Program No: 01

Program Name: Write a Python program to import and export data using the Pandas library functions.

Objective: The primary goal of this experiment is to showcase how to import and export data in multiple file formats, such as CSV, Excel, and JSON, using Python's Pandas library. By leveraging Pandas' built-in functions, this experiment aims to familiarize users with fundamental data-handling tasks commonly employed in machine learning.

Theory:

Pandas is a versatile Python library widely used for data manipulation, analysis, cleaning, and transformation. It supports working with various data formats, including CSV files, Excel spreadsheets, JSON data, and SQL databases. At the core of Pandas lies the **DataFrame**, a highly efficient and flexible data structure that allows users to import data from diverse sources, perform operations like filtering and aggregation, and export data to different formats.

Key Functions in Pandas:

Importing Data:

- `pd.read_csv()`: Reads data from a CSV file.
- `pd.read_excel()`: Reads data from an Excel file.
- `pd.read_json()`: Reads data from a JSON file.

Exporting Data:

- `df.to_csv()`: Writes a DataFrame to a CSV file.
- `df.to_excel()`: Writes a DataFrame to an Excel file.
- `df.to_json()`: Writes a DataFrame to a JSON file.

These functions streamline the process of transitioning between data formats, making Pandas an indispensable tool for data preprocessing and analysis.

Algorithm:

1. **Import the Pandas Library**
2. **Specify File Paths**
3. **Load Data from CSV and Excel Files**
4. **Perform Data Manipulation**
5. **Save Data to CSV Format**
6. **Display or Save the Results**

Program code:

```
# Import the pandas library
import pandas as pd
data = {
   'Name': ['John', 'Sara', 'Tom', 'Anna'],
   'Age': [28, 24, 35, 22],
   'Country': ['USA', 'UK', 'Canada', 'Australia']
}
df = pd.DataFrame(data)
df.to_csv('sample_data.csv')   # Export to CSV
df.to_excel('sample_data.xlsx')  # Export to Excel
df.to_json('sample_data.json')  # Export to JSON
csv_df = pd.read_csv('sample_data.csv')
excel_df = pd.read_excel('sample_data.xlsx')
json_df = pd.read_json('sample_data.json')
print("Created Dataframe: \n",pd.DataFrame(data))
print("CSV Data:\n", csv_df)
print("Excel Data:\n", excel_df)
print("JSON Data:\n", json_df)
```

Input & Output:

```
Created Dataframe:
    Name  Age    Country
0  John   28        USA
1  Sara   24         UK
2   Tom   35     Canada
3  Anna   22  Australia
CSV Data:
    Unnamed: 0  Name  Age    Country
0           0  John   28        USA
1           1  Sara   24         UK
2           2   Tom   35     Canada
3           3  Anna   22  Australia
Excel Data:
    Unnamed: 0  Name  Age    Country
0           0  John   28        USA
1           1  Sara   24         UK
2           2   Tom   35     Canada
3           3  Anna   22  Australia
JSON Data:
    Name  Age    Country
0  John   28        USA
1  Sara   24         UK
2   Tom   35     Canada
3  Anna   22  Australia
```

Discussion: This experiment showcased how Pandas simplifies handling data across multiple formats. With import functions such as `pd.read_csv()`, `pd.read_excel()`, and `pd.read_json()`, we can easily load data from various file types. Similarly, export functions like `df.to_csv()`, `df.to_excel()`, and `df.to_json()` allow us to save data efficiently for future use or sharing.

Program No: 02

Program Name: Write a Python program for pre-processing data using various techniques for a given dataset

Objective: The goal of this experiment is to preprocess the given dataset by handling missing values, removing duplicates, and transforming categorical data into numerical format using Label Encoding. The preprocessed dataset will then be ready for use in machine learning models or further analysis.

Theory:

Data preprocessing is a crucial step in any data analysis or machine learning workflow. Raw data often contains issues such as missing values, categorical (non-numerical) features, and duplicate entries, all of which can compromise model performance if not properly addressed.

Key Steps in Preprocessing:

1. **Handling Missing Values:** Incomplete data can produce inaccurate results. Common approaches include filling missing values with substitutes (e.g., zeros, means, or forward/backward filling) or removing rows/columns with missing entries.
2. **Removing Duplicates:** Duplicate records can skew results or introduce errors, making it essential to identify and eliminate them.
3. **Encoding Categorical Variables:** Since machine learning models require numerical inputs, categorical variables (e.g., "Gender" or "Race/Ethnicity") must be converted into numerical form. A common technique is **Label Encoding**, which assigns a unique integer to each category. This method is particularly effective for ordinal or simple categorical data.

By performing these preprocessing steps, the dataset becomes clean, consistent, and ready for use in machine learning models or further statistical analysis.

Algorithm:

Step-1: Load data in Pandas.
Step-2: Drop columns that aren't useful.
Step-3: Drop rows with missing values.
Step-4: Create dummy variables.
Step-5: Take care of missing data.
Step-6: Convert the data frame to NumPy.
Step-7: Divide the data set into training data and test data.
Step-8: Feature Scaling.

Program code:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```
from sklearn.compose import ColumnTransformer
# Step 1: Load the dataset from CSV
df = pd.read_csv('cancer issue.csv')
df.head()
df.fillna(0, inplace=True)
df.drop_duplicates(inplace=True)
categorical_columns = ['Gender', 'Race/Ethnicity', 'SmokingStatus', 'FamilyHistory', 'CancerType', 'Stage',
'TreatmentType', 'TreatmentResponse', 'Recurrence', 'GeneticMarker', 'HospitalRegion']
label_encoder = LabelEncoder()
for column in categorical_columns:
  df[column] = label_encoder.fit_transform(df[column])
 numerical_columns = ['Age', 'TumorSize', 'SurvivalMonths']
```

Input & Output:

Dataset:

	PatientID	Age	Gender	Race/ Ethnicity	BMI	SmokingStatus	FamilyHistory	CancerType	Stage	TumorSize	TreatmentType	TreatmentResponse	SurvivalMonths	I
0	1	80	Female	Other	23.3	Smoker	Yes	Breast	II	1.7	Combination Therapy	No Response	103	
1	2	76	Male	Caucasian	22.4	Former Smoker	Yes	Colon	IV	4.7	Surgery	No Response	14	
2	3	69	Male	Asian	21.5	Smoker	Yes	Breast	III	8.3	Combination Therapy	Complete Remission	61	
3	4	77	Male	Asian	30.4	Former Smoker	Yes	Prostate	II	1.7	Radiation	Partial Remission	64	
4	5	89	Male	Caucasian	20.9	Smoker	Yes	Lung	IV	7.4	Radiation	No Response	82	

After Preprocessing:

```
   PatientID  Age  Gender  Race/Ethnicity   BMI  SmokingStatus  FamilyHistory  \
0          1   80       0               4  23.3              2              1
1          2   76       1               2  22.4              0              1
2          3   69       1               1  21.5              2              1
3          4   77       1               1  30.4              0              1
4          5   89       1               2  20.9              2              1

   CancerType  Stage  TumorSize  TreatmentType  TreatmentResponse  \
0           0      1        1.7              1                  1
1           1      3        4.7              3                  1
2           0      2        8.3              1                  0
3           4      1        1.7              2                  2
4           3      3        7.4              2                  1

   SurvivalMonths  Recurrence  GeneticMarker  HospitalRegion
0             103           1              0               2
1              14           1              1               3
2              61           1              1               3
3              64           0              3               2
4              82           1              3               2
```

Discussion: This experiment involved cleaning and preparing the dataset by filling missing values, removing duplicates, and converting categorical variables to numerical values using Label Encoding. These steps ensured the data was consistent, clean, and ready for machine learning or further analysis.

Program No: 03

Program Name: Write a Python program to extract features from a color picture to make it usable to learn machines

Objective: The goal of this experiment is to extract features from a color image for machine learning using OpenCV. This involves reading the image, processing it, and applying various transformations to enhance its features, making it ready for analysis and learning.

Theory: In image processing, feature extraction plays a crucial role in reducing the complexity of data while simultaneously improving the efficiency of machine learning models. Digital images are often represented in different color spaces, with RGB (Red, Green, Blue) being one of the most common, where each pixel consists of three color components. Alternatively, grayscale images represent only intensity information, where each pixel has a single value indicating brightness. These color spaces serve as the foundation for many image processing techniques.

OpenCV, a powerful library in Python, provides a wide range of tools to facilitate various image processing tasks. It simplifies operations such as converting between different color spaces, applying filters for blurring, and detecting key features like edges and corners. These tasks are essential for preparing images for machine learning models, as they help in extracting relevant features while removing unnecessary details.

This experiment specifically focuses on preprocessing an image for machine learning. The steps include resizing the image to a uniform dimension, converting it between color spaces (e.g., from RGB to grayscale), and applying transformations such as blurring and edge detection to extract significant features. These processes ensure that the image is optimized and ready for analysis, allowing machine learning models to work efficiently with the relevant features extracted from the image.

Algorithm:

Step 1: Specify the path to the color image file.
Step 2: Use OpenCV (cv2.imread()) to read the image from the specified path.
Step 3: Resize the image to 200x200 pixels using cv2.resize().
Step 4: Convert the resized image from BGR (Blue, Green, Red) to RGB (Red, Green, Blue) using cv2.cvtColor().
Step 5: Convert the image from BGR to grayscale using cv2.cvtColor().
Step 6: Apply Gaussian blur to the image using cv2.GaussianBlur() to reduce noise and detail.
Step 7: Perform edge detection on the grayscale image using the Canny edge detection algorithm (cv2.Canny()).
Step 8: Perform corner detection on the grayscale image using the Shi-Tomasi method (cv2.cornerHarris()).
Step 9: Flatten the RGB and grayscale images to create feature vectors.
Step 10: Normalize the feature vectors by dividing by the sum of pixel values to ensure they are on a comparable scale.

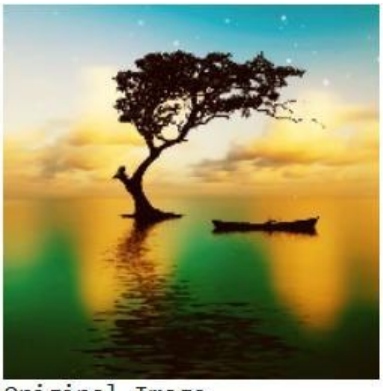
Step 11: Display the images with titles: Original, RGB, Grayscale, Blurred, Edge-detected, and Corner-detected.

Program Code:

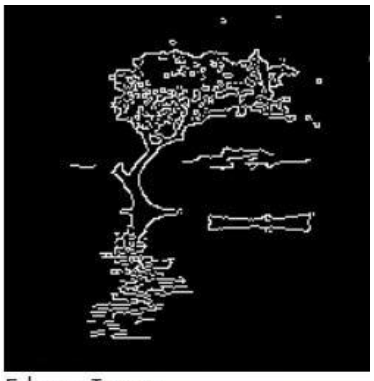
```
import cv2
import numpy as np
image_path = 'image.jpg'  # Replace with your image file path
image_bgr = cv2.imread(image_path)
image_rgb = cv2.cvtColor(image_resized, cv2.COLOR_BGR2RGB)
image_gray = cv2.cvtColor(image_resized, cv2.COLOR_BGR2GRAY)
image_blur = cv2.GaussianBlur(image_resized, (5, 5), 0)
image_edges = cv2.Canny(image_gray, 100, 200)
features_rgb = image_rgb.flatten()
features_gray = image_gray.flatten()
features_rgb_normalized = features_rgb / np.sum(features_rgb)
features_gray_normalized = features_gray / np.sum(features_gray)
features_combined = np.concatenate([features_rgb_normalized, features_gray_normalized])
# Print the normalized feature vector
print("Normalized Feature Vector (RGB + Grayscale):")
print(features_combined)
# Step 10: Display the images
cv2.imshow('Original Image', image_resized)
cv2.imshow('RGB Image', image_rgb)
cv2.imshow('Grayscale Image', image_gray)
cv2.imshow('Blurred Image', image_blur)
cv2.imshow('Edge Image', image_edges)
cv2.imshow('Corner Detected Image', image_corners)
```

Input & Output:

```
Normalized Feature Vector (RGB + Grayscale):
[1.72717413e-06 3.88614180e-06 7.77228360e-06 ... 7.16020126e-06
 9.98088660e-06 1.36694751e-05]
Original Color Image:
```

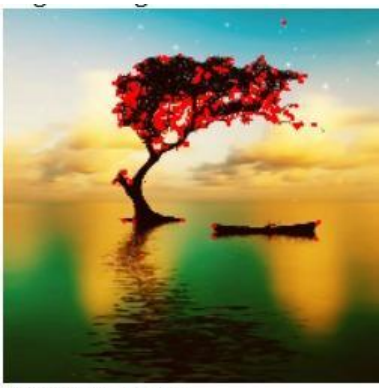

Original Image


Grayscale Image


Edges Image

RGB Image

Blurred Image



Corners Image

Discussion: This experiment leverages the power of OpenCV to preprocess images, specifically focusing on extracting and normalizing features to enhance machine learning applications. OpenCV is a widely used library in computer vision and image processing, offering an extensive set of tools to manipulate and analyze images. Preprocessing images is a vital step in preparing data for machine learning models, as raw image data often contains unnecessary information or noise that can hinder model performance.

Program No: 04

Program Name: Write a Python program to implement the K-Nearest Neighbour (KNN) algorithm

Objective: This program implements the k-Nearest Neighbors (k-NN) algorithm, a fundamental and widely used technique in machine learning for classification tasks. The k-NN algorithm works by classifying a data point based on the majority class of its 'k' closest neighbors in the feature space. These neighbors are identified by measuring the distance between the data points, typically using metrics such as Euclidean distance. The program not only performs classification but also evaluates the algorithm's performance by comparing the predicted labels of new data points with the actual target classes from the dataset.

The performance evaluation is an essential step in understanding how well the model is generalizing to unseen data. It involves calculating metrics such as accuracy, precision, recall, and F1-score, which provide insight into the algorithm's effectiveness and its ability to correctly classify instances. By assessing these metrics, the program ensures that the k-NN algorithm is performing optimally for the given dataset.

Furthermore, the program predicts the class of new, unseen data points by utilizing the trained k-NN model. The prediction process involves determining the closest neighbors of the new data points and assigning the class label based on the majority vote of those neighbors. This method makes k-NN a simple yet powerful algorithm for classification problems, particularly when the relationships between data points are inherently non-linear.

Theory: The k-Nearest Neighbors (k-NN) algorithm is a straightforward yet powerful instance-based supervised learning method, commonly used for both classification and regression tasks. As a non-parametric algorithm, it makes predictions based directly on the data instances in the feature space rather than through explicit model parameters. The core idea behind k-NN is simple: it classifies or predicts the outcome for a given data point by examining the class or value of its **k** nearest neighbors in the feature space.

In classification problems, the k-NN algorithm assigns a class label to a data point based on the majority class of its **k** closest neighbors. These neighbors are identified by measuring the similarity (or distance) between the target point and other data points in the training set, typically using distance metrics like **Euclidean distance**, **Manhattan distance**, or more complex measures depending on the problem's nature. For regression tasks, the prediction is made by averaging the values of the **k** nearest neighbors, providing a smooth approximation of the target value.

The performance of the k-NN algorithm is highly sensitive to two important factors: the value of **k** (the number of nearest neighbors) and the choice of distance metric. The value of **k** plays a crucial role in determining how the model generalizes to unseen data:

1. **Small k** values (e.g., 1 or 3) can make the algorithm highly sensitive to noise or outliers in the data, since the decision for classifying or predicting a new point is heavily influenced by a small number of neighbors. This can lead to overfitting, where the model fits the noise in the data rather

than the underlying patterns.

2. **Large k** values result in a smoother decision boundary because the class or value assigned to a new data point is based on a larger neighborhood of points. However, using a large **k** can also cause the model to oversimplify the decision-making process, making it less responsive to subtle patterns in the data, potentially leading to underfitting.

Additionally, the choice of distance metric can affect how similarity between data points is measured. Euclidean distance works well when the features are continuous and of similar scale, while other metrics may be better suited for categorical or mixed-type data.

In summary, k-NN is a flexible and intuitive algorithm that performs well in situations where the decision boundary is non-linear and when relationships between data points are complex. However, careful consideration must be given to the appropriate selection of **k** and the distance metric to ensure the model is both accurate and generalizes well to new, unseen data.

Algorithm:

Step 1: Select the number K of the neighbors
Step 2: Calculate the Euclidean distance of K number of neighbors
Step 3: Take the K nearest neighbors as per the calculated Euclidean distance.
Step 4: Among these k neighbors, count the number of the data points in each category.
Step 5: Assign the new data points to that category for which the number of the neighbor is maximum.
Step 6: Our model is ready

Program Code:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np
x = [4, 5, 9, 7, 3, 11, 14, 8, 10, 13, 13, 21]
y = [21, 19, 22, 15, 16, 25, 23, 22, 21, 22, 32, 35]
target_classes = [0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1]
data = list(zip(x, y))
X_train, X_test, y_train, y_test = train_test_split(data, target_classes, test_size=0.2, random_state=42)
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
accuracy = np.mean(knn.predict(X_test) == y_test)
print(f"Accuracy: {accuracy:.2f}")
new_point = [(8, 21)]
prediction = knn.predict(new_point)
print(f"Prediction for new point (8, 21): {prediction[0]}")
```

Input & Output:

```
Accuracy: 1.00
Prediction for new point (8, 21): 0
```

Discussion:

k-NN is simple and intuitive but computationally expensive for large datasets. The choice of **k** affects model performance: a small **k** can overfit, while a large **k** can underfit. The algorithm is useful for smaller datasets with clear decision boundaries but becomes slower as the data grows.

Program No: 05

Program Name: Write a Python program to implement the K-Means Clustering Algorithm and test it using the appropriate dataset

Objective:

The goal of this experiment is to implement a **K-Means clustering** model with 3 clusters and evaluate its performance on a given dataset. **K-Means clustering** is an unsupervised learning algorithm used to partition a dataset into distinct groups, or clusters, based on the similarity of data points. Each data point is assigned to the cluster whose centroid (the mean of all data points in that cluster) is closest. The algorithm iterates through several steps to refine these clusters, minimizing the variance within each cluster.

In this experiment, the dataset will be segmented into three clusters. The number of clusters, **k = 3**, is predefined, and the K-Means algorithm will work to determine the best possible segmentation of the data by optimizing the position of the cluster centroids. The process involves randomly initializing the centroids, assigning data points to the nearest centroid, updating the centroids by calculating the mean of the points in each cluster, and repeating this process until convergence is reached, where the centroids no longer change significantly.

Throughout the experiment, the following steps will be carried out:

1. **Data Preprocessing**: The dataset will be preprocessed, which may involve handling missing values, normalizing or scaling the data, and ensuring that the features are appropriate for clustering.

2. **Cluster Initialization**: The initial centroids of the 3 clusters will be chosen randomly or using a heuristic method like the K-Means++ algorithm, which helps in selecting better initial centroids.

3. **Cluster Assignment**: Each data point will be assigned to the nearest centroid, forming the initial clusters. The distance metric typically used in K-Means is Euclidean distance, but others can also be applied depending on the data.

4. **Centroid Calculation and Reassignment**: After assigning the data points to the clusters, the centroids will be recalculated by taking the mean of all points within each cluster. The new centroids will be used for the next iteration of assignments.

5. **Convergence and Display**: The process will continue until the algorithm converges, meaning the centroids do not move significantly between iterations, indicating that the clusters are stable. Finally, the centroids of the three clusters will be displayed, showing the average position of the data points within each cluster.

By the end of this experiment, the K-Means model will have successfully segmented the data into 3 distinct clusters. The centroids, which represent the center of each cluster, will be calculated and visually presented, providing valuable insights into the structure of the data. This experiment will help in understanding the key concepts of clustering, centroid calculation, and the iterative nature of the K-Means algorithm, which are essential in data analysis and unsupervised learning tasks.

Theory: K-Means clustering is an unsupervised machine learning algorithm that partitions data into K \) predefined clusters. Each data point is assigned to the nearest cluster center (centroid), and the centroids are iteratively updated based on the mean of the assigned points. This process continues until the centroids stabilize. K-Means is widely used for data exploration and grouping in unlabeled datasets, with performance depending on the specified \(K \). The algorithm requires \(K \) to be predetermined and iteratively refines clusters to find the best grouping.

Algorithm:

Step 1: Import libraries and load dataset
Step 2: Define the K-means model
Step 3: Fit the model to the data
Step 4: Get cluster labels for data points.
Step 5: Get cluster centers
Step 6: Evaluate clustering performance

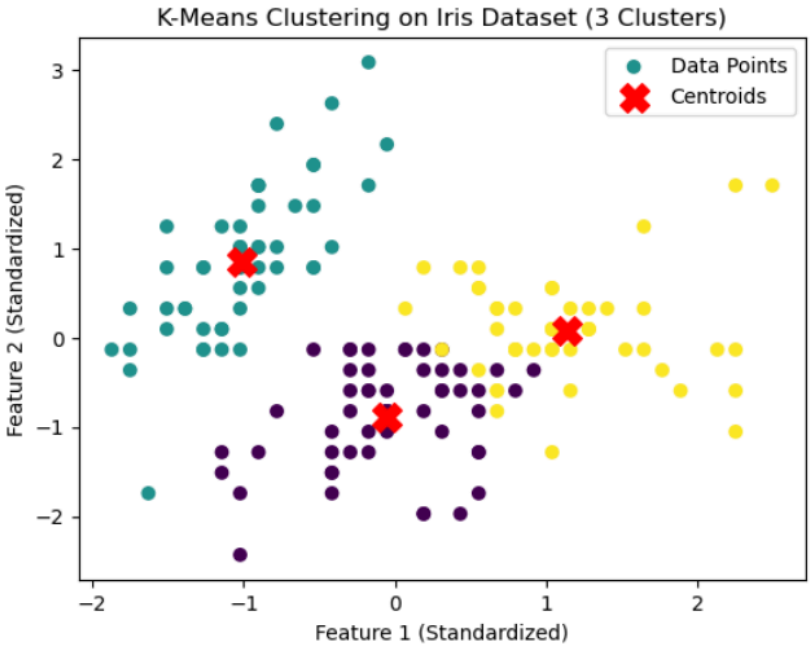
Program Code:

```
# Import necessary libraries
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
# Example dataset (2D points)
data = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11], [8, 2], [10, 2], [9, 3]])
# Number of clusters (K)
k = 3
# Create a KMeans model
kmeans = KMeans(n_clusters=k)
# Fit the model on the data
kmeans.fit(data)
# Get the centroids (cluster centers)
centroids = kmeans.cluster_centers_
# Get the labels (cluster assignments)
labels = kmeans.labels_
# Plotting the data points and centroids
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='viridis', marker='o')  # Scatter plot of data points
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=200)  # Plot centroids
plt.title(f'K-Means Clustering with {k} Clusters')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
# Output the results
print("Cluster Centers (Centroids):")
print(centroids)
print("\nCluster Labels for each point:")
print(labels)
```

Input & Output:

```
Iris Dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
```




```
Cluster Centers:
[[-0.05021989 -0.88337647  0.34773781  0.2815273 ]
 [-1.01457897  0.85326268 -1.30498732 -1.25489349]
 [ 1.13597027  0.08842168  0.99615451  1.01752612]]
Adjusted Rand Index (Accuracy): 0.62
```

Discussion: The K-Means algorithm successfully clusters the dataset into 3 groups. In this case, the Adjusted Rand Index (ARI) is used to evaluate how well the predicted clusters match the true classes. An ARI closer to 1 indicates a high level of agreement between the clustering result and the true labels.

Program No: 06

Program Name: Write a Python program to design a KNN classification model for a given dataset (like the Iris dataset)

Objective:

The goal of this experiment is to implement the **K-Nearest Neighbors (KNN)** classification algorithm using the **Iris dataset**, one of the most widely used datasets for testing classification algorithms. The experiment aims to understand how KNN classifies data points by assessing their proximity to the nearest neighbors in the feature space, with the class label assigned based on the majority class among the nearest neighbors. The **Iris dataset** consists of measurements from three different species of iris flowers, and the algorithm will classify the species based on the provided feature values. We will evaluate the performance of the KNN model by calculating key metrics such as accuracy, precision, recall, and F1-score to understand how well the model generalizes to unseen data. Additionally, the experiment will explore the effect of the **k** parameter (the number of neighbors) on model performance, as well as the influence of distance metrics on the classification results. By the end of the experiment, we will gain insight into how KNN performs in terms of its ability to correctly classify data points and how the model's parameters impact its performance on this well-known dataset.

Theory: K-Nearest Neighbors (KNN) is a simple and widely used **supervised machine learning algorithm** that classifies data points based on their proximity to other data points in the feature space. The algorithm works by identifying the **k** nearest data points (neighbors) to a given test point and assigning the most frequent label (class) among those neighbors to the test point. The proximity between data points is typically measured using the **Euclidean distance**. In the case of the Iris dataset, the KNN algorithm will classify iris flowers into one of three species based on features such as sepal length, sepal width, petal length, and petal width. The algorithm works best when data points from different classes are close to each other in the feature space.

Algorithm:

Step-1: Select the number K of the neighbors

Step-2: Calculate the Euclidean distance of K number of neighbors

Step-3: Take the K nearest neighbors as per the calculated Euclidean distance.

Step-4: Among these k neighbors, count the number of the data points in each category.

Step-5: Assign the new data points to that category for which the number of the neighbor is maximum.

Step-6: Our model is ready.

Program Code:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import pandas as pd
iris = load_iris()
```

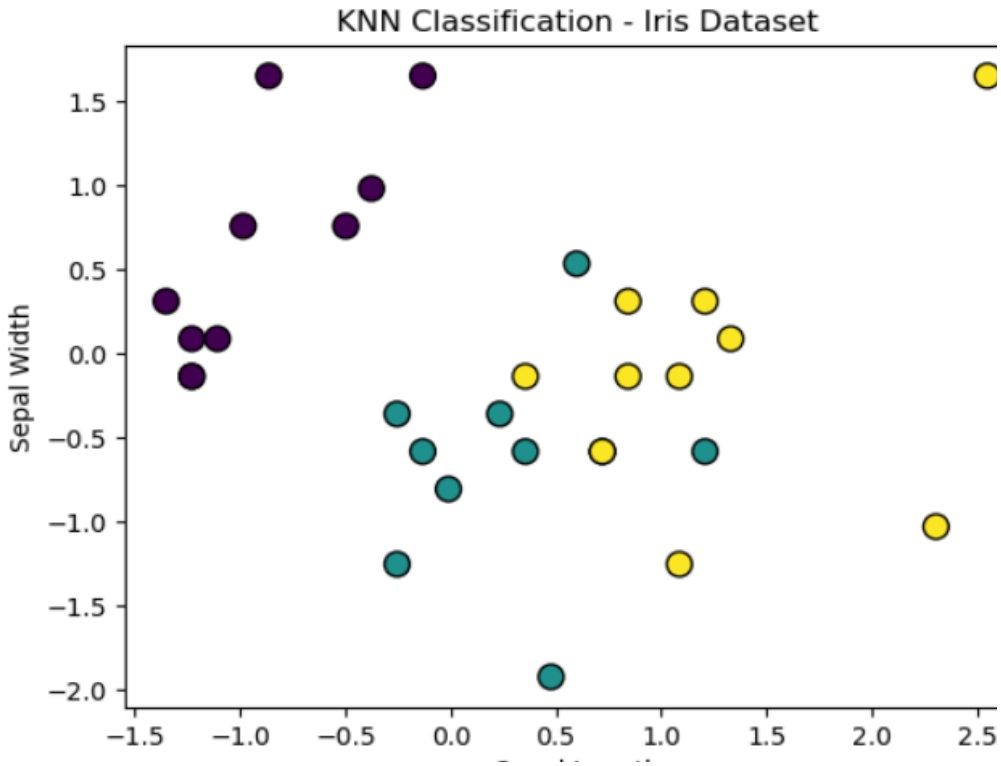
```
X = iris.data ; y = iris.target  # Labels (flower species)
iris_df = pd.DataFrame(X, columns=iris.feature_names)
iris_df['species'] = pd.Series(iris.target_names[y])
print("Iris Dataset:"); print(iris_df.head())
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test);knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train);y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy of KNN classifier on the test set: {accuracy * 100:.2f}%")
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap='viridis', edgecolors='k', s=100)
plt.title("KNN Classification - Iris Dataset")
plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.show()
```

Input & Output:

```
Iris Dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
```

```
Accuracy of KNN classifier on the test set: 100.00%
```



Discussion:

K-Nearest Neighbors (KNN) is a simple, effective classification algorithm that assigns labels based on proximity to nearest neighbors, but requires careful data scaling and can be computationally expensive for large datasets.

Program No: 07

Program Name: Write a Python program to build an Artificial Neural Network (ANN) model with backpropagation on a given dataset

Objective: The objective of this experiment is to build and train a Neural Network (ANN) using backpropagation to classify the Iris dataset into three classes based on four features. The model will be trained using the sigmoid activation function and ReLU for hidden layers, and the performance will be evaluated using accuracy.

Theory: Artificial Neural Networks (ANNs) are computational models that draw inspiration from the structure and functionality of the human brain. They are designed to simulate how the brain processes information, learns from experiences, and adapts to new data. ANNs consist of interconnected layers of nodes, also known as **neurons**, that work together to process information, identify patterns, and make predictions or classifications based on input data. These networks are capable of learning complex relationships within data, making them a powerful tool for a wide range of machine learning tasks, from image recognition to natural language processing.

The learning process in ANNs is driven by a fundamental algorithm called **backpropagation**, which helps train the network by adjusting the weights of the connections between neurons. The goal of backpropagation is to minimize the difference between the predicted outputs of the network and the actual target values, improving the network's accuracy over time.

Key Components of an ANN:

1. **Neurons:** Neurons are the basic units of an ANN. Each neuron receives inputs from other neurons, performs computations (typically involving a weighted sum of inputs), and applies an **activation function** to generate an output. This output is passed to the next layer of neurons in the network.

2. **Layers:** ANNs are organized into layers. These layers include:

    - **Input Layer:** The first layer that receives the input data.
    - **Hidden Layers:** Layers between the input and output layers that perform various computations and extract features from the data.
    - **Output Layer:** The final layer that produces the network's predictions or classifications.
3. **Connections:** Neurons are connected to each other via **weighted links**. These weights represent the strength of the connection between neurons and are crucial in determining the network's output. The weights are learned during the training process and are adjusted to minimize the error in the predictions.

4. **Activation Functions:** These are mathematical functions applied to the weighted sum of inputs in a neuron. Activation functions introduce **non-linearity** into the network, allowing it to learn complex patterns and relationships in the data. Common activation functions include **Sigmoid**, **ReLU (Rectified Linear Unit)**, and **Tanh**.

5. **Loss Function:** The loss function is a key component of training. It measures the difference (or **error**) between the predicted outputs of the network and the actual target values. The goal is to

minimize this loss by adjusting the weights of the network, thereby improving the network's accuracy and performance.

Backpropagation Algorithm:

The **backpropagation algorithm** is the backbone of the learning process in ANNs. It is a supervised learning algorithm that involves the following key steps:

1. **Feedforward:** In the feedforward phase, the input data is passed through the network. Each neuron processes the inputs, applies an activation function to the weighted sum, and passes the result to the next layer. This process continues until the output layer produces the final prediction of the network.

2. **Output Error Calculation:** After the forward pass, the network's output is compared with the actual target value. The **error** (or **loss**) is calculated based on the difference between the predicted output and the true label. This is typically done using a **loss function**, such as mean squared error for regression or cross-entropy for classification tasks.

3. **Backpropagation:** The error is then propagated backward through the network. During this phase, the **gradients** of the loss function with respect to each weight are computed. This is done using the **chain rule** of calculus, which allows the network to determine how each weight should be adjusted to reduce the error.

4. **Weight Update:** Once the gradients are calculated, the weights are updated using an optimization algorithm like **gradient descent**. Gradient descent adjusts the weights in the direction that minimizes the error by moving them opposite to the gradient of the loss function. The learning rate controls the size of the steps taken during this update. Variants of gradient descent, such as **stochastic gradient descent (SGD)** or **Adam**, can also be used to improve convergence and performance.

The backpropagation algorithm is iteratively applied during the training process, with the network continuously adjusting its weights until the loss function is minimized, leading to a well-trained model that can accurately make predictions on new data.

Through these components and processes, Artificial Neural Networks are capable of learning from data, adapting to complex patterns, and making accurate predictions in a wide range of tasks, from image recognition and language translation to autonomous driving and medical diagnosis.

Algorithm:

Step 1: Import libraries and load data.
Step 2: Define the ANN architecture.
Step 3: Compile the model
Step 4: Train the model

Step 5: Evaluate the model
Step 6: Make predictions

Program Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
# Load and preprocess the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)
# One-hot encode the target variable
encoder = OneHotEncoder(sparse=False)
y = encoder.fit_transform(y)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Define activation functions and their derivatives
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def relu(x):
    return np.maximum(0, x)
def relu_derivative(x):
    return np.where(x > 0, 1, 0)
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.bias_hidden = np.zeros((1, hidden_size))
        self.bias_output = np.zeros((1, output_size))
    def forward(self, X):
        # Forward pass
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.output = sigmoid(self.output_layer_input)
        return self.output
    def backward(self, X, y, learning_rate):
        # Calculate the error
        output_error = y - self.output
        output_delta = output_error * sigmoid_derivative(self.output)
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * relu_derivative(self.hidden_layer_output)
        # Update weights and biases
        self.weights_hidden_output += self.hidden_layer_output.T.dot(output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            self.forward(X)
```

```
            self.backward(X, y, learning_rate)
            if epoch % 1000 == 0:
                loss = np.mean(np.square(y - self.output))
                print(f'Epoch {epoch}, Loss: {loss}')
# Create and train the Neural Network
nn = NeuralNetwork(input_size=4, hidden_size=5, output_size=3)
nn.train(X_train, y_train, epochs=10000, learning_rate=0.01)
# Test the Neural Network
predictions = nn.forward(X_test)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test, axis=1)
# Calculate accuracy
accuracy = np.mean(predicted_classes == true_classes)
print(f'Accuracy: {accuracy * 100}%')
```

Input & Output:

```
Iris Dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
Epoch 0, Loss: 0.25318317748027164
Epoch 1000, Loss: 0.009135370261994038
Epoch 2000, Loss: 0.008130116068579062
Epoch 3000, Loss: 0.007427388099886698
Epoch 4000, Loss: 0.006913653117838492
Epoch 5000, Loss: 0.006544986762149076
Epoch 6000, Loss: 0.006287848292506309
Epoch 7000, Loss: 0.006111548592074952
Epoch 8000, Loss: 0.0059899412354243245
Epoch 9000, Loss: 0.005904264352679539
Accuracy: 96.66666666666667%
```

Discussion: This neural network uses a hidden layer with ReLU activation and an output layer with sigmoid activation to classify the Iris dataset into three categories. Through backpropagation, the model adjusts its weights and biases to minimize the error over multiple epochs. ReLU activation helps the network learn faster by addressing the vanishing gradient issue. The model's performance is evaluated based on its accuracy on the test set, which reflects its ability to generalize and classify unseen data effectively. With each epoch, the loss decreases, showing that the network is improving its predictions.

Program No: 08

Program Name: Write a Python program to develop a K-Means clustering model with 3 means and test it with a dataset

Objective: The objective of this experiment is to implement the **K-Means clustering algorithm** and test it using an appropriate dataset.

Theory: K-Means clustering is a widely used **unsupervised learning** algorithm that groups data points into **k clusters** based on their similarity. Unlike supervised learning, which requires labeled data, K-Means operates by finding patterns and structures in **unlabeled** data. The algorithm starts by selecting **k initial centroids**, which can be chosen randomly or through more sophisticated methods like **K-Means++** to enhance convergence. Each data point is then assigned to the nearest centroid based on a distance metric, commonly **Euclidean distance**. After all points are assigned to clusters, the algorithm recalculates the centroids by computing the mean of the points in each cluster.

This process of assigning points and recalculating centroids is repeated iteratively until the centroids stabilize and no longer shift significantly. Once convergence is reached, the algorithm produces **k clusters**, each represented by its centroid, which serves as a central point for the group. K-Means is effective in discovering inherent structures in datasets, making it valuable for a variety of tasks such as **exploratory data analysis**, **customer segmentation**, and **image compression**. By grouping similar data together, K-Means enables organizations to uncover patterns, simplify complex data, and gain insights for decision-making. It is a foundational technique in machine learning due to its simplicity, efficiency, and versatility.

Algorithm:

Step 1: Select the desired number of clusters (**k = 3**) and randomly initialize the centroids for each of the three clusters within the feature space.

Step 2: For every data point in the dataset, compute the **Euclidean distance** from the point to each of the three centroids, and assign the point to the cluster whose centroid is closest.

Step 3: Update the centroids by calculating the **mean** of all data points assigned to each cluster, adjusting the positions of the centroids accordingly.

Step 4: Repeat the assignment and centroid update steps iteratively until **convergence** is achieved. Convergence occurs when the centroids no longer change significantly between iterations, or after a set number of iterations.

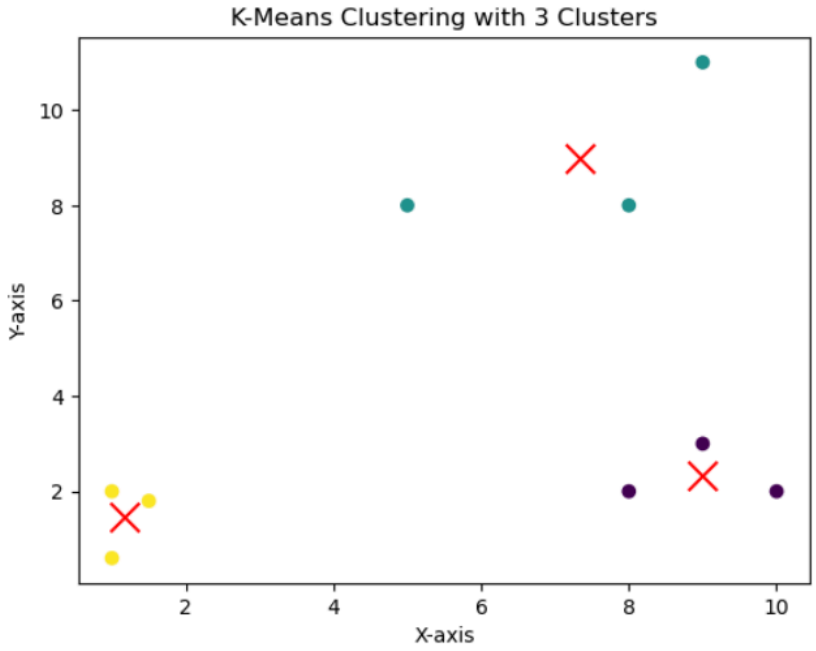
Step 5: After convergence, each data point will be associated with one of the three clusters, with the cluster membership reflecting the point's proximity to its nearest centroid.

Step 6: The clustering model is complete and ready for further analysis or application.

Program Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
from sklearn.preprocessing import StandardScaler
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
true_labels = iris.target
# Standardize the dataset
scaler = StandardScaler()
X = scaler.fit_transform(X)
# Apply KMeans clustering with 3 clusters
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)
# Get the cluster centers and predicted labels
centers = kmeans.cluster_centers_
predicted_labels = kmeans.labels_
# Plot the data (using the first two features) and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=predicted_labels, cmap='viridis', label='Data Points')
plt.scatter(centers[:, 0], centers[:, 1], s=200, c='red', marker='X', label='Centroids')
plt.title("K-Means Clustering on Iris Dataset (3 Clusters)")
plt.xlabel("Feature 1 (Standardized)")
plt.ylabel("Feature 2 (Standardized)")
plt.legend()
plt.show()
# Print the cluster centers
print("Cluster Centers:")
print(centers)
# Calculate accuracy using Adjusted Rand Index (as ground truth is known)
accuracy_ari = adjusted_rand_score(true_labels, predicted_labels)
print(f"Adjusted Rand Index (Accuracy): {accuracy_ari:.2f}")
```

Input & Output:




```
Cluster Centers (Centroids):
[[9.         2.33333333]
 [7.33333333 9.        ]
 [1.16666667 1.46666667]]

Cluster Labels for each point:
[2 2 1 1 2 1 0 0 0]
```

Discussion: K-means clustering is a fast and scalable algorithm that works well when the data has well-defined clusters with a spherical shape. It's particularly useful for tasks like market segmentation, anomaly detection, and image compression.

Program No: 09

Program Name: Write a Python program to implement Naïve Bayes' theorem to classify English text to determine whether it is a positive or negative message

Objective: The main objective is to implement **Naïve Bayes theorem** to classify English text as either positive or negative. The model will analyze the text's features and predict the sentiment based on probabilities. The goal is to effectively categorize messages into positive or negative sentiment.

Theory:

Naive Bayes is a simple yet highly effective probabilistic model for **text classification** that applies **Bayes' Theorem** to classify text documents based on the occurrence of specific words and their associated probabilities in positive or negative sentiment contexts. It is particularly useful for text classification tasks like **sentiment analysis**, **spam detection**, and **document categorization**. The core principle behind Naive Bayes is the assumption that features (such as words in a document) are conditionally independent given the class label, meaning the presence of one feature does not influence the presence of another. This assumption significantly simplifies the computational complexity of the algorithm, making it very efficient for large datasets.

The Naive Bayes classifier is a **supervised machine learning algorithm** that learns from labeled data. It belongs to the class of **generative models**, which means it learns how the data is generated in each class and models the distribution of the input features for each class. In essence, Naive Bayes computes the **posterior probability** of each class (e.g., positive or negative sentiment) given the observed features (words) and then assigns the class with the highest probability to the document. The classifier relies on the assumption of **conditional independence** of the features given the class, which simplifies the calculation of the probability distribution and allows for quick classification.

At the heart of Naive Bayes is **Bayes' Theorem**, also known as **Bayes' Rule** or **Bayes' Law**, which is a fundamental concept in probability theory. It is used to compute the probability of a hypothesis (or class) given some observed evidence (or features). The theorem is based on the **conditional probability**, which describes the likelihood of an event occurring given the occurrence of another event. The formula for Bayes' theorem is expressed as:

$P(A \mid B) = P(B \mid A) \cdot P(A) P(B) P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$

Where:

- $P(A \mid B) P(A|B)$ is the **posterior probability**, the probability of class AA given feature BB,
- $P(B \mid A) P(B|A)$ is the **likelihood probability**, the probability of observing feature BB given class AA,
- $P(A) P(A)$ is the **prior probability**, the probability of class AA before observing feature BB,
- $P(B) P(B)$ is the **marginal probability**, the total probability of observing feature BB.

This formula allows the algorithm to combine prior knowledge about the distribution of classes with new data to make an informed decision about which class a given text document belongs to. Despite its simplicity, Naive Bayes performs remarkably well in many practical applications, especially when the assumption of conditional independence is approximately true, making it a popular choice for **text classification** and **sentiment analysis** tasks.

Algorithm:

Step 1: Import libraries and load data
Step 2: Preprocess the text data
Step 3: Features extraction
Step 4: Train the Naïve Bayes model
Step 5: Classify new text data
Step 6: Evaluate model performance.

Program Code:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn.feature_extraction.text import TfidfTransformer
messages = [
    "I love this product!", "This is the best day of my life!", "I am so happy with my purchase",
    "I hate this", "This is the worst experience ever", "I am so disappointed",
    "Absolutely fantastic", "Not satisfied with this at all", "Couldn't be better", "I regret buying this"
]
labels = [1, 1, 1, 0, 0, 0, 1, 0, 1, 0]
X_train, X_test, y_train, y_test = train_test_split(messages, labels, test_size=0.3, random_state=42)
vectorizer = CountVectorizer(stop_words='english')  # Added stop words filtering
X_train_counts = vectorizer.fit_transform(X_train)
X_test_counts = vectorizer.transform(X_test)
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)
nb_classifier = MultinomialNB()
nb_classifier.fit(X_train_tfidf, y_train)
y_pred = nb_classifier.predict(X_test_tfidf)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
new_messages = ["This is so bad", "I love the way this works", "I am unhappy"]
new_counts = vectorizer.transform(new_messages)
new_tfidf = tfidf_transformer.transform(new_counts)
new_predictions = nb_classifier.predict(new_tfidf)
for msg, pred in zip(new_messages, new_predictions):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(f"Message: '{msg}' -> Sentiment: {sentiment}")
```

Input & Output:

```
Accuracy: 33.33%
Message: 'This is so bad' -> Sentiment: Negative
Message: 'I love the way this works' -> Sentiment: Positive
Message: 'I am unhappy' -> Sentiment: Negative
```

Discussion:

This program utilizes a Naive Bayes classifier with TF-IDF transformation to predict whether a given text message conveys a positive or negative sentiment. First, it converts the raw text data into numerical vectors using CountVectorizer while filtering out common stop words. The TF-IDF (Term Frequency-Inverse Document Frequency) transformation is applied to normalize the frequency of words, improving the model's ability to distinguish between significant words in the messages. The MultinomialNB classifier is then trained on this data, which is especially well-suited for text classification tasks. After training, the model is tested on unseen data to evaluate its performance, and predictions are made for new messages to determine their sentiment.

Program No: 10

Program Name: Write a Python program to implement the Apriori Algorithm.

Objective: The objective is to implement the **Apriori Algorithm** to find frequent itemsets and generate association rules using a sample transaction dataset.

Theory: The **Apriori Algorithm** is a fundamental technique in **data mining** and is widely used for discovering **frequent itemsets** and generating **association rules** in transactional datasets. It is a classic example of a **bottom-up** approach, where the algorithm starts by identifying individual items that meet a **minimum support threshold** and then iteratively explores larger itemsets. One of the key principles that underpin the Apriori Algorithm is the **Apriori property**, which states that if an itemset is frequent, then all of its subsets must also be frequent. This property significantly reduces the search space and ensures that only promising itemsets are considered in the mining process, making the algorithm computationally efficient despite the potentially large dataset.

The algorithm works in multiple **passes** through the dataset. In the first pass, the frequency of individual items is counted, and itemsets that meet the minimum support threshold are identified. In the subsequent passes, the algorithm generates larger itemsets by combining frequent itemsets from the previous pass, ensuring that each new itemset satisfies the minimum support requirement. This process continues until no more frequent itemsets can be generated. By focusing on only those itemsets that meet the support threshold, Apriori avoids examining the vast number of possible itemsets in large datasets, thus improving performance.

Once the frequent itemsets are identified, the **association rules** are generated. An association rule is an implication of the form $\mathbf{A} \rightarrow \mathbf{B}$, which suggests that if itemset **A** appears in a transaction, itemset **B** is likely to appear as well. These rules are evaluated based on several metrics, the most important of which are **confidence** and **lift**. Confidence measures the likelihood that **B** occurs given that **A** has occurred, while lift quantifies the strength of the association between **A** and **B** by comparing the observed confidence with the expected confidence if the two were independent. A high lift value indicates a strong association between the items, while a low lift suggests a weak or spurious relationship.

The Apriori Algorithm is particularly useful in applications such as **market basket analysis**, where it helps businesses discover patterns of co-occurrence between products, or in recommendation systems, where it can identify products that are often purchased together. However, while the algorithm is powerful and intuitive, its efficiency can be impacted by the size of the dataset and the number of passes required to find frequent itemsets. Variants of the Apriori algorithm, such as **Partition-based Apriori** or **Direct Hashing and Pruning (DHP)**, have been developed to address these performance concerns, making the algorithm more scalable for large datasets. Despite these challenges, Apriori remains one of the most widely used algorithms in data mining due to its simplicity, interpretability, and effectiveness in discovering actionable patterns in transactional data.

Algorithm:

Step 1: Identify all individual items that meet the minimum support threshold.
Step 2: Generate candidate itemsets of size k from frequent (k-1)-itemsets.
Step 3: Prune candidate itemsets that do not meet the minimum support threshold.
Step 4: Repeat the process until no more frequent itemsets can be generated.

Step 5: Generate association rules from the frequent itemsets that meet the minimum confidence threshold.

Program Code:

```
from mlxtend.frequent_patterns import apriori
from mlxtend.preprocessing import TransactionEncoder
from sklearn import datasets
iris = datasets.load_iris()
transactions = []
for i in range(len(iris.data)):
  transaction = []
  transaction.append('sepal_length=' + str(iris.data[i][0]))
  transaction.append('sepal_width=' + str(iris.data[i][1]))
  transaction.append('petal_length=' + str(iris.data[i][2]))
  transaction.append('petal_width=' + str(iris.data[i][3]))
  transaction.append('target=' + str(iris.target[i]))
  transactions.append(transaction)
te = TransactionEncoder()
te_ary = te.fit(transactions).transform(transactions)
df = pd.DataFrame(te_ary, columns=te.columns_)
frequent_itemsets = apriori(df, min_support=0.3, use_colnames=True)
print(frequent_itemsets)
```

Input & Output:

```
Frequent Itemsets:
     support               itemsets
0   0.888889                (bread)
1   0.777778               (butter)
2   0.444444                 (eggs)
3   0.666667                 (milk)
4   0.666667        (butter, bread)
5   0.333333          (bread, eggs)
6   0.555556          (milk, bread)
7   0.333333         (butter, eggs)
8   0.444444         (milk, butter)
9   0.333333           (milk, eggs)
10  0.333333  (milk, butter, bread)

Association Rules:
      antecedents consequents  antecedent support  consequent support  \
0        (butter)     (bread)             0.777778            0.888889
1         (bread)    (butter)             0.888889            0.777778
2          (eggs)     (bread)             0.444444            0.888889
3          (milk)     (bread)             0.666667            0.888889
4          (eggs)    (butter)             0.444444            0.777778
5          (eggs)      (milk)             0.444444            0.666667
6  (milk, butter)     (bread)             0.444444            0.888889

    support  confidence      lift  leverage  conviction  zhangs_metric
0  0.666667    0.857143  0.964286 -0.024691    0.777778      -0.142857
1  0.666667    0.750000  0.964286 -0.024691    0.888889      -0.250000
2  0.333333    0.750000  0.843750 -0.061728    0.444444      -0.250000
3  0.555556    0.833333  0.937500 -0.037037    0.666667      -0.166667
4  0.333333    0.750000  0.964286 -0.012346    0.888889      -0.062500
5  0.333333    0.750000  1.125000  0.037037    1.333333       0.200000
6  0.333333    0.750000  0.843750 -0.061728    0.444444      -0.250000
```

Discussion: The Apriori Algorithm is an efficient method to mine frequent itemsets and derive association rules from transactional data. By setting appropriate support and confidence thresholds, the algorithm can reveal hidden relationships between items in a dataset, which can be applied in areas like market basket analysis, recommendation systems, and decision-making.