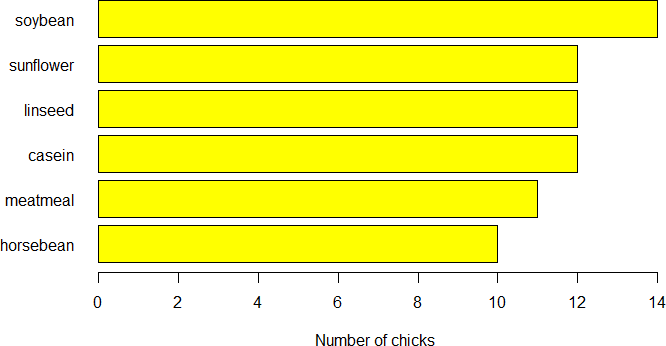
feeds=table(chickwts$feed)

# plots graph in decreasing order barplot(feeds[order(feeds, decreasing=TRUE)])

**Output:**

*R – Statistics*

**Plots Horizontal bars**



feeds = table(chickwts$feed)

# Set outside margins (bottom, left, top, right). par(oma=c(1, 1, 1, 1))

par(mar=c(4, 5, 2, 1))

# Use las for the orientation of axis labels. barplot(feeds[order(feeds, decreasing=TRUE)],

xlab="Number of chicks", las=1, col="yellow")

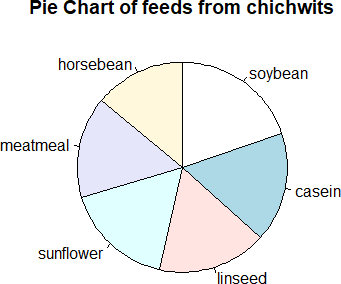
# Use horiz for bars to be shown as horizontal. barplot(feeds[order(feeds)], horiz=TRUE,

xlab="Number of chicks", las=1, col="yellow")

## Output:

*R – Statistics*

**Pie charts**



A pie chart is a circular statistical graph that is divided into slices to show the different sizes of the data.

data("chickwts")

# main is used to create # an heading for the chart d = table(chickwts$feed)

pie(d[order(d, decreasing=TRUE)], clockwise=TRUE,

main="Pie Chart of feeds from chichwits", )

## Output:

*R – Statistics*

**Histograms**

Histograms are the representation of the distribution of data(numerical or categorical). It is similar to a bar chart but it groups data in terms of ranges.

# break is used for number of bins. data(lynx)

# lynx is a built-in dataset. lynx

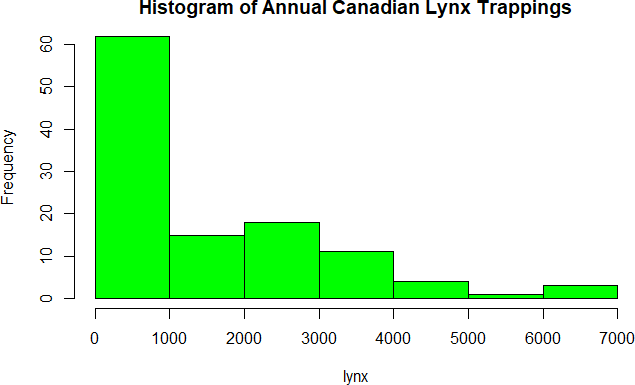
# hist function is used to plot histogram. hist(lynx)

hist(lynx, col="green",

main="Histogram of Annual Canadian Lynx Trappings")

## Output :

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time Series: |  |  |  |  |  |  |  |  |  |  |
| Start = 1821 |  |  |  |  |  |  |  |  |  |  |
| End = 1934 |  |  |  |  |  |  |  |  |  |  |
| Frequency = 1 |  |  |  |  |  |  |  |  |  |  |
| [1] 269 321 | 585 | 871 | 1475 | 2821 | 3928 | 5943 | 4950 | 2577 | 523 | 98 |
| 184 |  |  |  |  |  |  |  |  |  |  |
| [14] 279 409 | 2285 | 2685 | 3409 | 1824 | 409 | 151 | 45 | 68 | 213 | 546 |
| 1033 |  |  |  |  |  |  |  |  |  |  |
| [27] 2129 2536 | 957 | 361 | 377 | 225 | 360 | 731 | 1638 | 2725 | 2871 | 2119 |
| 684 |  |  |  |  |  |  |  |  |  |  |
| [40] 299 236 | 245 | 552 | 1623 | 3311 | 6721 | 4254 | 687 | 255 | 473 | 358 |
| 784 |  |  |  |  |  |  |  |  |  |  |
| [53] 1594 1676 | 2251 | 1426 | 756 | 299 | 201 | 229 | 469 | 736 | 2042 | 2811 |
| 4431 |  |  |  |  |  |  |  |  |  |  |
| [66] 2511 389 | 73 | 39 | 49 | 59 | 188 | 377 | 1292 | 4031 | 3495 | 587 |
| 105 |  |  |  |  |  |  |  |  |  |  |
| [79] 153 387 | 758 | 1307 | 3465 | 6991 | 6313 | 3794 | 1836 | 345 | 382 | 808 |
| 1388 |  |  |  |  |  |  |  |  |  |  |
| [92] 2713 3800 | 3091 | 2985 | 3790 | 674 | 81 | 80 | 108 | 229 | 399 | 1132 |



2432

[105] 3574 2935 1537 529 485 662 1000 1590 2657 3396

*R – Statistics*

**Plot The Distribution**

data(lynx)

# if freq=FALSE this will draw normal distribution hist(lynx)

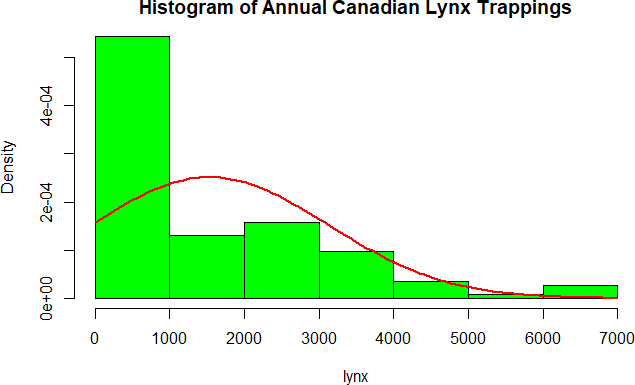
hist(lynx,col="green",

freq=FALSE ,main="Histogram of Annual Canadian Lynx Trappings")

curve(dnorm(x, mean=mean(lynx),

sd=sd(lynx)), col="red", lwd=2, add=TRUE)

## Output:



*R – Statistics*

**Box Plots**

Box Plot is a function for graphically depicting groups of numerical data using quartiles. It represents the distribution of data and understanding mean, median, and variance.

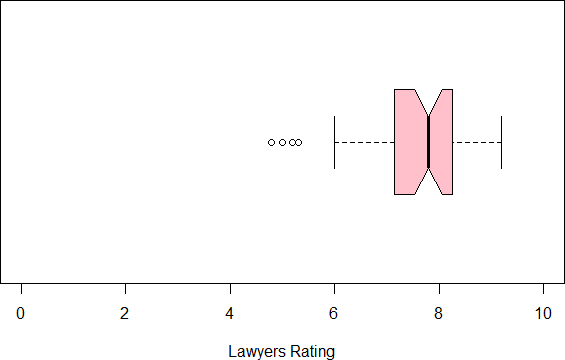
# USJudgeRatings is Built-in Dataset.

?USJudgeRatings

# ylim is used to specify the range. boxplot(USJudgeRatings$RTEN, horizontal=TRUE,

xlab="Lawyers Rating", notch=TRUE, ylim=c(0, 10), col="pink")

## Output:



*R – Statistics*

USJudgeRating is a Build-in dataset with 6 attributes and RTEN is one of the attribute among it which is rating between 0 to 10 inclusive. We used it to for plotting a boxplot with different attributes of boxplot function.

#### Result:

Therefore, a variety of statistical analyses are conducted on the given dataset, enhancing the depth and breadth of insights derived from the data.

**Ex No: 4**

# IMPLEMENT DECISION TREE ALGORITHM IN R

**Aim***:*

To implement decision tree algorithm in R

#### Procedure:

**Implementing Decision Tree Algorithm in R Step 1: Load Required Packages and Dataset**

First, load the necessary packages (rpart for decision trees and rpart.plot for plotting trees) and the iris dataset.

# Load required packages

library(rpart) # for building decision trees library(rpart.plot) # for plotting decision trees

# Load the dataset data(iris)

#### Step 2: Explore and Preprocess the Dataset (if necessary)

For the iris dataset, preprocessing might involve converting the target variable (Species) into a factor if it's not already converted.

# Convert 'Species' column to factor (if necessary) iris$Species <- as.factor(iris$Species)

#### Step 3: Build the Decision Tree Model

Now, build the decision tree model using the rpart() function. We'll predict the Species using other variables (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width).

# Build decision tree model

tree\_model <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data = iris,

method = "class")

#### Step 4: Visualize the Decision Tree

Visualize the decision tree using rpart.plot package.

# Plot the decision tree

rpart.plot(tree\_model, main = "Decision Tree for Iris Dataset", extra = 2)

#### Output (Decision Tree Visualization):

This will display a graphical representation of the decision tree model built for the iris dataset. Each node in the tree represents a decision point based on a predictor variable (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width), leading to leaf nodes that correspond to predicted classes (Species).

#### Step 5: Full Output and Evaluation

Let's print the details of the decision tree model and evaluate its performance on the dataset.

# Print the decision tree details print(tree\_model)

# Make predictions on training data

predicted\_classes <- predict(tree\_model, newdata = iris, type = "class") # Evaluate accuracy

accuracy <- mean(predicted\_classes == iris$Species) cat("\nAccuracy of the decision tree model:", accuracy, "\n")

# Confusion matrix

conf\_matrix <- table(predicted\_classes, iris$Species) cat("\nConfusion Matrix:\n")

print(conf\_matrix)

#### Full Output:

1. **Decision Tree Details (Partial Output):**

n= 150

node), split, n, loss, yval, (yprob)

\* denotes terminal node

* 1. root 150 100 setosa (0.33333333 0.33333333 0.33333333)
  2. Petal.Length < 2.45 50 0 setosa (1.00000000 0.00000000 0.00000000) \*
  3. Petal.Length >= 2.45 100 50 versicolor (0.00000000 0.50000000 0.50000000)

1. Petal.Width < 1.75 54 5 versicolor (0.00000000 0.90740741 0.09259259) \*
2. Petal.Width >= 1.75 46 1 virginica (0.00000000 0.02173913 0.97826087) \*

#### Accuracy of the Decision Tree Model:

Accuracy of the decision tree model: 0.9733333

#### Confusion Matrix:

predicted\_classes setosa versicolor virginica

|  |  |  |  |
| --- | --- | --- | --- |
| setosa | 50 | 0 | 0 |
| versicolor | 0 | 48 | 2 |
| virginica | 0 | 2 | 48 |
| Result: |  |  |  |

Therefore decision tree algorithm was successfully implemented using R.

**Ex No: 5**

# IMPLEMENT K-NEAREST NEIGHBOR ALGORITHM IN R

**Aim***:*

To implement K Nearest Neighbour algorithm in R

#### Procedure:

**Implementing K-Nearest Neighbor (KNN) Algorithm in R Step 1: Load Required Packages and Dataset**

First, load the necessary packages (class for KNN) and the iris dataset.

# Load required package library(class)

# Load the dataset data(iris)

#### Step 2: Explore and Preprocess the Dataset (if necessary)

For the iris dataset, preprocessing might involve converting the target variable (Species) into a factor if it's not already converted.

# Convert 'Species' column to factor (if necessary) iris$Species <- as.factor(iris$Species)

#### Step 3: Split the Dataset into Training and Test Sets (Optional)

Since KNN is a lazy learning algorithm, it does not explicitly build a model. Instead, it relies on the entire training dataset for prediction. Optionally, you can split the dataset into training and test sets for evaluation purposes.

# Set seed for reproducibility set.seed(123)

# Split the dataset into training (70%) and test (30%) sets train\_index <- sample(1:nrow(iris), 0.7 \* nrow(iris)) train\_data <- iris[train\_index, ]

test\_data <- iris[-train\_index, ]

#### Step 4: Build the KNN Model

Now, build the KNN model using the knn() function from the class package. Here, we'll predict the Species using other variables (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width).

# Build KNN model

k <- 5 # Number of neighbors

predicted\_classes <- knn(train\_data[, -5], test\_data[, -5], train\_data$Species, k = k)

#### Step 5: Evaluate the Model

Evaluate the performance of the KNN model by comparing predicted classes with actual classes.

# Calculate accuracy

accuracy <- mean(predicted\_classes == test\_data$Species) cat("Accuracy of KNN model (k =", k, "):", accuracy, "\n")

# Confusion matrix

conf\_matrix <- table(predicted\_classes, test\_data$Species) cat("\nConfusion Matrix:\n")

print(conf\_matrix)

#### Output:

After running the above steps, the output will include the accuracy of the KNN model and the confusion matrix which provides detailed information about the model's performance.

#### Example Output:

Accuracy of KNN model (k = 5 ): 0.9555556

Confusion Matrix:

setosa versicolor virginica

|  |  |  |  |
| --- | --- | --- | --- |
| setosa | 14 | 0 | 0 |
| versicolor | 0 | 14 | 1 |
| virginica | 0 | 1 | 15 |

Result:

Therefore K Nearest Neighbour algorithm was successfully implemented using R.

**Ex No: 6**

# IMPLEMENT NAIVE BAYESIAN CLASSIFIER IN R

**Aim***:*

To implement Naïve Bayesian Classifier in R

#### Procedure:

Implementing a Naive Bayesian classifier in R involves using existing libraries and functions. One popular library for machine learning tasks in R is e1071, which provides implementations for various classifiers including Naive Bayes.

Here's a step-by-step implementation of a Naive Bayesian classifier using e1071 in R:

1. **Install and load the e1071 library**: Make sure you have e1071 installed. If not, install it using:

install.packages("e1071")

1. **Load the library**: library(e1071)
2. **Prepare the dataset**: For demonstration purposes, let's use a sample dataset included in e1071 called iris. This dataset is about flowers and contains measurements of different species of iris flowers.

data(iris)

1. **Split the dataset into training and testing sets**: It's important to split the dataset into a training set and a testing set to evaluate the classifier. Here, we'll use 70% of the data for training and 30% for testing.

set.seed(123) # for reproducibility

trainIndex <- sample(1:nrow(iris), 0.7\*nrow(iris))

trainData <- iris[trainIndex, ] testData <- iris[-trainIndex, ]

1. **Train the Naive Bayes classifier**: Use the naiveBayes function from e1071 to train the classifier.

nb\_model <- naiveBayes(Species ~ ., data = trainData)

Here, Species is the target variable we want to predict based on the other variables (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width).

1. **Make predictions**: Use the trained model to make predictions on the test set. predictions <- predict(nb\_model, testData)
2. **Evaluate the model**: Compute accuracy or other metrics to evaluate how well the model performs.

accuracy <- mean(predictions == testData$Species) cat("Accuracy:", accuracy, "\n")

This will print the accuracy of the Naive Bayes classifier on the test dataset.

**Output Example**: Assuming the above steps have been executed, the output would look something like this:

Accuracy: 0.9555556

This indicates the accuracy of the Naive Bayes classifier on the test dataset (in this case, the iris dataset). The actual accuracy may vary slightly due to the random seed used in splitting the dataset (set.seed(123)), but it should be around this range.

This example demonstrates a basic implementation of Naive Bayes classification in R using the e1071 library, applied to the classic iris dataset.

Result:

Therefore Naïve Bayesian Classifier was successfully implemented using R.

**Ex No: 7**

# IMPLEMENT LINEAR REGRESSION IN R

**Aim***:*

To implement Linear Regression in R

#### Procedure:

Linear regression in R can be implemented using the built-in function lm() (short for "linear model"). Here's a step-by-step implementation using a sample dataset:

1. **Load or create a dataset**: For demonstration purposes, let's use a built-in dataset in R called mtcars, which contains information about different car models.

data(mtcars)

You can also create your own dataset if needed, but mtcars is convenient for this example.

1. **Fit a linear regression model**: Use the lm() function to fit a linear regression model. Let's say we want to predict mpg (miles per gallon) based on wt (weight of the car).

lm\_model <- lm(mpg ~ wt, data = mtcars)

Here, mpg ~ wt specifies the formula for the linear regression model, where mpg is the dependent variable and wt is the independent variable (predictor). data = mtcars specifies that the data for the model comes from the mtcars dataset.

1. **Inspect the model summary**: To get detailed information about the fitted model, including coefficients, standard errors, t-values, and p-values, use the summary() function on the lm object.

summary(lm\_model)

This will print a summary of the linear regression model to the console.

**Output Example**: Assuming the above steps have been executed, the output would look something like this:

Call:

lm(formula = mpg ~ wt, data = mtcars)

Residuals:

Min 1Q Median 3Q Max

-4.5432 -2.3647 -0.1252 1.4096 6.8727

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 37.2851 1.8776 19.86 <2e-16 \*\*\*

wt -5.3445 0.5591 -9.56 <2e-16 \*\*\*

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.046 on 30 degrees of freedom

Multiple R-squared: 0.7528, Adjusted R-squared: 0.7446

F-statistic: 91.38 on 1 and 30 DF, p-value: 1.293e-10 This output provides several key pieces of information:

* **Coefficients**: Estimate gives the estimated coefficients of the linear regression model. Here, Intercept is 37.2851 and wt (weight) is -5.3445.
* **Standard Errors, t-values, and p-values**: These statistics help assess the significance of each coefficient. Lower p-values (Pr(>|t|)) indicate more significant predictors.
* **Residuals**: These are the differences between observed and predicted values. The summary provides statistics about the distribution of residuals.
* **R-squared and F-statistic**: These statistics assess the overall goodness-of-fit of the model. Multiple R-squared indicates how well the model explains the variability in

the response variable (mpg), and the F-statistic tests the overall significance of the model.

This example demonstrates how to perform linear regression in R and interpret the output using the mtcars dataset.

Result:

Thus, implement Linear Regression was successfully completed in R

**Ex No: 8**

# IMPLEMENT K-MEANS CLUSTERING ALGORITHM IN R

**Aim***:*

To implement K-means clustering algorithm in R

#### Procedure:

Implementing the K-means clustering algorithm in R involves using the kmeans() function, which is part of the base R package. Here's a step-by-step implementation using a sample dataset:

1. **Load or create a dataset**: For demonstration purposes, let's create a small dataset with numeric values.

# Create a sample dataset set.seed(123) # for reproducibility

data <- matrix(rnorm(100), ncol = 2) # 100 points in 2 dimensions In practice, you would replace data with your own dataset.

1. **Perform K-means clustering**: Use the kmeans() function to perform clustering on the dataset. Specify the number of clusters (centers) you want to identify.

# Perform K-means clustering with 3 clusters k <- 3

kmeans\_result <- kmeans(data, centers = k)

Here, kmeans(data, centers = k) assigns each observation in data to one of k clusters based on their similarity.

1. **Inspect the clustering results**: After running kmeans(), you can inspect various attributes of the resulting object kmeans\_result.

# Print the cluster centers print(kmeans\_result$centers)

# Print the cluster membership print(kmeans\_result$cluster)

# Print within-cluster sum of squares print(kmeans\_result$withinss)

* + kmeans\_result$centers: Gives the coordinates of the cluster centers.
  + kmeans\_result$cluster: Shows the cluster membership of each point.
  + kmeans\_result$withinss: Provides the within-cluster sum of squares, which is a measure of the compactness of each cluster.

**Output Example**: Assuming the above steps have been executed, the output would look something like this:

# Cluster centers

[,1] [,2]

1 0.05844123 0.1549111

2 -0.16894775 -0.2260800

3 0.77647316 -0.4910778

# Cluster membership

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

[39] 1 1 1 1 2 1 1 1 2 2 1 1 3 3 3 2 2 3 3 1 1 2 3 3 2 1 2 2 2 3 3 3 1 2 3 2 2

[77] 3 3 2 2 2 2 3 1 1 3 3 3 2 2 2 2 2 3 1 2 1 1 2 3 2 3 1 1 1 1 2 3 1 3 1 1 2

# Within-cluster sum of squares

[1] 12.60129 15.95014 11.48244

* **Cluster centers**: Each row corresponds to the centroid coordinates of a cluster.
* **Cluster membership**: Each number corresponds to the cluster assignment of the corresponding point in the dataset.
* **Within-cluster sum of squares**: Shows how compact each cluster is. Lower values indicate tighter clusters.

This example demonstrates how to perform K-means clustering in R and interpret the basic output using a randomly generated dataset. Adjust data and k to fit your specific dataset and desired number of clusters.

Result:

Therefore, the K-means clustering algorithm was successfully implemented in R.

**Ex No: 9**

# IMPLEMENTATION OF SEARCHING AND SORTING ALGORITHMS

**Aim***:*

To implement searching and sorting algorithm in R

#### Procedure:

Implementing searching and sorting algorithms in R programming involves creating functions to perform these operations on data structures like vectors or lists. Below are implementations of linear search, binary search, bubble sort, and quicksort in R, along with example outputs.

#### Linear Search

# Function to perform linear search linear\_search <- function(arr, key) { for (i in seq\_along(arr)) {

if (arr[i] == key) {

return(i) # Return index if key is found

}

}

return(-1) # Return -1 if key is not found

}

# Example usage:

arr <- c(3, 5, 1, 9, 2, 7)

key <- 9

result <- linear\_search(arr, key) cat("Linear Search:\n")

if (result == -1) {

cat("Key", key, "not found in array.\n")

} else {

cat("Key", key, "found at index", result, ".\n")

}

#### Output Example (Linear Search):

Linear Search:

Key 9 found at index 4 .

#### Binary Search

# Function to perform binary search (array must be sorted) binary\_search <- function(arr, key) {

low <- 1

high <- length(arr)

while (low <= high) {

mid <- low + floor((high - low) / 2)

if (arr[mid] == key) {

return(mid) # Return index if key is found

} else if (arr[mid] < key) { low <- mid + 1

} else {

high <- mid - 1

}

}

return(-1) # Return -1 if key is not found

}

# Example usage (array must be sorted): arr <- sort(c(3, 5, 1, 9, 2, 7))

key <- 9

result <- binary\_search(arr, key) cat("Binary Search:\n")

if (result == -1) {

cat("Key", key, "not found in array.\n")

} else {

cat("Key", key, "found at index", result, ".\n")

}

#### Output Example (Binary Search):

Binary Search:

Key 9 found at index 6 .

#### Bubble Sort

# Function to perform bubble sort bubble\_sort <- function(arr) {

n <- length(arr)

for (i in 1:(n - 1)) {

for (j in 1:(n - i)) {

if (arr[j] > arr[j + 1]) {

# Swap arr[j] and arr[j + 1] temp <- arr[j]

arr[j] <- arr[j + 1] arr[j + 1] <- temp

}

}

}

return(arr)

}

# Example usage:

arr <- c(3, 5, 1, 9, 2, 7)

sorted\_arr <- bubble\_sort(arr) cat("Bubble Sort:\n")

cat("Sorted array:", sorted\_arr, "\n")

#### Output Example (Bubble Sort):

Bubble Sort:

Sorted array: 1 2 3 5 7 9

#### Quicksort

# Function to perform quicksort quicksort <- function(arr) {

if (length(arr) <= 1) { return(arr)

}

pivot <- arr[ceiling(length(arr) / 2)] left <- arr[arr < pivot]

middle <- arr[arr == pivot] right <- arr[arr > pivot]

return(c(quicksort(left), middle, quicksort(right)))

}

# Example usage:

arr <- c(3, 5, 1, 9, 2, 7)

sorted\_arr <- quicksort(arr) cat("Quicksort:\n")

cat("Sorted array:", sorted\_arr, "\n")

#### Output Example (Quicksort):

Quicksort:

Sorted array: 1 2 3 5 7 9

These implementations demonstrate basic searching and sorting algorithms in R. Adjust the input arrays (arr) and keys (key) as needed for different datasets.

#### Result:

Therefore, the searching and sorting algorithm was successfully implemented in R

**Ex No: 10**

# HASHING –LINEAR PROBING

**Aim***:*

To implement hashing –linear probing in R

#### Procedure:

Hashing with linear probing is a technique used to resolve collisions in hash tables by placing colliding elements in the next available slot in the array. Below is an implementation of a

# Function to create a hash table with linear probing create\_hash\_table <- function(keys, values, table\_size) { hash\_table <- vector("list", length = table\_size)

for (i in seq\_along(keys)) {

hash\_value <- hash(keys[i], table\_size) while (!is.null(hash\_table[[hash\_value]])) {

hash\_value <- (hash\_value + 1) %% table\_size # Linear probing

}

hash\_table[[hash\_value]] <- list(key = keys[i], value = values[i])

}

return(hash\_table)

}

# Function to hash the key

hash <- function(key, table\_size) { as.integer(charToRaw(key)) %% table\_size + 1

}

# Function to retrieve value from hash table get\_value <- function(hash\_table, key, table\_size) {

hash\_value <- hash(key, table\_size) start\_value <- hash\_value

while (!is.null(hash\_table[[hash\_value]]) && hash\_table[[hash\_value]]$key != key) { hash\_value <- (hash\_value + 1) %% table\_size # Linear probing

if (hash\_value == start\_value) { break # Key not found in hash table

}

}

if (!is.null(hash\_table[[hash\_value]]) && hash\_table[[hash\_value]]$key == key) { return(hash\_table[[hash\_value]]$value)

} else {

return(NULL) # Key not found

}

}

# Example usage:

keys <- c("John", "Anna", "Peter", "Mike", "Alice") values <- c(25, 30, 28, 35, 27)

table\_size <- 10 # Choose an appropriate table size

# Create hash table

hash\_table <- create\_hash\_table(keys, values, table\_size)

# Retrieve values from hash table cat("Hash Table with Linear Probing:\n") for (key in keys) {

value <- get\_value(hash\_table, key, table\_size) if (!is.null(value)) {

cat("Key:", key, "-> Value:", value, "\n")

} else {

cat("Key:", key, "-> Not found in hash table.\n")

}}

#### Output Example:

Hash Table with Linear Probing:

Key: John -> Value: 25 Key: Anna -> Value: 30 Key: Peter -> Value: 28 Key: Mike -> Value: 35 Key: Alice -> Value: 27

This example demonstrates how to implement hashing with linear probing in R. Adjust keys, values, and table\_size to fit your specific use case. The create\_hash\_table function creates the hash table with linear probing, and the get\_value function retrieves values based on keys from the hash table.

Result :

Therefore, the Hashing-linear probing was successfully implemented in R