Experiment 1:

AIM: To Compute Central Tendency Measures: Mean, Median, Mode Measure of Dispersion: Variance, Standard Deviation.

Description: To compute central tendency measures:

- **Mean**: Sum of all values divided by the number of values.
- **Median**: Middle value when data is sorted; if even number of values, average the two middle numbers.
- **Mode**: Value that appears most frequently in the dataset.
- **Variance**: It is a statistical measure that indicates how much the data points in a dataset differ from the mean (average) of that dataset. It quantifies the degree of spread or dispersion in the data.
- **Standard deviation:** It is the square root of variance and provides a measure of the average distance of each data point from the mean, giving insight into how spreads out the data points are.

Program:

```
import numpy as np
from scipy import stats
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mean = np.mean(data)
median = np.median(data)
mode = stats.mode(data).mode

variance = np.var(data, ddof=0)

std_deviation = np.sqrt(variance)

print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {std_deviation}")
```

Output:

```
Mean: 5.5
Median: 5.5
Mode: 1
Variance: 8.25
Standard Deviation: 2.8722813232690143
```

Experiment 2:

AIM: To apply the following Pre-processing techniques for a given dataset.

- **a. Attribute selection**
- **b. Handling Missing Values**
- **c. Discretization**
- **d. Elimination of Outliers**

Description:

Data preprocessing: It is a process of preparing the raw data and making it suitable for a machine learning model.

1.Attribute Selection: Involves selecting a subset of relevant features (attributes) from the dataset. We can use various techniques such as correlation-based selection or statistical tests like Chi-square.

2.Handling Missing Values: We can handle missing values by either dropping rows or columns with missing data or filling them with some meaningful values (e.g., mean, median,or mode).

3.Discretization: This technique is used to convert continuous attributes into categorical ones, usually by binning.

4.Elimination of Outliers: We can eliminate outliers by defining a threshold (e.g., using the Z-score or IQR method) to remove extreme values.

Program:

```
import pandas as pd
import numpy as np

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, None, 35, 1000],  # Missing value and an outlier (100)
    'Salary': [50000, 54000, 58000, None, 59000],  # Missing value
    'City': ['NY', 'LA', 'NY', 'SF', 'LA']
}

df = pd.DataFrame(data)
```

```
print("Original Dataset:")
print(df)
```

Original Dataset:

```
+---------+--------+---------+------+
| Name    |  Age   | Salary  | City |
+---------+--------+---------+------+
| Alice   |  25.0  | 50000.0 | NY   |
| Bob     |  30.0  | 54000.0 | LA   |
| Charlie |   NaN  | 58000.0 | NY   |
| David   |  35.0  |   NaN   | SF   |
| Eve     | 1000.0 | 59000.0 | LA   |
+---------+--------+---------+------+
```

```
df_numerical = df[['Age', 'Salary']]  # Select only numerical attributes
print("\nSelected Attributes (Numerical Columns):")
print(df_numerical)
```

```
+--------+---------+
|  Age   | Salary  |
+--------+---------+
|  25.0  | 50000.0 |
|  30.0  | 54000.0 |
|   NaN  | 58000.0 |
|  35.0  |   NaN   |
|1000.0  | 59000.0 |
+--------+---------+
```

```
df['Age'] = df['Age'].fillna(df['Age'].mean())  # Fill missing values in 'Age' with mean
df['Salary'] = df['Salary'].fillna(df['Salary'].mean())  # Fill missing values in 'Salary' with mean
print("\nDataset after Handling Missing Values:")
print(df)
```

Dataset after Handling Missing Values:

```
    +---------+--------+---------+------+
| Name    |  Age   | Salary  | City |
+---------+--------+---------+------+
| Alice   |  25.0  | 50000.0 | NY   |
| Bob     |  30.0  | 54000.0 | LA   |
| Charlie | 272.5  | 58000.0 | NY   |
| David   |  35.0  | 55250.0 | SF   |
| Eve     |1000.0  | 59000.0 | LA   |
+---------+--------+---------+------+
```

```
age_bins = [0, 18, 35, 60, 120]  # Define bins for Age
age_labels = ['Teen', 'Young Adult', 'Adult', 'Senior']  # Define labels for bins


df['Age_Group'] = pd.cut(df['Age'], bins=age_bins, labels=age_labels)  #
Discretize Age
print("\nDataset after Discretization (Age Groups):")
```

Dataset after Discretization (Age Groups):

```
mean_age = df['Age'].mean()  # Calculate mean of Age
std_age = df['Age'].std()    # Calculate standard deviation of Age
df['Z_score_Age'] = (df['Age'] - mean_age) / std_age  # Z-score formula

df_no_outliers = df[np.abs(df['Z_score_Age']) <= 3]
print("\nDataset after Eliminating Outliers (Z-Score Method for Age):")
print(df_no_outliers)
```

Dataset after Eliminating Outliers (Z-Score Method for Age):

```
+---------+--------+---------+------+-------------+--------------+
| Name    |  Age   | Salary  | City | Age_Group   | Z_score_Age  |
+---------+--------+---------+------+-------------+--------------+
| Alice   |  25.0  | 50000.0 | NY   | Young Adult |   -0.589234  |
| Bob     |  30.0  | 54000.0 | LA   | Young Adult |   -0.577330  |
| Charlie | 272.5  | 58000.0 | NY   |     NaN     |    0.000000  |
| David   |  35.0  | 55250.0 | SF   | Young Adult |   -0.565426  |
| Eve     |1000.0  | 59000.0 | LA   |     NaN     |    1.731989  |
+---------+--------+---------+------+-------------+--------------+
```

print(df) # Final dataset after the data preprocessing

Output :

```
  +---------+--------+---------+------+-------------+--------------+
| Name    |  Age   | Salary  | City | Age_Group   | Z_score_Age |
+---------+--------+---------+------+-------------+--------------+
| Alice   |  25.0  | 50000.0 | NY   | Young Adult |   -0.589234  |
| Bob     |  30.0  | 54000.0 | LA   | Young Adult |   -0.577330  |
| Charlie | 272.5  | 58000.0 | NY   |     NaN     |    0.000000  |
| David   |  35.0  | 55250.0 | SF   | Young Adult |   -0.565426  |
| Eve     |1000.0  | 59000.0 | LA   |     NaN     |    1.731989  |
+---------+--------+---------+------+-------------+--------------+
```

Experiment 3:

AIM: To implement K-Nearest Neighbors (KNN) algorithm to classify instances based on their nearest neighbors in the feature space and Apply KNN for regression by predicting a continuous value based on the average of its nearest neighbors.

Description:

- The k-nearest neighbor algorithm is imported from the scikit-learn package, it will be supported for both classification and regression problems, In this we have taken fruits dataset as a example, for KNN classification we need to classify the fruit based on the features and the result will be taken by majority voting and for KNN regression we need to predict sweetness of the fruit as it will change continously based on fruits in this final result will be based on average.

Program:

3a. KNN Classifcation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

# Sample Fruits Dataset: [Weight, Sweetness, Label]
# Label: 0 = Apple, 1 = Orange, 2 = Banana
data = np.array([
   [150, 7, 0],   # Apple
   [160, 8, 0],   # Apple
   [170, 7, 0],   # Apple
   [130, 6, 1],   # Orange
   [140, 5, 1],   # Orange
   [135, 6, 1],   # Orange
   [180, 9, 2],   # Banana
   [190, 10, 2],  # Banana
   [200, 10, 2]   # Banana
])

# Features (Weight, Sweetness) and Labels (Fruit Type)
X = data[:, :-1]  # Features: Weight and Sweetness
Y = data[:, -1]   # Labels: 0 = Apple, 1 = Orange, 2 = Banana
```

```
# Query Point: New fruit to classify
query_point = np.array([[165, 8]])  # Weight: 165g, Sweetness: 8

k = 3  # Number of nearest neighbors

# Initialize KNN classifier with different distance metrics
metrics = ["euclidean", "manhattan", "cosine"]
fruit_classes = {0: "Apple", 1: "Orange", 2: "Banana"}  # Mapping of labels to
fruit names

for metric in metrics:
    knn = KNeighborsClassifier(n_neighbors=k, metric=metric)
    knn.fit(X, Y)
    distances, neighbors = knn.kneighbors(query_point)

    print(f"Metric: {metric.capitalize()}")
    print(f"Distances: {distances[0]}")

    # Display the top-k nearest neighbors with distances and their class labels
    top_k_neighbors = sorted(zip(distances[0], neighbors[0]), key=lambda x: x[0])
    top_k_with_labels = [(dist, idx, fruit_classes[Y[idx]]) for dist, idx in
top_k_neighbors]
    print(f"Top-k Neighbors: {top_k_with_labels}")

    # Predict the class of the query point
    prediction = knn.predict(query_point)[0]
    print(f"Prediction: {fruit_classes[prediction]} (Class {prediction})\n")

    # Plot the dataset and query point
    plt.figure(figsize=(8, 6))
    colors = ['r', 'g', 'b']  # Red = Apple, Green = Orange, Blue = Banana
    labels = ['Apple', 'Orange', 'Banana']

    for i in range(3):  # We have 3 classes
        class_data = X[Y == i]
        plt.scatter(class_data[:, 0], class_data[:, 1], color=colors[i], label=labels[i])

    # Plot the query point (black star)
    plt.scatter(query_point[0, 0], query_point[0, 1], color='black', marker='*', s=200,
label='Query Point')

    # Highlight the top k nearest neighbors
```

```
    for _, idx, label in top_k_with_labels:
        plt.scatter(X[idx, 0], X[idx, 1], color='yellow', edgecolors='black', s=100,
label=f'Top-k Neighbor ({label})')


plt.scatter(query_point[0, 0], query_point[0, 1], color=colors[prediction],
marker='*', s=200, label=f'Predicted: {labels[prediction]}')

    plt.title(f"KNN Classification with {metric.capitalize()} Distance")
    plt.xlabel("Weight (g)")
    plt.ylabel("Sweetness")
    plt.legend()
    plt.grid(True)
    plt.show()
```

Output :

```
Metric: Euclidean
Distances: [ 5.  5.09901951   15.03329638]
Top-k Neighbors: [(5.0, 1, 'Apple'), (5.0990195135927845, 2, 'Apple'),
(15.033296378372908, 0, 'Apple')]
Prediction: Apple (Class 0)
```

3b. KNN Regression

```
import numpy as np
from sklearn.neighbors import KNeighborsRegressor

# Sample Fruits Dataset: [Weight, Sweetness, Label]
# Label: 0 = Apple, 1 = Orange, 2 = Banana
data = np.array([
    [150, 7, 0],   # Apple: Weight 150g, Sweetness 7
    [160, 8, 0],   # Apple
    [170, 7, 0],   # Apple
    [130, 6, 1],   # Orange: Weight 130g, Sweetness 6
    [140, 5, 1],   # Orange
    [135, 6, 1],   # Orange
    [180, 9, 2],   # Banana: Weight 180g, Sweetness 9
    [190, 10, 2],  # Banana
    [200, 10, 2]   # Banana
])

# Features (Weight, Sweetness) and Labels (Fruit Type)
```

```
X = data[:, 0].reshape(-1, 1)  # Features: Weight (reshaped to be 2D)
y = data[:, 1]  # Target: Sweetness values


# Initialize KNN Regressor
k = 3  # Number of nearest neighbors
knn_regressor = KNeighborsRegressor(n_neighbors=k)

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

# Predict sweetness for a new fruit with weight 165g
new_fruit_weight = np.array([[165]])  # New fruit with weight 165g
predicted_sweetness = knn_regressor.predict(new_fruit_weight)

print(f"Predicted Sweetness for the new fruit (Weight: 165g):
{predicted_sweetness[0]}")
```

Output :

Predicted Sweetness for the new fruit (Weight: 165g): 7.333333333333333

Experiment 4:

AIM: To demonstrate decision tree algorithm for a classification problem and perform parameter tuning for better results.

Description:

This code first loads the play Tennis dataset, splits it into training and testing sets, and then defines a Decision Tree classifier. It uses Entropy and Information gain as impurity measures. Finally, it predicts the test set on the basis of weather conditons to play tennis or not usng the dataset and classifies it as decision tree.

Program:

```
# Importing Required Libraries
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree

# Reading the dataset (Tennis-dataset)
data = pd.read_csv('Play.csv')

# Function to Highlight Yes/No Values
def highlight(cell_value):
    '''
    Highlight yes / no values in the dataframe
    '''
    color_1 = 'background-color: pink;'
    color_2 = 'background-color: lightgreen;'

    if cell_value == 'no':
        return color_1
    elif cell_value == 'yes':
        return color_2

# Displaying the Data with Style
display(data.style.applymap(highlight)
    .set_properties(subset=data.columns, **{'width': '100px'})
    .set_table_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'),
('border', '1px solid gray') ('font-weight', 'bold')]},
```

```
    {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid
black')]}]))

# Function to Calculate Entropy
def find_entropy(data):
    """
    Returns the entropy of the class or features
    formula: - ∑ P(X)logP(X)
    """
    entropy = 0
    for i in range(data.nunique()):
        x = data.value_counts()[i]/data.shape[0]
        entropy += (- x * math.log(x,2))
    return round(entropy,3)

# Function to Calculate Information Gain
def information_gain(data, data_):
    """
    Returns the information gain of the features
    """
    info = 0
    for i in range(data_.nunique()):
        df = data[data_ == data_.unique()[i]]
        w_avg = df.shape[0]/data.shape[0]
        entropy = find_entropy(df.play)
        x = w_avg * entropy
        info += x
    ig = find_entropy(data.play) - info
    return round(ig, 3)

# Function to Calculate Entropy and Information Gain for Each Feature
def entropy_and_infogain(datax, feature):
    """
    Grouping features with the same class and computing their
    entropy and information gain for splitting
    """
    for i in range(data[feature].nunique()):
        df = datax[datax[feature]==data[feature].unique()[i]]
        if df.shape[0] < 1:
            continue

        display(df[[feature, 'play']].style.applymap(highlight)
```

```
            .set_properties(subset=[feature, 'play'], **{'width':
'80px'})  .set_table_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'),
                                          ('border', '1px solid gray'),
                                          ('font-weight', 'bold')]},
                            {'selector': 'td', 'props': [('border', '1px solid gray')]},
                            {'selector': 'tr:hover', 'props': [('background-color', 'white'),
                                              ('border', '1.5px solid black')]}]))

      print(f'Entropy of {feature} - {data[feature].unique()[i]} =
{find_entropy(df.play)}')
   print(f'Information Gain for {feature} = {information_gain(datax,
datax[feature])}')

# Calculating Entropy and Information Gain for All Features
print(f'Entropy of the entire dataset: {find_entropy(data.play)}')
entropy_and_infogain(data, 'outlook')
entropy_and_infogain(data, 'temp')
entropy_and_infogain(data, 'humidity')
entropy_and_infogain(data, 'windy')

# Entropy and Information Gain for Subsets
sunny = data[data['outlook'] == 'sunny']
print(f'Entropy of the Sunny dataset: {find_entropy(sunny.play)}')
entropy_and_infogain(sunny, 'temp')
entropy_and_infogain(sunny, 'humidity')
entropy_and_infogain(sunny, 'windy')

rainy = data[data['outlook'] == 'rainy']
print(f'Entropy of the Rainy dataset: {find_entropy(rainy.play)}')
entropy_and_infogain(rainy, 'temp')
entropy_and_infogain(rainy, 'humidity')
entropy_and_infogain(rainy, 'windy')

# Preparing the data for DecisionTreeClassifier
X = pd.get_dummies(data.drop(columns=['play']))  # One-hot encoding for
categorical features
y = data['play']

# Training the Decision Tree Classifier
clf = DecisionTreeClassifier(criterion='entropy')
clf.fit(X, y)

# Visualizing the Decision Tree Classifier
```

```
plt.figure(figsize=(20,10))
plot_tree(clf, feature_names=X.columns, class_names=clf.classes_, filled=True)
plt.show()
```

Output :

	outlook	temp	humidity	windy	play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes
5	rainy	cool	normal	True	no
6	overcast	cool	normal	True	yes
7	sunny	mild	high	False	no
8	sunny	cool	normal	False	yes
9	rainy	mild	normal	False	yes
10	sunny	mild	normal	True	yes
11	overcast	mild	high	True	yes
12	overcast	hot	normal	False	yes
13	rainy	mild	high	True	no

Entropy of the entire dataset: 0.94

	outlook	play
0	sunny	no
1	sunny	no
7	sunny	no
8	sunny	yes
10	sunny	yes

Entropy of outlook - sunny = 0.971

	outlook	play

	outlook	play
2	overcast	yes
6	overcast	yes
11	overcast	yes
12	overcast	yes

Entropy of outlook - overcast = 0.0

	outlook	play
3	rainy	yes
4	rainy	yes
5	rainy	no
9	rainy	yes
13	rainy	no

Entropy of outlook - rainy = 0.971
Information Gain for outlook = 0.246

	temp	play
0	hot	no
1	hot	no
2	hot	yes
12	hot	yes

Entropy of temp - hot = 1.0

	temp	play
3	mild	yes
7	mild	no
9	mild	yes
10	mild	yes
11	mild	yes
13	mild	no

Entropy of temp - mild = 0.918

	temp	play
4	cool	yes
5	cool	no
6	cool	yes
8	cool	yes

Entropy of temp - cool = 0.811
Information Gain for temp = 0.029

	humidity	play
0	high	no
1	high	no
2	high	yes
3	high	yes
7	high	no
11	high	yes
13	high	no

Entropy of humidity - high = 0.985

	humidity	play
4	normal	yes
5	normal	no
6	normal	yes
8	normal	yes
9	normal	yes
10	normal	yes
12	normal	yes

Entropy of humidity - normal = 0.592
Information Gain for humidity = 0.151

	windy	play
0	False	no
2	False	yes
3	False	yes
4	False	yes
7	False	no
8	False	yes
9	False	yes
12	False	yes

Entropy of windy - False = 0.811

	windy	play
1	True	no
5	True	no
6	True	yes
10	True	yes
11	True	yes
13	True	no

Entropy of windy - True = 1.0
Information Gain for windy = 0.048
Entropy of the Sunny dataset: 0.971

	temp	play
0	hot	no
1	hot	no

Entropy of temp - hot = 0.0

	temp	play
7	mild	no
10	mild	yes

Entropy of temp - mild = 1.0

	temp	play
8	cool	yes

Entropy of temp - cool = 0.0
Information Gain for temp = 0.571

	humidity	play
0	high	no
1	high	no
7	high	no

Entropy of humidity - high = 0.0

	humidity	play
8	normal	yes
10	normal	yes

Entropy of humidity - normal = 0.0
Information Gain for humidity = 0.971

	windy	play
0	False	no
7	False	no
8	False	yes

Entropy of windy - False = 0.918

	windy	play
1	True	no
10	True	yes

Entropy of windy - True = 1.0
Information Gain for windy = 0.02
Entropy of the Rainy dataset: 0.971

	temp	play
3	mild	yes
9	mild	yes
13	mild	no

Entropy of temp - mild = 0.918

	temp	play
4	cool	yes
5	cool	no

Entropy of temp - cool = 1.0
Information Gain for temp = 0.02

	humidity	play
3	high	yes
13	high	no

Entropy of humidity - high = 1.0

	humidity	play
4	normal	yes
5	normal	no
9	normal	yes

Entropy of humidity - normal = 0.918

Information Gain for humidity = 0.02

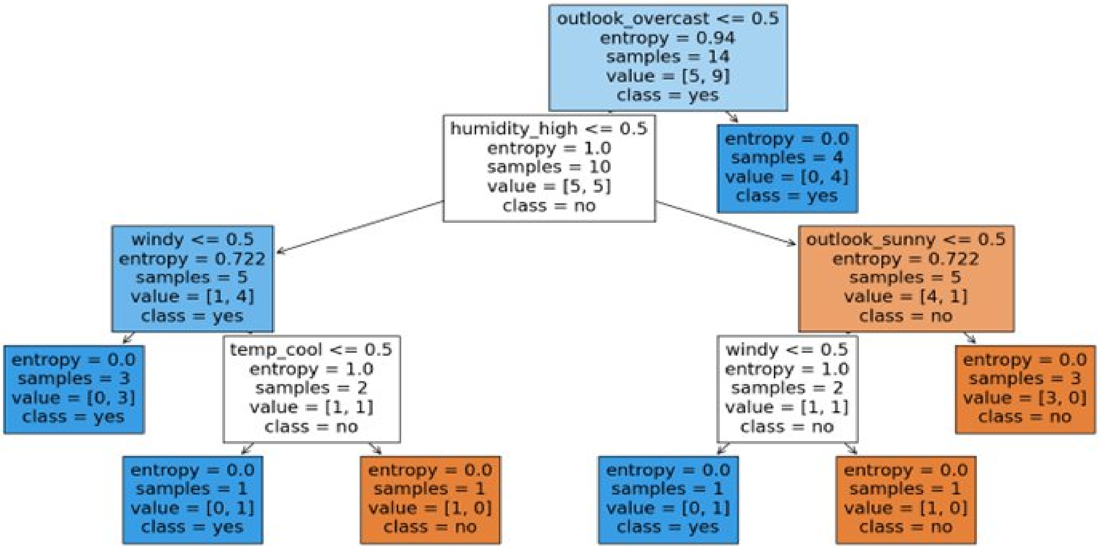
	windy	play
3	False	yes
4	False	yes
9	False	yes

Entropy of windy - False = 0.0

	windy	play
5	True	no
13	True	no

Entropy of windy - True = 0.0

Information Gain for windy = 0.971



Experiment 5:

AIM: Demonstrate decision tree algorithm for predicting continuous values in a regressionproblem using a suitable dataset.

Description:

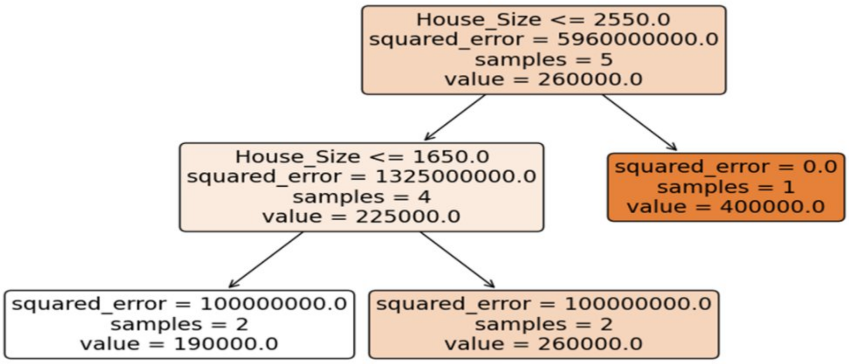
In this program we have taken house pricing data, splits it into training and test sets, trains a decision tree regressor, makes predictions on new samples based on Decision tree.

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree

# Step 1: Creating the Dataset
data = {
    'House_Size': [1500, 1800, 1200, 2100, 3000],
    'Bedrooms': [3, 4, 2, 4, 5],
    'Age': [10, 5, 15, 8, 2],
    'Price': [200000, 250000, 180000, 270000, 400000]  # Target variable
}

df = pd.DataFrame(data)

# Step 2: Define Features (X) and Target (y)
X = df[['House_Size']]  # Using only 'House_Size' for simplicity
y = df['Price']

# Step 3: Train the Decision Tree Regressor
regressor = DecisionTreeRegressor(max_depth=2)  # Limit depth for better
visualization
regressor.fit(X, y)

# Step 4: Visualizing the Decision Tree
plt.figure(figsize=(10, 6))
plot_tree(regressor, feature_names=['House_Size'], filled=True, rounded=True)
plt.show()
```

```
# Step 5: Making Predictions
house_sizes = np.array([1600, 2200, 2500]).reshape(-1, 1)
predictions = regressor.predict(house_sizes)

# Display Predictions
for size, price in zip(house_sizes.flatten(), predictions):
    print(f"Predicted price for {size} sq. ft. house: ${price:.2f}")
```

Output:



Predicted price for 1600 sq. ft. house: $190000.00
Predicted price for 2200 sq. ft. house: $260000.00
Predicted price for 2500 sq. ft. house: $260000.00

Experiment 6:
AIM: To Apply Random Forest algorithm for classification and regression
a) AIM: To Apply Random Forest algorithm for classification

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import plot_tree
from sklearn.metrics import accuracy_score

# Step 1: Create Dataset
data = pd.read_csv('Random forest.csv')

# Step 2: Encode Categorical Variables
color_encoder = LabelEncoder()
data['Color'] = color_encoder.fit_transform(data['Color'])  # Encode 'Red', 
'Yellow', 'Green' as 0, 1, 2

fruit_encoder = LabelEncoder()
data['Fruit'] = fruit_encoder.fit_transform(data['Fruit'])  # Encode 'Apple' (0), 
'Banana' (1), 'Watermelon' (2)

# Step 3: Split Data
X = data[['Weight', 'Color', 'Size']]
y = data['Fruit']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, 
random_state=42)

# Print Class Distribution
print("\nClass Distribution in Training Set:")
print(pd.Series(y_train).value_counts())

print("\nClass Distribution in Testing Set:")
print(pd.Series(y_test).value_counts())
```

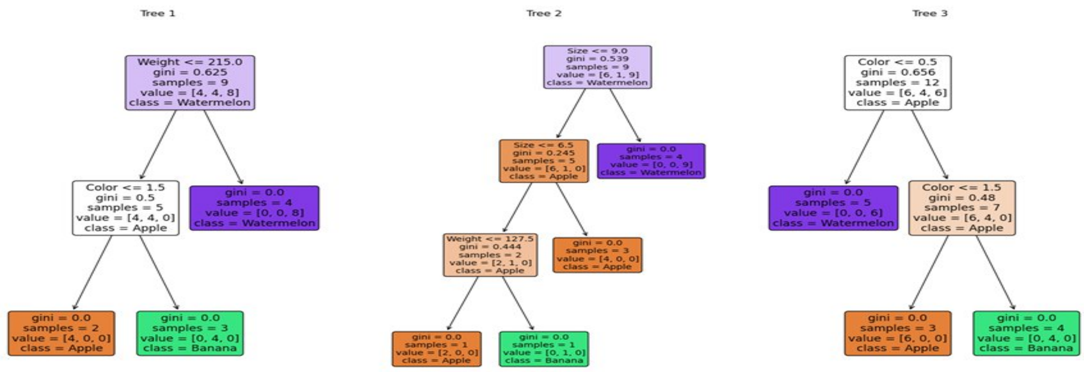
```
# Step 4: Train Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Step 5: Compute Model Accuracy
y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'\nAccuracy: {accuracy * 100:.2f}%')

# Step 6: Visualize 3 Trees from Random Forest
plt.figure(figsize=(20, 10))

for i in range(3):  # First 3 trees
    plt.subplot(1, 3, i + 1)
    plot_tree(rf_model.estimators_[i], feature_names=X.columns,
class_names=fruit_encoder.classes_, filled=True, rounded=True)
    plt.title(f"Tree {i+1}")

plt.show()

# Step 7: Majority Voting - Predict a New Sample
new_fruit = pd.DataFrame({'Weight': [160], 'Color':
[color_encoder.transform(['Red'])[0]], 'Size': [7]})

# Get predictions from first 3 trees
tree_predictions = [tree.predict(new_fruit)[0] for tree in rf_model.estimators_[:3]]

# Perform majority voting
final_prediction = np.bincount(tree_predictions).argmax()

# Decode the result back to fruit name
predicted_fruit = fruit_encoder.inverse_transform([final_prediction])[0]

print(f"\nPredicted Fruit (Majority Voting from 3 Trees): {predicted_fruit}")

# Step 8: Display Training Samples Used in Trees
train_samples = X_train.copy()
train_samples['Fruit'] = y_train
print("\nTraining Samples Used in Trees:")
print(train_samples.sort_values(by=['Weight', 'Size']))  # Sorted for easy analysis
```

Output :

Class Distribution in Training Set:
1   6
0   5
2   5
Name: Fruit, dtype: int64

Class Distribution in Testing Set:
0   2
2   1
1   1
Name: Fruit, dtype: int64

Accuracy: 100.00%



Predicted Fruit (Majority Voting from 3 Trees): Apple

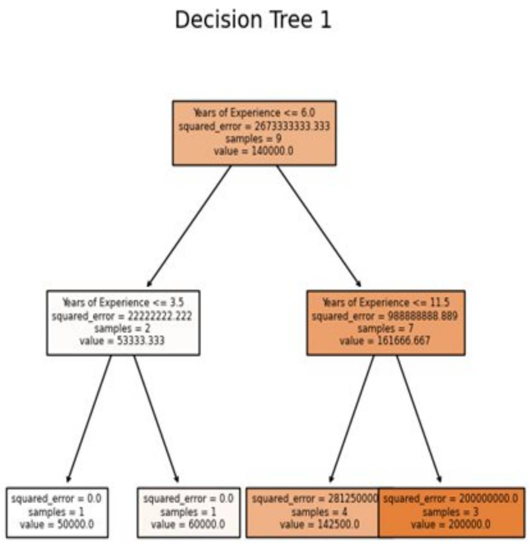
6b) AIM: To Apply Random Forest algorithm for regression

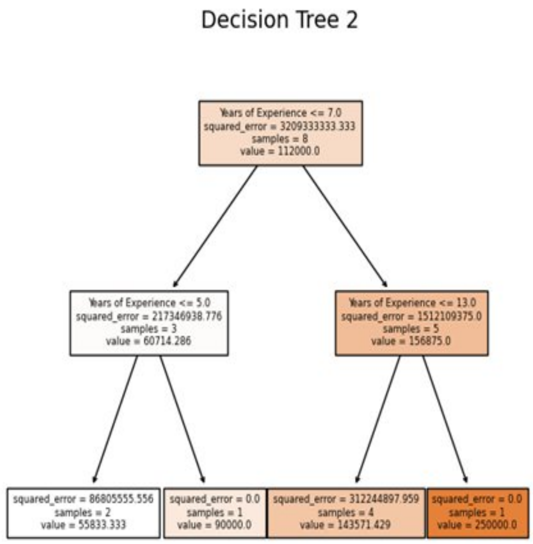
Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import plot_tree

# Creating dataset: Years of Experience vs. Salary

# Step 1: Create Dataset
data = pd.read_csv('Salary.csv')
```

```
# Feature (Years of Experience) and Target (Salary)
X = data[['Years of Experience']]
y = data['Salary ($)']

# Creating Random Forest with only 2 trees and max depth of 2
rf_regressor = RandomForestRegressor(n_estimators=2, max_depth=2,
random_state=42)
rf_regressor.fit(X, y)

# Visualizing the 2 Decision Trees with max depth 2
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14, 6))  # Adjust figure size

for i in range(2):  # Plotting only 2 trees
    plot_tree(rf_regressor.estimators_[i],
              feature_names=["Years of Experience"],
              filled=True,
              ax=axes[i],
              fontsize=6)  # Font size for readability
    axes[i].set_title(f"Decision Tree {i+1}", fontsize=14)  # Title font size

plt.show()

# New sample for prediction (7.5 years of experience)
new_sample = np.array([[7.5]])

# Get predictions from both trees
tree_predictions = [tree.predict(new_sample)[0] for tree in
rf_regressor.estimators_]

# Compute the average prediction
average_prediction = np.mean(tree_predictions)

# Print each tree's prediction and the final averaged output
print("Predictions from individual trees:", tree_predictions)
print(f"Final Averaged Prediction: ${average_prediction:,.2f}")
```

Output :





Predictions from individual trees: [142500.0, 143571.42857142858]
Final Averaged Prediction: $143,035.71

Experiment 7:

AIM: Demonstrate Naïve Bayes Classification algorithm.

Program:

```
import nltk
from nltk.corpus import movie_reviews
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Step 1: Download the dataset
nltk.download('movie_reviews')

# Step 2: Load IMDB movie reviews dataset
reviews = []
labels = []

for category in movie_reviews.categories():  # 'pos' and 'neg'
    for fileid in movie_reviews.fileids(category):
        reviews.append(movie_reviews.raw(fileid))  # Load review text
        labels.append(1 if category == 'pos' else 0)  # 1 = Positive, 0 = Negative

# Step 3: Convert text to numerical features (TF-IDF)
vectorizer = TfidfVectorizer(stop_words="english", max_features=5000)  # Limit
vocabulary size
X = vectorizer.fit_transform(reviews)  # Transform reviews into TF-IDF vectors

# Step 4: Train-Test Split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2,
random_state=42, shuffle=True)

# Step 5: Train Naïve Bayes classifier
classifier = MultinomialNB()
classifier.fit(X_train, y_train)

# Step 6: Evaluate Model
y_pred = classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

```
print(f'Accuracy: {accuracy * 100:.2f}')

# Step 7: Test with a new review
new_review = ["The movie was not much good, waste of time to watch"]
new_review_vectorized = vectorizer.transform(new_review)
prediction = classifier.predict(new_review_vectorized)

print('Sentiment Prediction:', 'Positive' if prediction[0] == 1 else 'Negative')

# Downloading nltk package
[nltk_data] Downloading package movie_reviews to
[nltk_data] C:\Users\SRAVANTHI\AppData\Roaming\nltk_data...
[nltk_data] Package movie_reviews is already up-to-date!
```

Output :

Accuracy: 79.50
Sentiment Prediction: Negative

Experiment 8:
AIM: To Apply Support Vector algorithm for classification

Program:

```
import numpy as np
from sklearn import svm

# Sample Data (Features: [free, money, discount, length])
X = np.array([[5, 3, 1, 120],  # Spam
              [0, 0, 0, 150],  # Not Spam
              [8, 5, 2, 200],  # Spam
              [0, 0, 1, 180],  # Not Spam
              [2, 1, 0, 130],  # Spam
              [0, 0, 0, 170]]) # Not Spam

# Labels (1 for Spam, -1 for Not Spam)
y = np.array([1, -1, 1, -1, 1, -1])

# Create a linear SVM model
model = svm.SVC(kernel='linear')

# Train the model
model.fit(X, y)

# Extract weights (w) and bias (b)
weights = model.coef_[0]  # Extract the weights (coefficients)
bias = model.intercept_[0]  # Extract the bias

# Print weights and bias
print("Weights (w):", weights)
print("Bias (b):", bias)

# New email for prediction
new_email = np.array([[4, 2, 1, 140]])  # Features of new email

# Predict whether it's spam (1) or not spam (-1)
prediction = model.predict(new_email)

# Use the decision function formula manually for demonstration
decision_value = np.dot(weights, new_email[0]) + bias  # w · x + b
```

```
print(f"Prediction for new email: {'Spam' if prediction == 1 else 'Not Spam'}")
print("Decision value (f(x)):", decision_value)
```

Output :

```
Weights (w): [ 0.39700193  0.22866404  0.06032616 -0.04881886]
Bias (b): 6.323465308186336
Prediction for new email: Spam
Decision value (f(x)): 1.5944871609081197
```

Experiment 9:

AIM: Implement simple linear regression to model the relationship between one dependent variable and one independent variable.

Program:

```
import numpy as np  # Used for multi-dimensional array storage
import pandas as pd  # Used for tabular data representation
from sklearn.model_selection import train_test_split  # To split data into training and testing sets
from sklearn.linear_model import LinearRegression  # Linear regression model
from sklearn.metrics import mean_absolute_error, mean_squared_error  # Evaluation metrics

# 1. Create a Dataset (Example Data)
data = {
    'Size (sq ft)': [1500, 2000, 2500, 1800, 2200],
    'Bedrooms': [3, 4, 5, 3, 4],
    'Location_Score': [8, 9, 7, 8, 9],  # Arbitrary scores for location quality
    'Price': [300000, 400000, 500000, 350000, 450000]
}
df = pd.DataFrame(data)

# 2. Define Features (X) and Target (y)
X = df[['Size (sq ft)', 'Bedrooms', 'Location_Score']]  # Input features
y = df['Price']  # Target variable

# 3. Split Dataset 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)

# 4. Train a Supervised Learning Model (Linear Regression)
model = LinearRegression()
model.fit(X_train, y_train)

# 5. Make Predictions
y_pred = model.predict(X_test)
```

```
# 6. Evaluate the Model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)  # RMSE is the square
root of MSE

# Display Performance Metrics
print("Mean Absolute Error (MAE):", mae)
print("Mean Squared Error (MSE):", mse)
print("Root Mean Squared Error (RMSE):", rmse)

# 7. Display Predictions vs Actual Values
print("Predicted Prices:", y_pred)
print("Actual Prices:", y_test.values)
```

Output :

```
Mean Absolute Error (MAE): 16666.666666666628
Mean Squared Error (MSE): 277777777.7777765
Root Mean Squared Error (RMSE): 16666.666666666628
Predicted Prices: [416666.66666667]
Actual Prices: [400000]
```

Experiment 10:

AIM: To apply logistic regression to model binary or multi-class classification problems by predicting probabilities of class membership.

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Create Dataset
data = {
    'Study Hours': [1, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 8, 9, 10],
    'Previous Scores': [40, 50, 55, 50, 60, 65, 70, 72, 75, 78, 80, 85, 90, 92, 95],
    'Pass': [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]  # 0 = Fail, 1 = Pass
}

df = pd.DataFrame(data)

# Split Data
X = df[['Study Hours', 'Previous Scores']]
y = df['Pass']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# train Model
model = LogisticRegression()
model.fit(X_train, y_train)

# Get Weights (Coefficients) and Bias (Intercept)
bias = model.intercept_[0]
weights = model.coef_[0]

print(f"Bias (Intercept): {bias:.4f}")
print(f"Weight for 'Study Hours': {weights[0]:.4f}")
print(f"Weight for 'Previous Scores': {weights[1]:.4f}")
```

```
# Predictions & Accuracy
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'\n Accuracy: {accuracy * 100:.2f}%')

# Predict for New Data
new_data = np.array([[6, 80], [2, 45], [8, 88]])
new_predictions = model.predict(new_data)

for i, pred in enumerate(new_predictions):
    status = "Pass" if pred == 1 else "Fail"
    print(f"Student {i+1}: Study Hours={new_data[i][0]}, Previous
Scores={new_data[i][1]} → Prediction: {status}")
```

Output:

```
Bias (Intercept): -46.3849
Weight for 'Study Hours': 0.0759
Weight for 'Previous Scores': 0.7377

 Accuracy: 100.00%
Student 1: Study Hours=6, Previous Scores=80 → Prediction: Pass
Student 2: Study Hours=2, Previous Scores=45 → Prediction: Fail
Student 3: Study Hours=8, Previous Scores=88 → Prediction: Pass
```

Experiment 11:

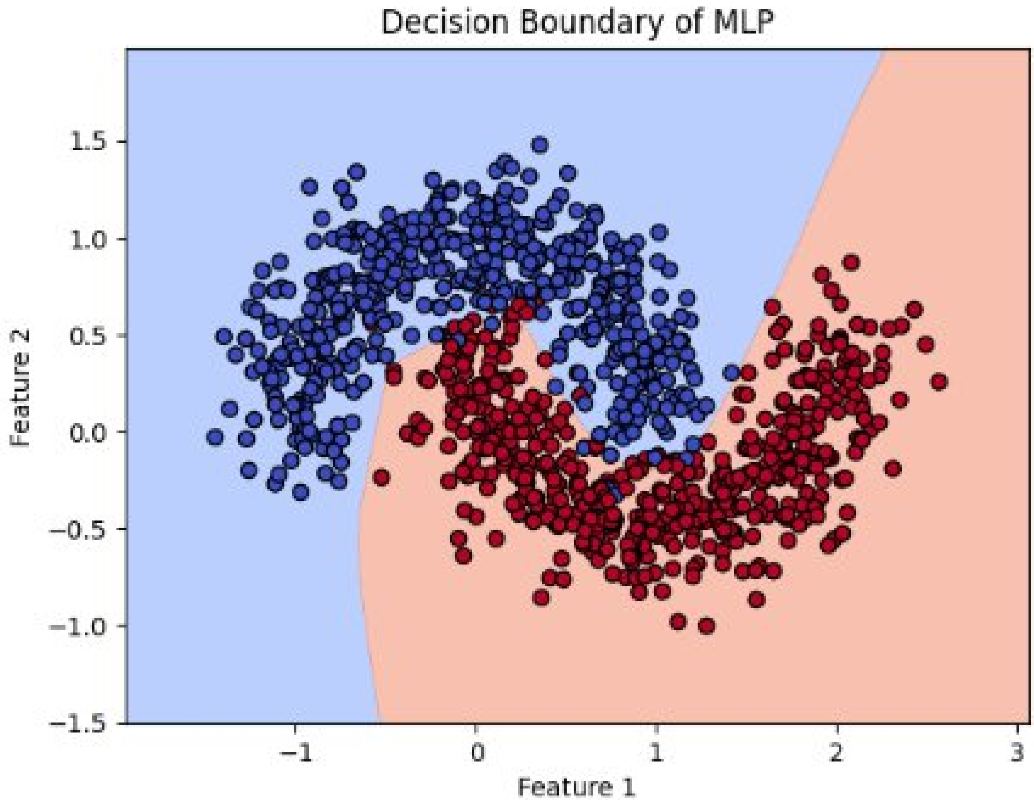
AIM: Implement a Multi-layer Perceptron (MLP), a type of artificial neural network, for complex non-linear classification tasks using backpropagation for training.

Program :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from tensorflow import keras
from tensorflow.keras import layers

# Generate non-linearly separable dataset (Moons Dataset)
X, y = make_moons(n_samples=1000, noise=0.2, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Build an MLP model for classification
model = keras.Sequential([
layers.Dense(128, activation='relu', input_shape=(2,)), # Hidden Layer 1
layers.Dense(64, activation='relu'), # Hidden Layer 2
layers.Dense(1, activation='sigmoid') # Output Layer (Binary Classification)
])

# Compile the model (Binary Crossentropy loss & Adam optimizer)
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train the model using backpropagation
```

```
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test), verbose=1)
# Plot decision boundary
def plot_decision_boundary(model, X, y):
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max,
100))

Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, levels=[0, 0.5, 1], cmap=plt.cm.coolwarm, alpha=0.6)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, edgecolors='k')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Decision Boundary of MLP")
plt.show()

# Visualize decision boundary
plot_decision_boundary(model, X, y)
# Evaluate performance on test set
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print(f"\nTest Accuracy: {test_acc *100:.4f}")
```

Output:




```
7/7 - 0s - 9ms/step - accuracy: 0.9850 - loss: 0.0479

Test Accuracy: 98.5000
```

Experiment 12:

AIM: Implement a Multi-layer Perceptron (MLP), a type of artificial neural network, for complex non-linear classification tasks using backpropagation for training.

Program :

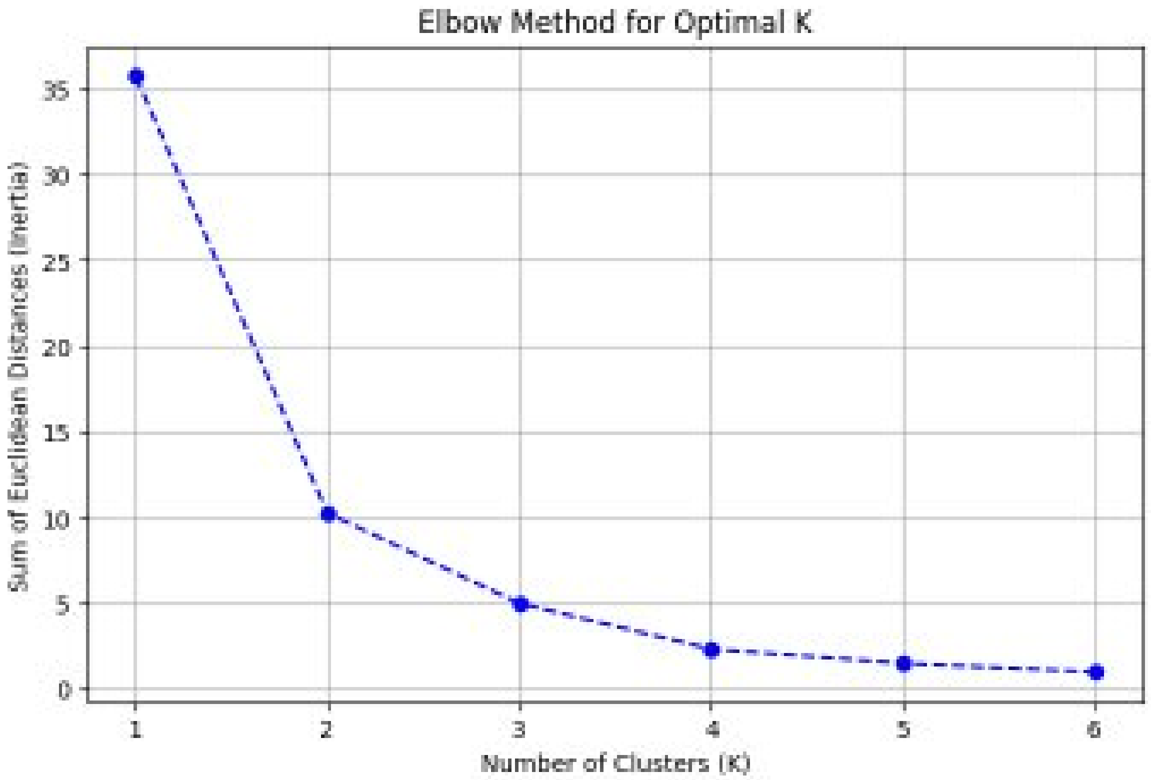
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import LabelEncoder
from mpl_toolkits.mplot3d import Axes3D

# Step 1: Define the dataset
fruit_data = pd.DataFrame({
   'Fruit': ['Apple', 'Orange', 'Banana', 'Mango', 'Grapes', 'Watermelon', 'Pineapple', 
'Strawberry'],
   'Color': ['Red', 'Orange', 'Yellow', 'Yellow', 'Purple', 'Green', 'Brown', 'Red'],
   'Size': ['Medium', 'Medium', 'Long', 'Medium', 'Small', 'Large', 'Large', 'Small'],
   'Texture': ['Smooth', 'Rough', 'Smooth', 'Smooth', 'Smooth', 'Rough', 'Rough', 
'Smooth']
})

# Step 2: Convert categorical values into numerical values
label_enc = LabelEncoder()
fruit_data['Color'] = label_enc.fit_transform(fruit_data['Color'])
fruit_data['Size'] = label_enc.fit_transform(fruit_data['Size'])
fruit_data['Texture'] = label_enc.fit_transform(fruit_data['Texture'])
```

```
# Step 3: Select features for clustering
X = fruit_data[['Color', 'Size', 'Texture']]

#ans for different K values
max_k = 6 #Function to compute clustering and inertia for different K values
def evaluate_kmeans(X, max_k=6):
    inertias = []
    for k in range(1, max_k + 1):
        kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
        kmeans.fit(X)
        inertias.append(kmeans.inertia_)  # Sum of squared distances to cluster
centers
    return inertias

# Step 4: Apply K-Means
inertias = evaluate_kmeans(X, max_k)

# Step 5: Plot inertia vs. K to find the optimal number of clusters
plt.figure(figsize=(8, 5))
plt.plot(range(1, max_k + 1), inertias, marker='o', linestyle='--', color='b')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Sum of Euclidean Distances (Inertia)')
plt.title('Elbow Method for Optimal K')
plt.xticks(range(1, max_k + 1))
plt.grid(True)
plt.show()
```

```
# Step 6: Apply K-Means with optimal K
optimal_k = 3  # Choose based on elbow plot
kmeans_optimal = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
fruit_data['Cluster'] = kmeans_optimal.fit_predict(X)

# Step 7: Display clusters
print("\nClustered Fruits:")
for cluster in sorted(fruit_data['Cluster'].unique()):
    print(f"\nCluster {cluster}:")
    print(fruit_data[fruit_data['Cluster'] == cluster][['Fruit', 'Color', 'Size',
'Texture']])

# Step 8: 3D Visualization of Clusters
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for each cluster
scatter = ax.scatter(
    fruit_data['Color'],
    fruit_data['Size'],
    fruit_data['Texture'],
    c=fruit_data['Cluster'],
    cmap='rainbow',
    s=150,
    edgecolors='k'
)
```

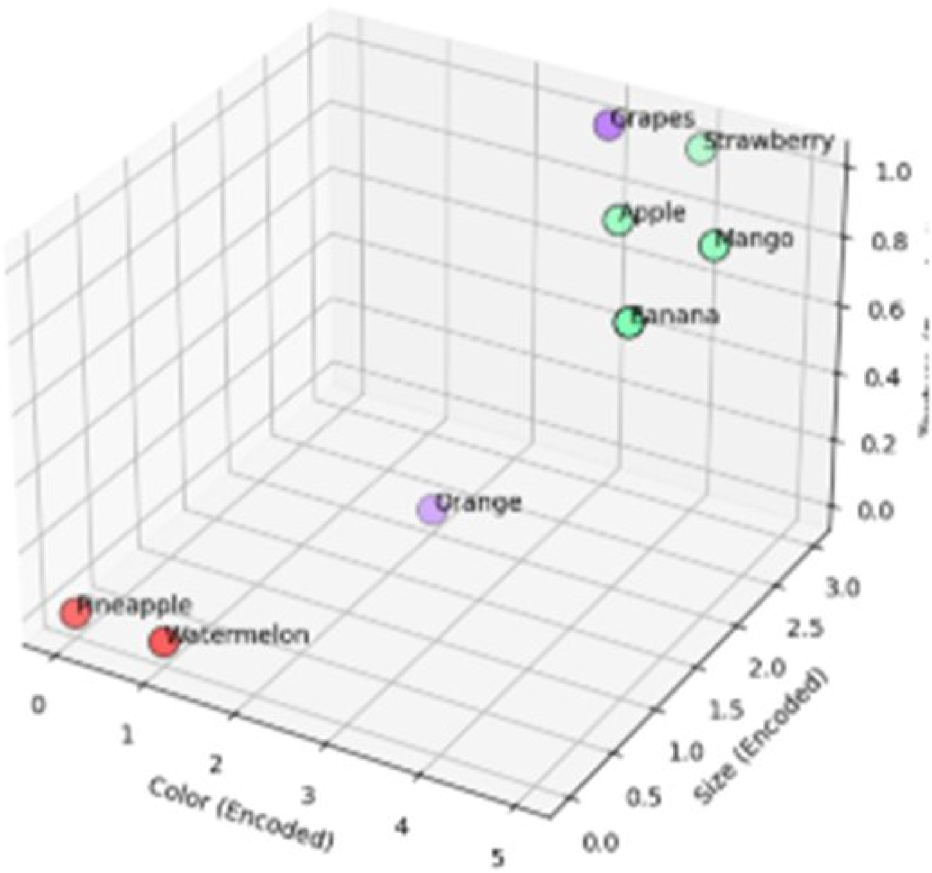
```
# Annotate each point with fruit names
for i, row in fruit_data.iterrows():
    ax.text(row['Color'], row['Size'], row['Texture'], row['Fruit'], fontsize=10,
ha='left')

# Labels and title
ax.set_xlabel("Color (Encoded)")
ax.set_ylabel("Size (Encoded)")
ax.set_zlabel("Texture (Encoded)")
ax.set_title(f"3D Clustering of Fruits with K={optimal_k}")

# Show plot
plt.show()
```

Output:



```
Clustered Fruits:

Cluster 0:
     Fruit  Color  Size  Texture
1  Orange      2     2        0
4  Grapes      3     3        1

Cluster 1:
         Fruit  Color  Size  Texture
0        Apple      4     2        1
2       Banana      5     1        1
3        Mango      5     2        1
7   Strawberry      4     3        1

Cluster 2:
         Fruit  Color  Size  Texture
5   Watermelon      1     0        0
6    Pineapple      0     0        0
```

3D Clustering of Fruits with K=3



Experiment 13:

AIM: Demonstrate the use of Fuzzy C-Means Clustering

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import skfuzzy as fuzz

# --- Fruit Data ---
data = np.array([
    [180, 8, 0.6],     # Apple
    [150, 7, 0.55],    # Banana
    [200, 6, 0.5],     # Orange
    [300, 9, 0.7],     # Mango
    [5, 10, 0.9],      # Blueberry
    [50, 9, 0.75],     # Grape
    [1500, 5, 0.4],    # Pineapple
    [2000, 4, 0.3],    # Watermelon
    [90, 8, 0.7],      # Kiwi
    [20, 7, 0.5]       # Strawberry
])

fruits = ['Apple', 'Banana', 'Orange', 'Mango', 'Blueberry', 'Grape',
          'Pineapple', 'Watermelon', 'Kiwi', 'Strawberry']

# --- FCM Clustering with skfuzzy ---
n_clusters = 3
```

```
m = 2.0   # Fuzziness parameter
error = 1e-6
max_iter = 1000

# Perform Fuzzy C-Means clustering
cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    data.T, n_clusters, m, error, max_iter, seed=42
)

# Assign clusters based on max membership
predicted_labels = np.argmax(u, axis=0) + 1  # 1-based indexing

# --- Create DataFrame ---
cluster_df = pd.DataFrame(data, columns=['Weight (g)', 'Sweetness', 'Color
Intensity'])
cluster_df['Fruit'] = fruits
cluster_df['Cluster'] = predicted_labels

# Add membership values
membership_df = pd.DataFrame(u.T, columns=[f'Cluster {i+1}' for i in
range(n_clusters)])
cluster_df = pd.concat([cluster_df, membership_df], axis=1)

# --- Display Final Cluster Membership Table ---
print("\n--- Final Cluster Membership Table ---\n")
print(cluster_df.to_string(index=False))

# --- Display Fruits in Each Cluster ---
print("\n--- Fruits in Each Cluster ---\n")
for i in range(1, n_clusters + 1):
```

```
    fruits_in_cluster = cluster_df[cluster_df['Cluster'] == i]['Fruit'].tolist()
    print(f"Cluster {i}: {', '.join(fruits_in_cluster)}")

# --- Visualization ---
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Plotting the clusters
colors = ['r', 'g', 'b']
for i in range(n_clusters):
    cluster_points = data[predicted_labels == i + 1]
    axes[0].scatter(cluster_points[:, 0], cluster_points[:, 1],
                    label=f'Cluster {i + 1}', color=colors[i])

# Plotting cluster centers
axes[0].scatter(cntr[:, 0], cntr[:, 1], c='black', marker='X', label='Centers')
axes[0].set_title('Clusters (Weight vs Sweetness)')
axes[0].set_xlabel('Weight (g)')
axes[0].set_ylabel('Sweetness')
axes[0].legend()

# Membership plot
axes[1].plot(membership_df.values)
axes[1].set_title('Cluster Membership for Each Fruit')
axes[1].set_xlabel('Fruit Index')
axes[1].set_ylabel('Membership Value')
axes[1].legend([f'Cluster {i + 1}' for i in range(n_clusters)])

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

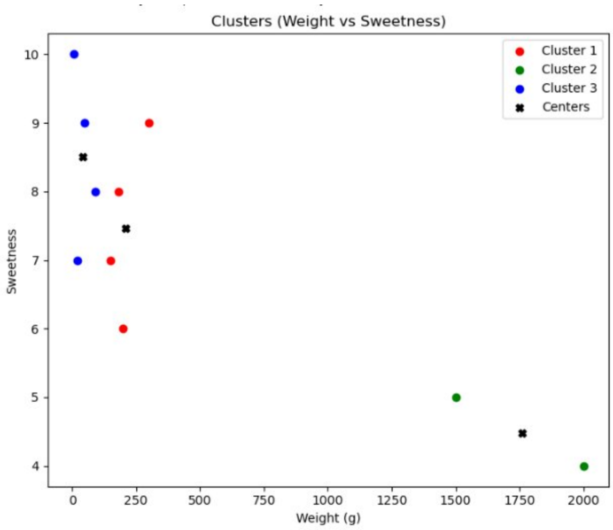
Output:

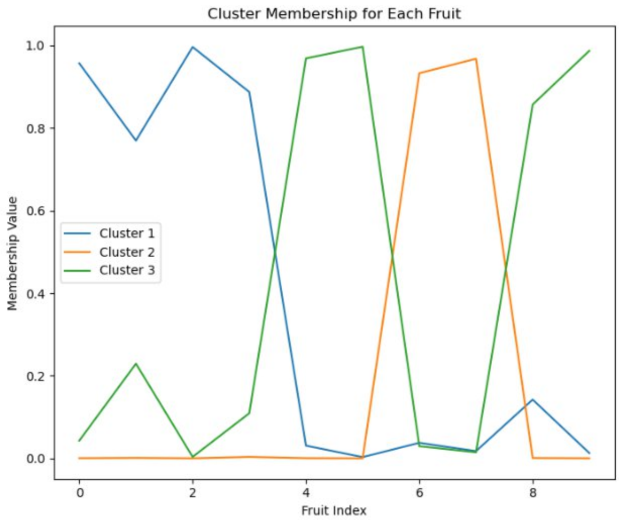
--- Final Cluster Membership Table ---

Weight (g)	Sweetness	Color Intensity	Fruit	Cluster	Cluster 1	Cluster 2	Cluster 3
180.0	8.0	0.60	Apple	1	0.956936	0.000329	0.042735
150.0	7.0	0.55	Banana	1	0.769655	0.001045	0.229301
200.0	6.0	0.50	Orange	1	0.996464	0.000036	0.003500
300.0	9.0	0.70	Mango	1	0.887360	0.003430	0.109210
5.0	10.0	0.90	Blueberry	3	0.030785	0.000418	0.968797
50.0	9.0	0.75	Grape	3	0.002924	0.000025	0.997050
1500.0	5.0	0.40	Pineapple	2	0.037616	0.932929	0.029455
2000.0	4.0	0.30	Watermelon	2	0.017505	0.967862	0.014633
90.0	8.0	0.70	Kiwi	3	0.142329	0.000727	0.856944
20.0	7.0	0.50	Strawberry	3	0.012666	0.000150	0.987184

--- Fruits in Each Cluster ---

```
Cluster 1: Apple, Banana, Orange, Mango
Cluster 2: Pineapple, Watermelon
Cluster 3: Blueberry, Grape, Kiwi, Strawberry
```





Experiment 14:

AIM: Demonstrate the use of Expectation Maximization based clustering algorithm

Program:

```
import numpy as np
import pandas as pd
from scipy.stats import binom
from sklearn.mixture import GaussianMixture
from tabulate import tabulate  # For formatted output display

# Set random seed for reproducibility
np.random.seed(42)

# Coin probabilities
p_A = 0.7  # Probability of heads for Coin A
p_B = 0.4  # Probability of heads for Coin B
n_flips = 10         # Number of flips per experiment
n_experiments = 100  # Number of experiments

# Generate the data
data = []
true_labels = []

for i in range(n_experiments):
    coin = np.random.choice(['A', 'B'])  # Randomly select coin
    if coin == 'A':
        heads = np.random.binomial(n_flips, p_A)
```

```
        true_labels.append(0)  # Label for Coin A
    else:
        heads = np.random.binomial(n_flips, p_B)
        true_labels.append(1)  # Label for Coin B
    data.append(heads)

# Reshape data for GMM
data = np.array(data).reshape(-1, 1)

# Apply GMM clustering
gmm = GaussianMixture(n_components=2, random_state=42)
gmm.fit(data)

# Cluster predictions
labels = gmm.predict(data)
probs = gmm.predict_proba(data)

# Create DataFrame with experiment info
results = pd.DataFrame({
    'Experiment': np.arange(1, n_experiments + 1),
    'Number of Heads': data.flatten(),
    'Cluster': labels,
    'Probability of Coin A': probs[:, 0],
    'Probability of Coin B': probs[:, 1],
    'True Label': true_labels
})

# Select 10 random samples for display
sample_results = results.sample(n=10, random_state=42)
```

```
# Calculate clustering accuracy
accuracy = np.mean(labels == true_labels)

# Format the output table
output_table = sample_results[['Experiment', 'Number of Heads', 'Cluster',
                               'Probability of Coin A', 'Probability of Coin B']]

# Display the formatted table using tabulate
print("\n🔥 First 10 Experiments with Cluster Assignments and Probabilities 🔥\n")
print(tabulate(output_table, headers='keys', tablefmt='fancy_grid', floatfmt=".4f"))

# Display the accuracy
print(f"\n✅ Clustering Accuracy: {accuracy * 100:.2f}%")
```

Output:

🔥 First 10 Experiments with Cluster Assignments and Probabilities 🔥

	Experiment	Number of Heads	Cluster	Probability of Coin A	Probability of Coin B
83	84.0000	7.0000	0.0000	0.5777	0.4223
53	54.0000	6.0000	1.0000	0.2328	0.7672
70	71.0000	7.0000	0.0000	0.5777	0.4223
45	46.0000	9.0000	0.0000	0.9065	0.0935
44	45.0000	8.0000	0.0000	0.8127	0.1873
39	40.0000	3.0000	1.0000	0.0004	0.9996
22	23.0000	6.0000	1.0000	0.2328	0.7672
80	81.0000	2.0000	1.0000	0.0000	1.0000
10	11.0000	7.0000	0.0000	0.5777	0.4223
0	1.0000	6.0000	1.0000	0.2328	0.7672

✅ Clustering Accuracy: 80.00%