21CSC305P MACHINE LEARNING

LAB MANUAL

PROGRAM 1:

AIM: To implement a program to load and view the dataset

ALGORITHM:

1. Download a dataset from Kaggle

2. Import the dataset or load in the jupyter book or in google colab

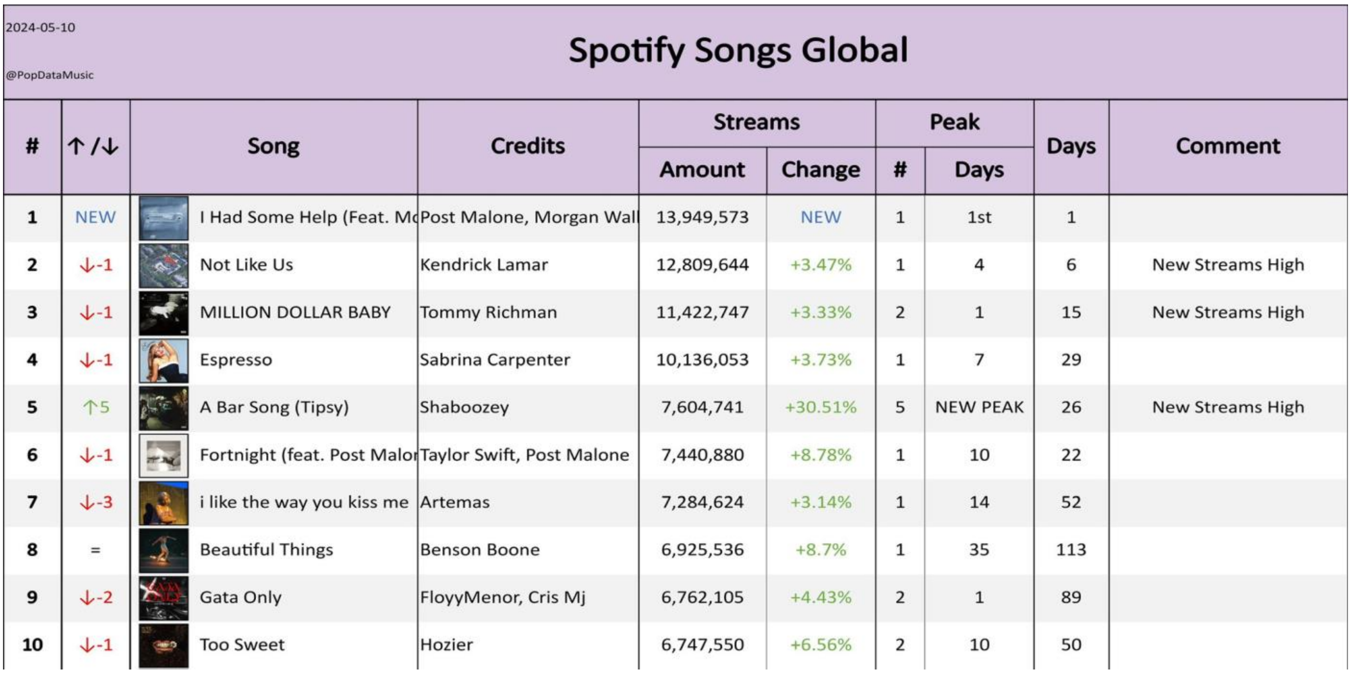
3. Create a duplicate set.

4. Display mean,median and other statistics using describe function().

PROGRAM:

```
import pandas as pd
import matplotlib.pylot as plt
af = pd.read_csv(“Most streamed spotify songs 2024.csv”, encoding=’unicode_escape’)
new_af=af.copy()
new_af=new_af[[“Artist”, “release date”, “All time rank”, “spotify playlist reach”, “youtube
views”]]
new_af.head(10)
```

OUTPUT:



2024-05-10

@PopDataMusic

Spotify Songs Global

#	↑/↓	Song	Credits	Streams		Peak		Days	Comment
				Amount	Change	#	Days		
1	NEW	I Had Some Help (Feat. M	Post Malone, Morgan Wall	13,949,573	NEW	1	1st	1	
2	↓-1	Not Like Us	Kendrick Lamar	12,809,644	+3.47%	1	4	6	New Streams High
3	↓-1	MILLION DOLLAR BABY	Tommy Richman	11,422,747	+3.33%	2	1	15	New Streams High
4	↓-1	Espresso	Sabrina Carpenter	10,136,053	+3.73%	1	7	29	
5	↑5	A Bar Song (Tipsy)	Shaboozey	7,604,741	+30.51%	5	NEW PEAK	26	New Streams High
6	↓-1	Fortnight (feat. Post Malo	Taylor Swift, Post Malone	7,440,880	+8.78%	1	10	22	
7	↓-3	i like the way you kiss me	Artemas	7,284,624	+3.14%	1	14	52	
8	=	Beautiful Things	Benson Boone	6,925,536	+8.7%	1	35	113	
9	↓-2	Gata Only	FloyyMenor, Cris Mj	6,762,105	+4.43%	2	1	89	
10	↓-1	Too Sweet	Hozier	6,747,550	+6.56%	2	10	50	

RESULT: Thus the desired output of most streamed songs has been successfully executed.

PROGRAM 2

AIM: To display the summary and statistics of the dataset.

ALGORITHM:

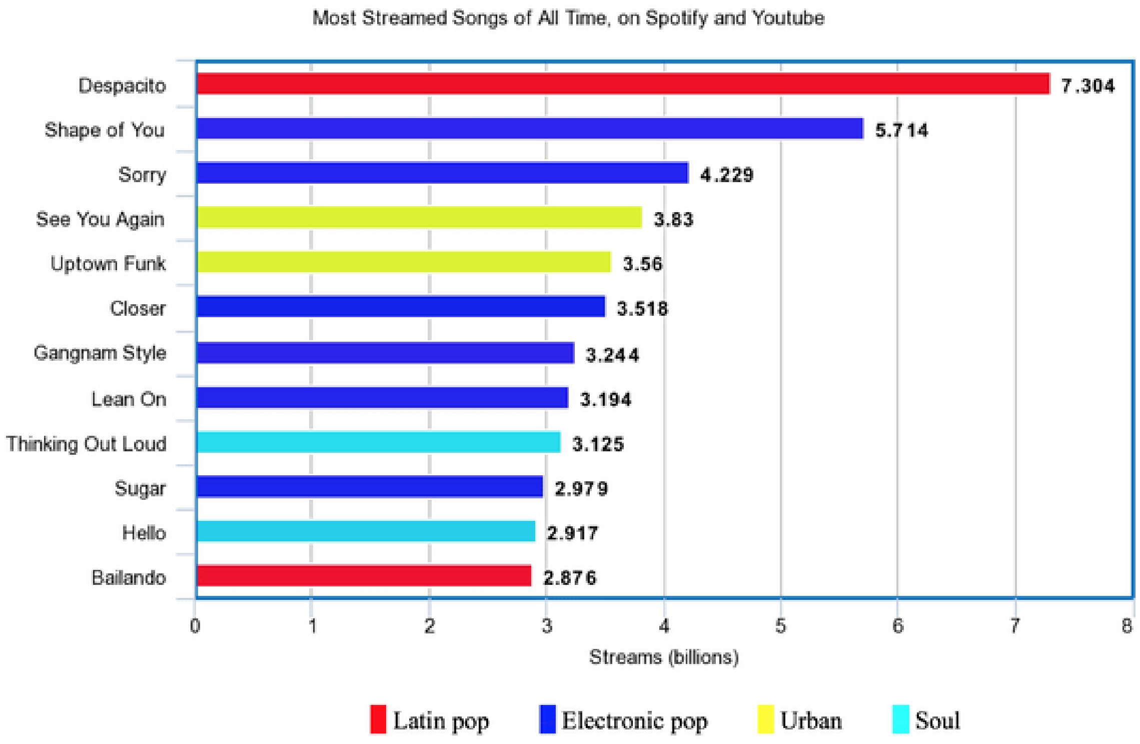
1. Download a dataset from Kaggle
2. Import the dataset or load in the jupyter book or in google colab
3. Create a duplicate set.
4. Display mean,median and other statistics using describe function().

PROGRAM:

```
median = af["spotify popularity"].median()
print("Median is ", median)

new_af.describe()
```

OUTPUT:

```
Median is 67.0
```



RESULT: Thus the program to display the summary and statistics has been successfully verified and executed.

PROGRAM 3

AIM: To implement linear regression to perform prediction.

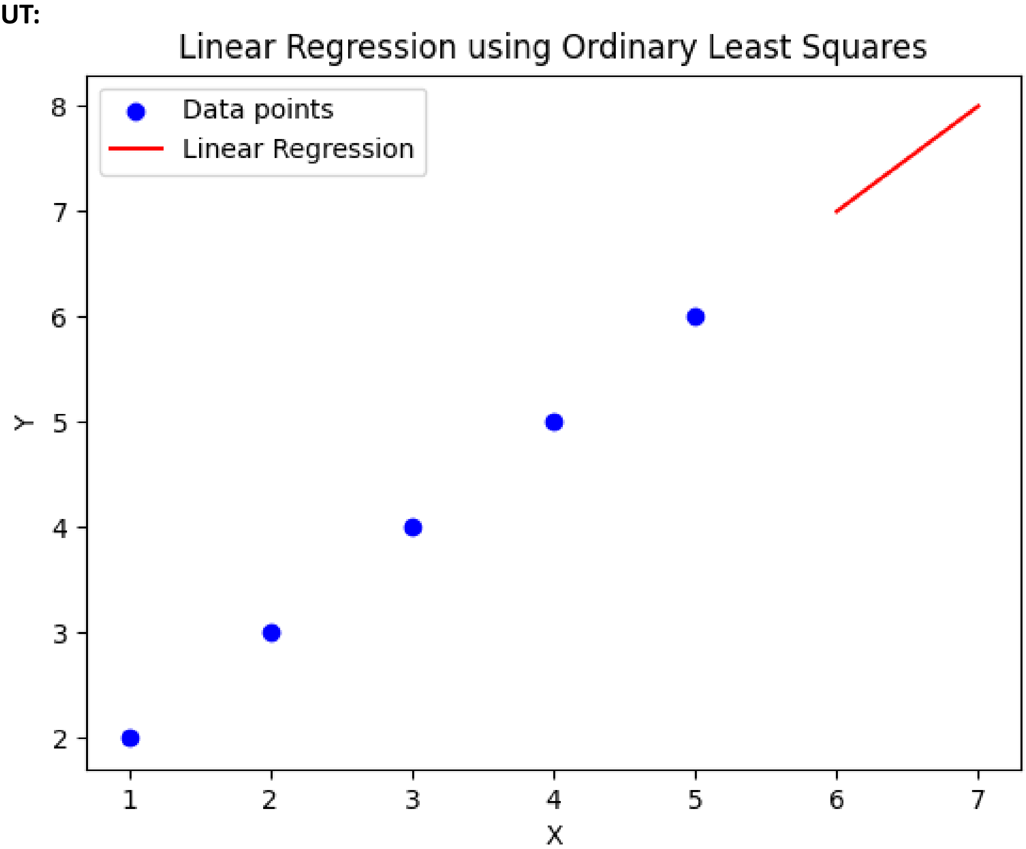
ALGORITHM:

1. Initialize Parameters**:** Start by initializing the parameters like coefficients (slope and intercept).
2. Input Data**:** Gather the dataset containing the independent variable (X) and dependent variable (Y).
3. Feature Scaling (Optional)**:** Normalize or standardize the input data if necessary to ensure better convergence.
4. Split Data**:** Divide the dataset into training and testing sets to evaluate the model.
5. Model Training**:** Implement a method to optimize the parameters (coefficients) based on the training data. This can be done using techniques like Gradient Descent, Normal Equations, or using libraries like scikit-learn.
6. Prediction**:** Use the learned parameters to predict outcomes for new data points or the test set.
7. Evaluation**:** Measure the performance of the model using evaluation metrics like Mean Squared Error (MSE), R-squared, etc.

PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt


# Sample dataset

X = np.array([1, 2, 3, 4, 5])

Y = np.array([2, 3, 4, 5, 6])


# Function to perform linear regression using Ordinary Least Squares
def linear_regression_ols(X, Y):

    # Add a column of ones to X for the intercept term
    X_b = np.c_[np.ones((len(X), 1)), X]

    # Calculate theta using the Normal Equation: theta = (X^T * X)^(-1) * X^T * Y
    theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(Y)
    return theta


# Function to make predictions
def predict(X, theta):

    # Add a column of ones to X for the intercept term
    X_b = np.c_[np.ones((len(X), 1)), X]

    # Predict Y_hat = X_b * theta
    Y_pred = X_b.dot(theta)


    return Y_pred

# Perform linear regression

theta = linear_regression_ols(X, Y)


# Make predictions
X_new = np.array([6, 7])

predictions = predict(X_new, theta)


# Plotting the original data and the linear regression line
plt.scatter(X, Y, color='blue', label='Data points')

plt.plot(X_new, predictions, color='red', label='Linear Regression')
plt.xlabel('X')

plt.ylabel('Y')
plt.legend()

plt.title('Linear Regression using Ordinary Least Squares')
plt.show()


print(f"Predictions for X_new: {predictions}")
```

OUTPUT:



RESULT: Thus the implementation of linear regression to perform prediction has been successfully executed.

PROGRAM 4.1

AIM: To implement Bayesian logistic regression for classification

ALGORITHM:

8. **Initialize Parameters**: Start with prior distributions for the model parameters (e.g., coefficients, intercept) and likelihood distributions based on the data.
9. **Input Data**: Gather the dataset containing features (X) and corresponding binary labels (Y).
10. **Posterior Calculation**: Use Bayesian inference techniques such as Markov Chain Monte Carlo (MCMC) or Variational Inference to compute the posterior distribution over the parameters given the data.
11. **Prediction**: Use the posterior distribution to predict the probability of classes for new data points.
12. **Evaluation**: Measure the performance of the model using metrics such as accuracy, precision, recall, and F1-score.

PROGRAM:

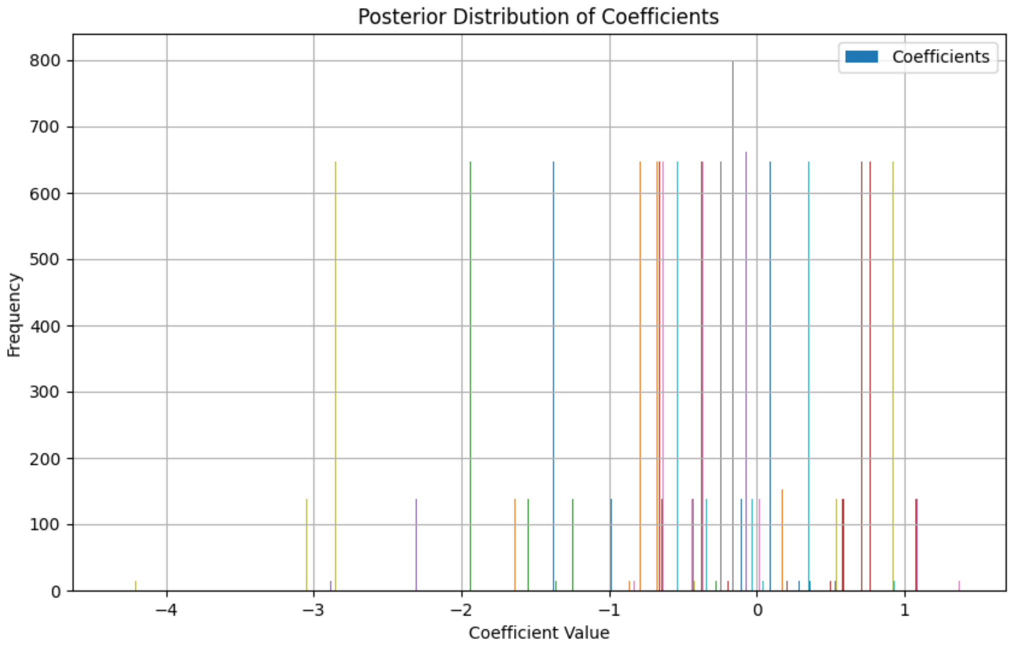
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate synthetic data
X, Y = make_classification(n_samples=1000, n_features=20, random_state=42)

# Split data into train and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

# Add intercept to X_train for bias term
X_train = np.c_[np.ones((len(X_train), 1)), X_train]
```

```
# Define sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Initialize parameters randomly
np.random.seed(42)
theta = np.random.randn(X_train.shape[1])

# Bayesian Logistic Regression with Metropolis-Hastings sampling
def bayesian_logistic_regression(X, Y, num_samples=1000, burn_in=200):
    m, n = X.shape
    trace = np.zeros((num_samples, n))  # Trace to store samples of theta
    theta_current = theta.copy()
    acceptance_count = 0

    for i in range(num_samples):
        # Generate proposal from Gaussian distribution
        proposal = theta_current + np.random.randn(n)

        # Calculate prior probabilities (assuming uninformative priors)
        prior_current = np.sum(-np.log(1 + np.exp(-(X.dot(theta_current)))))
        prior_proposal = np.sum(-np.log(1 + np.exp(-(X.dot(proposal)))))

        # Calculate likelihoods
        likelihood_current = np.sum(Y * X.dot(theta_current) - np.log(1 +
np.exp(X.dot(theta_current))))
        likelihood_proposal = np.sum(Y * X.dot(proposal) - np.log(1 + np.exp(X.dot(proposal))))

        # Calculate posterior probabilities
        posterior_current = likelihood_current + prior_current
        posterior_proposal = likelihood_proposal + prior_proposal

        # Accept or reject the proposal
        acceptance_prob = np.exp(posterior_proposal - posterior_current)
        accept = np.random.rand() < acceptance_prob

        if accept:
            theta_current = proposal
            acceptance_count += 1

        trace[i] = theta_current

    acceptance_rate = acceptance_count / num_samples
    print(f'Acceptance rate: {acceptance_rate}')
    return trace[burn_in:]

# Perform Bayesian Logistic Regression
trace = bayesian_logistic_regression(X_train, Y_train)

# Predictions for test data
X_test = np.c_[np.ones((len(X_test), 1)), X_test]
logits = X_test.dot(trace.mean(axis=0))
Y_pred = (sigmoid(logits) >= 0.5).astype(int)

# Calculate accuracy
accuracy = accuracy_score(Y_test, Y_pred)
print(f'Accuracy: {accuracy}')

# Plotting the coefficients distribution
```

```
plt.figure(figsize=(10, 6))

plt.hist(trace[:, 1:], bins=30, label='Coefficients')

plt.xlabel('Coefficient Value')

plt.ylabel('Frequency')

plt.title('Posterior Distribution of Coefficients')

plt.legend()

plt.grid(True)

plt.show()
```

OUTPUT:

```
Acceptance rate: 0.005

Accuracy: 0.69
```



RESULT: Thus the program for to implement Bayesian logistic regression for classification is been successfully executed.

PROGRAM 4.2

AIM: To implement the SVM for classification

ALGORITHM:

i. **Initialize Parameters**: Start with setting parameters such as kernel type (linear, polynomial, radial basis function (RBF)), regularization parameter (C), and kernel coefficients (gamma).
ii. **Input Data**: Gather the dataset containing features (X) and corresponding binary labels (Y).
iii. **Model Training**: Use the training data to fit the SVM model, adjusting the parameters to maximize the margin between classes.
iv. **Prediction**: Use the learned SVM model to predict classes for new data points.
v. **Evaluation**: Measure the performance of the model using metrics such as accuracy, precision, recall, and F1-score.

PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score, classification_report


# Generate synthetic data

X, Y = make_classification(n_samples=1000, n_features=20, random_state=42)


# Split data into train and test sets

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)


# Define SVM model

svm_model = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)


# Train SVM model
```

```
svm_model.fit(X_train, Y_train)

# Predictions for test data
Y_pred = svm_model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(Y_test, Y_pred)
print(f'Accuracy: {accuracy}')

# Classification report
print(classification_report(Y_test, Y_pred))

# Plotting decision boundary (for 2D data)
if X.shape[1] == 2:
    # Plot decision boundary
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=Y, cmap='viridis', s=50, alpha=0.6)

    # Create grid to evaluate model
    xlim = plt.gca().get_xlim()
    ylim = plt.gca().get_ylim()
    xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                         np.linspace(ylim[0], ylim[1], 100))
    Z = svm_model.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary and margins
    plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
                linestyles=['--', '-', '--'])
    plt.title('SVM Decision Boundary')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()
```

OUTPUT:

```
Accuracy: 0.845
              precision    recall   f1-score   support


           0       0.80       0.89       0.84          93
           1       0.90       0.80       0.85         107


    accuracy                          0.84       200
   macro avg       0.85       0.85       0.84       200
weighted avg       0.85       0.84       0.85       200
```

RESULT: The given program for SVM classification is executed and verified successfully.

PROGRAM 5.1

AIM : To implement the K-means clustering to categorize the data

ALGORITHM:

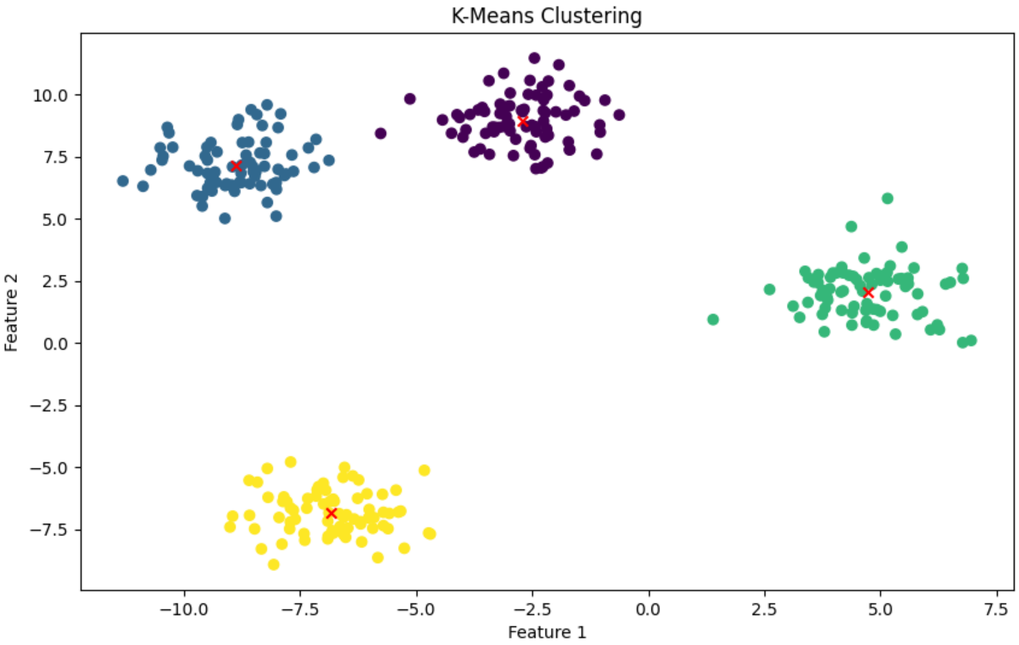
1. **Initialize**: Randomly select KKK initial centroids from the data.
2. **Assignment**: Assign each data point to the nearest centroid, forming KKK clusters.
3. **Update**: Recalculate the centroids of the clusters by taking the mean of all data points in each cluster.
4. **Repeat**: Repeat steps 2 and 3 until the centroids no longer change (convergence) or for a fixed number of iterations.

PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt


# Generate synthetic data

def generate_data(n_samples=300, n_centers=4, random_seed=42):

   np.random.seed(random_seed)

   points_per_center = n_samples // n_centers

   centers = np.random.uniform(-10, 10, (n_centers, 2))

   X = np.vstack([center + np.random.randn(points_per_center, 2) for center in centers])

   return X


# K-Means Clustering

def k_means(X, k, max_iters=100):

   centroids = X[np.random.choice(X.shape[0], k, replace=False)]

   for _ in range(max_iters):

      # Assign each point to the nearest centroid

      distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
```

```
        clusters = np.argmin(distances, axis=1)

        # Calculate new centroids
        new_centroids = np.array([X[clusters == j].mean(axis=0) for j in range(k)])

        # Check for convergence
        if np.all(centroids == new_centroids):
            break

        centroids = new_centroids

    return clusters, centroids

# Generate data and run K-Means
X = generate_data()
clusters_kmeans, centroids_kmeans = k_means(X, k=4)

# Plot K-Means results
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=clusters_kmeans, cmap='viridis', marker='o')
plt.scatter(centroids_kmeans[:, 0], centroids_kmeans[:, 1], c='red', marker='x')
plt.title('K-Means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

OUTPUT:



RESULT: Thus the program for K-means clustering is been executed successfully

PROGRAM 5.2

AIM: To implement the mixture of gaussian models to categorize the data

ALGORITHM:

1. **Initialize**: Choose initial parameters for the Gaussian components (means, covariances, and mixing coefficients).
2. **Expectation (E-step)**: Calculate the probability of each data point belonging to each Gaussian component.
3. **Maximization (M-step)**: Update the parameters of the Gaussian components using the probabilities computed in the E-step.
4. **Repeat**: Repeat the E-step and M-step until convergence.

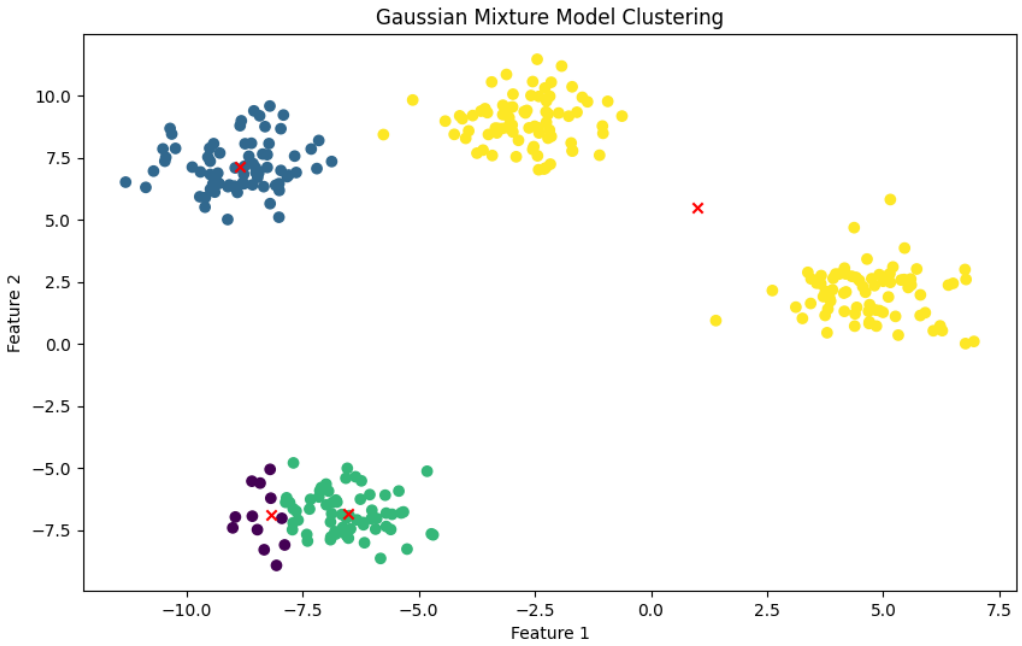
PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt
```

```
# Generate synthetic data
def generate_data(n_samples=300, n_centers=4, random_seed=42):
    np.random.seed(random_seed)
    points_per_center = n_samples // n_centers
    centers = np.random.uniform(-10, 10, (n_centers, 2))
    X = np.vstack([center + np.random.randn(points_per_center, 2) for center in centers])
    return X

# Gaussian Mixture Model (GMM)
def gmm(X, k, max_iters=100):
    n_samples, n_features = X.shape

    # Initialize parameters
    np.random.seed(42)
    weights = np.ones(k) / k
    means = X[np.random.choice(X.shape[0], k, replace=False)]
    covariances = np.array([np.eye(n_features) for _ in range(k)])

    def gaussian(x, mean, cov):
        n = x.shape[0]
        diff = x - mean
        return (np.exp(-0.5 * np.dot(diff.T, np.linalg.solve(cov, diff))) /
                np.sqrt((2 * np.pi) ** n * np.linalg.det(cov)))

    def e_step(X, weights, means, covariances):
        responsibilities = np.zeros((n_samples, k))
        for i in range(n_samples):
            for j in range(k):
                responsibilities[i, j] = weights[j] * gaussian(X[i], means[j], covariances[j])
            responsibilities[i] /= np.sum(responsibilities[i])
        return responsibilities

    def m_step(X, responsibilities):
        weights = np.mean(responsibilities, axis=0)
        means = np.dot(responsibilities.T, X) / np.sum(responsibilities, axis=0)[:, np.newaxis]
        covariances = []
        for j in range(k):
            diff = X - means[j]
            cov = np.dot(responsibilities[:, j] * diff.T, diff) / np.sum(responsibilities[:, j])
            covariances.append(cov)
        return weights, means, np.array(covariances)

    for _ in range(max_iters):
        responsibilities = e_step(X, weights, means, covariances)
        weights, means, covariances = m_step(X, responsibilities)

    clusters = np.argmax(responsibilities, axis=1)
    return clusters, means

# Generate data and run GMM
```

```
X = generate_data()
clusters_gmm, means_gmm = gmm(X, k=4)

# Plot GMM results
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=clusters_gmm, cmap='viridis', marker='o')
plt.scatter(means_gmm[:, 0], means_gmm[:, 1], c='red', marker='x')
plt.title('Gaussian Mixture Model Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

OUTPUT:



RESULT:

Thus the program is been executed successfully.

PROGRAM 5.3

AIM: To implement the hierarchial clustering to categorize the data

ALGORITHM:

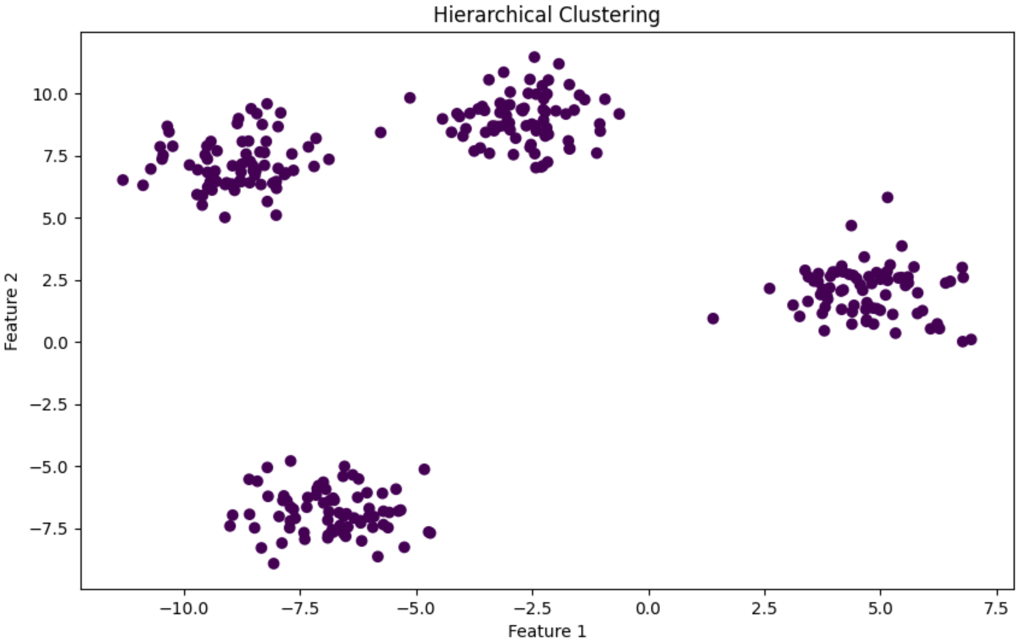
1. **Start**: Treat each data point as a singleton cluster.
2. **Merge**: Find the pair of clusters that are closest and merge them into a single cluster.

3. **Repeat**: Repeat step 2 until only a single cluster remains or a stopping criterion is met (e.g., a desired number of clusters).
4. **Cut**: Cut the dendrogram at the desired level to extract clusters.

PROGRAM:

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


# Generate synthetic data
def generate_data(n_samples=300, n_centers=4, random_seed=42):
    np.random.seed(random_seed)
    points_per_center = n_samples // n_centers
    centers = np.random.uniform(-10, 10, (n_centers, 2))
    X = np.vstack([center + np.random.randn(points_per_center, 2) for center in centers])
    return X


# Calculate Euclidean distance
def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))


# Compute the distance matrix
def compute_distance_matrix(X):
    n_samples = X.shape[0]
    distances = np.zeros((n_samples, n_samples))
    for i in range(n_samples):
        for j in range(i + 1, n_samples):
            distances[i, j] = euclidean_distance(X[i], X[j])

            distances[j, i] = distances[i, j]

    return distances


# Hierarchical Clustering

def hierarchical_clustering(X):

    distances = compute_distance_matrix(X)

    n_samples = len(X)


    # Initialize clusters

    clusters = [[i] for i in range(n_samples)]


    while len(clusters) > 1:

        # Find the two closest clusters

        min_dist = float('inf')

        to_merge = (None, None)


        for i in range(len(clusters)):

            for j in range(i + 1, len(clusters)):

                d = np.min([distances[p][q] for p in clusters[i] for q in clusters[j]])

                if d < min_dist:

                    min_dist = d

                    to_merge = (i, j)


        # Merge the two clusters
        i, j = to_merge
        clusters[i].extend(clusters[j])
        del clusters[j]

    return clusters

# Function to extract cluster labels
def extract_clusters(clusters, n_samples):
    labels = np.zeros(n_samples)
    for cluster_id, cluster in enumerate(clusters):
        for index in cluster:
            labels[index] = cluster_id
    return labels

# Generate data and perform hierarchical clustering
X = generate_data()
final_clusters = hierarchical_clustering(X)

# Extract final cluster labels
cluster_labels = extract_clusters(final_clusters, len(X))

# Plot Hierarchical Clustering results
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=cluster_labels, cmap='viridis', marker='o')
plt.title('Hierarchical Clustering')
plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.show()
```

OUTPUT:



RESULT: Thus the desired program have been successfully executed .

PROGRAM 6

AIM: To create a program to perform PCA

ALGORITHM:

1. **Standardize the Data:** Center the data by subtracting the mean of each feature from the data. Optionally, scale each feature to unit variance.
2. **Compute the Covariance Matrix:** Calculate the covariance matrix of the centered data.
3. **Calculate Eigenvalues and Eigenvectors:** Find the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors determine the directions of the new feature space, and the eigenvalues determine their magnitude.
4. **Sort Eigenvalues and Eigenvectors:** Sort the eigenvectors by decreasing eigenvalues and select the top k eigenvectors to form a matrix (principal components).
5. **Transform the Data:** Project the original data onto the new feature space using the matrix of principal components.

PROGRAM:

```
import numpy as np


def pca(X, n_components):
    # Step 1: Center the Data
    X_centered = X - np.mean(X, axis=0)


    # Step 2: Compute the Covariance Matrix
    cov_matrix = np.cov(X_centered, rowvar=False)


    # Step 3: Calculate Eigenvalues and Eigenvectors
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)


    # Step 4: Sort Eigenvalues and Eigenvectors
    sorted_index = np.argsort(eigenvalues)[::-1]
    sorted_eigenvectors = eigenvectors[:, sorted_index]
    sorted_eigenvalues = eigenvalues[sorted_index]

    # Step 5: Select Top n_components Eigenvectors
    eigenvector_subset = sorted_eigenvectors[:, :n_components]

    # Step 6: Transform the Data
    X_reduced = np.dot(X_centered, eigenvector_subset)

    return X_reduced, sorted_eigenvalues, eigenvector_subset

# Example usage
if __name__ == "__main__":
    # Example data
    X = np.array([[2.5, 2.4],
            [0.5, 0.7],
            [2.2, 2.9],
            [1.9, 2.2],
            [3.1, 3.0],
            [2.3, 2.7],
            [2, 1.6],
            [1, 1.1],
            [1.5, 1.6],
            [1.1, 0.9]])
    # Perform PCA
    X_reduced, eigenvalues, eigenvectors = pca(X, n_components=2)

    print("Reduced Data:\n", X_reduced)
    print("Eigenvalues:\n", eigenvalues)
    print("Eigenvectors:\n", eigenvectors)
```

OUTPUT:

```
Reduced Data:
 [[ 0.82797019 -0.17511531]
 [-1.77758033  0.14285723]
 [ 0.99219749  0.38437499]
 [ 0.27421042  0.13041721]
 [ 1.67580142 -0.20949846]
 [ 0.9129491   0.17528244]
 [-0.09910944 -0.3498247 ]
 [-1.14457216  0.04641726]
 [-0.43804614  0.01776463]
 [-1.22382056 -0.16267529]]
Eigenvalues:
 [1.28402771 0.0490834 ]
Eigenvectors:
 [[ 0.6778734  -0.73517866]
 [ 0.73517866  0.6778734 ]]
```

RESULT: Thus the given program has been successfully executed

PROGRAM 7

AIM : To implement HMM to predict the sequential data.

ALGORITHM :

```
import numpy as np


class SimpleHMM:

    def __init__(self, states, observations, start_prob, trans_prob, emit_prob):

        self.states = states

        self.observations = observations

        self.start_prob = start_prob

        self.trans_prob = trans_prob

        self.emit_prob = emit_prob


    def viterbi(self, obs_sequence):

        n_states = len(self.states)

        n_obs = len(obs_sequence)


        # Initialize the dynamic programming tables

        viterbi_table = np.zeros((n_states, n_obs))

        backpointer_table = np.zeros((n_states, n_obs), dtype=int)


        # Initialization step

        first_obs = obs_sequence[0]

        for s in range(n_states):

            viterbi_table[s, 0] = self.start_prob[s] * self.emit_prob[s, first_obs]
            backpointer_table[s, 0] = 0

        # Recursion step
        for t in range(1, n_obs):
            for s in range(n_states):
                probabilities = viterbi_table[:, t-1] * self.trans_prob[:, s] * self.emit_prob[s, obs_sequence[t]]
                viterbi_table[s, t] = np.max(probabilities)
                backpointer_table[s, t] = np.argmax(probabilities)

        # Termination step
        best_path_prob = np.max(viterbi_table[:, n_obs-1])
        best_last_state = np.argmax(viterbi_table[:, n_obs-1])

        # Path backtracking
        best_path = np.zeros(n_obs, dtype=int)
        best_path[-1] = best_last_state
        for t in range(n_obs-2, -1, -1):
            best_path[t] = backpointer_table[best_path[t+1], t+1]

        return best_path, best_path_prob

# Example usage
if __name__ == "__main__":
    # Define the states, observations, and model parameters
    states = ['Rainy', 'Sunny']
    observations = ['Walk', 'Shop', 'Clean']

    start_probability = np.array([0.6, 0.4])

    transition_probability = np.array([
        [0.7, 0.3],  # From Rainy to Rainy/Sunny
        [0.4, 0.6],  # From Sunny to Rainy/Sunny
    ])

    emission_probability = np.array([
        [0.1, 0.4, 0.5],  # Probabilities of Walk/Shop/Clean from Rainy
        [0.6, 0.3, 0.1],  # Probabilities of Walk/Shop/Clean from Sunny
    ])

    # Create the HMM model
    hmm = SimpleHMM(states, observations, start_probability, transition_probability,
emission_probability)

    # Encode the observation sequence as integers
    obs_map = {obs: i for i, obs in enumerate(observations)}
    obs_sequence = np.array([obs_map['Walk'], obs_map['Shop'], obs_map['Clean'],
obs_map['Walk']])

    # Predict the most likely sequence of states
    state_sequence, probability = hmm.viterbi(obs_sequence)


    # Decode state sequence into state names

    state_names = [states[state] for state in state_sequence]


    print("Most likely states sequence:", state_names)

    print("Probability of the best path:", probability)
```

OUTPUT:

```
Most likely states sequence: ['Sunny', 'Rainy', 'Rainy', 'Sunny']

Probability of the best path: 0.0024192
```

RESULT: Thus the program to implement HMM has been successfully executed .

PROGRAM 8

AIM: To implement the CART learning algorithms to perform categorization.

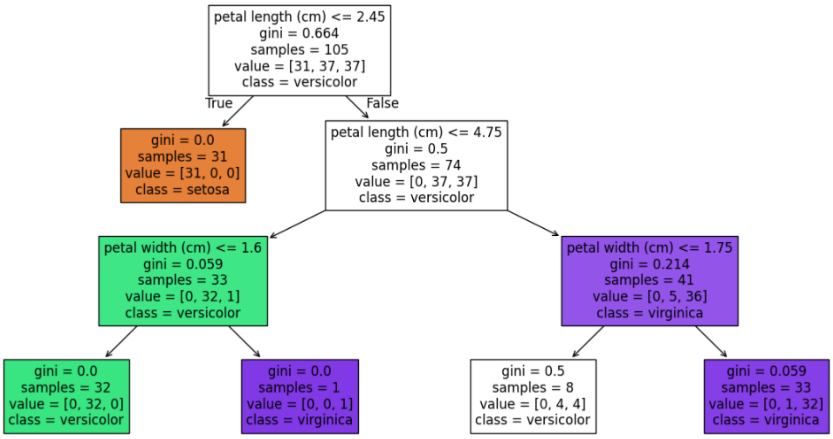
ALGORITHM:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
import matplotlib.pyplot as plt

# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create a Decision Tree classifier
clf = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)

# Train the classifier
clf.fit(X_train, y_train)
```

```
# Make predictions on the test set

y_pred = clf.predict(X_test)


# Evaluate the model

accuracy = clf.score(X_test, y_test)

print(f'Accuracy: {accuracy:.2f}')


# Visualize the Decision Tree

plt.figure(figsize=(12, 8))

tree.plot_tree(clf, feature_names=data.feature_names, class_names=data.target_names, filled=True)

plt.show()
```

OUTPUT:

```
Accuracy:1.0
```



RESULT: Thus the given program for the implementation of CART learning algorithms to perform categorization has been successfully executed.

PROGRAM 9

AIM: To implement Ensemble learning models to perfom classification

ALGORITHM:

Bagging (e.g., Random Forest)

1. **Create multiple subsets of the training data** by sampling with replacement (bootstrap sampling).
2. **Train a base model** (e.g., decision tree) on each subset independently.
3. **Aggregate the predictions** of all models by taking a majority vote (for classification) or averaging (for regression).

Boosting (e.g., AdaBoost)

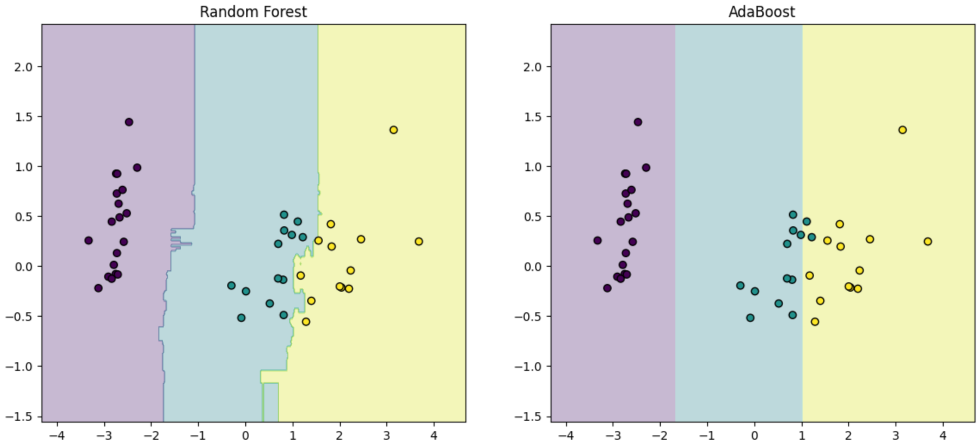
1. **Initialize weights** for all training examples.
2. **Train a base model** on the training data, weighted according to their current weights.
3. **Evaluate the model** and increase the weights of misclassified examples.
4. **Train subsequent models** iteratively, focusing more on difficult examples.
5. **Combine the models** by giving more weight to better-performing models.

PROGRAM:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier

from sklearn.decomposition import PCA
```

```
# Load Iris dataset and split into train and test sets
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Reduce dimensionality for visualization purposes
pca = PCA(n_components=2)
X_train_2d = pca.fit_transform(X_train)
X_test_2d = pca.transform(X_test)

# Train Random Forest classifier
rf_clf = RandomForestClassifier(random_state=42)
rf_clf.fit(X_train_2d, y_train)

# Train AdaBoost classifier
ada_clf = AdaBoostClassifier(random_state=42)
ada_clf.fit(X_train_2d, y_train)

# Function to plot decision boundaries
def plot_decision_boundaries(clf, X, y, ax, title):
    h = .02  # Step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3)

    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k', marker='o')

    ax.set_title(title)


# Create plots

fig, axs = plt.subplots(1, 2, figsize=(14, 6))


# Plot Random Forest decision boundaries

plot_decision_boundaries(rf_clf, X_test_2d, y_test, axs[0], 'Random Forest')


# Plot AdaBoost decision boundaries

plot_decision_boundaries(ada_clf, X_test_2d, y_test, axs[1], 'AdaBoost')


plt.show()
```

OUTPUTS:



RESULT: Thus the program for implementation to Ensemble learning models to perfom classification is been successfully executed and verified.