# R PROGRAMMING LAB

### EXPERIMENT-1

**Aim:** Implementation of DataFrames and Lists**.**

**Requirements:**

* R-studio
* R-Language

**Description:**

DataFrames: DataFrames are data displayed in a format as a table. DataFrames can have different types of data inside it. While the first column can be character the send and third can be numeric or logic. Use the dataframe() function to create a data frame.

Lists:

A list in r can contain many different data types inside it. A list is a collection of data which is ordered and changeable. To create a list of Dataframes we use the **list()** function in R and then pass each of the data frame you have created as arguments to the function.

**Source Code:**

Implementation of dataframe:

# Create a data frame Data\_Frame <- data.frame (

Training = c("Strength", "Stamina", "Other"), Pulse = c(100, 150, 120),

Duration = c(60, 30, 45)

)

# Print the data frame Data\_Frame

Implementation of list:

# List of strings

thislist <- list("apple", "banana", "cherry")

# Print the list thislist Output:

|  |  |
| --- | --- |
| Implementation of dataframes | Implementation of list |
| Training Pulse Duration | [[1]] |
| 1.Strength 100 60 | [1] "apple" |
| 2.Stamina 150 30 |  |
| 3.Other 120 45 | [[2]] |
|  | [1] "banana" |
|  | [[3]] |
|  | [1] "cherry" |

### Experiment-2

**Aim**:Implementation of Matrix operations.

**Requirements:**

* R-studio
* R-Language

##### Description:

A matrix is a two dimensional data set with columns and rows.

A column is a vertical representation of data, while a row is a horizontal representation of data.A matrix function in R is **a 2-dimensional array that has m number of rows and n number of columns**.

A matrix can be created with the matrix() function. Specify the nrow and ncol parameters to get the amount of rows and columns.

##### Operations on Matrices

There are four basic operations i.e. DMAS (Division, Multiplication, Addition, Subtraction) that can be done with matrices. Both the matrices involved in the operation should have the same number of rows and columns.

##### Matrices Addition

The addition of two same ordered matrices and yields a matrix where every element is the sum of corresponding elements of the input matrices.

##### Source code:

# R program to add two matrices # Creating 1st Matrix

B = matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3) # Creating 2nd Matrix

C = matrix(c(7, 8, 9, 10, 11, 12), nrow = 2, ncol = 3) # Getting number of rows and columns num\_of\_rows = nrow(B)

num\_of\_cols = ncol(B)

# Creating matrix to store results

sum = matrix(, nrow = num\_of\_rows, ncol = num\_of\_cols) # Printing Original matrices

print(B) print(C)

**Using ‘+’ operator for matrix addition:** Similarly, the following R script uses the in-built operator +:

# R program for matrix addition # using '+' operator

# Creating 1st Matrix

B = matrix(c(1, 2 + 3i, 5.4, 3, 4, 5), nrow = 2, ncol = 3) # Creating 2nd Matrix

C = matrix(c(2, 0i, 0.1, 3, 4, 5), nrow = 2, ncol = 3) # Printing the resultant matrix

print(B + C)

R provides the basic inbuilt operator to add the matrices. In the above code, all the elements in the resultant matrix are returned as complex numbers, even if a single element of a matrix is a complex number. **Properties of Matrix Addition:**

* **Commutative:** B + C = C + B
* **Associative:** For n number of matrices A + (B + C) = (A + B) + C
* Order of the matrices involved must be same.

##### Matrices Subtraction:

The subtraction of two same ordered matrices and yields a matrix

where every element is the difference of corresponding elements of the second input matrix from the first.

# Creating 1st Matrix

B = matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3) # Creating 2nd Matrix

C = matrix(c(7, 8, 9, 10, 11, 12), nrow = 2, ncol = 3) # Getting number of rows and columns num\_of\_rows = nrow(B)

num\_of\_cols = ncol(B)

# Creating matrix to store results

diff = matrix(, nrow = num\_of\_rows, ncol = num\_of\_cols) # Printing Original matrices

print(B) print(C)

# Calculating diff of matrices for(row in 1:num\_of\_rows)

{

for(col in 1:num\_of\_cols)

{

diff[row, col] <- B[row, col] – C[row, col]

}

}

# Printing resultant matrix print(diff)

**Using ‘-‘ operator for matrix subtraction:** Similarly, the following R script uses the in-built operator ‘-‘:

# Creating 1st Matrix

B = matrix(c(1, 2 + 3i, 5.4, 3, 4, 5), nrow = 2, ncol = 3) # Creating 2nd Matrix

C = matrix(c(2, 0i, 0.1, 3, 4, 5), nrow = 2, ncol = 3) # Printing the resultant matrix

print(B - C)

##### Properties of Matrix Subtraction:

* **Non-Commutative:** B – C != C – B
* **Non-Associative:** For n number of matrices A – (B – C) != (A – B) – C
* Order of the matrices involved must be same.

##### Matrices Multiplication:

:The multiplication of two same ordered matrices and yields a matrix where every element is the product of corresponding elements of the input matrices. # Creating 1st Matrix

B = matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3) # Creating 2nd Matrix

C = matrix(c(7, 8, 9, 10, 11, 12), nrow = 2, ncol = 3) # Getting number of rows and columns num\_of\_rows = nrow(B)

num\_of\_cols = ncol(B)

# Creating matrix to store results

prod = matrix(, nrow = num\_of\_rows, ncol = num\_of\_cols) # Printing Original matrices

print(B) print(C)

# Calculating product of matrices for(row in 1:num\_of\_rows)

{

for(col in 1:num\_of\_cols)

{

prod[row, col] <- B[row, col] \* C[row, col]

}

}

# Printing resultant matrix print(prod)

**Using ‘\*’ operator for matrix multiplication:** Similarly, the following R script uses the in- built operator \*:

# Creating 1st Matrix

B = matrix(c(1, 2 + 3i, 5.4), nrow = 1, ncol = 3) # Creating 2nd Matrix

C = matrix(c(2, 1i, 0.1), nrow = 1, ncol = 3) # Printing the resultant matrix

print (B \* C)

##### Properties of Matrix Multiplication:

* **Commutative:** B \* C = C \* B
* **Associative:** For n number of matrices A \* (B \* C) = (A \* B) \* C
* Order of the matrices involved must be same.

##### Matrices Division:

The division of two same ordered matrices and yields a matrix where every element is the quotient of corresponding elements of the first matrix element divided by the second.

# Creating 1st Matrix

B = matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3) # Creating 2nd Matrix

C = matrix(c(7, 8, 9, 10, 11, 12), nrow = 2, ncol = 3) # Getting number of rows and columns num\_of\_rows = nrow(B)

num\_of\_cols = ncol(B)

# Creating matrix to store results

div = matrix(, nrow = num\_of\_rows, ncol = num\_of\_cols) # Printing Original matrices

print(B) print(C)

# Calculating product of matrices for(row in 1:num\_of\_rows)

{

for(col in 1:num\_of\_cols)

{

div[row, col] <- B[row, col] / C[row, col]

}

}

# Printing resultant matrix print(div)

**Using ‘/’ operator for matrix division:** Similarly, the following R script uses the in-built operator /:

# Creating 1st Matrix

B = matrix(c(4, 6i, -1), nrow = 1, ncol = 3)

# Creating 2nd Matrix

C = matrix(c(2, 2i, 0), nrow = 1, ncol = 3)

# Printing the resultant matrix print (B / C)

##### Properties of Matrix Division:

* **Non-Commutative:** B / C != C / B
* **Non-Associative:** For n number of matrices A / (B / C) != (A / B) / C
* Order of the matrices involved must be same. output:

|  |  |  |  |
| --- | --- | --- | --- |
| Using Sum | Using + operator | Subtraction | Using – |
| [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6  [,1] [,2] [,3]  [1,] 7 9 11  [2,] 8 10 12  [,1] [,2] [,3]  [1,] 8 12 16  [2,] 10 14 18 | [,1] [,2] [,3]  [1,] 3+0i 5.5+0i 8+0i [2,] 2+3i 6.0+0i 10+0i | [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6  [,1] [,2] [,3]  [1,] 7 9 11  [2,] 8 10 12  [,1] [,2] [,3]  [1,] -6 -6 -6  [2,] -6 -6 -6 | [,1] [,2] [,3]  [1,] -1+0i 5.3+0i 0+0i [2,] 2+3i 0.0+0i 0+0i |
| Using Multiplication | Using \* | Using Division | Using / |
| [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6  [,1] [,2] [,3]  [1,] 7 9 11  [2,] 8 10 12  [,1] [,2] [,3]  [1,] 7 27 55  [2,] 16 40 72 | [,1] [,2] [,3]  [1,] 2+0i -3+2i 0.54+0i | [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6  [,1] [,2] [,3]  [1,] 7 9 11  [2,] 8 10 12  [,1] [,2]  [,3]  [1,] 0.1428571  0.3333333 0.4545455  [2,] 0.2500000  0.4000000 0.5000000 | [,1] [,2] [,3]  [1,] 2+0i 3+0i -Inf+NaNi |

**Aim:Implementation of factors.**

**Experiment -3**

**Requirements:**

* R-studio
* R-Language

**Description: Factors in R Programming Language** are data structures that are implemented to categorize the data or represent categorical data and store it on multiple levels.

Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like "Male, "Female" and True, False etc. They are useful in data analysis for statistical modeling.

Factors are created using the **factor ()** function by taking a vector as input. **Source code:**

# Create the vectors for data frame.

height <- c(132,151,162,139,166,147,122) weight <- c(48,49,66,53,67,52,40)

gender <- c("male","male","female","female","male","female","male")

# Create the data frame.

input\_data <- data.frame(height,weight,gender) print(input\_data)

# Test if the gender column is a factor. print(is.factor(input\_data$gender))

# Print the gender column so see the levels. print(input\_data$gender)

**output:**

height weight gender

|  |  |
| --- | --- |
| 1 132 | 48 male |
| 2 151 | 49 male |
| 3 162 | 66 female |
| 4 139 | 53 female |
| 5 166 | 67 male |
| 6 147 | 52 female |
| 7 122 | 40 male |

[1] TRUE

[1] male male female female male female male Levels: female male

## Experiment-4

**Aim**:Implementation of Quick Sort and Merge sort.

**Requirements:**

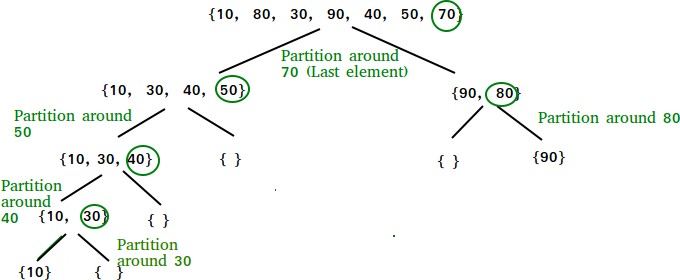
* R-studio
* R-Language

**Description**:

**QuickSort** is a [Divide and Conquer algorithm.](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/) It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick the first element as a pivot.
* Always pick the last element as a pivot (implemented below)
* Pick a random element as a pivot.
* Pick median as the pivot.

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



**Source Code**:

quick\_sort<-**function**(x)

{

**if**(length(x)<=1) **return**(x) pivot<-x[1]

rest<-x[-1]

pivot\_less<-quick\_sort(rest[rest<pivot]) pivot\_greater<-quick\_sort(rest[rest>=pivot]) **return**(c(pivot\_less,pivot,pivot\_greater))

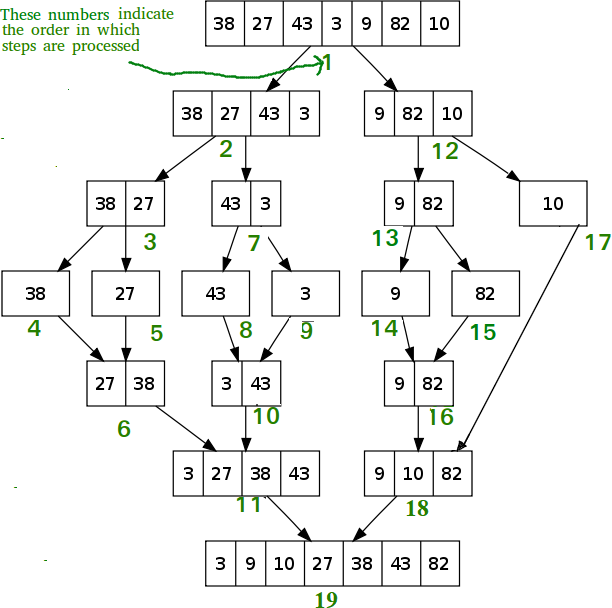
}

quick\_sort(c(5,4,12,13,3,8,88))

**Output**:

## [1] 3 4 5 8 12 13 88

**Merge sort**:

**Merge sort** is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. The process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

**Souce code**:

# function to merge two sorted arrays merge <- function(a, b) {

# create temporary array

temp <- numeric(length(a) + length(b))

# take two variables which initially points to # starting of the sorted sub arrays

# and j which points to starting of starting # of temporary array

astart <- 1

bstart <- 1

j <- 1

for(j in 1 : length(temp)) {

# if a[astart] < b[bstart] if((astart <= length(a) &&

a[astart] < b[bstart]) ||

bstart > length(b)) {

# insert a[astart] in temp and increment

# astart pointer to next temp[j] <- a[astart] astart <- astart + 1

}

else {

}

}

temp

}

temp[j] <- b[bstart] bstart <- bstart + 1

# function to sort the array mergeSort <- function(arr) {

# if length of array is greater than 1, # then perform sorting if(length(arr) > 1) {

# find mid point through which # array need to be divided

mid <- ceiling(length(arr)/2)

# first part of array will be from 1 to mid a <- mergeSort(arr[1:mid])

# second part of array will be # from (mid+1) to length(arr)

b <- mergeSort(arr[(mid+1):length(arr)])

# merge above sorted arrays merge(a, b)

}

# else just return arr with single element else {

arr

}

}

# take sample input

arr <- sample(1:100, 10)

# call mergeSort function result <- mergeSort(arr)

# print result Result Output:

[1] 6 8 16 19 21 24 35 38 74 90

**Experiment:5**

**Aim**:Implementation of Binary Search Tree.

**Requirements:**

* R-studio
* R-Language

##### Description:

R doesn't have a built-in binary search function, but writing such a function isn't too difficult. The first statement creates an integer vector with five values. The second statement sets up a target value for which to search. The third statement uses the built-in %in% operator to search for the target

##### Source Code:

Binary search=function(arr,item)

{

Low1;highlength(arr) While(lowhigh)

{

Midas.integer(round((low+high)/2)) If(abs(arr[Mid]-item)==0)

{

Return(mid)

}

Else if (arr[mid]<item)

{

Lowmid+1

}

Else

{

Highmid+1

}

Return 0

} Arr(4,0,3,1,5,6,7)

Sorted-arrsort(arr) Item4

Cat(“Array”,arr,”In sorted array”,sorted-arr,”in item= “,item); Indexbinary search (sorted-arr,item)

If (index!=0)

{

Cat(“Element is present at index”,index)

}

Else

{

Cat(“element not found”)

}

##### Output:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Array | 4 | 0 | 3 | 1 | 5 | 6 | 2 |
| Sorted Array | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Item=4 |  |  |  |  |  |  |  |

Elements in present at index 5.

**Experiment-6**

**Aim**:Implementation of reading and writing files.

**Requirements:**

* R-studio
* R-Language

##### Description:

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.

**Syntax:** read.delim(file, header = TRUE, sep = “\t”, dec = “.”, …)

##### Parameters:

* 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.

##### R – Writing to Files:

**Writing Data to CSV files in R Programming Language:**

CSV stands for Comma Separated Values. These files are used to handle a large amount of statistical data. Following is the syntax to write to a CSV file:

##### Syntax:

**write.table(my\_data, file = "my\_data.txt", sep = "")**

***Here,***

*csv() and csv2() are the function in R programming.*

* ***write.csv()*** *uses “.” for the decimal point and a comma (“, ”) for the separator.*
* ***write.csv2()*** *uses a comma (“, ”) for the decimal point and a semicolon (“;”) for the separator.*

##### Source code:

# Read a text file using read.delim()

myData = read.delim("ccc.txt", header = FALSE) print(myData)

write.table(mydata=”ccc.text”,sep=”,”) print(myData)

**Output**:

|  |  |
| --- | --- |
| Original ccc.txt | Hello! Welcome to R programming |
| Read ccc.txt | Hello! Welcome to R programming |
| Write.txt | Hello!,Welcome,to,R,programming |

# EXPERIMENT-7

**Aim:** To Implementation of Descriptive and Summary Statistics.

**Requirements:**

* R-studio
* R-Language

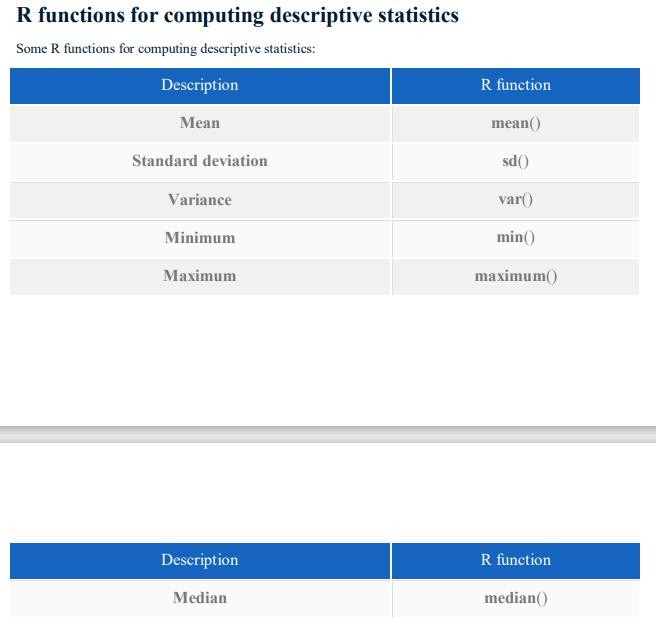
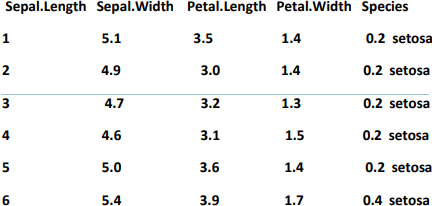
**Description:**

Syntax : # Store the data in the variable my\_data my\_data <- iris

Check your data You can inspect your data using the functions head() and tails(), which will display the first and the last part of the data, respectively.

**Syntax**:

# Print the first 6 rows head(my\_data, 6)

**Output**:

**Measure of central tendency:** mean, median, mode Roughly speaking, the central tendency measures the “average” or the “middle” of your data.

The most commonly used measures include:

the mean: the average value. It’s sensitive to outliers.

the median: the middle value. It’s a robust alternative to mean. and the mode: the most frequent value Syntax:

# Compute the mean value mean(my\_data$Sepal.Length) Output: [1] 5.843333

Syntax: # Compute the median value median(my\_data$Sepal.Length) Output: [1] 5.8

Range: minimum & maximum Range corresponds to biggest value minus the smallest value. It gives you the full spread of the data.

Syntax: # Compute the minimum value min(my\_data$Sepal.Length) Output: [1] 4.3

Syntax: # Compute the maximum value max(my\_data$Sepal.Length) Output: [1] 7.9

Syntax: # Range range(my\_data$Sepal.Length) Output: [1] 4.3 7.9

**Variance and standard deviation:**The variance represents the average squared deviation from the mean. The standard deviation is the square root of the variance. It measures the average deviation of the values, in the data, from the mean value

Syntax:

# Compute the variance var(my\_data$Sepal.Length)

# Compute the standard deviation =

# square root of th variance sd(my\_data$Sepal.Length) Output: [1] 0.6856935 [2] 0.8280661

Syntax:

# Compute the median median(my\_data$Sepal.Length)

# Compute the median absolute deviation mad(my\_data$Sepal.Length)

Output: [1] 5.8 [2] 1.0378

**Experiment-8**

**Aim**: Implementation of chart-Bar(side by Side,stacked),Line.

**Requirements:**

* R-studio
* R-Language

##### Description:

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

The **Syntax**

basic syntax to create a bar-chart in R is − barplot(H,xlab,ylab,main, names.arg,col)

Following is the description of the parameters used −

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

##### Source code:

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

# Give the chart file a name png(file = "barchart.png") # Plot the bar chart barplot(H)

# Save the file dev.off()

#SIDE BY SIDE LINE

#Define data-set columns x1 <- c(31,13,25,31,16)

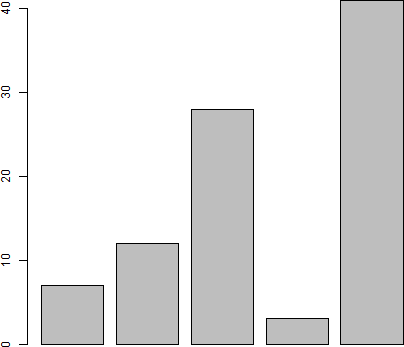
x2 <- c(12,23,43,12,22,45,32)

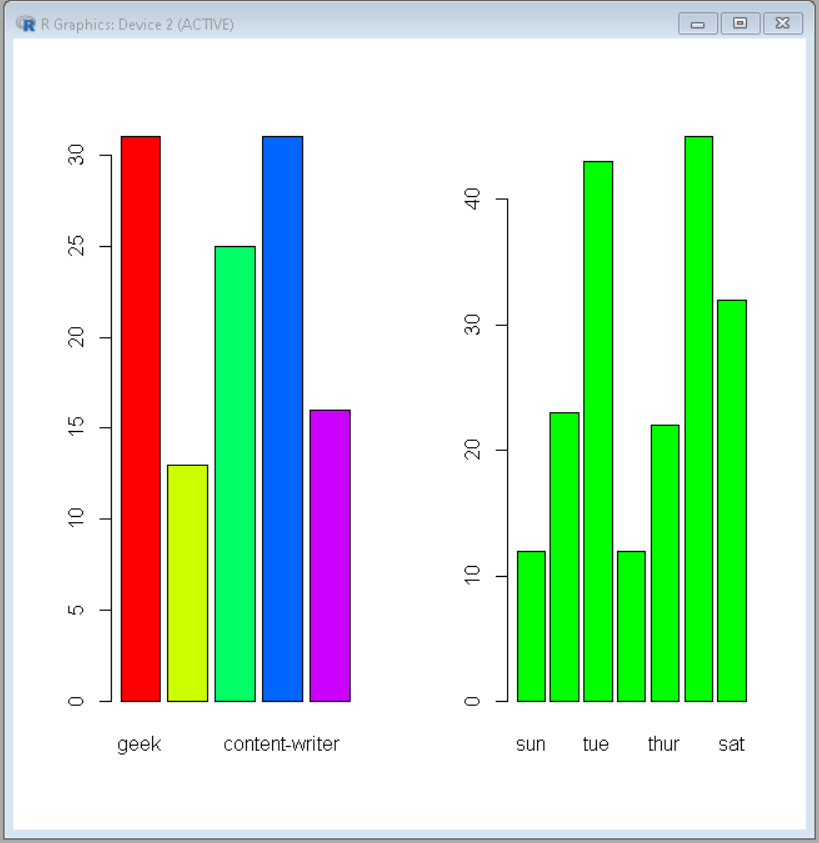
label1 <- c('geek','geek-i-knack','technical-scripter', 'content-writer','problem-setter')

label2 <- c('sun','mon','tue','wed','thur','fri','sat') # set the plotting area into a 1\*2 array par(mfrow=c(1,2))

# Draw the two bar chart using above datasets barplot(x1, names.arg = label1,col=rainbow(length(x1))) barplot(x2, names.arg = label2,col ="green")

**Output:**





**Experiment-9**

**Aim**:Implementation of Correlation,T-Test,ANOVA

**Requirements:**

* R-studio
* R-Language

##### Description:

Correlation on a statistical basis is **the method of finding the relationship between the variables in terms of the movement of the data**. That is, it helps us analyze the effect of changes made in one variable over the other variable of the dataset.

There are mainly two types of correlation:

* **Parametric Correlation** – [Pearson correlation(r)](https://www.geeksforgeeks.org/python-pearson-correlation-test-between-two-variables/): It measures a linear dependence between two variables (x and y) is known as a parametric correlation test because it depends on the distribution of the data.
* **Non-Parametric Correlation** – [Kendall(tau)](https://www.geeksforgeeks.org/python-kendall-rank-correlation-coefficient/?ref=rp) and [Spearman(rho)](https://www.geeksforgeeks.org/spearman-correlation-testing-in-r-programming/): They are rank-based correlation coefficients, are known as non-parametric correlation.

##### T-TEST:

**Classification:**

* One Sample T-test
* Two sample T-test
* Paired sample T-test One-sample T-Test:

The One-Sample T-Test is used to test the statistical difference between a sample mean and a known or assumed/hypothesized value of the mean in the population.

##### Two sample T-test:

It is used to help us to understand that the difference between the two means is real or simply by chance.

The general form of the test is t.test(y1, y2, paired=FALSE). By default, R assumes that the variances of y1 and y2 are unequal, thus defaulting to Welch’s test. To toggle this, we use the flag var.equal=TRUE.

##### Paired Sample T-test:

This is a statistical procedure that is used to determine whether the mean difference between two sets of observations is zero. In a paired sample t-test, each subject is measured two times, resulting in pairs of observations.

The test is run using the syntax t.test(y1, y2, paired=TRUE)

##### ANOVA

ANOVA test involves setting up:

* **Null Hypothesis:** All population means are equal.
* **Alternate Hypothesis:** Atleast one population mean is different from other. ANOVA tests are of two types:
* **One way ANOVA:** It takes one categorical group into consideration.
* **Two way ANOVA:** It takes two categorical group into consideration.

##### Source code:

Correlation:

X=c(1,2,3,4,5,6,7)

Y=c(1,3,6,2,7,4,5)

Result=corr(x,y,method=”pearson”) Cat(“pearson correlation coefficient is: “,result)

One-sample T-test set.seed(0)

sweetSold <- c(rnorm(50, mean = 140, sd = 5)) t.test(sweetSold, mu = 150) # Ho: mu = 150 Two-sample T-Test

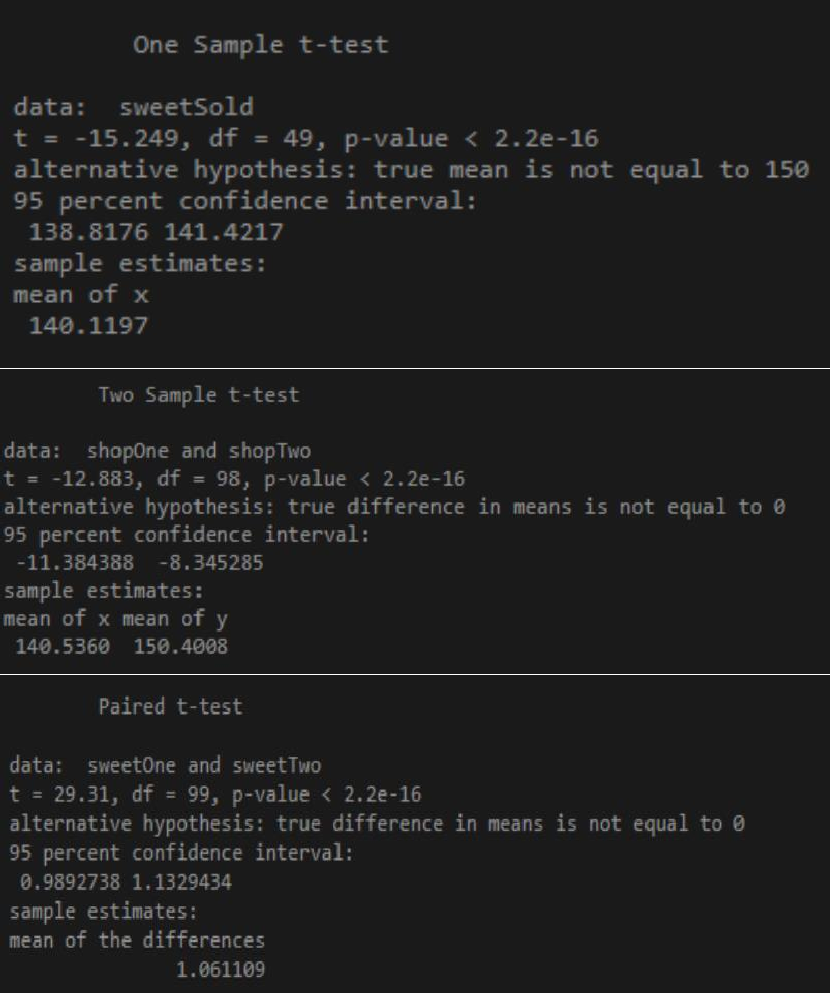
set.seed(0)

shopOne <- rnorm(50, mean = 140, sd = 4.5) shopTwo <- rnorm(50, mean = 150, sd = 4) t.test(shopOne, shopTwo, var.equal = TRUE)

Paled Sample T-Test set.seed(2820)

sweetOne <- c(rnorm(100, mean = 14, sd = 0.3)) sweetTwo <- c(rnorm(100, mean = 13, sd = 0.2)) t.test(sweetOne, sweetTwo, paired = TRUE) ANOVA T-TEST

install.packages("dplyr") library(dplyr)

boxplot(mtcars$disp~factor(mtcars$gear),xlab = "gear", ylab = "disp") mtcars\_aov <- aov(mtcars$disp~factor(mtcars$gear))summary(mtcars\_aov) OUTPUT:

|  |  |
| --- | --- |
| Correlation | Pearson correlation coefficient is : 0.5357143 |
| One-sample T-test |  |
| Two-sample T-Test |  |
| Paled Sample T-Test |  |

|  |  |
| --- | --- |
| ANOVA T-Test | Rplot2 (1).png |

**Experiment-10**

**Aim:**Implementation of Decision tree, Support Vector Classifications.

**Requirements:**

* R-studio
* R-Language

**Description:**

**Decision Tree:**

A decision tree is a type of supervised machine learning algorithm that is used for classification and regression analysis. It is a tree-like model that represents decisions and their possible consequences. The model starts with a single node, called the root, and branches out to multiple nodes, each of which represents a decision or a test of a particular feature or attribute.

At each node of the tree, the algorithm makes a decision based on the values of the input features, and then follows the appropriate branch of the tree to the next node. This process is repeated until a leaf node is reached, which represents the final decision or output of the algorithm.

The decision tree algorithm is particularly useful when the data has a hierarchical structure, where the features can be grouped into a hierarchy. The decision tree algorithm can be used to automatically learn the hierarchy and the decision rules based on the training data. The resulting model can be used to predict the outcome of new data with high accuracy, and is also easy to interpret and visualize.

**Support vector classification:**

Support Vector Classification (SVC) is a type of supervised machine learning algorithm that is used for classification problems. It is a non-probabilistic binary linear classifier, which means that it assigns input data points to one of two categories based on a linear boundary.

The SVC algorithm works by finding the hyperplane that best separates the input data into different classes. The hyperplane is a decision boundary that maximizes the margin between the two classes. The margin is the distance between the hyperplane and the closest data points from each class. The goal of the algorithm is to find the hyperplane that has the largest margin, as this is expected to generalize well on new, unseen data.

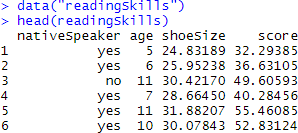
SVC can handle both linear and nonlinear classification problems through the use of kernel functions. The kernel function maps the input data to a higher-dimensional feature space, where a linear boundary can be found to separate the classes. The most commonly used kernel functions are linear, polynomial, and radial basis function (RBF) kernels.

One of the key advantages of SVC is its ability to handle high-dimensional datasets with relatively few training examples. It is also known for its robustness to outliers, and its ability to handle non-linearly separable data by using kernel functions. However, SVC can be sensitive to the choice of kernel function and its associated parameters, and can be computationally expensive for large datasets.

**Source code:**

# implementation of decision tree library(datasets)

library(caTools) library(party) library(dplyr) library(magrittr)

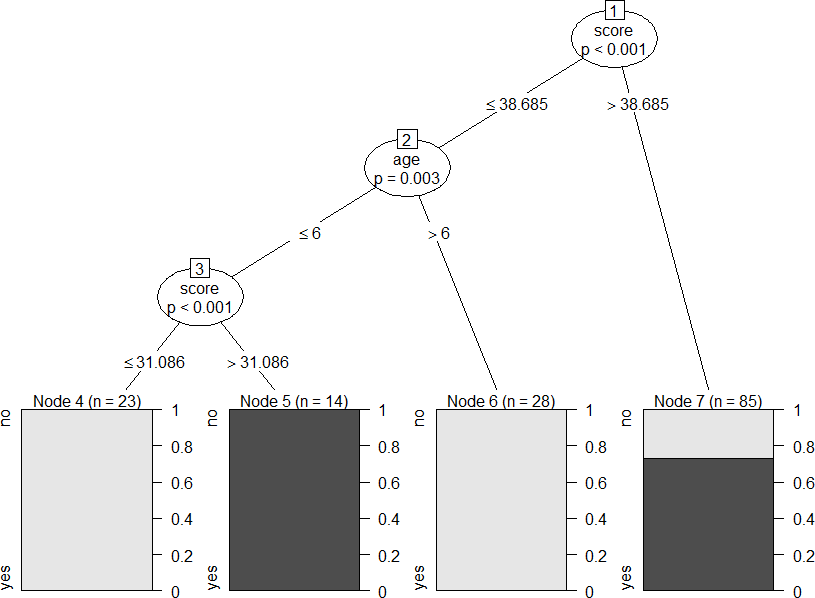
data("readingSkills") head(readingSkills) **Output:**

sample\_data = sample.split(readingSkills, SplitRatio = 0.8) train\_data<- subset(readingSkills, sample\_data == TRUE) test\_data<- subset(readingSkills, sample\_data == FALSE) model<- ctree(nativeSpeaker ~ ., train\_data)

plot(model)

ctree(formula, data)

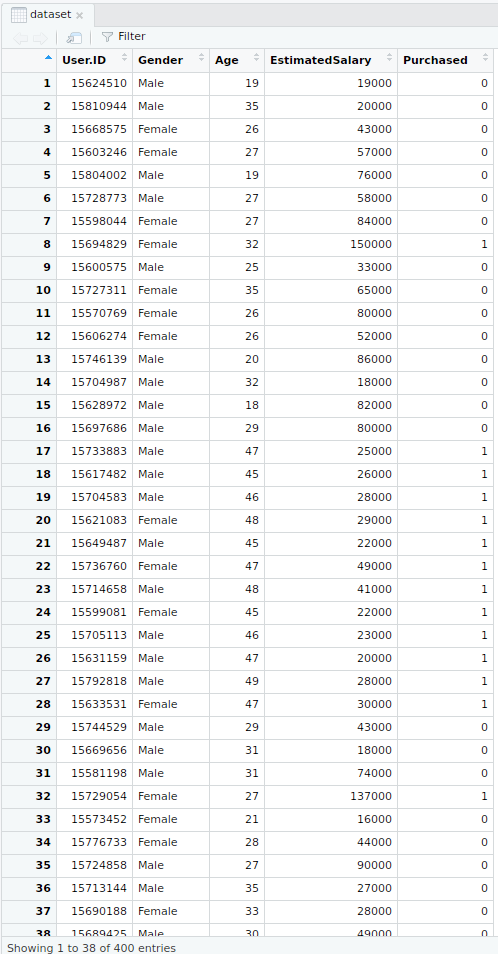
##### Output:



# testing the people who are native speakers # and those who are not

predict\_model<-predict(ctree\_, test\_data)

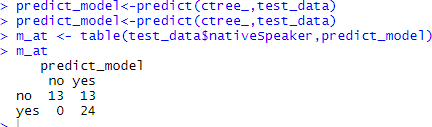
# creates a table to count how many are classified



# as native speakers and how many are not

m\_at<- table(test\_data$nativeSpeaker, predict\_model) m\_at

**output:**



### support vector classification:

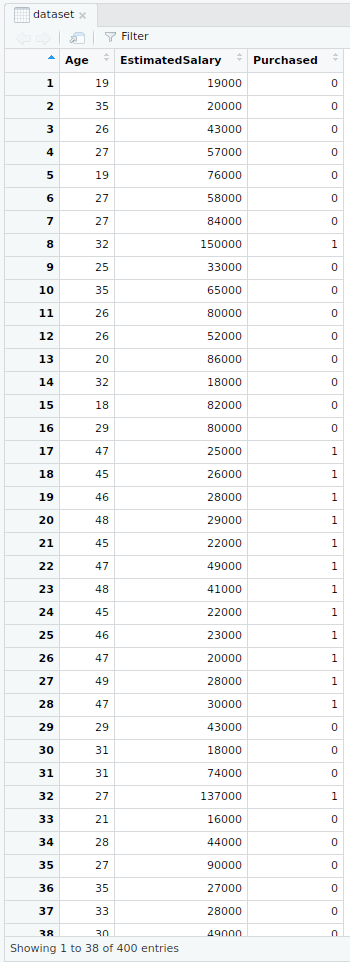
# Importing the dataset

dataset = read.csv('Social\_Network\_Ads.csv') dataset = dataset [3:5]

# Taking columns 3-5 dataset = dataset[3:5]

# Encoding the target feature as factor

dataset$Purchased = factor(dataset$Purchased, levels = c(0, 1))



# Splitting the dataset into the Training set and Test set install.packages('caTools')

library(caTools) set.seed(123)

split = sample.split(dataset$Purchased, SplitRatio = 0.75) training\_set = subset(dataset, split == TRUE)

test\_set = subset(dataset, split == FALSE) # Feature Scaling

training\_set[-3] = scale(training\_set[-3]) test\_set[-3] = scale(test\_set[-3])

# Fitting SVM to the Training set install.packages('e1071') library(e1071)

classifier = svm(formula = Purchased ~ .,

data = training\_set, type = 'C-classification', kernel = 'linear')

# Predicting the Test set results

y\_pred = predict(classifier, newdata = test\_set[-3]) # Making the Confusion Matrix

cm = table(test\_set[, 3], y\_pred)

# installing library ElemStatLearn library(ElemStatLearn)

# Plotting the training data set results set = training\_set

X1 = seq(min(set[, 1]) - 1, max(set[, 1]) + 1, by = 0.01)

X2 = seq(min(set[, 2]) - 1, max(set[, 2]) + 1, by = 0.01) grid\_set = expand.grid(X1, X2)

colnames(grid\_set) = c('Age', 'EstimatedSalary') y\_grid = predict(classifier, newdata = grid\_set) plot(set[, -3],

main = 'SVM (Training set)',

xlab = 'Age', ylab = 'Estimated Salary', xlim = range(X1), ylim = range(X2))

contour(X1, X2, matrix(as.numeric(y\_grid), length(X1), length(X2)), add = TRUE) points(grid\_set, pch = '.', col = ifelse(y\_grid == 1, 'coral1', 'aquamarine')) points(set, pch = 21, bg = ifelse(set[, 3] == 1, 'green4', 'red3'))

set = test\_set

X1 = seq(min(set[, 1]) - 1, max(set[, 1]) + 1, by = 0.01)

X2 = seq(min(set[, 2]) - 1, max(set[, 2]) + 1, by = 0.01) grid\_set = expand.grid(X1, X2)

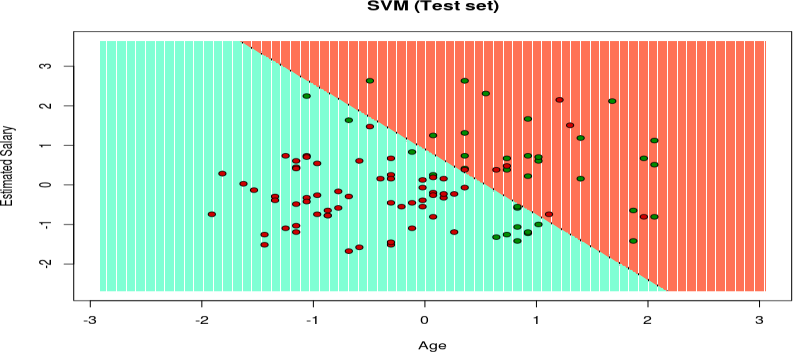
colnames(grid\_set) = c('Age', 'EstimatedSalary') y\_grid = predict(classifier, newdata = grid\_set)

plot(set[, -3], main = 'SVM (Test set)',

xlab = 'Age', ylab = 'Estimated Salary', xlim = range(X1), ylim = range(X2))

contour(X1, X2, matrix(as.numeric(y\_grid), length(X1), length(X2)), add = TRUE) points(grid\_set, pch = '.', col = ifelse(y\_grid == 1, 'coral1', 'aquamarine')) points(set, pch = 21, bg = ifelse(set[, 3] == 1, 'green4', 'red3'))

**output:**



**EXPERIMENT-11**

**Aim:**Implementation of Linear, Random Forest Regressions.

**Requirements:**

* R-studio
* R-Language

**Description:**

**Linear regression:**

Linear regression is a type of supervised machine learning algorithm used for predicting a continuous target variable. It is a statistical approach that models the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data.

In simple linear regression, there is only one independent variable, and the linear equation takes the form: y = mx + b

where y is the target variable, x is the independent variable, m is the slope of the line, and b is the y- intercept. The goal of linear regression is to find the values of m and b that minimize the difference between the predicted values and the actual values of the target variable.

In multiple linear regression, there are multiple independent variables, and the linear equation takes the form:

y = b0 + b1x1 + b2x2 + ... + bnxn

where y is the target variable, xi are the independent variables, and bi are the coefficients of the linear equation. The goal of multiple linear regression is to find the values of bi that minimize the difference between the predicted values and the actual values of the target variable.

Linear regression is a widely used algorithm in machine learning and statistical modeling due to its simplicity, interpretability, and ability to capture linear relationships between variables. However, it is important to note that linear regression assumes a linear relationship between the independent and dependent variables, and may not be appropriate for non-linear relationships.

**Random Forest :**

Random Forest is an ensemble learning algorithm used for classification and regression tasks. It is a combination of multiple decision trees, where each tree is built using a random subset of features and a random subset of training samples. The output of the algorithm is the mode of the predicted classes for classification tasks or the average of the predicted values for regression tasks.

The main idea behind Random Forest is to reduce the variance of the decision trees by introducing randomness in the feature selection and the training samples. By doing so, the algorithm is less prone to overfitting and can generalize better to new data.

To build a Random Forest model, the following steps are taken:

1. Select a random subset of features from the dataset.
2. Randomly select a subset of training samples from the dataset.
3. Build a decision tree using the selected features and samples.
4. Repeat steps 1-3 multiple times to build multiple decision trees.

5. For classification tasks, the predicted class is the mode of the predicted classes of the individual trees. For regression tasks, the predicted value is the average of the predicted values of the individual trees..

**SOURCE CODE:**

# Create a dataset of age and height

age <- c(18, 20, 22, 24, 26, 28, 30, 32, 34, 36)

height <- c(68, 69, 71, 72, 73, 74, 75, 76, 77, 78)

data <- data.frame(age, height)

# Plot the data to visualize the relationship plot(height ~ age, data = data)

# Fit a linear regression model

model <- lm(height ~ age, data = data) # View the model summary summary(model)

# Make predictions using the model new\_data<- data.frame(age = c(25, 28, 31)) predictions <- predict(model, new\_data) predictions

# Plot the predicted values lines(new\_data**$age, predictions, col = "red") Output:**

Call:

lm(formula = height ~ age, data = data) Residuals:

1 2 3 4 5 6 7 8 9 10

-0.83333 -0.33333 0.16667 0.66667 1.16667 1.66667 -0.83333 -0.33333 0.16667 0.66667

**Coefficients**:

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

(Intercept) 64.00000 2.52489 25.334 1.55e-08 \*\*\*

age 0.50000 0.10541 4.735 0.00182 \*\*

---

signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1 Residual standard error: 0.9428 on 8 degrees of freedom Multiple R-squared: 0.7183, Adjusted R-squared: 0.6711

F-statistic: 22.44 on 1 and 8 DF, p-value: 0.001818

1 2 3

71.16667 74.16667 77.16667

**Source code for Random Forest:**

# Load the randomForest library library(randomForest)

# Create a dataset of age, gender, and height age <- c(18, 20, 22, 24, 26, 28, 30, 32, 34, 36)

gender <- factor(c("M", "F", "M", "F", "M", "F", "M", "F", "M", "F")) height <- c(68, 69, 71, 72, 73, 74, 75, 76, 77, 78)

data <- data.frame(age, gender, height)

# Split the data into training and testing sets set.seed(123)

train\_index<- sample(1:nrow(data), nrow(data) \* 0.7) train\_data<- data[train\_index, ]

test\_data<- data[-train\_index, ] # Fit a random forest model

model <- randomForest(height ~ age + gender, data = train\_data, ntree = 500, mtry = 2)

# View the model summary print(model)

# Make predictions using the model predictions <- predict(model, test\_data)

# Calculate the mean squared error of the predictions

mse<- mean((test\_data$height - predictions) ^ 2) mse

output:

Call:

randomForest(formula = height ~ age + gender, data = train\_data, ntree = 500, mtry = 2) Type of random forest: regression

Number of trees: 500 No. of variables tried at each split: 2

Mean of squared residuals: 0.13

% Var explained: 93.88

**Aim:** Implementation of clustering.

**Requirements**:

* R-Studio
* R-language

### EXPERIMENT-12

**Description:**

[**Clustering**](https://www.geeksforgeeks.org/clustering-in-machine-learning/) **in** [**R**](https://www.geeksforgeeks.org/introduction-to-r-programming-language/) **Programming Language** is an unsupervised learning technique in which the data set is partitioned into several groups called as clusters based on their similarity. Several clusters of data are produced after the segmentation of data. All the objects in a cluster share common characteristics. During data mining and analysis, clustering is used to find similar datasets.

##### Methods of Clustering:

There are 2 types of clustering in R programming:

* + **Hard clustering:** In this type of clustering, the data point either belongs to the cluster totally or not and the data point is assigned to one cluster only. The algorithm used for hard clustering is k-means clustering.
  + **Soft clustering:** In soft clustering, the probability or likelihood of a data point is assigned in the clusters rather than putting each data point in a cluster. Each data point exists in all the clusters with some probability. The algorithm used for soft clustering is the fuzzy clustering method or soft k-means.

##### K-Means Clustering in R Programming language:

K-Means is an iterative hard clustering technique that uses an unsupervised learning algorithm. In this, total numbers of clusters are pre-defined by the user and based on the similarity of each data point, the data points are clustered. This algorithm also finds out the centroid of the cluster.

**Syntax:** kmeans(x, centers, nstart)

**where,**

* + **x** represents numeric matrix or data frame object
  + **centers** represents the **K** value or distinct cluster centers

#### **nstart** represents number of random sets to be chosen

**Source code:** install.packages("factoextra") library(factoextra)

# Loading dataset df <- mtcars

# Omitting any NA values df <- na.omit(df)

# Scaling dataset df <- scale(df)

# output to be present as PNG file png(file = "KMeansExample.png")

\

km <- kmeans(df, centers = 4, nstart = 25)

# Visualize the clusters fviz\_cluster(km, data = df)

# saving the file dev.off()

# output to be present as PNG file png(file = "KMeansExample2.png")

km <- kmeans(df, centers = 5, nstart = 25) # Visualize the clusters

fviz\_cluster(km, data = df)

# saving the file dev.off()

**Output:**

|  |  |
| --- | --- |
| When k = 4 | When k = 5 |
| https://media.geeksforgeeks.org/wp-content/uploads/20200515070746/KMeansExample1.png | https://media.geeksforgeeks.org/wp-content/uploads/20200515070830/KMeansExample2.png |