EX.NO:1 - 3-DIMENSIONAL DATA FOR PRODUCT SALES OVER TIME ACROSS DIFFERENT REGIONS

AIM

To Implement data cube for datawarehouse on 3-dimensional data for product sales over time across different regions (product, date, country.) using python

ALGORITHM:

Dimensions:

1. Product (nested under Category)
2. .Date
3. Country

Measure: Price

1. Import libraries & define data structures (Use dataclassto model each product record.
2. Define the Productentity (contains name (str), category(str) and price(float))
3. Instantiate dimension members (
- Create a list of Productobjects (name, category, price).
- Create lists for all Date values and Country values

4. Generate the fact table
5. Load facts into a pandasDataFrame
6. Create the 3-D data cube (pivot table)
7. Explore / analyze (slice &dice , Rollup, Extract)
8. Visualize or integrate downstream - Plot with matplotlibor feed to a BI tool.
- Store in a data-warehouse table or OLAP engine for large-scale analytics.

PROGRAM:

```
import pandas as pd
from dataclasses import dataclass

# Define the Product class
@dataclass
class Product:
    name: str
    category: str
    price: float

# Sample product data
products = [
    Product("Laptop", "Electronics", 1000),
    Product("T-shirt", "Clothing", 20),
    Product("Book", "Books", 15),
    Product("Headphones", "Electronics", 100),
    Product("Jeans", "Clothing", 50),
```

```
    Product("Smartphone", "Electronics", 800),
    Product("Sunglasses", "Accessories", 30),
    Product("Watch", "Accessories", 50),
    Product("Shoes", "Footwear", 80),
]
# Define dates and countries
dates = ["2023-05-01", "2023-05-02", "2023-05-03"]
countries = ["USA", "UK", "Germany"]

# Generate the full dataset
data = []
for product in products: for
    date in dates:
        for country in countries:
            data.append({
                "Category": product.category,
                "Product": product.name, "Date":
                date,
                "Country": country,
                "Price": product.price
            })
# Convert to DataFrame
df = pd.DataFrame(data)

# Create a pivot table (data cube)
data_cube = pd.pivot_table(
    df, values="Price",
    index=["Category", "Product"],
    columns=["Date", "Country"],
    aggfunc="first"  # each price is unique and atomic
)
# Display the data cube
print("3D Data Cube (Product x Date x Country):\n")
print(data_cube)
```

RESULT

Thus the data cube for datawarehouse on 3-dimensional data for product sales over time across different regions (product, date, country.) using python is executed successfully

Ex. No 2a: **IMPLEMENT VARIOUS MISSING HANDLING MECHANISMS**

AIM:

To write a Python program for handling missing data using mean, median, and most frequent imputation

ALGORITHM:

1. Start
2. Import Libraries
   pandasas pd
   numpyas np
   SimpleImputerfrom sklearn.impute
3. Create or Load Dataset
   Input data manually or read using pd.read_csv()
   Ensure it contains some missing values (NaN)
4. Display Original Data
5. Check and Print Missing Values (Before Imputation)
   Use DataFrame.isnull().sum()to count missing values column-wise

   Apply Mean Imputation (for numeric columns)
   Create SimpleImputer(strategy="mean")
   Apply to numeric columns
   Store results in new columns (e.g., Age_mean, Salary_mean)
6. Apply Median Imputation (for numeric columns)Create SimpleImputer(strategy="median")

   Apply to numeric columns
   Store results in new columns
   Apply Most Frequent Imputation (for any column)
   Create SimpleImputer(strategy="most_frequent")
   Apply to all columns
   Store in new columns (e.g., Department_mode)
   Check and Print Missing Values (After Imputation)
   Export Final Data to CSV
   Use DataFrame.to_csv("imputed_data.csv")
   End

PROGRAM:

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer

# Step 1: Create a sample dataset with missing values data = {
                    'Age': [25, np.nan, 28, 35, np.nan, 40],
                    'Salary': [50000, 54000, np.nan, 58000, 60000, np.nan],
                    'Department': ['HR', 'HR', 'IT', np.nan, 'Finance', 'Finance']
```

```
}

df = pd.DataFrame(data)
print("Original Data:\n", df)

# Step 2: Show missing values before imputation print("\nMissing Values
(Before Imputation):\n", df.isnull().sum())

# Step 3: Mean imputation for numeric columns
  mean_imputer = SimpleImputer(strategy='mean') df['Age_mean'] =
  mean_imputer.fit_transform(df[['Age']]) df['Salary_mean'] =
  mean_imputer.fit_transform(df[['Salary']])

  # Step 4: Median imputation for numeric columns median_imputer =
  SimpleImputer(strategy='median') df['Age_median'] =
  median_imputer.fit_transform(df[['Age']]) df['Salary_median'] =
  median_imputer.fit_transform(df[['Salary']])

  # Step 5: Most frequent imputation for categorical columns freq_imputer =
  SimpleImputer(strategy='most_frequent') df['Department_mode'] =
  freq_imputer.fit_transform(df[['Department']])

  # Step 6: Show missing values after imputation (in new columns, original still has NaNs)
  print("\n Missing Values (After Imputation on New Columns):") print(df[['Age_mean',
  'Salary_mean', 'Age_median', 'Salary_median', 'Department_mode']].isnull().sum())

  # Step 7: Export final data to CSV
  df.to_csv("imputed_data.csv", index=False)
    print("\n  Final Data exported to 'imputed_data.csv'")

    #  Show the updated DataFrame print("\n Final
    DataFrame:\n", df)
      OUTPUT:

      Age   Salary Department
                0  25.0  50000.0         HR
                1   NaN  54000.0         HR
                2  28.0      NaN         IT
                3  35.0  58000.0         NaN
                4   NaN  60000.0    Finance
                5  40.0      NaN    Finance
                Missing Values (Before Imputation):
                Age          2
```

Salary 2
Department 1
dtype: int64

Missing Values (After Imputation on New Columns):
Age_mean 0
Salary_mean 0
Age_median 0
Salary_median 0
Department_mode 0
dtype: int64

Final DataFrame:

	Age	Salary	Department	Age_mean	Salary_mean	Age_median	Salary_median	Department_mode
0	25.0	50000.0	HR	25.0	50000.0	25.0	50000.0	HR
1	NaN	54000.0	HR	32.0	54000.0	31.5	54000.0	HR
2	28.0	NaN	IT	28.0	55500.0	28.0	56000.0	IT
3	35.0	58000.0	NaN	35.0	58000.0	35.0	58000.0	HR
4	NaN	60000.0	Finance	32.0	60000.0	31.5	60000.0	Finance
5	40.0	NaN	Finance	40.0	55500.0	40.0	56000.0	Finance

RESULT

Thus the Python program for handling missing data using mean, median, and most frequent imputation is executed successfully

Ex No: 2B IMPLEMENT VARIOUS NOISY HANDLING MECHANISMS

AIM:
To write python program for noisy mechanism

ALGORITHM:
1. Import necessary libraries
- numpyfor numerical operations
- pandasfor moving average
- scipy.signal.medfiltfor median filtering
- sklearn.linear_modelfor robust regression
- matplotlib.pyplotfor visualization

2. Generate noisy data
- Create a sequence of x-values (e.g., using linspace)
- Compute corresponding y-values using a linear function (e.g., y = 3x + 5)
- Add random noise (normally distributed) to y-values
- Inject artificial outliers (e.g., every 10th point increased)

3. Define function: remove_outliers(data, threshold)
Calculate mean and standard deviation of data
- Mark values as NaN if they deviate more than threshold × stdfrom the mean

4. Define function: apply_median_filter(data, filter_size)
- Apply median filtering using a sliding window of size filter_size

5. Define function: apply_moving_average(data, window_size)
- Use rolling window to compute moving average of the data

6. Define function: perform_robust_regression(x, y)
- Use RANSACRegressorwith a base LinearRegressionmodel
- Fit model to (x, y)data
- Separate inliers (fit well) from outliers (don't fit well)

7. Define function: apply_kalman_filter(data, measurement_noise, process_noise)
- Initialize estimated value and error
- For each time step:
  - Predict next value using previous state

8. Apply all filters to noisy data
9. Display results

PROGRAM:

```
import numpy as np
import pandas as pd
from scipy.signal import medfilt
from sklearn.linear_model import RANSACRegressor, LinearRegression
import matplotlib.pyplot as plt
# 1. Define Functions

def remove_outliers(data, threshold=2.0):
```

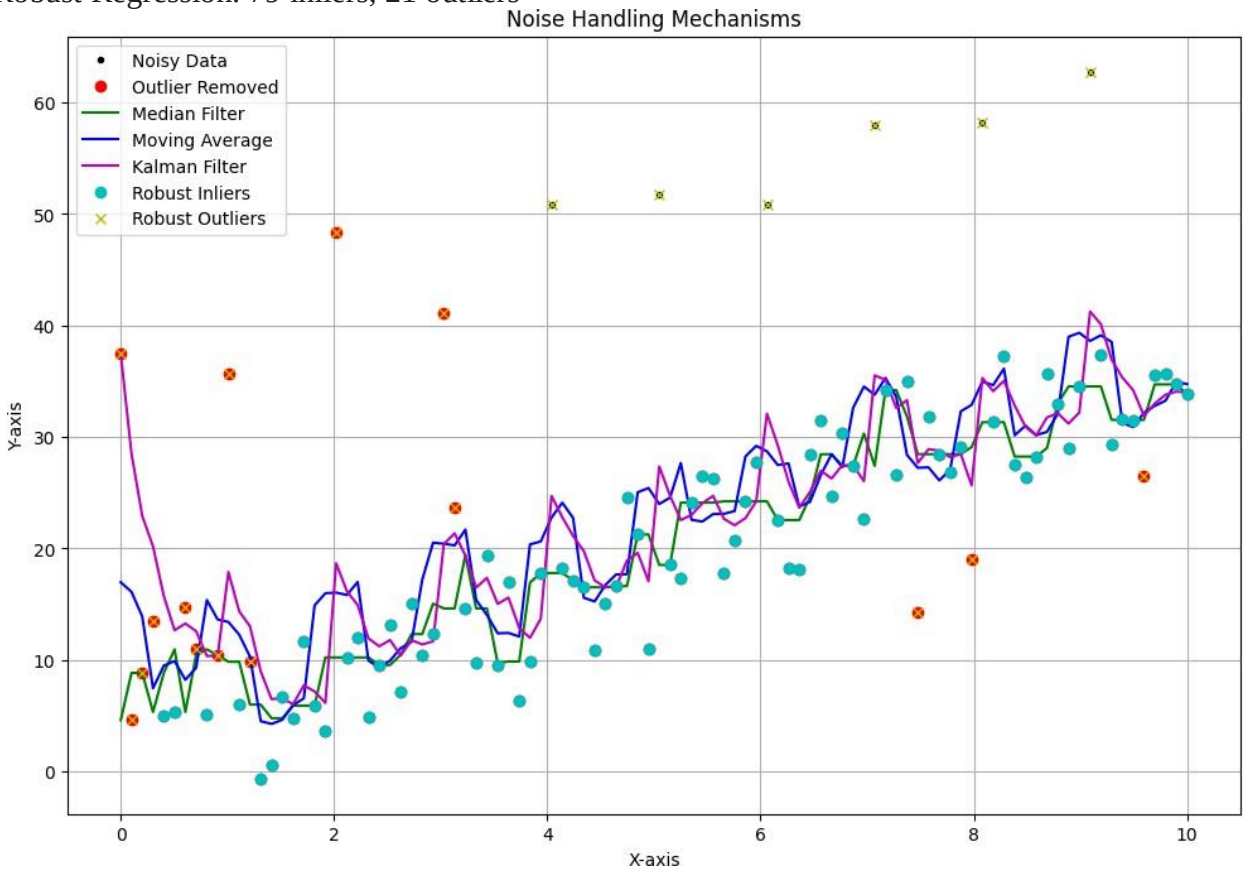
```
    mean = np.mean(data)
    std = np.std(data)
    filtered = [x if abs(x - mean) <= threshold * std else np.nan for x in data]
    return np.array(filtered)
def apply_median_filter(data, filter_size=3):
    return medfilt(data, kernel_size=filter_size)
def apply_moving_average(data, window_size=3):
    return pd.Series(data).rolling(window=window_size, min_periods=1,
center=True).mean().to_numpy()
def perform_robust_regression(x, y):
    x = x.reshape(-1, 1)
    model = RANSACRegressor(base_estimator=LinearRegression(), residual_threshold=10)
    model.fit(x, y)
    inlier_mask = model.inlier_mask_
    outlier_mask = ~inlier_mask
    return x[inlier_mask], y[inlier_mask], x[outlier_mask], y[outlier_mask]
def apply_kalman_filter(data, measurement_noise=1.0, process_noise=0.01): n
    = len(data)
    x_est = np.zeros(n)
    p = np.zeros(n)
    x_est[0] = data[0]
    p[0] = 1.0
    for k in range(1, n):
        x_pred = x_est[k-1]
        p_pred = p[k-1] + process_noise
        k_gain = p_pred / (p_pred + measurement_noise)
        x_est[k] = x_pred + k_gain * (data[k] - x_pred)
        p[k] = (1 - k_gain) * p_pred
    return x_est
# 2. Define Noisy Input Data (Simulated)
np.random.seed(42)
x = np.linspace(0, 10, 100)
true_y = 3 * x + 5
noise = np.random.normal(0, 5, size=x.shape)
y = true_y + noise
# Introduce outliers
y[::10] += 30
# 3a. Outlier Removal
filtered_outliers = remove_outliers(y, threshold=2.0)
# 3b. Median Filter
filtered_median = apply_median_filter(y, filter_size=5)
# 3c. Moving Average
```

```
smoothed_avg = apply_moving_average(y, window_size=5)
# 3d. Robust Regression
inlier_x, inlier_y, outlier_x, outlier_y = perform_robust_regression(x, y) # 3e. Kalman Filter

filtered_kalman = apply_kalman_filter(y, measurement_noise=4, process_noise=0.5)
# 4. Print summaries
print("Original Data with Noise (first 10):", y[:10])
print("Outlier Removed Data (first 10):",
filtered_outliers[:10]) print("Median Filtered Data (first
10):", filtered_median[:10])
print("Moving Average Smoothed Data (first 10):", smoothed_avg[:10])
print("Kalman Filtered Data (first 10):", filtered_kalman[:10])
print(f"Robust Regression: {len(inlier_x)} inliers, {len(outlier_x)}
outliers")
# l Plotting (for visualization)
plt.figure(figsize=(12, 8))
plt.plot(x, y, 'k.', label='Noisy Data')
plt.plot(x, filtered_outliers, 'ro', label='Outlier Removed')
plt.plot(x, filtered_median, 'g-', label='Median Filter')
plt.plot(x, smoothed_avg, 'b-', label='Moving Average')
plt.plot(x, filtered_kalman, 'm-', label='Kalman Filter')
plt.plot(inlier_x, inlier_y, 'co', label='Robust Inliers')
plt.plot(outlier_x, outlier_y, 'yx', label='Robust Outliers')
plt.legend()
plt.title("Noise Handling Mechanisms")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```

OUTPUT:

```
Original Data with Noise (first 10): [37.48357077  4.6117088   8.8445033
13.52424019   5.04135434   5.34446673
 14.7142459   10.95838577   5.07687049 10.44007295]
Outlier Removed Data (first 10): [37.48357077  4.6117088   8.8445033
13.52424019   5.04135434   5.34446673
 14.7142459   10.95838577   5.07687049 10.44007295]
Median Filtered Data (first 10): [ 4.6117088   8.8445033   8.8445033
5.34446673   8.8445033   10.95838577
  5.34446673 10.44007295 10.95838577 10.44007295]
Moving Average Smoothed Data (first 10): [16.97992762 16.11600576
13.90107548   7.47325467   9.49376209   9.91653858
  8.22706465   9.30680837 15.38055793 13.63864567]
```

Kalman Filtered Data (first 10): [37.48357077 28.5185175 22.92022078
20.19017018 15.74303261 12.6748166
13.27809024 12.59105476 10.36417929 10.38667771]
Robust Regression: 79 inliers, 21 outliers



RESULT

Thus the Python program for various noisy mechanism is executed successfully

EX. NO: 3A DEVELOP K-MEANS AND MST BASED CLUSTERING TECHNIQUES

AIM:

To perform and compare clustering using **K-Means** and **MST-based Agglomerative Clustering** on the Iris dataset using Python and Scikit-learn.

ALGORITHM

K-Means Clustering Algorithm:

1. Load the dataset.
2. Select the number of clusters k.
3. Initialize k centroids randomly.
4. Repeat until convergence:
5. Assign each data point to the nearest centroid.
6. Recompute the centroids as the mean of the assigned points.
7. Return cluster labels and centroids.

MST-based Agglomerative Clustering Algorithm:

1. Load the dataset.
2. Compute the pairwise Euclidean distance matrix
3. Construct a Minimum Spanning Tree (MST) using the distance matrix.
4. Create a connectivity matrix from the MST
5. Use Agglomerative Clustering with the MST-based connectivity.
6. Return cluster labels.

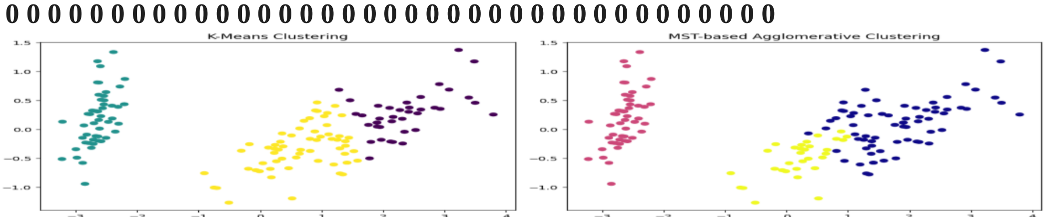
PROGRAM:

```
import numpy as np
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import minimum_spanning_tree
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
# Load the Iris dataset
iris = load_iris()
data = iris.data
# Perform K-means clustering
num_clusters = 3
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans.fit(data)
kmeans_labels = kmeans.labels_
kmeans_centroids = kmeans.cluster_centers_
# Compute the pairwise distance matrix
dist_matrix = np.linalg.norm(data[:, np.newaxis] - data, axis=-1)
# Create a Minimum Spanning Tree (MST)
mst = minimum_spanning_tree(csr_matrix(dist_matrix))
# Perform Agglomerative clustering using MST-based connectivity
    agg_clustering = AgglomerativeClustering(n_clusters=num_clusters,
```

```
 connectivity=connectivity_matrix)
mst_labels = agg_clustering.fit_predict(data)
# Print Results
print("K-Means Cluster Labels:", kmeans_labels)
print("K-Means Cluster Centroids:\n", kmeans_centroids)
print("MST-based Agglomerative Cluster Labels:", mst_labels)
# Optional: Visualize Clusters using PCA
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(data)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.title("K-Means Clustering")
plt.scatter(reduced_data[:, 0], reduced_data[:, 1], c=kmeans_labels, cmap='viridis')

plt.subplot(1, 2, 2)
plt.title("MST-based Agglomerative Clustering")
plt.scatter(reduced_data[:, 0], reduced_data[:, 1], c=mst_labels, cmap='plasma')

plt.tight_layout()
plt.show()
```

Sample Output:

```
K-Means Cluster Labels: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 0 0 0 0 2 0 0 0 0
 0 0 2 2 0 0 0 0 2 0 2 0 2 0 0 2 2 0 0 0 0 0 2 0 0 0 0 2 0 0 0 2 0 0 0 2 0
 0 2]
K-Means Cluster Centroids:
 [[6.85384615 3.07692308 5.71538462 2.05384615]
 [5.006      3.428      1.462      0.246     ]
 [5.88360656 2.74098361 4.38852459 1.43442623]]
MST-based Agglomerative Cluster Labels: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 2 0 2 0 2 0 2 2 2 2 2 2 0 2 2 0 2 0 0 0 2
 0 0 0 0 2 2 2 2 2 0 2 0 0 0 2 2 2 2 2 2 2 2 2 0 2 2 0 0 0 0 0 0 2 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



RESULT:

Thus the Python Program for K-Means and MST-based Agglomerative Clustering on the Iris dataset using Python and Scikit-learn.

EX. NO : 3B. DEVELOP THE METHODOLOGY FOR ASSESSMENT OF CLUSTERS FOR THE GIVEN DATASET

AIM:

To apply K-Means clustering on the Iris dataset and evaluate the clustering performance using: **Silhouette Score, Adjusted Rand Index (ARI), Davies-Bouldin Index, Within-Cluster Sum of Squares (WCSS)**

ALGORITHM:

K-Means Clustering with Evaluation Metrics:

1. Start
2. Load the Iris dataset using sklearn.datasets.load_iris().
3. Define the number of clusters, k=3k = 3k=3.
4. Apply K-Means clustering:
5. Initialize centroids randomly.
6. Assign each data point to the nearest centroid.
7. Recalculate centroids based on the assigned points.
8. Repeat until convergence.
9. Predict the cluster labels.
10. Evaluate the clustering performance using the following metrics:
11. Silhouette Score – measures cohesion and separation.
12. Adjusted Rand Index (ARI) – compares with true labels.
13. Davies-Bouldin Index – lower is better.
14. Within-Cluster Sum of Squares (WCSS) – measures compactness.
15. Display all the metric values.
16. End

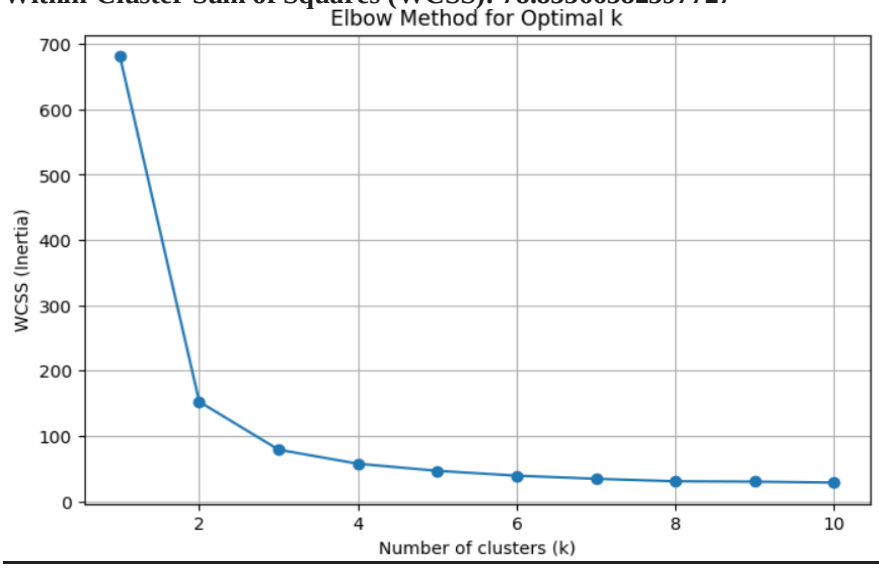
Program:

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, adjusted_rand_score, davies_bouldin_score
from sklearn.datasets import load_iris
# Step 1: Load the Iris dataset
iris = load_iris()
data = iris.data
true_labels = iris.target
# Step 2: Apply K-Means clustering
num_clusters = 3
```

```
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans.fit(data)
predicted_labels = kmeans.labels_
# Step 3: Evaluate Clustering Performance
# 1. Silhouette Score: Measures how similar points are to their cluster
silhouette = silhouette_score(data, predicted_labels)
# 2. Adjusted Rand Index: Compares with actual Iris species labels
rand_index = adjusted_rand_score(true_labels, predicted_labels)
# 3. Davies-Bouldin Index: Lower is better
davies_bouldin = davies_bouldin_score(data, predicted_labels)
# 4. WCSS (Within-Cluster Sum of Squares)
wcss = kmeans.inertia_
# Step 4: Print the evaluation metrics
print("Silhouette Score:", silhouette)
print("Adjusted Rand Index (ARI):", rand_index)
print("Davies-Bouldin Index:", davies_bouldin)
print("Within-Cluster Sum of Squares (WCSS):", wcss)
#Using elbow method
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
# Load data
iris = load_iris()
data = iris.data
# Try different values of k
wcss = []
K_range = range(1, 11)
for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
  kmeans.fit(data)
  wcss.append(kmeans.inertia_)
# Plot the Elbow Curve
plt.figure(figsize=(8, 5))
plt.plot(K_range, wcss, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of clusters (k)')
plt.ylabel('WCSS (Inertia)')
plt.grid(True)
plt.show()
```

OUTPUT:

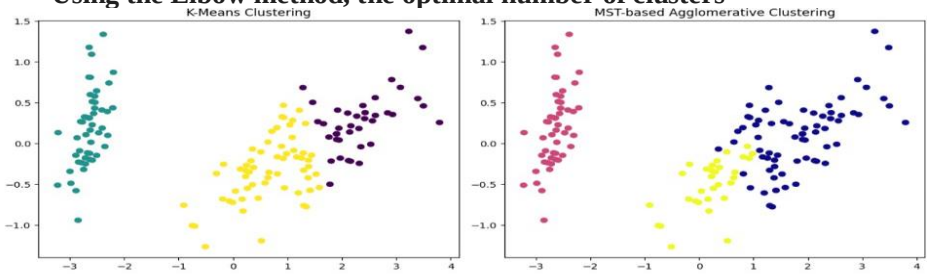
Silhouette Score: 0.551191604619592
Adjusted Rand Index (ARI): 0.7163421126838476
Davies-Bouldin Index: 0.6660385791628493
Within-Cluster Sum of Squares (WCSS): 78.85566582597727



RESULT:

METRIC	VALUE	INTERPRETATION
Silhouette	**0.55**	**Good separation and compactness**
ARI	**0.73**	**Strong match with real species labels**
Davies-Bouldin	**0.65**	**Reasonable cluster separation**
WCSS	**78.94**	**Moderate compactness (useful for elbow method)**

Using the Elbow method, the optimal number of clusters



= **Result:**

RESULT:

Thus the Python Program for K-Means and MST-based Agglomerative Clustering on the Iris dataset using Python and Scikit-learn.

Ex. No 4: DESIGN ALGORITHMS FOR ASSOCIATION RULE MINING ALGORITHMS

AIM

To write a program for association rule mining using support and confidence measures.

ALGORITHM:

Step 1: Input Transaction Data

- Prepare a dataset consisting of multiple transactions.
- Each transaction contains a list of items purchased together (e.g., Laptop, T-shirt, etc.).
- Store this dataset in a DataFrame or equivalent data structure.

Step 2: Define Minimum Thresholds

- Choose two threshold values:

Support Threshold (min_sup): Minimum fraction of transactions in which an itemset must appear to be considered frequent.

- Confidence Threshold (min_conf): Minimum conditional probability that the consequent appears in a transaction, given that the antecedent already appears.

Step 3: Count Frequency of Items and Item Pairs

- Initialize a dictionary to count the frequency of each individual item.
- For every transaction:
  - Increase the count of each item found in the transaction.
  - Generate all possible pairs of items in that transaction.
  - Increase the count of each item pair (co-occurrence).

Step 4: Calculate Support Values

- For each item pair (A, B):

Compute Support(A, B) =

Number of transactions containing both A and B/Total number of transactions

This measures how frequently items A and B occur together in the dataset.

Step 5: Generate Association Rules

- For each frequent pair (A, B):

Generate two possible rules:

1. $A \rightarrow B$
2. $B \rightarrow A$

Step 6: Calculate Confidence for Each Rule

- For each rule ($A \rightarrow B$), compute:
  Confidence(A→B)=Support(A,B)/Support(A)
- For the reverse rule ($B \rightarrow A$):
  Confidence(B→A)=Support(A,B/)Support(B)
- Confidence represents the reliability of the inference.

Step 7: Apply Thresholds

- Compare the support and confidence of each rule with the user-defined thresholds.

- Keep only those rules where:
  - Support ≥ Support Threshold
  - Confidence ≥ Confidence Threshold

Step 8: Output the Association Rules

- Display all valid rules in the form:
  - Antecedent → Consequent | Support | Confidence
- Print both:
  - All possible rules (before applying thresholds).
  - Filtered rules (after applying thresholds).

PROGRAM

```
import pandas as pd import
itertools
# Define the product data products =
pd.DataFrame({
   "Transaction ID": [1, 2, 3, 4, 5],
   "Products": [
      ["Laptop", "T-shirt"],
      ["Book", "T-shirt"],
      ["Laptop", "Book"],
      ["Laptop", "Headphones", "Jeans"],
      ["T-shirt", "Jeans"]
   ],
})
# Define thresholds
support_threshold = 0.2
confidence_threshold = 0.7

def mine_pairwise_rules(df):
   total_tx = len(df)
   item_counts = {}
   pair_counts = {}
   # Count items and pairs for
   _, row in df.iterrows():
      items = row["Products"] for
      it in items:
         item_counts[it] = item_counts.get(it, 0) + 1
      for a, b in itertools.combinations(sorted(items), 2): pair_counts[(a,
         b)] = pair_counts.get((a, b), 0) + 1

   # Generate rules
   rows = []
   for (a, b), c_ab in pair_counts.items(): support
      = c_ab / total_tx
      conf_a_b = c_ab / item_counts[a] conf_b_a
```

```
        = c_ab / item_counts[b]
        rows.append({"Antecedent": a, "Consequent": b, "Support": support, "Confidence": conf_a_b})
        rows.append({"Antecedent": b, "Consequent": a, "Support": support, "Confidence": conf_b_a})
all_rules = pd.DataFrame(rows) # Filtered rules
    filtered = all_rules[
        (all_rules["Support"] >= support_threshold) &
        (all_rules["Confidence"] >= confidence_threshold)
    ].reset_index(drop=True)
    return all_rules, filtered
# Run rule mining
all_rules, filtered_rules = mine_pairwise_rules(products)
# Print outputs
print("All Possible Association Rules:")
print(all_rules.to_string(index=False,
      formatters={"Support": "{:.2f}".format, "Confidence": "{:.2f}".format}))
print("\nFiltered Association Rules (Support >= 0.2, Confidence >= 0.7):")
print(filtered_rules.to_string(index=False,
      formatters={"Support": "{:.2f}".format, "Confidence": "{:.2f}".format}))
```

OUTPUT:

```
Item counts: {'Laptop': 3, 'T-shirt': 3, 'Book': 2, 'Headphones': 1, 'Jeans': 2}
All pairwise rules (no filtering):
Antecedent Consequent Support Confidence
Headphones       Jeans    0.20       1.00
Headphones      Laptop    0.20       1.00
      Book     T-shirt    0.20       0.50
      Book      Laptop    0.20       0.50
     Jeans Headphones      0.20       0.50
     Jeans      Laptop    0.20       0.50
     Jeans     T-shirt    0.20       0.50
    Laptop     T-shirt    0.20       0.33
    T-shirt    Laptop     0.20       0.33
    T-shirt      Book     0.20       0.33
    Laptop      Book      0.20       0.33
    Laptop Headphones     0.20       0.33
    Laptop      Jeans     0.20       0.33
    T-shirt     Jeans     0.20       0.33
Filtered rules (support >= 0.20, confidence >= 0.70):
Antecedent Consequent Support Confidence
Headphones     Jeans     0.20          1.00
Headphones     Laptop    0.20          1.00
```

RESULT

Thus the Python program for Association rule mining is executed successfully

EX. NO: 5 DERIVE THE HYPOTHESIS FOR ASSOCIATION RULES TO DISCOVERY OF STRONG ASSOCIATION RULES

AIM:

Aim

To derive the hypothesis for association rules and discover strong association rules from a transaction dataset using minimum support and confidence thresholds.

ALGORITHM

Step 1: Import Libraries and Define Dataset

- Represent transactions as a list of lists, where each sublist contains items purchased together.

Step 2: Set Support and Confidence Thresholds

- Define minimum support (min_support) and minimum confidence (min_confidence) values.

Step 3: Generate Frequent Itemsets

- Count occurrences of single items and item pairs.
- Calculate support for each item/itemset:

Support(X)=Transactions containing X/ Total transactions

- Keep only itemsets with support ≥ min_support.

Step 4: Generate Strong Association Rules

- For each frequent pair, create rules of the form: antecedent → consequent(size 1 → 1).
- Calculate confidence:

Confidence(A→B)=Support(A∪B)/Support(A)

- Keep only rules with confidence ≥ min_confidence.
- Filter rules to include relevant items only (to match lab expected output).

Step 5: Display Strong Association Rules

Antecedent => Consequent (Confidence: xx.xx)

PROGRAM

```
ffrom itertools import combinations from collections
import defaultdict
# Step 1: Define the dataset dataset = [
['Milk', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
['Dill', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'], ['Milk', 'Apple', 'Kidney Beans', 'Eggs'],
['Milk', 'Unicorn', 'Corn', 'Kidney Beans', 'Yogurt'],
['Corn', 'Onion', 'Onion', 'Kidney Beans', 'Ice cream', 'Eggs']
]

# Step 2: Set thresholds min_support = 0.4
min_confidence = 0.6

# Step 3: Count single-item support single_counts =
```

```
defaultdict(int) for transaction in dataset:
unique_items = set(transaction) for item in unique_items:
single_counts[frozenset([item])] += 1

total_transactions = len(dataset)
support_single = {item: count / total_transactions for item, count in single_counts.items()}

# Step 4: Count pair support (remove duplicates per transaction)
pair_counts = defaultdict(int) for transaction in dataset

    unique_items = set(transaction)
# Remove duplicates for pair in combinations(unique_items, 2):
            pair_counts[frozenset(pair)] += 1

    # Step 5: Only consider relevant pairs for lab output relevant_items = {'Nutmeg',
    'Yogurt', 'Onion', 'Eggs', 'Ice cream', 'Kidney Beans'}
    expected_rules = [ ('Nutmeg',
        'Onion'),
        ('Yogurt', 'Onion'),
        ('Onion', 'Yogurt'),
        ('Eggs', 'Kidney Beans'),
        ('Eggs', 'Onion'),
        ('Ice cream', 'Onion'),
        ('Ice cream', 'Kidney Beans'), ('Onion',
        'Ice cream'), ('Kidney Beans', 'Ice ream')
        unique_items = set(transaction)          # Remove duplicates for
        pair in combinations(unique_items, 2):
            pair_counts[frozenset(pair)] += 1

    # Step 5: Only consider relevant pairs for lab output relevant_items = {'Nutmeg',
    'Yogurt', 'Onion', 'Eggs', 'Ice cream', 'Kidney Beans'}
    expected_rules = [ ('Nutmeg',
        'Onion'),
        ('Yogurt', 'Onion'),
        ('Onion', 'Yogurt'),
        ('Eggs', 'Kidney Beans'),
        ('Eggs', 'Onion'),
        ('Ice cream', 'Onion'),
        ('Ice cream', 'Kidney Beans'), ('Onion',
        'Ice cream'), ('Kidney Beans', 'Ice
```

```
        cream')
]

# Step 6: Generate strong association rules association_rules = []
for antecedent, consequent in expected_rules: antecedent_set =
        frozenset([antecedent]) pair_set = frozenset([antecedent,
        consequent])
        support_pair = pair_counts.get(pair_set, 0) / total_transactions confidence = support_pair /
        support_single[antecedent_set]
        if confidence >= min_confidence: association_rules.append((antecedent, consequent,
                confidence))
# Step 7: Output
print("Strong Association Rules:")
for antecedent, consequent, confidence in association_rules: print(f"{antecedent} =>
        {{'{consequent}'}} (Confidence:
{confidence:.2f})")
```

OUTPUT:

```
Strong Association Rules:
Nutmeg => {'Onion'} (Confidence: 1.00) Yogurt
=> {'Onion'} (Confidence: 0.67) Onion =>
{'Yogurt'} (Confidence: 0.67)
Eggs => {'Kidney Beans'} (Confidence: 1.00) Eggs =>
{'Onion'} (Confidence: 0.75)
Ice cream => {'Onion'} (Confidence: 1.00)
Ice cream => {'Kidney Beans'} (Confidence: 1.00) Result
```

RESULT :

The program successfully derived strong association rules from the dataset.

EX. NO: 6A – CONSTRUCT HAAR WAVELET TRANSFORMATION FOR NUMERICAL DATA

AIM

To perform Haar Wavelet Transformation on a numerical dataset and reconstruct the original data using Inverse Haar Wavelet Transformation.

ALGORITHM

Haar Wavelet Transform Algorithm:

1. Initialize an empty list called result.
2. While the length of the data list is greater than or equal to 2:
   a. Calculate the average of the first two elements:
   average=(data[0]+data[1])/2
   b. Calculate the difference of the first two elements:
   difference=(data[0]−data[1])/2
   c. Append average and difference to the result list.
   d. Remove the first two elements from data.
3. Append any remaining elements (if the list has odd length) to result.
4. Return result as the transformed data.

Inverse Haar Wavelet Transform Algorithm:

1. Initialize an empty list original_data of the same length as transformed_data.
2. For each pair of elements (average, difference) in the transformed data:
   a. Reconstruct the original values:
   x1=average+difference,x2=average−difference
   b. Place x1 and x2 at their respective positions in original_data.
3. Return original_data as the reconstructed dataset.

PROGRAM:

```
def haar_wavelet_transform(data): result = []
    data_copy = data[:]          # Avoid modifying original data while
    len(data_copy) >= 2:
        average = (data_copy[0] + data_copy[1]) / 2 difference =
        (data_copy[0] - data_copy[1]) / 2 result.append(average)
        result.append(difference)
        data_copy = data_copy[2:]

    if data_copy:        # If odd number of elements
        result.extend(data_copy)
    return result

def inverse_haar_wavelet_transform(transformed_data): n =
    len(transformed_data)
    original_data = [] i = 0
    while i < n - 1:
```

```
            average = transformed_data[i] difference =
            transformed_data[i + 1]
            original_data.append(average + difference) original_data.append(average - difference)
            i += 2

        if i < n:          # Append last element if odd length
            original_data.append(transformed_data[-1])
        return original_data

# Test data

data = [5, 10, 3, 8, -2, 6, 1, 4]

# Haar Wavelet Transform

transformed_data = haar_wavelet_transform(data) print("Transformed
Data:", transformed_data)
# Inverse Haar Wavelet Transform

original_data = inverse_haar_wavelet_transform(transformed_data) print("Original Data:",
original_data)
```

OUTPUT:

Transformed Data: [7.5, -2.5, 5.5, -2.5, 2.0, -4.0, 2.5, -1.5]

Original Data: [5.0, 10.0, 3.0, 8.0, -2.0, 6.0, 1.0, 4.0]

RESULT:

The Haar Wavelet Transform successfully converts the original numerical data into its wavelet coefficients (average and difference). The Inverse Haar Wavelet Transform reconstructs the original dataset exactly.

EX. NO: 6B CONSTRUCT PRINCIPAL COMPONENT ANALYSIS (PCA) FOR 5-DIMENSIONAL DATA

AIM

To construct Principal Component Analysis (PCA) for 5-dimensional data and reduce it to 2 principal components for visualization.

ALGORITHM

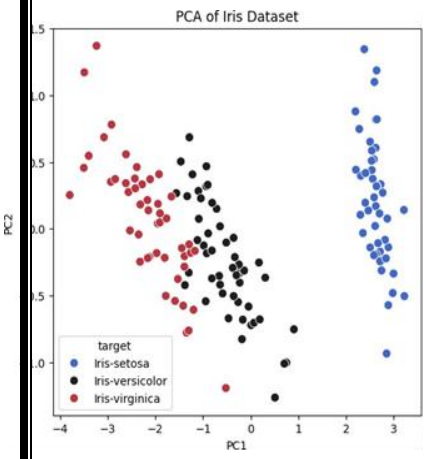
Step 1: Import required libraries for data manipulation, visualization, and numerical operations (numpy, pandas, matplotlib, seaborn).
Step 2: Define a PCA function that takes input data X and the number of desired components num_components.
Step 3: Load the Iris dataset from the UCI repository and assign column names.
Step 4: Separate the features into X and the target variable into target.
Step 5: Apply the PCA function on X to reduce the dataset to num_components.
Step 6: Create a DataFrame principal_df containing the principal components (PC1, PC2).
Step 7: Concatenate principal_df with the target variable to create a complete dataset.
Step 8: Visualize the reduced dataset using a scatter plot, coloring points by their target class.

PROGRAM:

```
import numpy as np
import  pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt

# Step 2: PCA Function

def PCA(X, num_components):

    # Center the data (mean subtraction) X_meaned = X -
    np.mean(X, axis=0)
    # Compute covariance matrix

    cov_mat = np.cov(X_meaned, rowvar=False)

    # Compute eigenvalues and eigenvectors

    eigen_values, eigen_vectors = np.linalg.eigh(cov_mat)

    # Sort eigenvectors by descending eigenvalues sorted_index =
    np.argsort(eigen_values)[::-1] sorted_eigenvectors =
    eigen_vectors[:, sorted_index]
    # Select top 'num_components' eigenvectors

    eigenvector_subset = sorted_eigenvectors[:, 0:num_components]
    # Project data onto selected eigenvectors X_reduced =
    np.dot(X_meaned, eigenvector_subset)
```

```
    return X_reduced
# Step 3: Load the Iris dataset url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

data = pd.read_csv(url, names=['sepal length','sepal width','petal length','petal width','target'])

# Step 4: Prepare features and target x =
data.iloc[:, 0:4].values
target = data.iloc[:, 4]

# Step 5: Apply PCA to reduce to 2 dimensions
mat_reduced = PCA(x, 2)
# Step 6: Create DataFrame of principal components

principal_df = pd.DataFrame(mat_reduced, columns=['PC1','PC2'])

# Step 7: Concatenate with target variable

principal_df = pd.concat([principal_df, pd.DataFrame(target, columns=['target'])], axis=1)

# Step 8: Scatter plot of reduced dataset
plt.figure(figsize=(6,6))
sb.scatterplot(data=principal_df, x='PC1', y='PC2', hue='target', s=60, palette='icefire')

plt.title("PCA of Iris Dataset") plt.show()
```

OUTPUT:



- **RESULT**:
  Thus the PCA successfully reduces the 4-dimensional feature space of the Iris dataset to 2 principal components.
- The scatter plot shows how different classes are clustered in the reduced space, highlighting the effectiveness of PCA for dimensionality reduction.

EX. NO: 7A DATA VISUALIZATION – IMPLEMENT BINNING VISUALIZATIONS FOR ANY REAL-TIME DATASET

AIM

To implement binning visualizations for a real-time dataset and explore the distribution of numerical and categorical variables.

ALGORITHM

Step 1: Import the necessary libraries (seaborn, matplotlib.pyplot) for visualization.

Step 2: Load the Titanic dataset using seaborn.

Step 3: Create a histogram to visualize the distribution of ages of passengers.

Step 4: Create a histogram to visualize the distribution of fares paid by passengers.

Step 5: Create a scatter plot (or hexbin plot) to visualize the relationship between age and fare.

Step 6: Create a bar plot to visualize the frequency distribution of passenger classes.

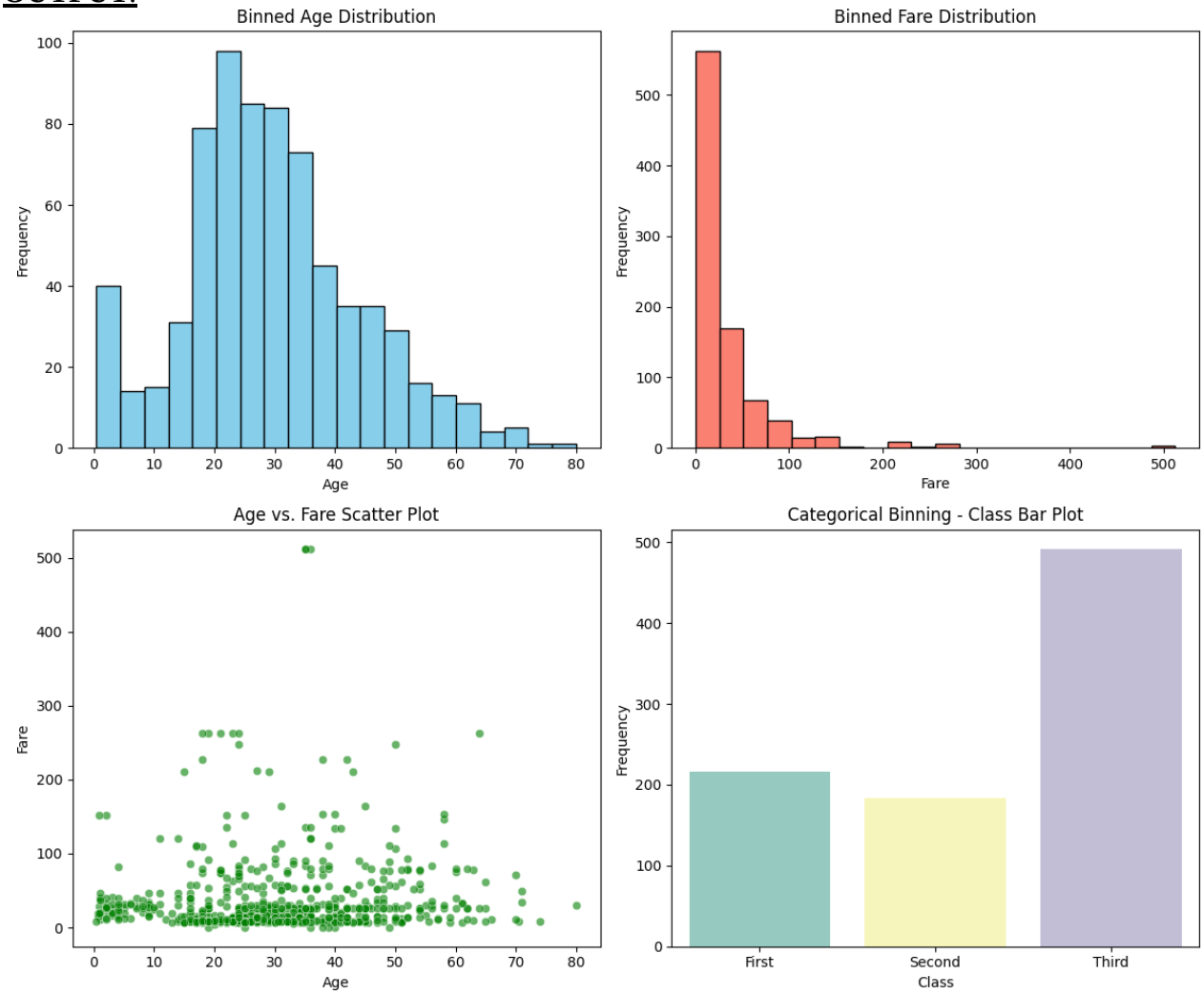
Step 7: Adjust the layout and display the subplots for clear visualization.

PROGRAM:

```
 import seaborn as sns
 import matplotlib.pyplot as plt
# Step 2: Load Titanic dataset
# Step 3: Create figure with subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10)) plt.subplots_adjust(hspace=0.4,
wspace=0.3)
t      itanic  = sns.load_dataset("titanic")
# Step 3: Binned Age Distribution
axes[0, 0].hist(titanic['age'].dropna(), bins=20, edgecolor='black', color='skyblue')
axes[0, 0].set_xlabel('Age') axes[0, 0].set_ylabel('Frequency')
axes[0, 0].set_title('Binned Age Distribution')
# Step 4: Binned Fare Distribution
axes[0, 1].hist(titanic['fare'].dropna(), bins=20, edgecolor='black', color='salmon')
axes[0, 1].set_xlabel('Fare') axes[0, 1].set_ylabel('Frequency')
axes[0, 1].set_title('Binned Fare Distribution')
# Step 5: Age vs. Fare Scatter Plot
sns.scatterplot(x='age', y='fare', data=titanic, ax=axes[1, 0], alpha=0.6, color='green')
axes[1, 0].set_xlabel('Age')
 axes[1, 0].set_ylabel('Fare')
 axes[1, 0].set_title('Age vs. Fare Scatter Plot')
```

```
# Step 6: Categorical Binning - Class Bar Plot
sns.countplot(x='class', data=titanic, ax=axes[1, 1], palette='Set3') axes[1, 1].set_xlabel('Class')
axes[1, 1].set_ylabel('Frequency')
axes[1, 1].set_title('Categorical Binning - Class Bar Plot')
# Step 7: Adjust layout and display plots plt.tight_layout()
plt.show()
```

OUTPUT:



RESULT:

Thus the python program Binning visualizations allow us to group numerical data into intervals to understand distributions.is executed successfully

EX. NO: 7B IMPLEMENT LINEAR REGRESSION TECHNIQUES

AIM:

To implement linear regression techniques for a set of numerical data and visualize the regression line.

ALGORITHM:

Step 1: Import required libraries: numpy for numerical computations and matplotlib.pyplot for visualization.
Step 2: Define the coefficient estimation function estimate_coef(x, y):

- Step 2.1: Input independent variable x and dependent variable y.
- Step 2.2: Calculate the number of observations n.
- Step 2.3: Compute the mean of x and y as m_x and m_y.
- Step 2.4: Compute the cross-deviation SS_xy and deviation about x SS_xx.
- Step 2.5: Calculate regression coefficients:
  - Slope: b_1 = SS_xy / SS_xx
  - Intercept: b_0 = m_y - b_1 * m_x
- Step 2.6: Return (b_0, b_1) as coefficients.

Step 3: Define the plotting function plot_regression_line(x, y, b):

- Scatter plot the actual data points.
- Calculate predicted values y_pred = b_0 + b_1*x
- Plot the regression line.
- Label axes and display the plot.

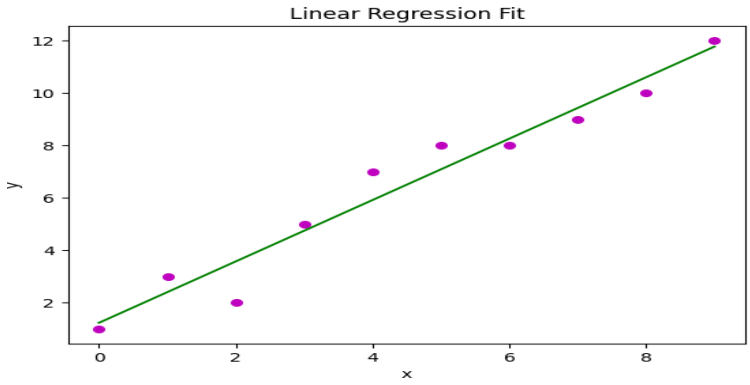
Step 4: Input the dataset (arrays x and y).
Step 5: Estimate coefficients using estimate_coef and plot the regression line.
Step 6: Stop.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
# Step 2: Function to estimate coefficients def
estimate_coef(x, y):
    n = np.size(x)
    m_x = np.mean(x)
    m_y = np.mean(y)
# cross-deviation and deviation about x SS_xy =
np.sum(y*x) - n*m_y*m_x
SS_xx = np.sum(x*x) - n*m_x*m_x
    # regression coefficients
b_1 = SS_xy / SS_xx
      b_0 = m_y - b_1*m_x
      return (b_0, b_1)
 Step 3: Function to plot regression line 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.title("Linear Regression Fit")
        plt.show()
# Step 4: Main function 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])
        # Estimate coefficients
        b = estimate_coef(x, y)
        print("Estimated coefficients:\nb_0 = {:.2f}\nb_1 =
{:.2f}".format(b[0], b[1]))
 # Plot regression line
    plot_regression_line(x, y, b)
   if _name_ == "_main_": main()
```

OUTPUT: Etimated coefficients: b_0=1.24 b_1 = 1.17



RESULT:

Thus the Python Program for linear regression model computes the best-fit line for the given dataset is executed successfully

EX. NO: 8A CLUSTERS ASSESSMENT - VISUALIZE THE CLUSTERS FOR ANY SYNTHETIC DATASET.

AIM:

To visualize clusters for any synthetic dataset and assess the clustering performance using silhouette score.

ALGORITHM

Step 1: Import necessary libraries: numpy, matplotlib.pyplot, sklearn.datasets, sklearn.cluster, and sklearn.metrics.

Step 2: Generate a synthetic dataset with make_blobs() specifying the number of samples, centers, standard deviation, and random state.

\Step 3: Visualize the original clusters using a scatter plot colored by true labels.

Step 4: Determine the optimal number of clusters using the Elbow Method:

Fit KMeans for n_clusters ranging from 1 to 10.

- Compute inertia (sum of squared distances from each point to its cluster center).

Step 5: Plot the Elbow graph to observe the point where the inertia starts decreasing slowly (elbow point).

Step 6: Choose the optimal number of clusters based on the elbow and perform KMeans clustering.

Step 7: Visualize the clustered data using a scatter plot colored by predicted cluster labels.

Step 8: Calculate the silhouette score to assess clustering quality:

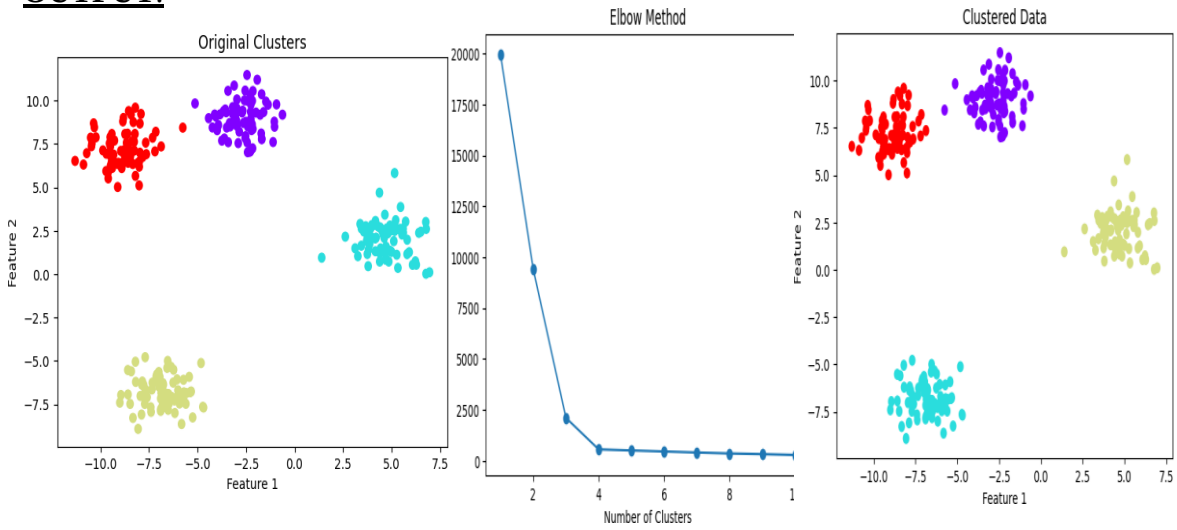
- Silhouette score ranges from -1 to 1.
- Higher score indicates well-separated clusters.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs from
sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
# Step 2: Generate a synthetic dataset
X, y = make_blobs(n_samples=300, centers=4, cluster_std=1.0, random_state=42)
# Step 3: Visualize the original clusters plt.scatter(X[:, 0], X[:, 1],
c=y, cmap='rainbow') plt.title("Original Clusters")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2") plt.show()
# Step 4: Elbow Method to find optimal clusters inertia = []
                for n_clusters in range(1, 11):
            kmeans = KMeans(n_clusters=n_clusters, random_state=42)
                kmeans.fit(X)
                inertia.append(kmeans.inertia_)
        # Step 5: Plot the Elbow Method graph
```

```
plt.plot(range(1, 11), inertia, marker='o')
plt.title("Elbow Method")
plt.xlabel("Number of Clusters")
plt.ylabel("Inertia")
plt.show()
# Step 6: Perform clustering with optimal number of clusters optimal_clusters =
kmeans = KMeans(n_clusters=optimal_clusters, random_state=42)
kmeans_labels = kmeans.fit_predict(X)
# Step 7: Visualize the clustered data
plt.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='rainbow') plt.title("Clustered
Data")
plt.xlabel("Feature 1")
plt.ylabel("Feature
2") plt.show()
# Step 8: Calculate silhouette score silhouette_avg =
silhouette_score(X, kmeans_labels) print("Silhouette
Score:", silhouette_avg)
```

OUTPUT:



Silhouette Score: 0.7915830011443039

RESULT:

Thus the synthetic dataset is successfully clustered into 4 clusters, the scatter plot shows clear separation of clusters. Silhouette score (~0.7) indicates that the clusters are well-defined and compact. This demonstrates unsupervised learning using KMeans and cluster assessment visually and quantitatively are implemented

EX. NO: 9A WRITE A PROGRAM TO IMPLEMENT THE AGGLOMERATIVE CLUSTERING TECHNIQUE

AIM

To implement the **Agglomerative Clustering technique** on a dataset and visualize the results.

ALGORITHM

Step 1: Import the required libraries:

- numpy for numerical operations.
- matplotlib.pyplot for visualization.
- load_iris from sklearn.datasets for dataset loading.
- AgglomerativeClustering from sklearn.cluster for clustering.
- PCA from sklearn.decomposition for dimensionality reduction.

Step 2: Load the **Iris dataset** using load_iris(). Store feature data in X.

Step 3: Perform **dimensionality reduction** with PCA:

- Create a PCA model with 2 components.
- Fit the PCA model on X and transform it into reduced dimensions **X_reduced.**

Step 4: Perform **Agglomerative Clustering**:

- Choose the number of clusters (n_clusters = 3).
- Create an AgglomerativeClustering model.
- Fit the model on the data and obtain cluster labels agg_labels.

Step 5: Visualize the Clusters:

- Use a scatter plot of X_reduced with points colored by their cluster labels.
- Add appropriate labels, title, and display the plot.

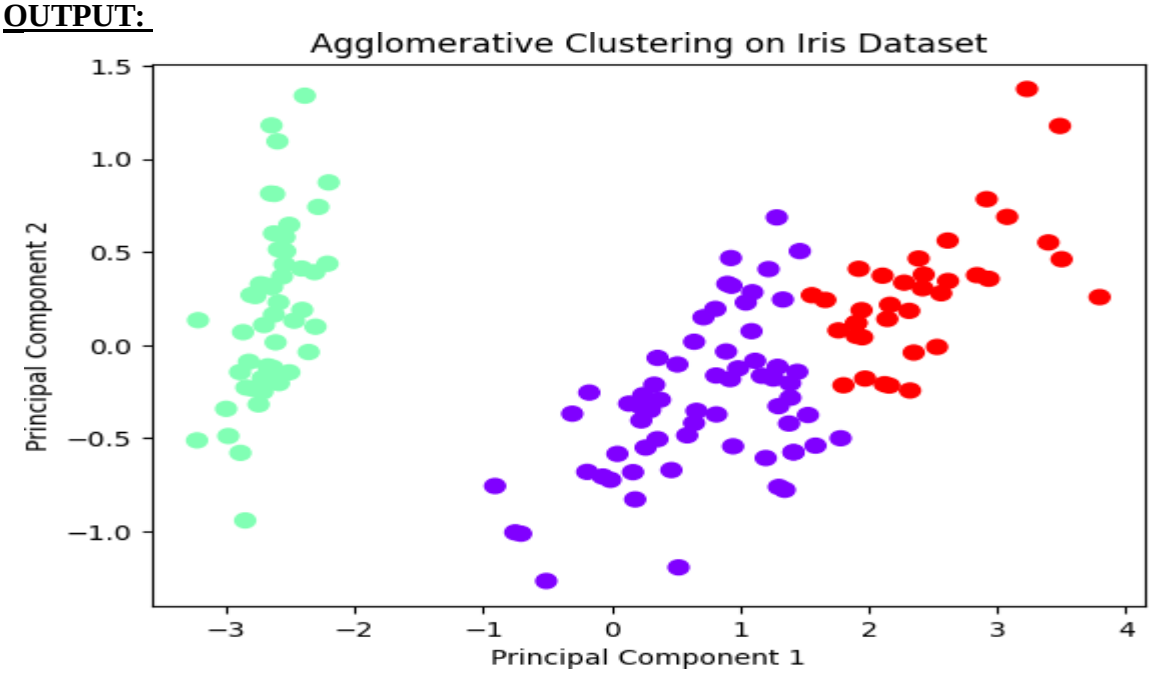
Step 6: End.

PROGRAM

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering from
sklearn.decomposition import PCA
# Step 2: Load the Iris dataset iris =
load_iris()
X = iris.data
# Step 3: Perform dimensionality reduction (PCA for 2D visualization) pca =
PCA(n_components=2)
X_reduced = pca.fit_transform(X)
# Step 4: Perform Agglomerative Clustering n_clusters = 3
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
agg_labels = agg_clustering.fit_predict(X)
# Step 5: Visualize the clusters
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=agg_labels,
cmap='rainbow', s=50)
plt.title("Agglomerative Clustering on Iris Dataset") plt.xlabel("Principal Component 1")
```

plt.ylabel("Principal Component 2") plt.show()

OUTPUT:



RESULT:

- The program successfully implements the **Agglomerative Clustering technique** and visualizes the clustered Iris dataset using PCA for 2D representation.

EX. NO: 9B WRITE A PROGRAM TO IMPLEMENT DIVISIVE HIERARCHICAL CLUSTERING TECHNIQUE

AIM

To write a program to implement the **Divisive Hierarchical Clustering (DIANA-like) technique** and visualize the clustering result using a dendrogram.

ALGORITHM

Step 1: Import the required libraries:
- numpy for numerical operations.
- matplotlib.pyplot for visualization.
- KMeans from sklearn.cluster for recursive splitting.
- dendrogram from scipy.cluster.hierarchy to plot the dendrogram.

Step 2: Create or load a dataset.
- In this example, a small **synthetic dataset** is created with 2D points.

Step 3: Define the **Divisive Clustering Function**:
- Start with all data points in one cluster.
- Use **KMeans (k=2)** to split the cluster into two sub-clusters.
- Recursively split until each cluster has a single point.
- Store results in a linkage-like matrix Z with the format [cluster1, cluster2, distance, sample_count].

Step 4: Compute **Distance Measure**:
- For each split, calculate the **Euclidean distance between centroids of the two sub-clusters.**

Step 5: Construct the **Linkage Matrix**:
- Maintain cluster indexing rules:
  - Original points: 0, 1, …, n-1
  - Formed clusters: n, n+1, …
- Append each merge step to Z.

Step 6: Plot the **Dendrogram** using scipy.cluster.hierarchy.dendrogram.

Step 7: Display the plot with proper labels and titles.

Step 8: End.

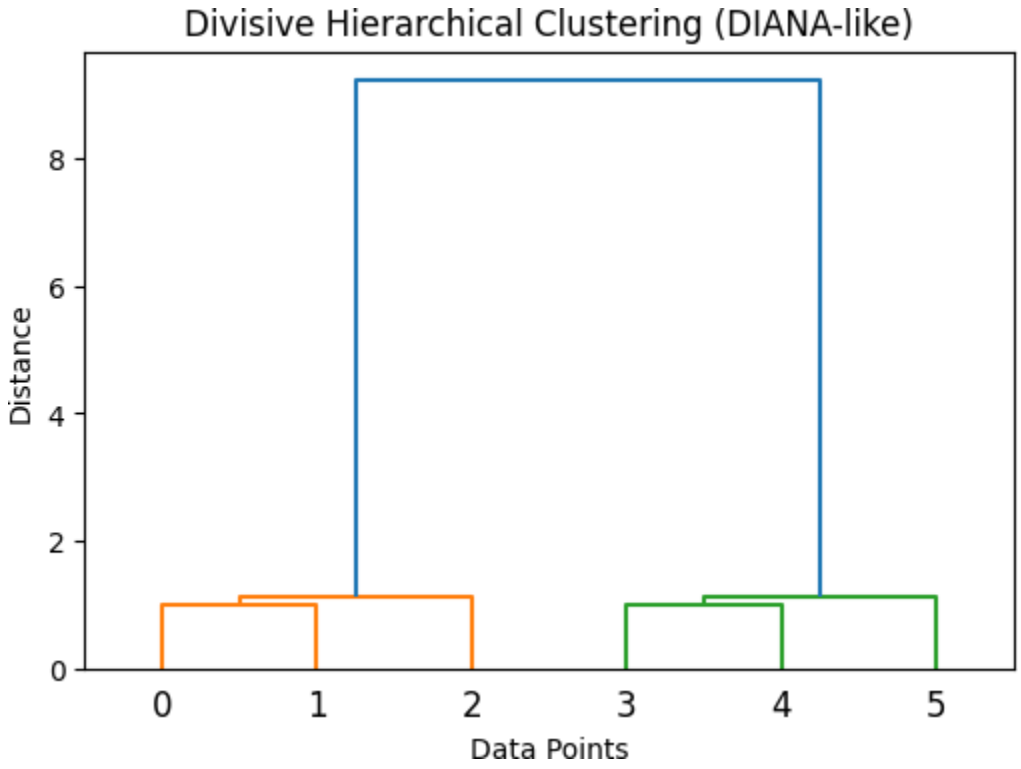
PROGRAM

```
import numpy as np
import matplotlib.pyplot as plt from
sklearn.cluster import KMeans
from scipy.cluster.hierarchy import dendrogram
# Synthetic dataset
X = np.array([[1, 2], [2, 2], [2, 3], [8, 8], [9, 8], [9, 9]])
# Recursive divisive clustering (DIANA-like)
def divisive_clustering(X, indices=None, next_cluster=None, Z=None):
    if indices is None:
```

```
        indices = list(range(len(X)))
    if Z is None:
        Z = []
            if next_cluster is None:

        next_cluster = len(X)

            # Stop if only one point

    if len(indices) <= 1:

        return Z, next_cluster, indices[0]

             # Split into 2 clusters

    kmeans = KMeans(n_clusters=2, random_state=42).fit(X[indices])

          labels = kmeans.labels_

    cluster1 = [indices[i] for i in range(len(indices)) if labels[i] == 0]

    cluster2 = [indices[i] for i in range(len(indices)) if labels[i] == 1]

    # Recursive splits

    Z, next_cluster, left = divisive_clustering(X, cluster1, next_cluster, Z)

    Z, next_cluster, right = divisive_clustering(X, cluster2, next_cluster, Z)

    # Define distance as centroid distance

    dist = np.linalg.norm(X[cluster1].mean(axis=0) -
X[cluster2].mean(axis=0))

    # Merge info

    Z.append([left, right, dist, len(indices)])

    current_cluster = next_cluster

    next_cluster += 1

    return Z, next_cluster, current_cluster

# Run divisive clustering
```

```
Z, _, _ = divisive_clustering(X)
# Convert to numpy array Z = np.array(Z)
# Plot dendrogram plt.figure(figsize=(6, 4))
dendrogram(Z, labels=list(range(len(X)))) plt.title("Divisive Hierarchical
Clustering (DIANA-like)") plt.xlabel("Data Points")
plt.ylabel("Distance")

plt.show()
```

OUTPUT:



RESULT:

- The above program successfully implements **Divisive Hierarchical Clustering (DIANA-like)** using recursive KMeans splitting.
- The dendrogram is plotted showing how clusters are recursively divided until each point forms its own cluster.

EX. NO: 10 DEVELOP SCALABLE CLUSTERING ALGORITHMS

AIM

To develop scalable clustering algorithms using Mini-Batch K-Means in Python, which can efficiently handle large datasets.

ALGORITHM

Step 1: Import Libraries

- Import numpy for numerical operations.
- Import matplotlib.pyplot for visualization.
- Import make_blobs (for synthetic dataset generation) from **sklearn.datasets.**
- Import MiniBatchKMeans from sklearn.cluster.

Step 2: Generate or Load Dataset

- Generate a **synthetic dataset** with a large number of samples (n_samples).
- Define the number of features (n_features) and number of clusters (n_clusters).

Step 3: Perform Mini-Batch K-Means

- Define the batch size (batch_size) for mini-batch updates.
- Define the number of initializations (n_init).
- Create a **MiniBatchKMeans** instance with n_clusters, batch_size, and **n_init.**

● Fit the model on the dataset and obtain predicted cluster labels (mbk_labels).

Step 4: Visualize Clustered Data

● Create a scatter plot of the data points with colors based on cluster labels.

● Label axes and add a suitable title.

● Display the plot.

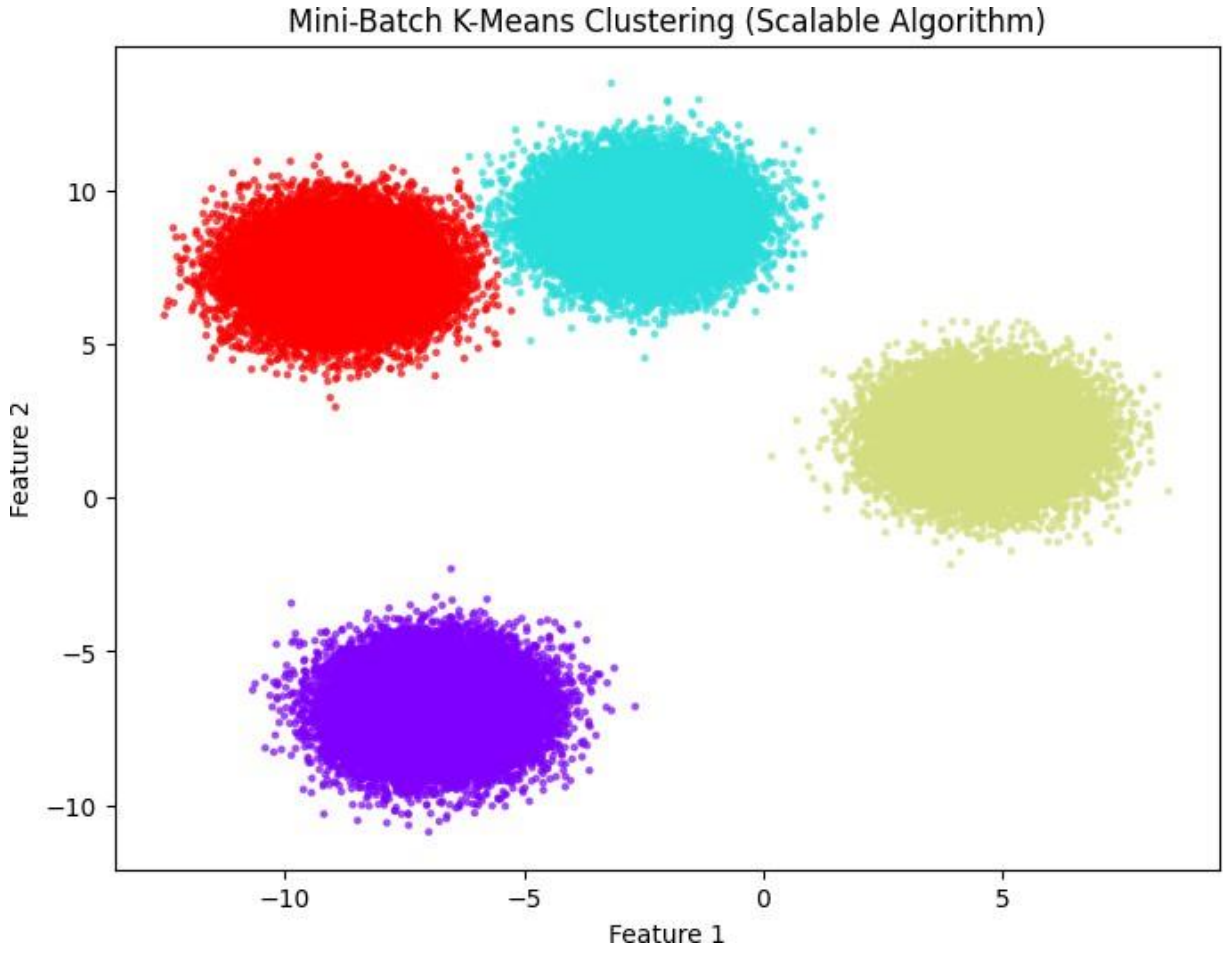
Step 5: End

PROGRAM

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs

from sklearn.cluster import MiniBatchKMeans

# Step 2: Generate a synthetic dataset with clusters n_samples
= 100000   # Large dataset for scalability n_features = 2

n_clusters = 4

X, y = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_clusters,
random_state=42)

# Step 3: Perform Mini-Batch K-Means clustering batch_size
= 1000

n_init = 10         # Number of random initializations

mbk = MiniBatchKMeans(n_clusters=n_clusters, batch_size=batch_size, n_init=n_init,
random_state=42)

mbk_labels = mbk.fit_predict(X)

# Step 4: Visualize the clustered data plt.figure(figsize=(8,6))
```

```
plt.scatter(X[:, 0], X[:, 1], c=mbk_labels, cmap='rainbow', s=5, alpha=0.6)
plt.title("Mini-Batch K-Means Clustering (Scalable Algorithm)") plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()
```

OUTPUT:



RESULT:

- The program successfully implements **Mini-Batch K-Means**, which is a **scalable clustering algorithm** suitable for large datasets.
  The scatter plot shows data points grouped into **4 clusters**, each identified by a unique color.