



Dr. Vishwanath Karad

**MIT WORLD PEACE
UNIVERSITY** | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Submitted Through

Department of Computer Science and Applications

MCA SCIENCE 2023 – 24

SYMCA DIV – A

ASSIGNMENTS

ON

**COS6074B - Lab on Artificial Intelligence
and Machine Learning**

SUBMITTED BY:

Dhanani Rohan J.
Roll No :- 57
PRN :- 1132220432

SUBMITTED TO:

Prof. Apurva Sharma

Date : 11/12/2023

Certificate

This is to certify that Mr./Ms. **Dhanani Rohan J.** Roll. No. **57**, a student of MCA (Science), Semester III has successfully completed **9** no. of assignments in Lab Course on MCA120B Lab on Artificial Intelligence and Machine Learning during A.Y. 2023-24.

Subject Teacher

Head of the Department

Principal

INDEX

Sr. No.	Title of the Assignment	Date	Page No.	Remark	Signature
MCAB Lab on AI&ML					
1	Breadth – First Search		1		
2	Depth – First Search		4		
3	Best – First Search		6		
4	Linear Regression		9		
5	Decision Tree		13		
6	Logistic Regression		16		
7	Naïve Bayes Algorithm		20		
8	Apriori Algorithm		24		
9	K – Means Algorithm		28		

Q – 1) Program to implement the Breadth-First Search (BFS) algorithm.

Aim :-

To put into practice the Breadth-First Search (BFS) method in order to traverse and determine the shortest path between the source and target nodes.

Objective:-

1. Understand the BFS algorithm and its application in graph traversal.
2. Implement BFS in Python.
3. Visualize the traversal of a sample graph.

Introduction :-

Breadth-First Search (BFS) is an algorithm that explores a graph by systematically visiting all neighbors of a node before moving on to their neighbors. It's often used to find the shortest path and is suitable for tasks that require a level-by-level examination of a graph's structure.

Algorithm :-

1. Create a queue and a set to keep track of visited nodes.
2. Initialize the queue with the source node.
3. Create a set or array to keep track of visited nodes.
4. While the queue is not empty:
 - a) De queue the a node from front of the queue.
 - b) If the node is the target node return the path.
 - c) If the node has not been visited:
 1. Mark the node as visited.
 2. En queue all unvisited neighbour of the nodes.
5. If the target node is not found, return an appropriate message.

Program :-

```
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        v = queue.pop(0)
        print(v)
        for u in graph[v]:
            if u not in visited:
                visited.append(u)
                queue.append(u)

print("")
print("Breadth-First Search Result, starting from vertex 1:-")

g = {
    '1': ('2', '3'),
    '2': ('4', '5'),
    '3': ('6'),
    '4': (),
    '5': ('6'),
    '6': ()
}

visited = []
queue = []

bfs(visited, g, '1')
```

Output :-

```
OUTPUT  PROBLEMS  TERMINAL  PORTS  SEARCH TERMINAL OUTPUT  COMMENTS  DEB  
● PS D:\SEM - 3\AI & ML> python BFS.py  
  
Breadth-First Search Result, starting from vertex 1:-  
1  
2  
3  
4  
5  
6  
○ PS D:\SEM - 3\AI & ML> █
```

Q – 2) Program to implement the Depth-First Search (DFS) algorithm.

Aim :-

To implement the Breadth-First Search (BFS) algorithm for traversing and find shortest path from source node to target node.

Objectives:-

1. Understand the DFS algorithm and its application in graph traversal.
2. Implement DFS in Python.
3. Visualize the traversal of a sample graph.

Introduction :-

Depth-First Search (DFS) is a graph traversal algorithm that explores as deeply as possible along a branch before backtracking. It starts at a node, visits unexplored neighbors, marks visited nodes, and backtracks when no more unvisited neighbors are available. DFS is useful for tasks like finding paths, topological sorting, and exploring connected components in a graph.

Algorithm :-

1. Create a data structure to Keep a track of nodes to be visited.
2. Start from a source node.
3. Mark the source node as visited.
4. Push the source node onto the stack.
5. While the stack is not empty:
 - a) Pop a node from the stack.
 - b) Process the node.
6. For each unvisited neighbour of the current node:
 - a) Mark the neighbour node as visited.
 - b) Push the neighbour node onto the stack.
7. Continue the step 5-6 until the stack is empty.

Program :-

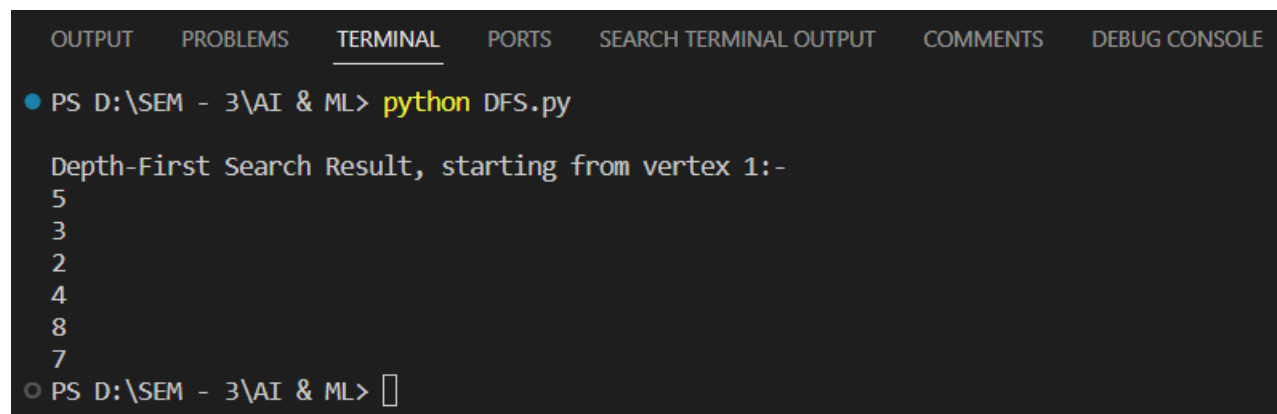
```
def dfs(visited, graph, node):
    if node not in visited:
        print(node)
        visited.add(node)
        for n in graph[node]:
            dfs(visited, graph, n)

graph = {
    '5' : ['3', '7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [],
    '4' : ['8'],
    '8' : []
}

visited = set()
print("")
print("Depth-First Search Result, starting from vertex 1:-")

dfs(visited, graph, '5')
```

Output :-



```
OUTPUT  PROBLEMS  TERMINAL  PORTS  SEARCH TERMINAL OUTPUT  COMMENTS  DEBUG CONSOLE

● PS D:\SEM - 3\AI & ML> python DFS.py

Depth-First Search Result, starting from vertex 1:-
5
3
2
4
8
7
○ PS D:\SEM - 3\AI & ML> 
```


Q – 3) Program to implement the Best-First Search algorithm.

Aim :-

To implement the Best-First Search (BFS) algorithm for finding the most promising or best path in a search space.

Objectives:-

1. Understand the BFS algorithm and its application in finding the best path.
2. Implement Best-First Search in Python.
3. Visualize the traversal of a sample search space.

Introduction :-

Best-First Search is a graph traversal algorithm that selects the most promising node to explore next based on a heuristic function. Instead of exploring all neighbors like in BFS, it prioritizes nodes that seem most likely to lead to the goal. This makes it well-suited for tasks like finding the shortest path or optimizing search in applications such as artificial intelligence and navigation systems.

Algorithm :-

1. Maintain a priority queue to prioritize nodes.
2. Begin at the source node and add it to the priority queue with a cost of 0.
3. Repeatedly select and explore the node with the lowest cost from the priority queue.
4. If the selected node is the target, the search is complete.
5. Explore the unvisited neighbors of the selected node and add them to the priority queue with their heuristic costs.
6. Continue until reaching the target node or exploring all possibilities.
7. The algorithm uses heuristics to guide the search efficiently, but a specific heuristic function is not provided in the code.

Program :-

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

def bestFS(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True

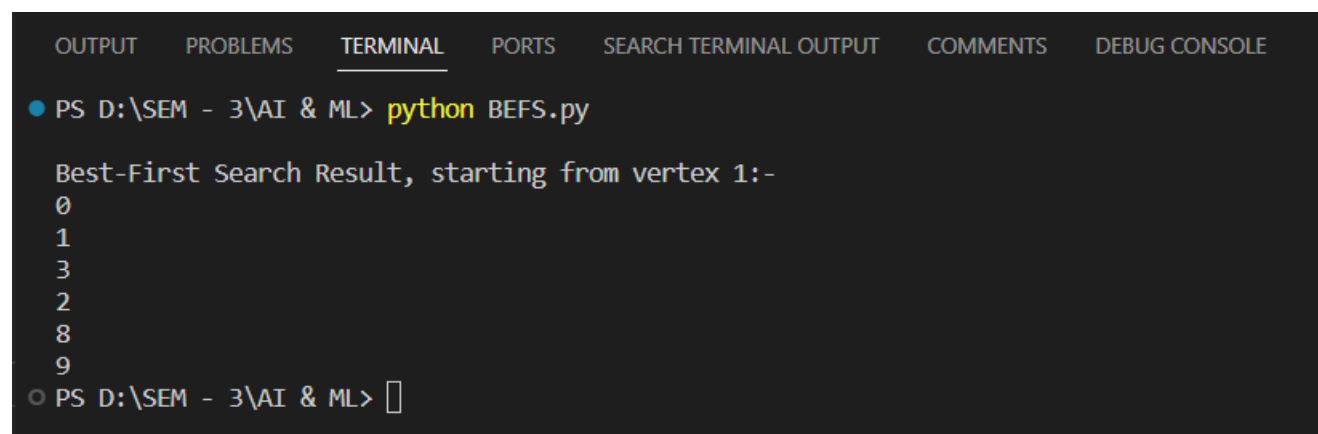
    while pq.empty() == False:
        u = pq.get()[1]
        print(u)
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
```

```
source = 0
target = 9

print("")
print("Best-First Search Result, starting from vertex 1:-")
bestFS(source, target, v)
```

Output :-



The screenshot shows a terminal window with a dark background. At the top, there are several tabs: OUTPUT, PROBLEMS, TERMINAL (which is active and underlined), PORTS, SEARCH TERMINAL OUTPUT, COMMENTS, and DEBUG CONSOLE. Below the tabs, the terminal shows a command prompt where the user has entered 'python BEFS.py'. The output of the program is displayed below the command, showing the text 'Best-First Search Result, starting from vertex 1:-' followed by a list of numbers: 0, 1, 3, 2, 8, and 9. The prompt 'PS D:\SEM - 3\AI & ML>' is visible at the bottom of the terminal window.

```
PS D:\SEM - 3\AI & ML> python BEFS.py

Best-First Search Result, starting from vertex 1:-
0
1
3
2
8
9
PS D:\SEM - 3\AI & ML> 
```

Q – 4) Write a program to implement linear regression by building basic regression model that will fit a line to the data.

Aim :-

To gain a basic understanding of "linear regression" and how to use it to fit a straight line to a given dataset or set of datapoints. creating a Python program that does the same and illustrates it as well.

Objectives:-

1. Understand the concept of linear regression and its mathematical representation.
2. Implement a simple linear regression algorithm from scratch.
3. Fit a line to a sample dataset using the implemented algorithm.
4. Evaluate the model's performance using relevant metrics.
5. Visualize the data and the fitted line.

Introduction :-

A supervised machine learning method called linear regression is used to model the connection between a dependent variable, or target, and one or more independent variables. The goal of simple linear regression is to minimize the sum of squared differences between the observed and predicted values by fitting a straight line, or linear equation, to the data points.

Algorithm :-

1. Import necessary libraries
2. Define a function `estimate_coef(x, y)` to estimate the coefficients (intercept and slope) of a linear regression model given the input data `x` and corresponding output data `y`.
 - Calculate the number of data points `n`.
 - Compute the means of `x` and `y`, denoted as `m_x` and `m_y`.
 - Calculate the sums of squares `SS_xy` and `SS_xx`.
 - Use the formula to calculate the slope `b_1` and the intercept `b_0`.

- Return the calculated coefficients as a tuple.

3. Define a function `plot_regression_line(x, y, b)` to create a scatter plot of the data points and overlay the regression line.

- Scatter plot the data points using `plt.scatter`.
- Calculate the predicted values `y_pred` based on the coefficients `b`.
- Plot the regression line using `plt.plot`.
- Label the axes and display the plot using `plt.show`.

4. Define the `main()` function to:

- Create numpy arrays for input data `x` and output data `y`.
- Call `estimate_coef` to calculate the coefficients of the linear regression model.
- Print the estimated coefficients.
- Call `plot_regression_line` to create and display the scatter plot with the regression line.

5. Use the `if __name__ == "__main__":` block to execute the `main()` function when the script is run.

Program :-

```
import numpy as np
import matplotlib.pyplot as plt

def estimate_coef(x, y):

    n = np.size(x)

    m_x = np.mean(x)
```

```

m_y = np.mean(y)

SS_xy = np.sum(y*x) - n*m_y*m_x
SS_xx = np.sum(x*x) - n*m_x*m_x

b_1 = SS_xy / SS_xx
b_0 = m_y - b_1*m_x

return (b_0, b_1)

def plot_regression_line(x, y, b):

    plt.scatter(x, y, color="m",
                marker="o", s=30)

    y_pred = b[0] + b[1]*x

    plt.plot(x, y_pred, color="g")

    plt.xlabel('x')
    plt.ylabel('y')

    plt.show()

def main():

    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

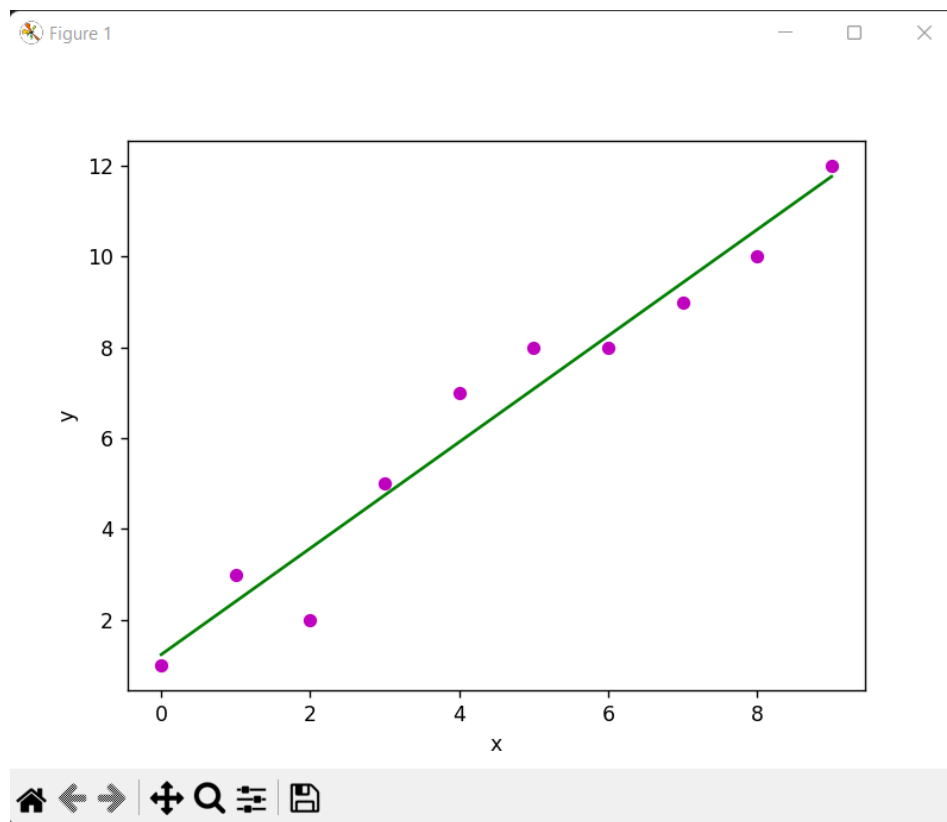
    b = estimate_coef(x, y)
    print("Liner Regression model Estimated coefficients:\nb_0 = {} \
        \nb_1 = {}".format(b[0], b[1]))

    plot_regression_line(x, y, b)

if __name__ == "__main__":
    main()

```

Output :-



```
OUTPUT  PROBLEMS  TERMINAL  PORTS  SEARCH TERMINAL OUTPUT  COMMENTS  DEBUG CONSOLE
```

```
PS D:\SEM - 3\AI & ML> python Liner.py
Liner Regression model Estimated coefficients:
b_0 = 1.2363636363636363
b_1 = 1.1696969696969697
█
```

Q – 5) Write a program to implement Decision Tree.

Aim :-

Understanding and applying the well-liked machine learning technique, Decision Tree, to classification tasks. Here, we will train a decision tree classifier and assess its performance using any dataset.

Introduction :-

Machine learning techniques that are flexible enough to be applied to both regression and classification problems are decision trees. They produce a tree-like structure of decision nodes and leaf nodes by recursively dividing the dataset according to the most informative attributes. Decision trees can handle numerical and categorical data and are interpretable.

In this, a decision tree classifier will be constructed using any dataset. In order to forecast the target variable, which is either the class of the corresponding dataset, the classifier will discover patterns and relationships in the data.

Algorithm :-

1. Import the necessary libraries
2. Read a dataset from a CSV file into a pandas DataFrame and display the first few rows using `df.head()`. Check the shape of the dataset using `df.shape`.
3. Extract the target variable (`y`) and features (`X`) from the DataFrame:
 - `y` is the "diagnosis" column encoded using `LabelEncoder`.
 - `X` is all columns except "diagnosis," "id," and "Unnamed: 32."
4. Split the data into training and testing sets using `train_test_split`, ensuring stratified splitting based on the target variable (`y`) and setting a random seed for reproducibility.
5. Create a Decision Tree classifier with the default maximum depth and fit it to the training data using `dt.fit`.
6. Make predictions on the training and testing sets using the trained model and calculate accuracy for both sets with `accuracy_score`.

7. Print the accuracy of the Decision Tree model on the training and testing sets. Note the presence of overfitting, as the training accuracy is significantly higher than the testing accuracy.
8. Create a new Decision Tree classifier with a specified maximum depth and fit it to the training data.
9. Make predictions on the training and testing sets using the updated model and calculate accuracy for both sets.
10. Print the updated accuracy of the Decision Tree model on the training and testing sets, which should show reduced overfitting.

Program :-

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

df = pd.read_csv("data.csv")
df.head()
df.shape

y = df.loc[:, "diagnosis"].values
X = df.drop(["diagnosis", "id", "Unnamed: 32"], axis=1).values

le = LabelEncoder()
y = le.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=0)

dt = DecisionTreeClassifier(random_state = 42)
dt.fit(X_train, y_train)

y_train_pred = dt.predict(X_train)
y_test_pred = dt.predict(X_test)

tree_train = accuracy_score(y_train, y_train_pred)
```

```

tree_test = accuracy_score(y_test, y_test_pred)

print(f'Decision tree train/test accuracies: {tree_train:.3f}/{tree_test:.3f}')

# The score on the training set is 100%, but the score on the test set is 95%. This
# means that our model has an overfitting problem.
dt = DecisionTreeClassifier(max_depth=2)
dt.fit(X_train, y_train)

# Now, let's take a look at the performance of this model on the training and the test
# set again.
y_train_pred=dt.predict(X_train)
y_test_pred=dt.predict(X_test)
tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print(f'Decision tree train/test accuracies: {tree_train:.3f}/{tree_test:.3f}')

```

Output :-

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_me
0	842302	M	17.99	10.38	122.80	1001.0	0.11840
1	842517	M	20.57	17.77	132.90	1326.0	0.08474
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960
3	84348301	M	11.42	20.38	77.58	386.1	0.14250
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030

```
Decision tree train/test accuracies:1.000/0.951
```

```
Decision tree train/test accuracies:0.951/0.923
```

Q-6) Write a program to implement logistic regression by building basic regression model that will fit a line to the data.

Aim :-

The aim of implementing logistic regression is to create a model that can effectively classify data into binary classes, estimate probabilities, and provide insights into the relationship between input features and the outcome. It is a fundamental tool for various binary classification tasks in machine learning and statistics.

Objectives:-

The primary objective is to build a logistic regression model that can accurately classify data into one of two classes (e.g., 0 or 1) and visualize the logistic curve that represents the probability estimates.

Data Preparation: We'll start by preparing the dataset, ensuring it contains relevant features and a binary target variable.

Logistic Regression Model: We'll create a logistic regression model using scikit-learn, which will be trained to model the probability of the binary outcome based on the input features.

Model Training: The model will be trained using the prepared data, learning to make predictions based on the features.

Model Evaluation: We'll evaluate the model's performance using appropriate classification metrics, ensuring it can make accurate predictions.

Decision Threshold Selection: We'll discuss the concept of choosing an appropriate decision threshold for classification.

Visualization: Finally, we'll visualize the data points and the logistic curve, showing how the model estimates probabilities and makes classification decisions.

Introduction :-

In this program, we'll implement logistic regression, a powerful tool for binary classification tasks. Logistic regression models the probability of an observation belonging to one of two classes, and in the context of this introduction, we'll consider it as "fitting a line" to distinguish between the two classes.

Algorithm :-

1.Data Generation:

- Generate a feature matrix `X` with random values.
- Create a binary target variable `y` based on a threshold.

2. Logistic Regression Model:

- Initialize a logistic regression model with the 'liblinear' solver.

3. Model Training:

- Train the logistic regression model on the data.

4. Predictions:

- Make predictions using the trained model.

5. Data Visualization:

- Create a scatter plot for actual data points (blue) and predicted values (red).
- Generate the logistic curve (probability estimates) and plot it in green.

6. Display the Plot:

- Show the plot to visualize the results.

Program :-

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

# Generate some example data
np.random.seed(0)
X = np.random.rand(100, 1) # Feature
y = (X > 0.5).astype(int).ravel() # Binary target variable

# Create a logistic regression model
model = LogisticRegression(solver='liblinear')

# Fit the model to the data
model.fit(X, y)

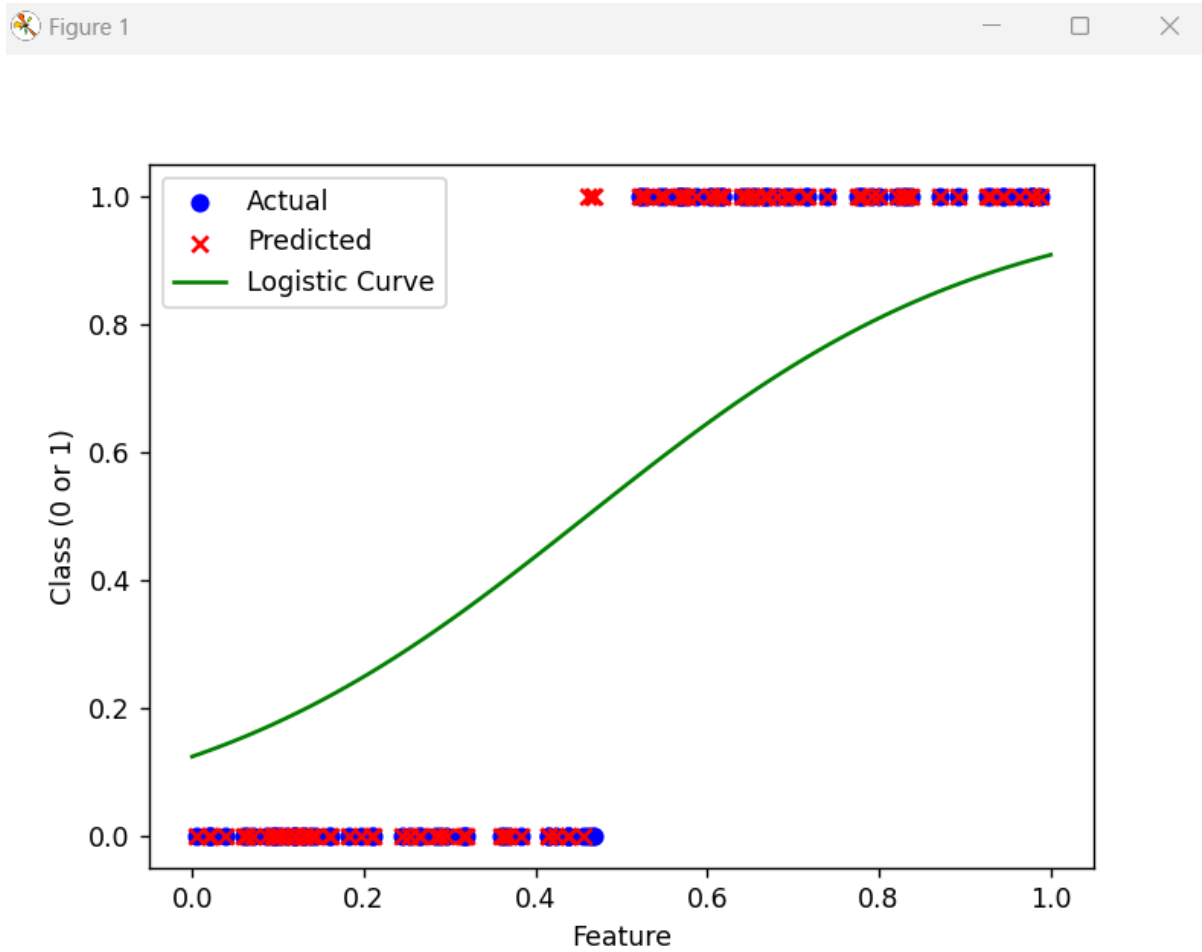
# Make predictions on the same data for visualization
y_pred = model.predict(X)

# Plot the data points
plt.scatter(X, y, c='blue', marker='o', label='Actual')
plt.scatter(X, y_pred, c='red', marker='x', label='Predicted')

# Plot the logistic curve
X_test = np.linspace(0, 1, 100)
y_prob = model.predict_proba(X_test.reshape(-1, 1))[:, 1]
plt.plot(X_test, y_prob, color='green', label='Logistic Curve')

plt.xlabel('Feature')
plt.ylabel('Class (0 or 1)')
plt.legend()
plt.show()
```

Output :-



Q-7) Implementation of Naive Bayes Algorithm using Weka tool

Aim :-

The aim of this experiment is to implement the Naive Bayes algorithm using the Weka tool and analyze its performance in classifying data. This involves understanding the basic concepts of the Naive Bayes algorithm and utilizing the features of the Weka tool to apply the algorithm to a dataset.

Objectives:-

Naive Bayes Algorithm:

- Naive Bayes is a probabilistic algorithm based on Bayes' theorem, which calculates the probability of a hypothesis given the evidence.
- Naive Bayes is a popular machine learning algorithm used for classification tasks, particularly in natural language processing and text classification.
- In the context of classification, it is commonly used for text categorization and spam filtering.
- In statistics, naive Bayes classifiers are considered as simple probabilistic classifiers that apply Bayes' theorem.
- The "naive" assumption in Naive Bayes is that the features used to describe an observation are conditionally independent, given the class label.
- Despite this simplifying assumption, Naive Bayes often performs well in practice.

Weka Tool:

- Weka is a collection of machine learning algorithms for data mining tasks.
- It provides a graphical user interface for easy experimentation with different algorithms and data.

- Weka supports various machine learning tasks, including classification, clustering, regression, and feature selection.

Algorithm :-

1. Initialization: Calculate the prior probability for each class based on the training dataset.

2. Training:

For each feature in the dataset:

- Calculate the likelihood of each feature given the class.

3. Prediction:

For a new instance:

- Calculate the posterior probability of each class given the features.
- Assign the class with the highest posterior probability as the predicted class.

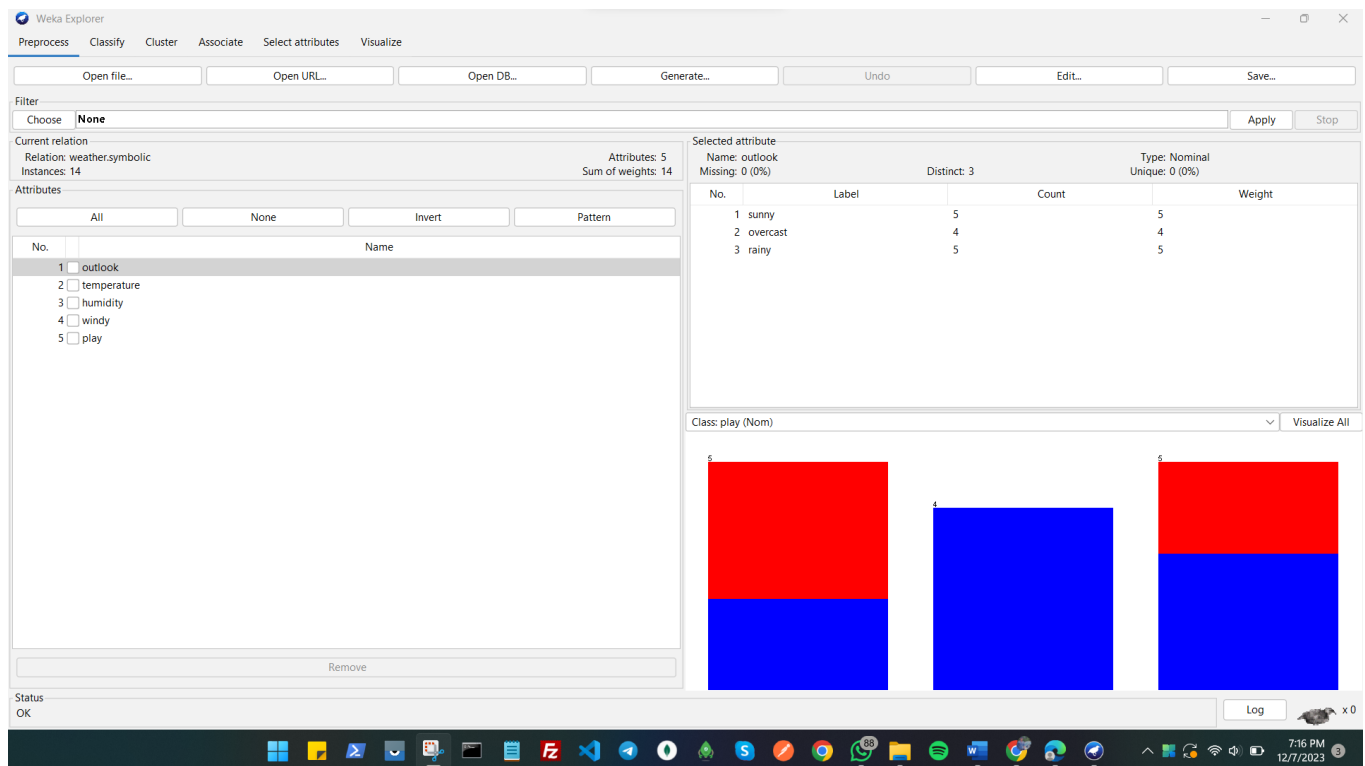
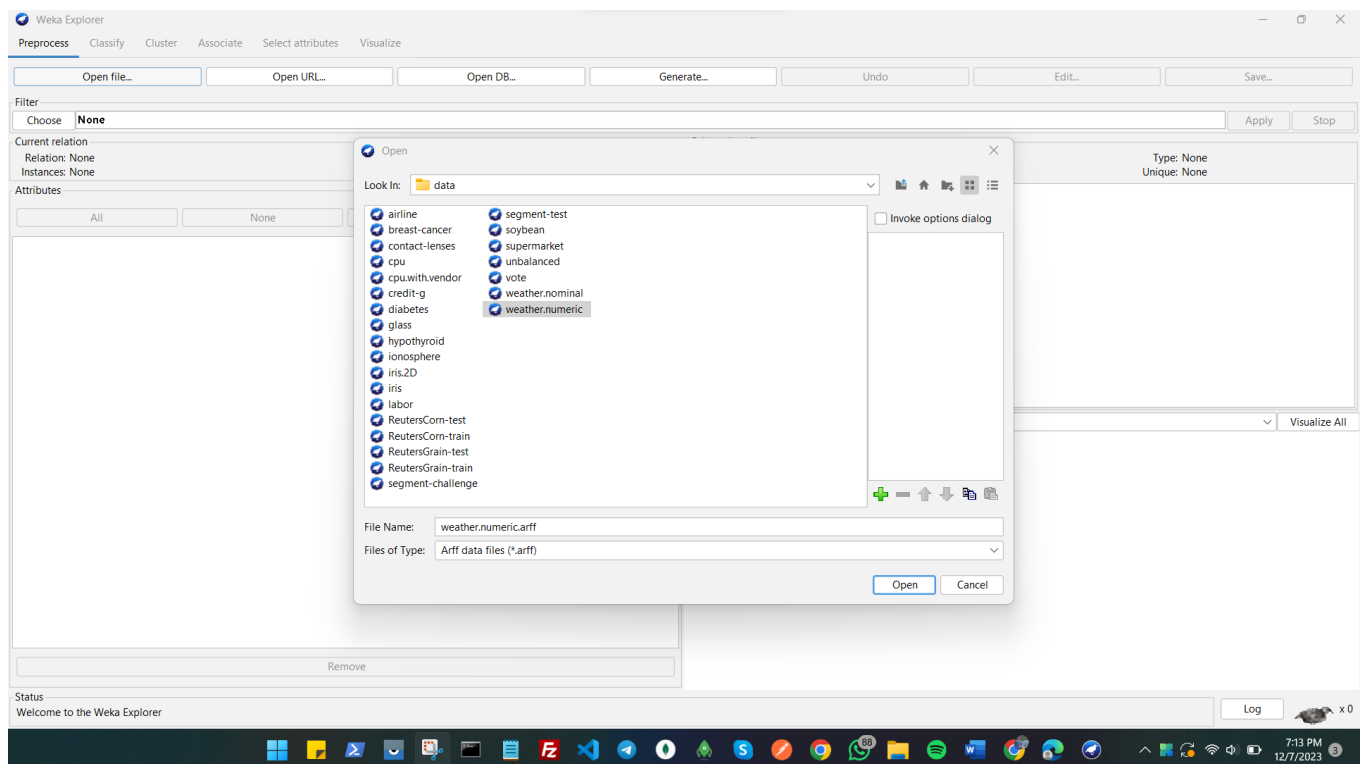
4. Evaluation:

- Assess the performance of the Naive Bayes classifier using metrics such as accuracy, precision, recall, and F1 score.

When using Weka for Naive Bayes implementation, you can follow these general steps:

- i. Load your dataset into Weka.
- ii. Choose the Naïve Bayes classifier.
- iii. Set any relevant options or parameters.
- iv. Apply the classifier to your dataset.
- v. Evaluate the performance of the classifier.

Output :-



Weka Explorer

Preprocess Classify Cluster Associate Select attributes Visualize

Classifier: Choose NaiveBayes

Test options:

- ☐ Use training set
- ☐ Supplied test set
- ☒ Cross-validation Folds: 10
- ☐ Percentage split % 66

More options...

(Nom) play

Start Stop

Result list (right-click for options):

19:21:44 - bayes.NaiveBayes

Classifier output:

```

=== Run information ===

Scheme:      weka.classifiers.bayes.NaiveBayes
Relation:    weather.symbolic
Instances:   14
Attributes:  5
    outlook
    temperature
    humidity
    windy
    play
Test mode:   10-fold cross-validation

=== Classifier model (full training set) ===

Naive Bayes Classifier

Attribute      Class
              yes    no
              (0.63) (0.38)
=====
outlook
  sunny        3.0    4.0
  overcast     5.0    1.0
  rainy        4.0    3.0
  [total]     12.0    8.0

temperature
  hot          3.0    3.0
  mild         5.0    3.0
  cool         4.0    2.0
  [total]     12.0    8.0

humidity
  high         4.0    5.0
  normal       7.0    2.0
  [total]     11.0    7.0
  
```

Status: OK

Log x0

7:22 PM 12/7/2023

Weka Explorer

Preprocess Classify Cluster Associate Select attributes Visualize

Classifier: Choose NaiveBayes

Test options:

- ☐ Use training set
- ☐ Supplied test set
- ☒ Cross-validation Folds: 10
- ☐ Percentage split % 66

More options...

(Nom) play

Start Stop

Result list (right-click for options):

19:21:44 - bayes.NaiveBayes

Classifier output:

```

=====
outlook
  sunny        3.0    4.0
  overcast     5.0    1.0
  rainy        4.0    3.0
  [total]     11.0    7.0

windy
  TRUE         4.0    4.0
  FALSE        7.0    3.0
  [total]     11.0    7.0

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances      8          57.1429 %
Incorrectly Classified Instances    6          42.8571 %
Kappa statistic                    -0.0244
Mean absolute error                 0.4374
Root mean squared error             0.4916
Relative absolute error             91.8631 %
Root relative squared error         99.6492 %
Total Number of Instances          14

=== Detailed Accuracy By Class ===

              TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
0.778    0.800    0.636    0.778    0.700    -0.026    0.578    0.697    yes
0.200    0.222    0.333    0.200    0.250    -0.026    0.578    0.557    no
Weighted Avg.   0.571    0.594    0.528    0.571    0.539    -0.026    0.578    0.647

=== Confusion Matrix ===

a b  <-- classified as
7 2 | a = yes
4 1 | b = no
  
```

Status: OK

Log x0

7:22 PM 12/7/2023

Q – 8) Implementation of Apriori Algorithm using Weka tool.

Aim:-

The aim of this experiment is to implement the Apriori Algorithm using the Weka tool for association rule mining. Association rule mining is a data mining technique used to discover interesting relationships and patterns within large datasets. The Apriori Algorithm is specifically employed for finding frequent itemsets and generating association rules based on the concept of support and confidence.

Theory:-

Apriori Algorithm :-

- The Apriori Algorithm is a classic algorithm for association rule mining.
- It works by iteratively discovering frequent itemsets and then generating association rules based on these frequent itemsets.
- The Apriori algorithm uses frequent itemsets to generate association rules, and it is designed to work on the databases that contain transactions.
- With the help of these association rule, it determines how strongly or how weakly two objects are connected.
- It is the iterative process for finding the frequent itemsets from the large dataset

Weka Tool:

- Weka is a collection of machine learning algorithms for data mining tasks.
- It provides a graphical user interface for easy experimentation with different algorithms and data.
- Weka supports various machine learning tasks, including classification, clustering, regression, and feature selection.

Algorithm:-

1. Initialization:

- Begin with individual items as the initial candidate itemsets. Scan the database to determine the support of each item.

2. Generate Candidate Itemsets:

Combine frequent itemsets of size k to generate candidate itemsets of size $k+1$.

3. Prune Candidate Itemsets:

- Eliminate candidate itemsets that contain infrequent subsets.

4. Calculate Support:

- Scan the database to determine the support of each candidate itemset.

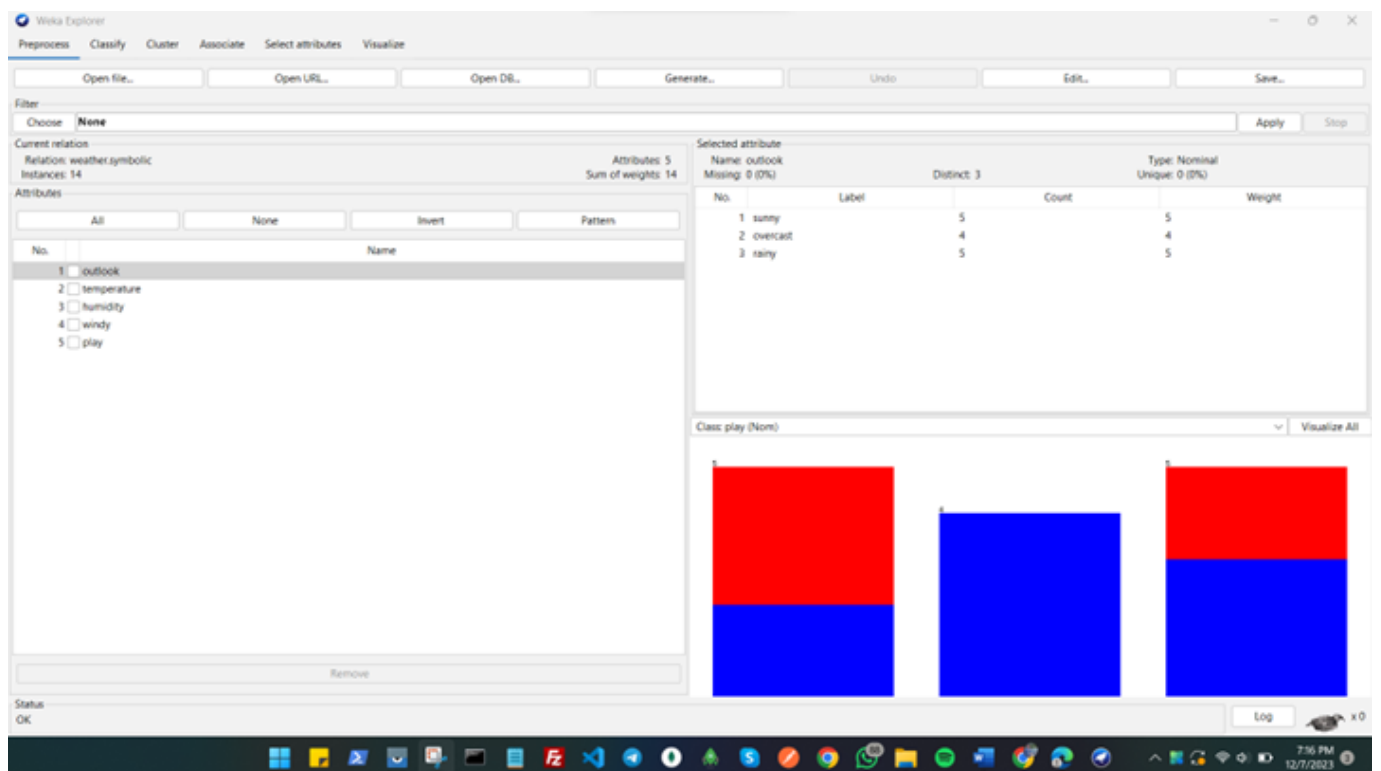
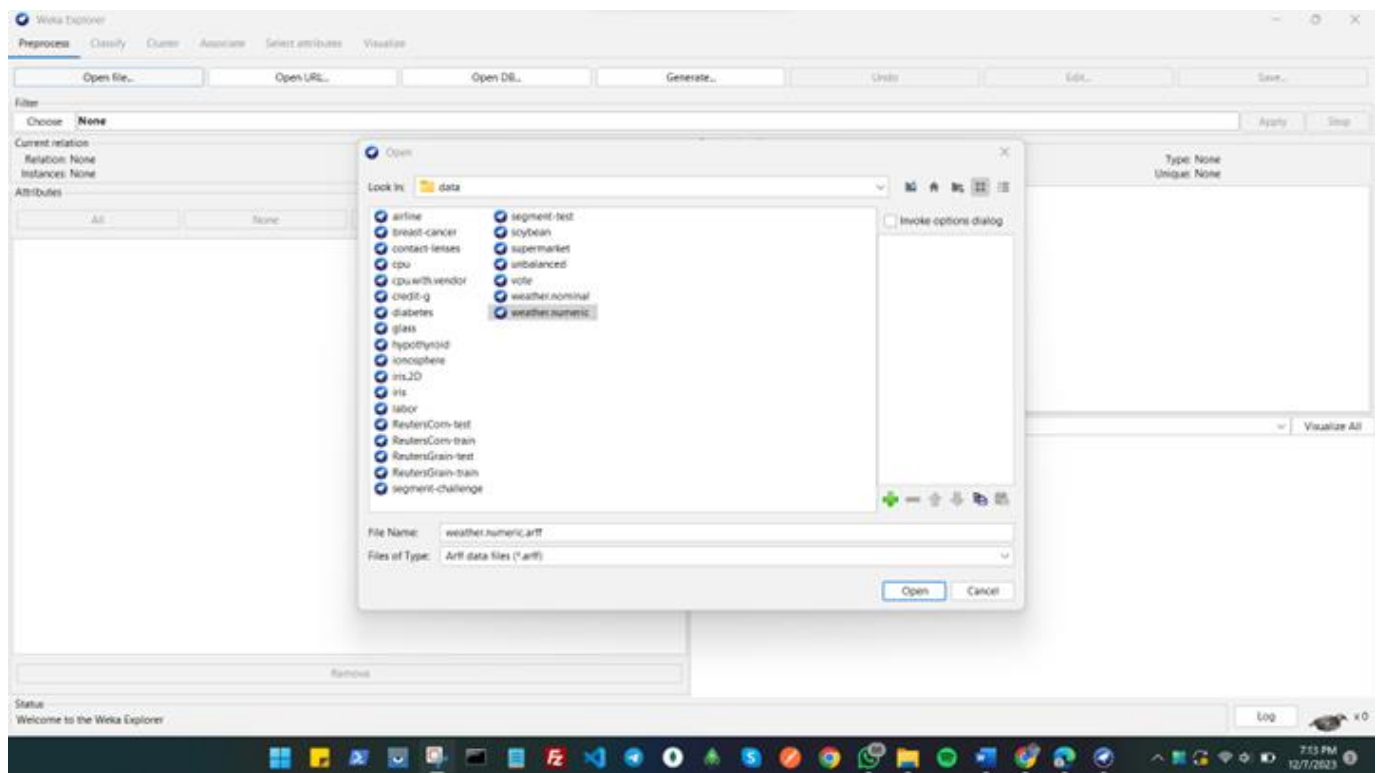
5. Repeat:

- Repeat steps 2-4 until no new frequent itemsets can be generated.

6. Generate Association Rules:

- From the final set of frequent itemsets, generate association rules with a user-specified minimum confidence threshold.

Output :-



Weka Explorer

Preprocess Classify Cluster Associate Select attributes Visualize

Associator

Choose **Apriori** -N 10 -T 0 -C 0.9 -D 0.05 -U 1.0 -M 0.1 -S -1.0 -c -1

Start Stop

Result list (right-click for ...)

19:25:05 - Apriori

Associator output

```

=== Run information ===

Scheme:      weka.associations.Apriori -N 10 -T 0 -C 0.9 -D 0.05 -U 1.0 -M 0.1 -S -1.0 -c -1
Relation:    weather.symbolic
Instances:    14
Attributes:   5
              outlook
              temperature
              humidity
              windy
              play

=== Associator model (full training set) ===

Apriori
=====

Minimum support: 0.15 (2 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 17

Generated sets of large itemsets:

Size of set of large itemsets L(1): 12
Size of set of large itemsets L(2): 47
Size of set of large itemsets L(3): 39
Size of set of large itemsets L(4): 6

Best rules found:

1. outlook=overcast 4 ==> play=yes 4    <conf: (1)> lift: (1.56) lev: (0.1) [1] conv: (1.43)
2. temperature=cool 4 ==> humidity=normal 4    <conf: (1)> lift: (2) lev: (0.14) [2] conv: (2)
3. humidity=normal windy=FALSE 4 ==> play=yes 4    <conf: (1)> lift: (1.56) lev: (0.1) [1] conv: (1.43)
4. outlook=sunny play=no 3 ==> humidity=high 3    <conf: (1)> lift: (2) lev: (0.11) [1] conv: (1.5)

```

Status OK

Log x0

7:25 PM 12/7/2023

Weka Explorer

Preprocess Classify Cluster Associate Select attributes Visualize

Associator

Choose **Apriori** -N 10 -T 0 -C 0.9 -D 0.05 -U 1.0 -M 0.1 -S -1.0 -c -1

Start Stop

Result list (right-click for ...)

19:25:05 - Apriori

Associator output

```

temperature
humidity
windy
play

=== Associator model (full training set) ===

Apriori
=====

Minimum support: 0.15 (2 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 17

Generated sets of large itemsets:

Size of set of large itemsets L(1): 12
Size of set of large itemsets L(2): 47
Size of set of large itemsets L(3): 39
Size of set of large itemsets L(4): 6

Best rules found:

1. outlook=overcast 4 ==> play=yes 4    <conf: (1)> lift: (1.56) lev: (0.1) [1] conv: (1.43)
2. temperature=cool 4 ==> humidity=normal 4    <conf: (1)> lift: (2) lev: (0.14) [2] conv: (2)
3. humidity=normal windy=FALSE 4 ==> play=yes 4    <conf: (1)> lift: (1.56) lev: (0.1) [1] conv: (1.43)
4. outlook=sunny play=no 3 ==> humidity=high 3    <conf: (1)> lift: (2) lev: (0.11) [1] conv: (1.5)
5. outlook=sunny humidity=high 3 ==> play=no 3    <conf: (1)> lift: (2.8) lev: (0.14) [1] conv: (1.93)
6. outlook=rainy play=yes 3 ==> windy=FALSE 3    <conf: (1)> lift: (1.75) lev: (0.09) [1] conv: (1.29)
7. outlook=rainy windy=FALSE 3 ==> play=yes 3    <conf: (1)> lift: (1.56) lev: (0.08) [1] conv: (1.07)
8. temperature=cool play=yes 3 ==> humidity=normal 3    <conf: (1)> lift: (2) lev: (0.11) [1] conv: (1.5)
9. outlook=sunny temperature=hot 2 ==> humidity=high 2    <conf: (1)> lift: (2) lev: (0.07) [1] conv: (1)
10. temperature=hot play=no 2 ==> outlook=sunny 2    <conf: (1)> lift: (2.8) lev: (0.09) [1] conv: (1.29)

```

Status OK

Log x0

7:25 PM 12/7/2023

Q – 9) Implement the “K- means Clustering” algorithm.

Aim :-

The aim of implementing the K-means clustering algorithm is to partition a given dataset into K clusters, where each data point belongs to the cluster with the nearest mean. The algorithm aims to minimize the within-cluster variance and assign data points to clusters in a way that reflects inherent patterns or structures in the data.

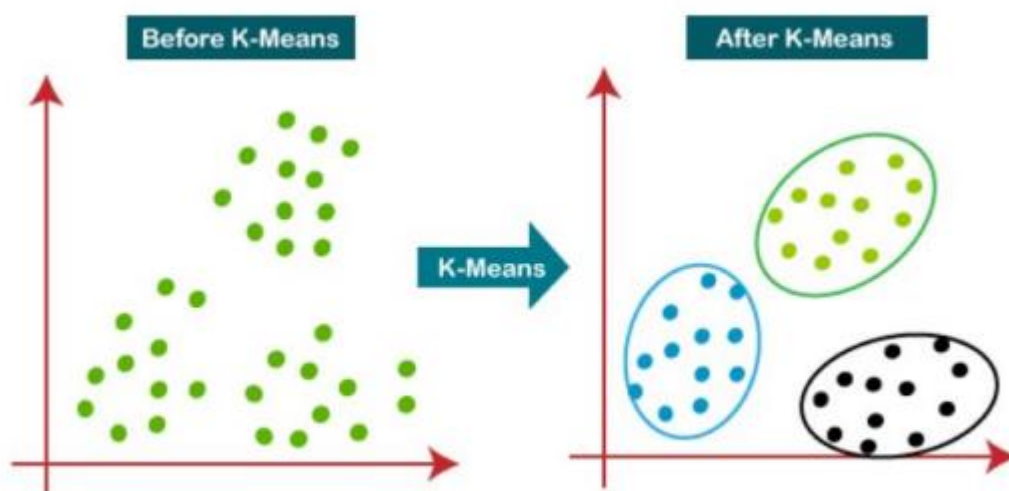
Objectives:-

- Develop a K-means clustering model with the objective of partitioning the dataset into K clusters that minimize within-cluster variance.
- Implement the K-means clustering algorithm to determine the optimal number of clusters (K) by exploring various possibilities and selecting the configuration that best captures the inherent patterns and structures within the data.
- K-means clustering begins with the description of a cost function over a parameterized set of possible clustering, and the objective of the clustering algorithm is to find a minimum cost partitioning (clustering).
- The clustering function is turned into an optimization problem under this model.

Introduction :-

- K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if $K=2$, there will be two clusters, and for $K=3$, there will be three clusters, and so on.
- It is an iterative algorithm that divides the unlabeled dataset into k different clusters in such a way that each dataset belongs only one group that has similar properties.

- It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training.
- It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.
- The algorithm takes the unlabeled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.
- The k-means clustering algorithm mainly performs two tasks:
 - Determines the best value for K center points or centroids by an iterative process.
 - Assigns each data point to its closest k-center. Those data points which are near to the particular k-center, create a cluster.
- The below diagram explains the working of the K-means Clustering Algorithm:



Algorithm :-

The working of the K-Means algorithm is explained in the below steps:

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be other from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

Step-4: Calculate the variance and place a new centroid of each cluster.

Step-5: Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

Step-7: The model is ready.

Program :-

```
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
```

```
dataset = pd.read_csv('/content/Mall_Customers.csv')
print(dataset)
```

```
print(dataset)
```

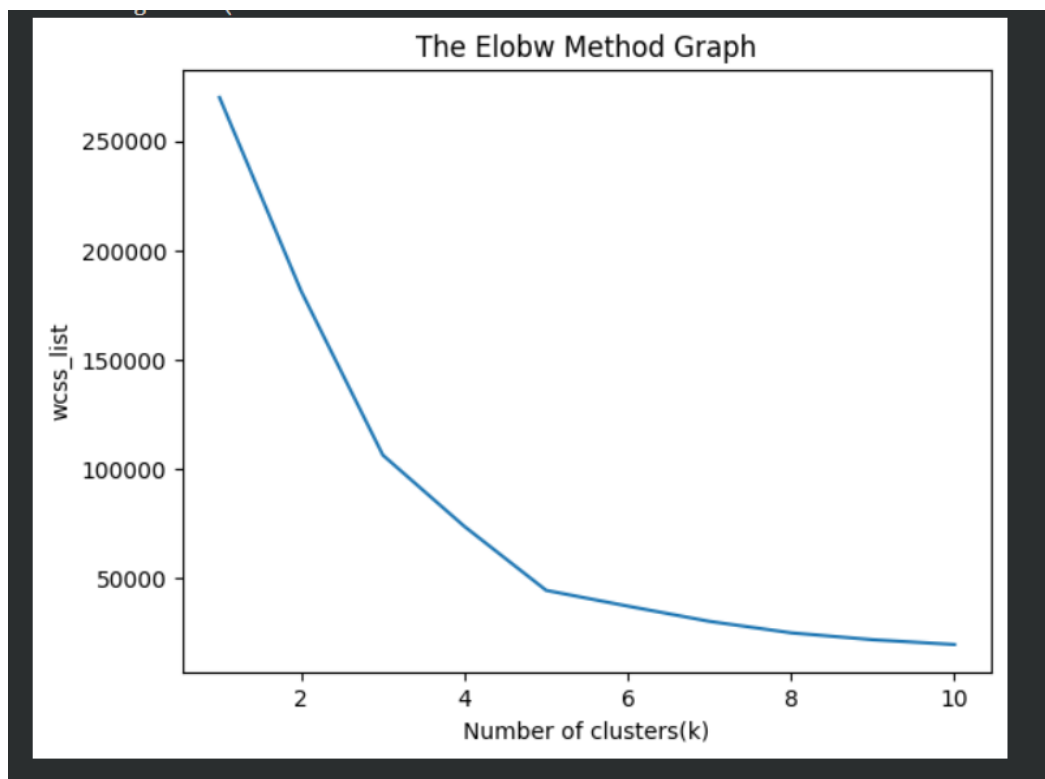
	CustomerID	Gender	Age	Annual Income (k\$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
..
195	196	Female	35	120	79
196	197	Female	45	126	28
197	198	Male	32	126	74
198	199	Male	32	137	18
199	200	Male	30	137	83

```
[200 rows x 5 columns]
```

```
x = dataset.iloc[:, [3, 4]].values
```

```
from sklearn.cluster import KMeans
wcss_list= []

#Using for loop for iterations from 1 to 10.
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state= 42)
    kmeans.fit(x)
    wcss_list.append(kmeans.inertia_)
mtp.plot(range(1, 11), wcss_list)
mtp.title('The Elbow Method Graph')
mtp.xlabel('Number of clusters(k)')
mtp.ylabel('wcss_list')
mtp.show()
```



#training the K-means model on a dataset

```
kmeans = KMeans(n_clusters=5, init='k-means++', random_state= 42)
y_predict= kmeans.fit_predict(x)
```

```
mtp.scatter(x[y_predict == 0, 0], x[y_predict == 0, 1], s = 100, c = 'blue', label =
'Cluster 1')
mtp.scatter(x[y_predict == 1, 0], x[y_predict == 1, 1], s = 100, c = 'green', label =
'Cluster 2')
mtp.scatter(x[y_predict == 2, 0], x[y_predict == 2, 1], s = 100, c = 'red', label =
'Cluster 3')
mtp.scatter(x[y_predict == 3, 0], x[y_predict == 3, 1], s = 100, c = 'cyan', label =
'Cluster 4')
mtp.scatter(x[y_predict == 4, 0], x[y_predict == 4, 1], s = 100, c = 'magenta', label =
'Cluster 5')
mtp.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1], s = 300, c =
'yellow', label = 'Centroid')
mtp.title('Clusters of customers')
mtp.xlabel('Annual Income (k$)')
mtp.ylabel('Spending Score (1-100)')
mtp.legend()
mtp.show()
```

Output :-

