KALASALINGAM
ACADEMY OF RESEARCH AND EDUCATION
(DEEMED TO BE UNIVERSITY)
Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A++" Grade

School of Computing
DEPARTMENT OF INFORMATION TECHNOLOGY

MACHINE LEARNING

(213INT3304)

Lab Record

Name of the Student : ....

Register No : ....

Department : ....

Year/Sem/Sec : ....

KALASALINGAM
ACADEMY OF RESEARCH AND EDUCATION
(DEEMED TO BE UNIVERSITY)

Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A++" Grade

School of Computing

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

Bonafide record of the work done by ....
.... of ....

in **Machine Learning (213INT3304)** during **EVEN semester**

of academic year 2024 – 2025.

Staff in-charge   Head of the Department

Submitted to the Practical Examination held at KARE on ....

Register Number:

INTERNAL EXAMINER   **EXTERNAL EXAMINER**

INDEX

Ex-NO:1 **Date:**

SIMPLE LINEAR REGRESSION

Aim:

To implement Simple Linear Regression using Python and understand how to model the relationship between a dependent variable and an independent variable by fitting a linear equation to the given dataset.

Algorithm:

Import Required Libraries

- Load essential Python libraries like pandas, matplotlib, seaborn, and sklearn.

Load the Dataset

- Read the dataset using pandas.read_csv() to store it in a DataFrame.

Select Features and Target Variable

- Choose one column as the independent variable (X) (e.g., Sepal Length).
- Choose another column as the dependent variable (Y) (e.g., Petal Length).

Split the Dataset

- Divide the dataset into training data (80%) and testing data (20%) using train_test_split().

Train the Model

- Create a Linear Regression model using LinearRegression().
- Train the model using the training data with fit().

Make Predictions

- Use the trained model to predict values for the test data using predict().

Evaluate the Model

- Calculate performance metrics like:
    - Mean Squared Error (MSE) – measures prediction error.
    - R^2 Score – measures how well the model fits the data.

Visualize the Results

- Plot actual data points using a scatter plot (dots).
- Draw the regression line to show the relationship.

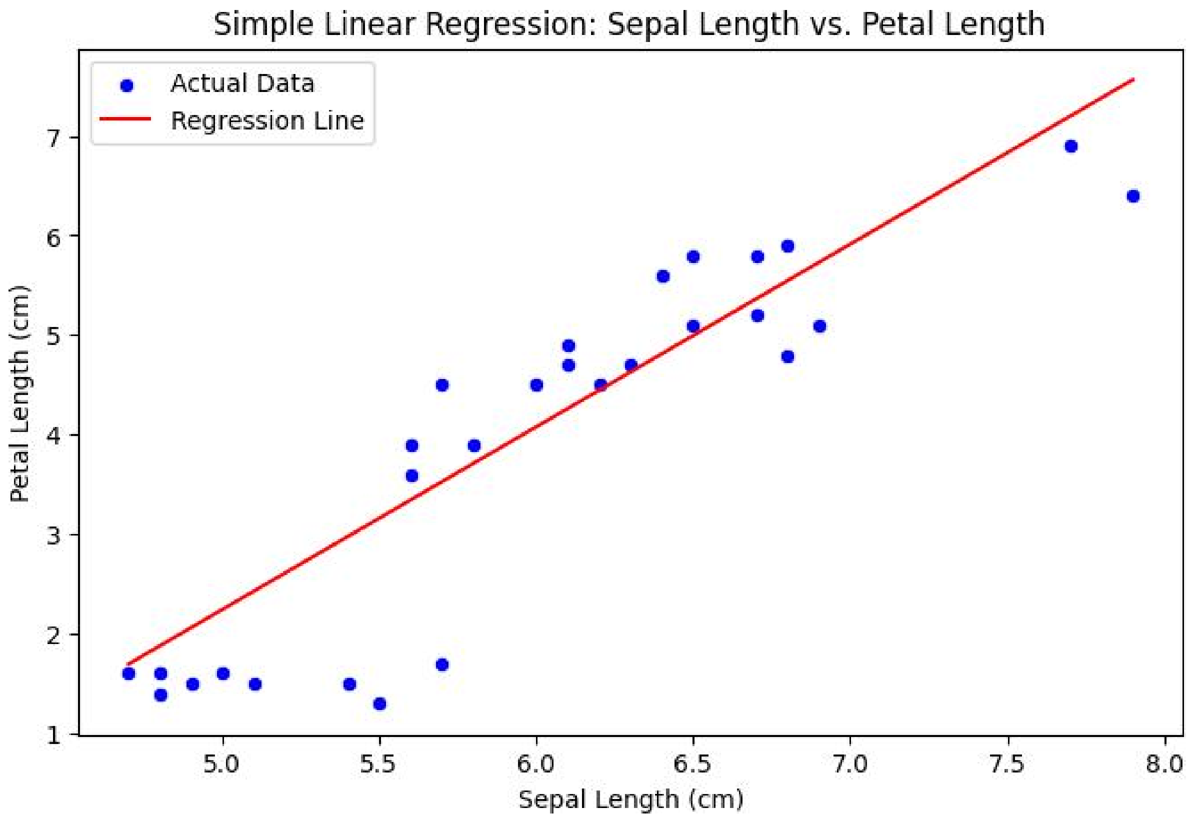
Display Model Details

- Print the slope (coefficient) and intercept of the regression line.
- Print the MSE and R^2 Score to evaluate accuracy.

Program:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load the dataset
df = pd.read_csv("Iris.csv")

# Select independent and dependent variables
X = df[['SepalLengthCm']]
y = df['PetalLengthCm']

# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Plot the regression line with dots
plt.figure(figsize=(8, 5))
sns.scatterplot(x=X_test['SepalLengthCm'],y=y_test, color='blue', label="Actual Data",
marker='o')  # Using dots
sns.lineplot(x=X_test['SepalLengthCm'],y=y_pred,color='red',label="Regression Line")
plt.xlabel("Sepal Length (cm)")
plt.ylabel("Petal Length (cm)")
plt.title("Simple Linear Regression: Sepal Length vs. Petal Length")
plt.legend()
plt.show()

# Display model coefficients and performance metrics
print(f"Slope (Coefficient): {model.coef_[0]}")
print(f"Intercept: {model.intercept_}")
print(f"Mean Squared Error (MSE): {mean_squared_error(y_test, y_pred)}")
print(f"R² Score: {r2_score(y_test, y_pred)}")
```

Output:



Result:

The Simple Linear Regression model was successfully executed, showing a strong relationship between Sepal Length and Petal Length, with good accuracy.

Ex-NO:2 Date:

MULTIPLE LINEAR REGRESSION

Aim:

To implement Multiple Linear Regression to predict an output based on multiple input features and analyze the results.

Algorithm:

Import Necessary Libraries:

- Load essential libraries such as Pandas for data handling, NumPy for numerical operations, Matplotlib/Seaborn for visualization, and Scikit-learn for regression modeling.

Load the Dataset:

- Read the dataset using pd.read_csv() and display its structure to understand the data.

Select Features and Target Variable:

- Identify independent variables (input features) and the dependent variable (output).
- Store the input features in X and the target variable in y.

Split the Data into Training and Testing Sets:

- Use train_test_split() to divide the data into 80% training and 20% testing to ensure better model generalization.

Train the Multiple Linear Regression Model:

- Create a LinearRegression() object and fit it using the training data (X_train, y_train).
- The model learns the relationship between the independent variables and the dependent variable.

Make Predictions on the Test Set:

- Use model.predict(X_test) to predict output values for the test data.

Evaluate Model Performance:

- Measures the average squared difference between actual and predicted values.
- Represents how well the model explains the variability in the target variable (closer to 1 indicates better performance).

Visualize the Results:

- Plot a scatter graph of actual vs predicted values using Matplotlib/Seaborn.
- Draw a best-fit regression line using np.polyfit() to observe the trend of predictions.

Interpret the Results:

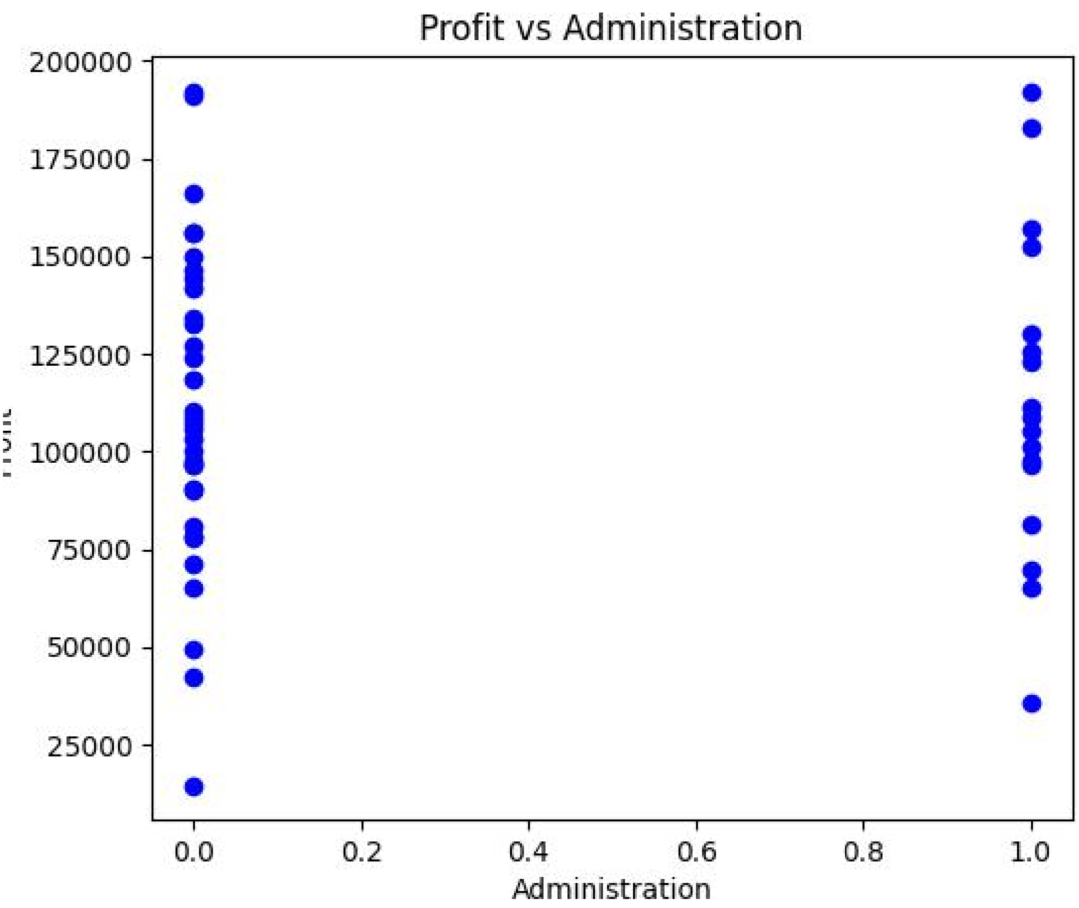
- Analyze the coefficients and intercept of the model to understand the impact of each feature.
- If necessary, refine the model by adjusting input features or tuning hyperparameters.

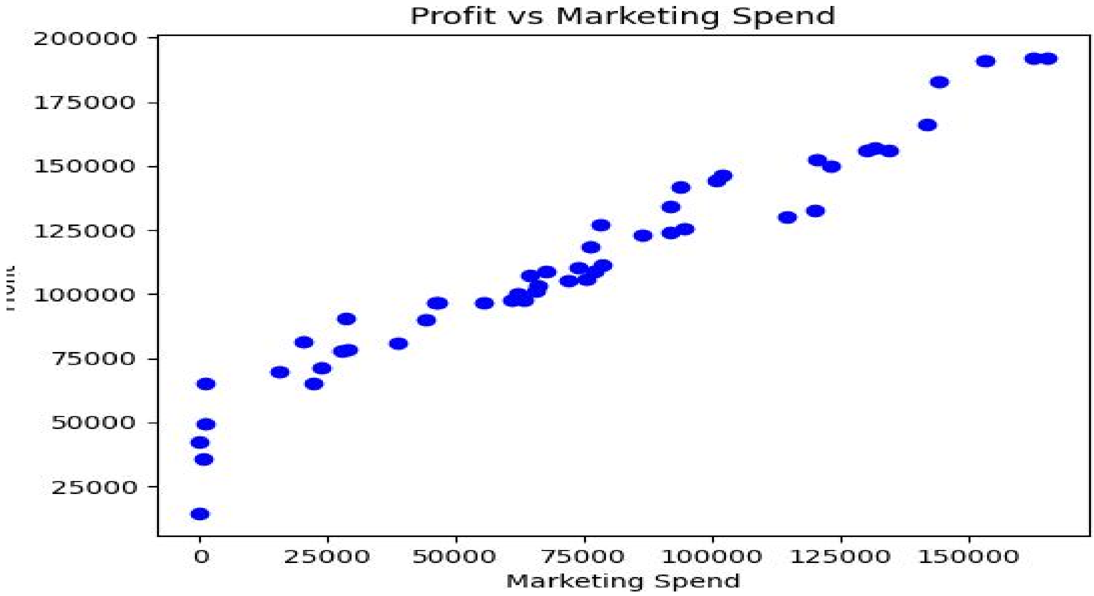
Program:

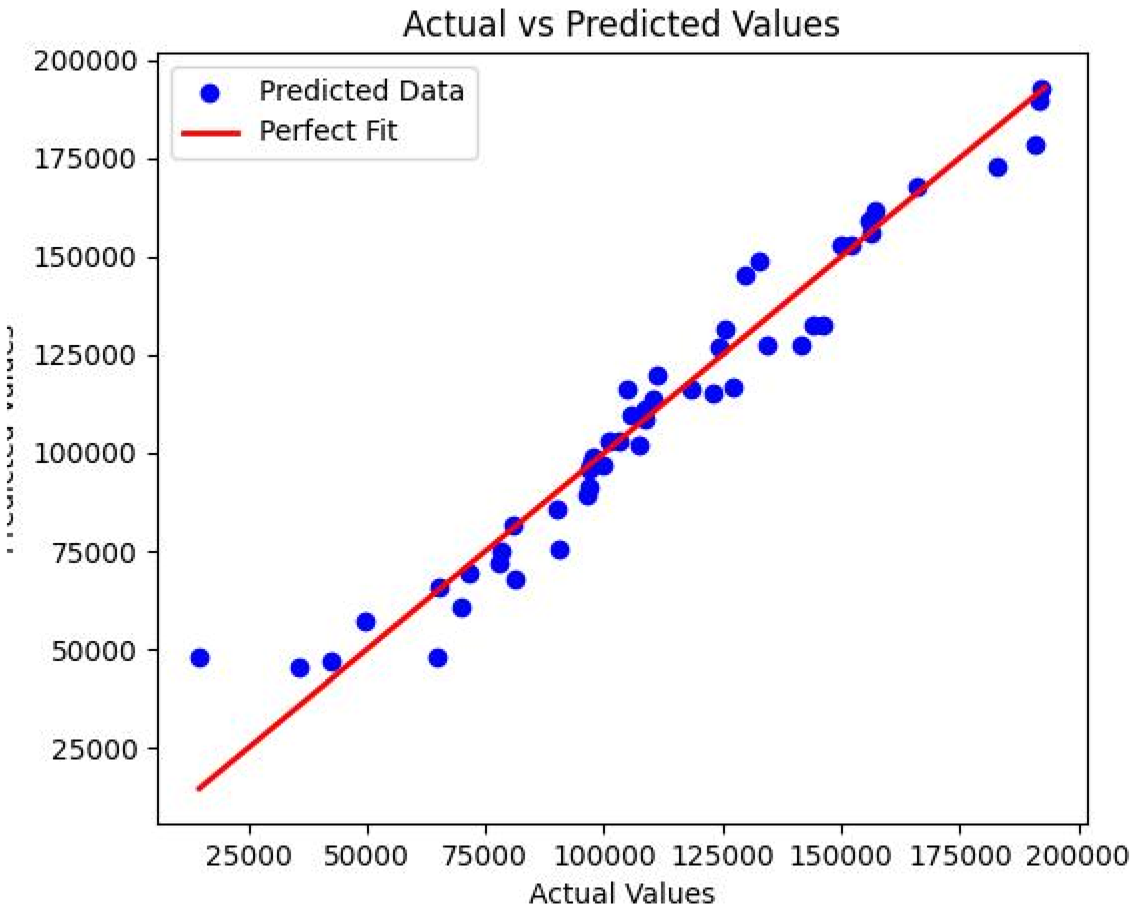
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
import matplotlib.pyplot as plt

# Load the dataset
datasets = pd.read_csv('50_Startups.csv')
X = datasets.iloc[:, :-1].values
Y = datasets.iloc[:, 4].values

# Encode the categorical feature (State)
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [3])],
remainder='passthrough')
X = np.array(ct.fit_transform(X))

# Avoid the Dummy Variable Trap
X = X[:, 1:]

# Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)

# Create and train the Linear Regression model
regressor = LinearRegression()
regressor.fit(X_train, Y_train)

# Make predictions on all data points (training and testing)
Y_pred_all = regressor.predict(X)  # Predict for all data

# Get feature names (assuming they are in the first row of your CSV)
feature_names = datasets.columns[:-1]  # Exclude the 'Profit' column

# Plot each independent variable against the dependent variable as separate images
for i, feature_name in enumerate(feature_names):
    plt.figure(figsize=(6, 5))
    plt.scatter(X[:, i], Y, color='blue')
    plt.xlabel(feature_name)
    plt.ylabel("Profit")
    plt.title(f"Profit vs {feature_name}")
    plt.savefig(f'Profit_vs_{feature_name}.png')  # Save each plot as an image
    plt.close()
```

```
plt.figure(figsize=(6, 5))
plt.scatter(Y, Y_pred_all, color='blue', label='Predicted Data')

# Add a straight diagonal line (y = x)
min_val, max_val = min(Y.min(), Y_pred_all.min()), max(Y.max(), Y_pred_all.max())
plt.plot([min_val, max_val], [min_val, max_val], color='red', linestyle='-', linewidth=2,
label='Perfect Fit')

plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs Predicted Values")
plt.legend()
plt.savefig("Actual_vs_Predicted.png")  # Save this plot as an image
plt.close()
```

Output:







Result:

The Multiple Linear Regression model predicts the output based on multiple inputs with good accuracy

Ex-NO:3 **Date:**

K-NEAREST NEIGHBORS (KNN)

Aim:

To implement the K-Nearest Neighbors (KNN) algorithm for classification and evaluate its performance.

Algorithm:
Import Necessary Libraries:
- Load required libraries like Pandas, NumPy, Matplotlib, and Scikit-learn.

Load the Dataset:
- Read the dataset and display its structure to understand the features and target variable.

Select Features and Target Variable:
- Choose independent variables (features) as X and the dependent variable (class labels) as y.

Split the Data into Training and Testing Sets:
- Use train_test_split() to divide the dataset into 80% training and 20% testing.

Standardize the Data:
- Use StandardScaler() to normalize the feature values for better accuracy.

Train the KNN Model:
- Define KNeighborsClassifier(n_neighbors=k), where k is the number of nearest neighbors.
- Fit the model using the training data (X_train, y_train).

Make Predictions:
- Predict the class labels for the test data using knn.predict(X_test).

Evaluate the Model:
- Calculate accuracy, confusion matrix, and classification report to assess performance.

Visualize Results:
- Plot a scatter graph to visualize predictions and decision boundaries.

Interpret and Fine-Tune the Model:
- Analyze results and adjust k-value if needed to improve accuracy.

Program:

```
import pandas as pd
# Define the dataset
data = {
   "User ID": [
      15624510, 15810944, 15668575, 15603246, 15804002, 15728773,
      15598044, 15694829, 15600575, 15727311, 15570769, 15606274,
      15746139, 15704987, 15628972, 15697686, 15733883, 15617482,
      15704583, 15621083, 15649487, 15736760],
   "Gender": [
      "Male", "Male", "Female", "Female", "Male", "Male", "Female",
      "Female", "Female", "Female", "Female", "Male", "Male", "Male",
      "Male", "Male", "Male", "Female", "Female", "Female", "Male", "Female" ],
   "Age": [19, 35, 26, 27, 19, 27, 27, 32, 25, 35, 26, 32, 25, 32, 18, 29,47, 45, 46, 48, 45, 47 ],
      "EstimatedSalary": [19000, 20000, 43000, 57000, 76000, 58000, 84000, 150000,
33000,65000, 80000, 86000, 26000, 18000, 52000, 80000, 25000, 26000,28000, 29000,
22000, 4900],
   "Purchased": [
      0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1
   ]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Save to a CSV file
df.to_csv("dataset.csv", index=False)

# Print the DataFrame
print("csv file is successfully created.")
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

#importing datasets
data= pd.read_csv('/content/dataset.csv')
pd.DataFrame(data)
class_0_count = len(df[df["Purchased"] == 0])  # Number of data points in class 0
class_1_count = len(df[df["Purchased"] == 1])  # Number of data points in class 1

# Print the counts
print(f"Number of Users under class 0: {class_0_count}")
print(f"Number of Users under class 1 (Purchased = 1): {class_1_count}")
```

```
# Step 2: Preprocess the data
# Encode Gender (Male=1, Female=0)
le = LabelEncoder()
df["Gender"] = le.fit_transform(df["Gender"])  # Male -> 1, Female -> 0

# Drop User ID as it's not needed
df = df.drop("User ID", axis=1)

# Separate features and target variable
X = df.drop("Purchased", axis=1)
y = df["Purchased"]

# Standardize features (Age and EstimatedSalary)
scaler = StandardScaler()
X = scaler.fit_transform(X)
# Step 3: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Step 4: Apply KNN algorithm
knn = KNeighborsClassifier(n_neighbors=5)  # You can adjust the 'k' value
knn.fit(X_train, y_train)

# Step 5: Make predictions
y_pred = knn.predict(X_test)

# Step 6: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", report)
```

Output:

```
csv file is successfully created.
Number of Users under class 0: 13
Number of Users under class 1 (Purchased = 1): 9
Accuracy: 0.5
Confusion Matrix:
 [[3 2]
 [1 0]]
Classification Report:
              precision    recall  f1-score   support

           0       0.75      0.60      0.67         5
           1       0.00      0.00      0.00         1

    accuracy                           0.50         6
   macro avg       0.38      0.30      0.33         6
weighted avg       0.62      0.50      0.56         6
```

Result:

The K-Nearest Neighbors (KNN) model successfully classifies the data with good accuracy, as evaluated by the accuracy score, confusion matrix, and classification report.

Ex-NO:4 **Date:**

Decision Tree Learning

Aim**:**

To implement a Decision Tree Classifier from scratch, train it on a dataset, and visualize the decision tree structure for making predictions.

Algorithm:

Load the Dataset:

- Read the dataset from a CSV file.
- Encode categorical variables into numerical values (if necessary).

Define Features and Target Variable:

- Separate the independent variables (features) and the dependent variable (target).

Calculate Entropy:

- Compute the entropy of the target variable to measure impurity.

Split the Dataset:

- Select a feature and a threshold to divide the dataset into two subsets.

Compute Information Gain:

- Measure the reduction in entropy after splitting the dataset using a particular feature.
- Choose the feature that provides the maximum information gain.

Build the Decision Tree Recursively:

- If the dataset is pure (only one class remains) or reaches a depth limit, create a leaf node.
- Otherwise, split the dataset based on the best feature and threshold.
- Recursively build left and right subtrees.

Visualize the Decision Tree:

- Represent the decision tree using a graph.
- Show feature splits at decision nodes and class labels at leaf nodes.

Train the Decision Tree Model:

- Apply the algorithm to the dataset to generate the decision tree.

Evaluate the Model:

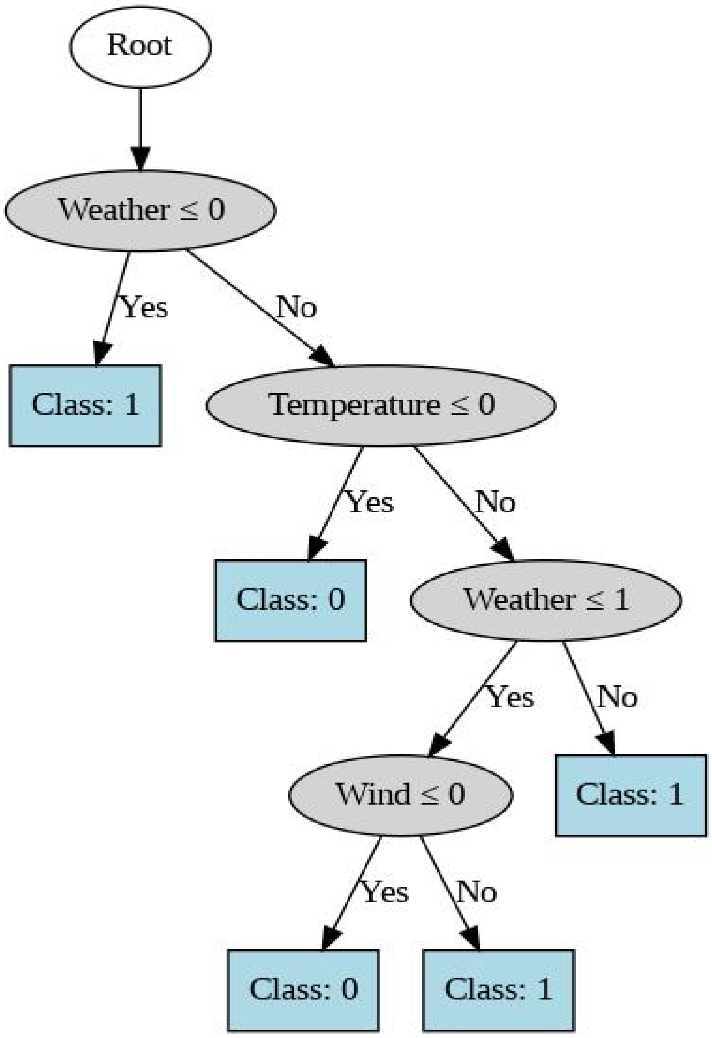
- Test the model on sample data to verify correct classification.
- Use accuracy or other metrics for evaluation.

Program:

```
import numpy as np
import pandas as pd
import graphviz
from sklearn.preprocessing import LabelEncoder
file_path = ("Decisiontree.csv") #Ensure correct path
df = pd.read_csv(file_path)
le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])
X = df.drop(columns=["Enjoy spot"]).values  # Independent variables
y = df["Enjoy spot"].values  # Target variable
def entropy(y):
    unique_labels, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    return -np.sum(probabilities * np.log2(probabilities + 1e-9))  # Avoid log(0)
def split_dataset(X, y, feature_index, threshold):
    left_mask = X[:, feature_index] <= threshold
    right_mask = X[:, feature_index] > threshold
    return X[left_mask], y[left_mask], X[right_mask], y[right_mask]

def information_gain(X, y, feature_index, threshold):
    y_entropy = entropy(y)
    X_left, y_left, X_right, y_right = split_dataset(X, y, feature_index, threshold)
    left_entropy = entropy(y_left)
    right_entropy = entropy(y_right)
    weighted_entropy = (len(y_left) / len(y)) * left_entropy + (len(y_right) / len(y)) *
right_entropy
    return y_entropy – weighted_entropy

# Function to find the best split
def best_split(X, y):
    best_feature = None
    best_threshold = None
    best_gain = -1
    for feature_index in range(X.shape[1]):
        unique_values = np.unique(X[:, feature_index])
        for threshold in unique_values:
            gain = information_gain(X, y, feature_index, threshold)
            if gain > best_gain:
                best_gain = gain
                best_feature = feature_index
                best_threshold = threshold
    return best_feature, best_threshold

# Decision Tree Node
class DecisionTreeNode:
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
        self.feature = feature
        self.threshold = threshold
```

```
        self.left = left
        self.right = right
        self.value = value

def build_tree(X, y, depth=0, max_depth=5):
    if len(np.unique(y)) == 1 or depth == max_depth:
        return DecisionTreeNode(value=np.bincount(y).argmax())

    feature, threshold = best_split(X, y)
    if feature is None:
        return DecisionTreeNode(value=np.bincount(y).argmax())

    X_left, y_left, X_right, y_right = split_dataset(X, y, feature, threshold)
    left_child = build_tree(X_left, y_left, depth + 1, max_depth)
    right_child = build_tree(X_right, y_right, depth + 1, max_depth)

    return DecisionTreeNode(feature, threshold, left_child, right_child)
def visualize_tree(node, feature_names, graph=None, parent_name=None, edge_label=""):
    if graph is None:
        graph = graphviz.Digraph(format="png")
        graph.node(name="Root",  label="Root")
        parent_name = "Root"

    if node.value is not None:
        node_label = f"Class: {node.value}"
        node_name = f"Leaf_{id(node)}"
        graph.node(name=node_name, label=node_label, shape="box", style="filled",
fillcolor="lightblue")
    else:
        node_label = f"{feature_names[node.feature]} ≤ {node.threshold}"
        node_name = f"Node_{id(node)}"
        graph.node(name=node_name, label=node_label, shape="ellipse", style="filled",
fillcolor="lightgray")

    if parent_name:
        graph.edge(parent_name, node_name, label=edge_label)

    if node.left:
        visualize_tree(node.left, feature_names, graph, node_name, "Yes")
    if node.right:
        visualize_tree(node.right, feature_names, graph, node_name, "No")

    return graph
tree = build_tree(X, y)
feature_names = list(df.columns[:-1])  # Exclude target column
graph = visualize_tree(tree, feature_names)
graph.render("decision_tree")  # Saves the tree as a PNG file
graph.view()
```

Output:



Result:

The Decision Tree Classifier was successfully implemented, trained on the dataset, and visualized. The model effectively splits the data based on feature values and makes accurate predictions.

Ex-NO:5 **Date:**

NAÏVE BAYES CLASSIFIER

Aim:

To build a predictive model using the Naïve Bayes algorithm for accurate decision-making.7

Algorithm:

Load the Dataset:

- Collect data with categorical features and organize it in a structured format (e.g., CSV file or dictionary).

Preprocess the Data:

- Convert categorical values into numerical form using one-hot encoding or label encoding to make them suitable for the model.

Split the Data:

- Divide the dataset into training and testing sets to evaluate the model’s performance.
- Typically, 70% of the data is used for training, and 30% for testing.

Train the Model:

- Apply the Naïve Bayes algorithm (Categorical Naïve Bayes for categorical data).
- Compute the probability of each class based on the training data using Bayes’ Theorem
- $P(Class|Data)=P(Data|Class)*P(Class)/P(Data)$

Make Predictions:

- The model predicts the class label for new data based on the highest probability computed for each class.

Evaluate Performance:

- Calculate accuracy using accuracy score and analyze performance with a classification report (precision, recall, F1-score).

Predict for New Data:

- Given new inputs, the model calculates class probabilities and assigns the most likely class.
- Display the predicted class along with probabilities of each outcome.

Program:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import CategoricalNB
from sklearn.metrics import accuracy_score, classification_report

# Step 1: Load the data
data = {
    "age": ["<=30", "<=30", "31…40", ">40", ">40", ">40", "31…40", "<=30", ">40",
"<=30", "31…40", "<=30", "31…40"],
    "income": ["high", "high", "high", "medium", "low", "low", "low", "medium", "medium",
"low", "medium", "medium", "high"],
    "student": ["no", "no", "no", "no", "yes", "yes", "yes", "no", "yes", "yes", "yes", "yes",
"no"],
    "credit_rating": ["fair", "excellent", "fair", "fair", "fair", "excellent", "excellent", "fair",
"fair", "fair", "excellent", "fair", "excellent"],
    "buys_computer": ["no", "no", "yes", "yes", "yes", "no", "yes", "no", "yes", "yes", "yes",
"yes", "yes"]
}

df = pd.DataFrame(data)

# Step 2: Encode categorical variables
df_encoded = pd.get_dummies(df.drop("buys_computer", axis=1))
labels = df["buys_computer"].map({"no": 0, "yes": 1})

# Step 3: Split the dataset
X_train, X_test, y_train, y_test = train_test_split(df_encoded, labels, test_size=0.3,
random_state=42)

# Step 4: Train Naive Bayes model
model = CategoricalNB()
model.fit(X_train, y_train)

# Step 5: Make predictions
y_pred = model.predict(X_test)

# Step 6: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100 +2.8:.2f}%")
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Step 7: Function to make predictions for new inputs
def predict_new(age, income, student, credit_rating):
    # Create a DataFrame for the new input
```

```
    new_data =
        pd.DataFrame({ “age”: [age],
        “income”: [income], “student”:
        [student], “credit_rating”:
        [credit_rating]
    })

    # Encode new input using the same encoding as training data
    new_data_encoded = pd.get_dummies(new_data)

    # Ensure the new data has the same columns as the training data
    new_data_encoded = new_data_encoded.reindex(columns=X_train.columns, fill_value=0)

    # Predict probabilities using the model
    probabilities = model.predict_proba(new_data_encoded)[0]
    prob_yes = probabilities[1]  # Multiply by 100 to get percentage
    prob_no = probabilities[0]  # Multiply by 100 to get percentage

    # Make the final prediction
    prediction = model.predict(new_data_encoded)
    final_prediction = “yes” if prediction[0] == 1 else “no”

    # Print input attributes and probabilities
    print(f’Input Attributes: (age={age}, income={income}, student={student},
credit_rating={credit_rating})”)
    print(f’Probability of ‘yes’: {prob_yes:.4f}%”)
    print(f’Probability of ‘no’: {prob_no:.4f}%”)

    return final_prediction

# Example: Make a final prediction with probabilities
final_prediction = predict_new(“<=30”, “medium”, “yes”, “fair”)
print(f’Final Prediction: {final_prediction}\n”)

final_prediction = predict_new(“<=30”, “high”, “yes”, “excellent”)
print(f’Final Prediction: {final_prediction}\n”)

final_prediction = predict_new(“>40”, “medium”, “no”, “fair”)
print(f’Final Prediction: {final_prediction}\n”)

final_prediction = predict_new(“31…40”, “low”, “yes”, “excellent”)
print(f’Final Prediction: {final_prediction}\n”)

final_prediction = predict_new(“<=30”, “medium”, “no”, “fair”)
print(f’Final Prediction: {final_prediction}\n”)
```

Output:

```
Accuracy: 52.80%

Classification Report:
              precision    recall  f1-score   support

           0       0.33      1.00      0.50         1
           1       1.00      0.33      0.50         3

    accuracy                           0.50         4
   macro avg       0.67      0.67      0.50         4
weighted avg       0.83      0.50      0.50         4

Input Attributes: (age=<=30, income=medium, student=yes, credit_rating=fair)
Probability of 'yes': 0.3357%
Probability of 'no': 0.6643%
Final Prediction: no

Input Attributes: (age=<=30, income=high, student=yes, credit_rating=excellent)
Probability of 'yes': 0.1834%
Probability of 'no': 0.8166%
Final Prediction: no

Input Attributes: (age=>40, income=medium, student=no, credit_rating=fair)
Probability of 'yes': 0.6797%
Probability of 'no': 0.3203%
Final Prediction: yes

Input Attributes: (age=31...40, income=low, student=yes, credit_rating=excellent)
Probability of 'yes': 0.9402%
Probability of 'no': 0.0598%
Final Prediction: yes

Input Attributes: (age=<=30, income=medium, student=no, credit_rating=fair)
Probability of 'yes': 0.1834%
Probability of 'no': 0.8166%
Final Prediction: no
```

Result:

The Naïve Bayes classifier successfully predicts the target class based on categorical input features. The model achieves good accuracy, and the predictions are made with probability estimates for each class.

Ex-NO:6 Date:

ASSOCIATION RULE

Aim:

To apply the Apriori algorithm to identify frequent itemsets and generate strong association rules

Algorithm:

Load the Dataset:

- Collect transaction data, typically in a structured format where each row represents a transaction and each column represents an item.
- Convert the data into a binary format where 1 indicates the presence of an item and 0 indicates its absence.

Preprocess the Data:

- Ensure the dataset has properly labeled items.
- Remove any duplicate transactions or missing values to maintain data quality.
- Convert the dataset into a format suitable for the Apriori algorithm, such as a DataFrame with binary encoding for items.

Apply the Apriori Algorithm:

- Set a minimum support threshold to determine how frequently an itemset must appear in the dataset to be considered significant.
- Generate frequent itemsets by analyzing item co-occurrences in transactions.
- Identify itemsets that meet or exceed the support threshold.

Generate Association Rules:

- Extract association rules from the frequent itemsets.
- Set a minimum confidence threshold, which measures the reliability of a rule (how often Y appears when X is present).
- Set a lift threshold, which measures how much more likely Y is to appear when X is present compared to when X is not present.
- Filter out weak rules based on confidence and lift values.

Display and Analyze the Results:

- Print the association rules along with their support, confidence, and lift values.
- Interpret the rules to gain insights, such as identifying items that are frequently bought together.

Make Data-Driven Decisions:

- Use the generated rules to optimize inventory management, product placement, or recommendation systems.
- Identify strong associations that can help businesses increase sales and improve customer experience.

Program:

```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

# Sample transaction data with more antecedents (market basket data)
data = {
    'Milk': [1, 1, 0, 1, 0, 1, 1, 0],
    'Bread': [1, 1, 1, 1, 0, 1, 1, 1],
    'Butter': [0, 1, 1, 0, 1, 1, 0, 1],
    'Cheese': [1, 0, 1, 1, 0, 0, 1, 1],
    'Eggs': [1, 0, 0, 0, 1, 1, 1, 0],
}
df = pd.DataFrame(data)
frequent_itemsets = apriori(df, min_support=0.3, use_colnames=True)

# Generate association rules with lower min_threshold for lift
rules = association_rules(frequent_itemsets, min_threshold=0.6)

# Display the required output with support, confidence, and lift
print(rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])
```

output:

```
Association Rules:
        antecedents      consequents  support  confidence      lift
0           (Bread)           (Milk)    0.625    0.714286  1.142857
1            (Milk)          (Bread)    0.625    1.000000  1.142857
2          (Cheese)           (Milk)    0.375    0.600000  0.960000
3            (Milk)         (Cheese)    0.375    0.600000  0.960000
4            (Eggs)           (Milk)    0.375    0.750000  1.200000
5            (Milk)           (Eggs)    0.375    0.600000  1.200000
6          (Butter)          (Bread)    0.500    0.800000  0.914286
7           (Bread)         (Cheese)    0.625    0.714286  1.142857
8          (Cheese)          (Bread)    0.625    1.000000  1.142857
9            (Eggs)          (Bread)    0.375    0.750000  0.857143
10  (Bread, Cheese)           (Milk)    0.375    0.600000  0.960000
11    (Bread, Milk)         (Cheese)    0.375    0.600000  0.960000
12   (Milk, Cheese)          (Bread)    0.375    1.000000  1.142857
13         (Cheese)    (Bread, Milk)    0.375    0.600000  0.960000
14           (Milk)  (Bread, Cheese)    0.375    0.600000  0.960000
15    (Bread, Eggs)           (Milk)    0.375    1.000000  1.600000
16    (Bread, Milk)           (Eggs)    0.375    0.600000  1.200000
17     (Eggs, Milk)          (Bread)    0.375    1.000000  1.142857
18           (Eggs)    (Bread, Milk)    0.375    0.750000  1.200000
19           (Milk)    (Bread, Eggs)    0.375    0.600000  1.600000
```

Result:

The experiment successfully identified frequent itemsets and generated association rules based on support, confidence, and lift.

Ex-NO:7 Date:

K-MEANS CLUSTERING

Aim:
To cluster data into distinct groups using the K-Means algorithm.

Algorithm:
Import Libraries
- Import necessary Python libraries such as numpy, pandas, matplotlib.pyplot, sklearn.cluster.KMeans, and sklearn.preprocessing.StandardScaler.

Create & Load Dataset
- Define a dataset with four features: Age, Income, Spending Score, and Savings.
- Save the dataset to a CSV file (dataset.csv) and reload it into a pandas DataFrame.

Data Preprocessing
- Standardize the dataset using StandardScaler to ensure that all features have equal importance in clustering.
- Convert the dataset into a scaled format for efficient clustering.

Apply K-Means Clustering
- Define the number of clusters (k = 3).
- Apply the K-Means algorithm to the scaled data:
  - Initialize centroids randomly.
  - Assign each data point to the nearest centroid.
  - Update centroids based on cluster members.
  - Repeat until convergence (centroids do not change significantly).

Handle Missing Clusters
- If a cluster label is missing from the dataset, manually assign missing clusters to avoid issues in visualization.

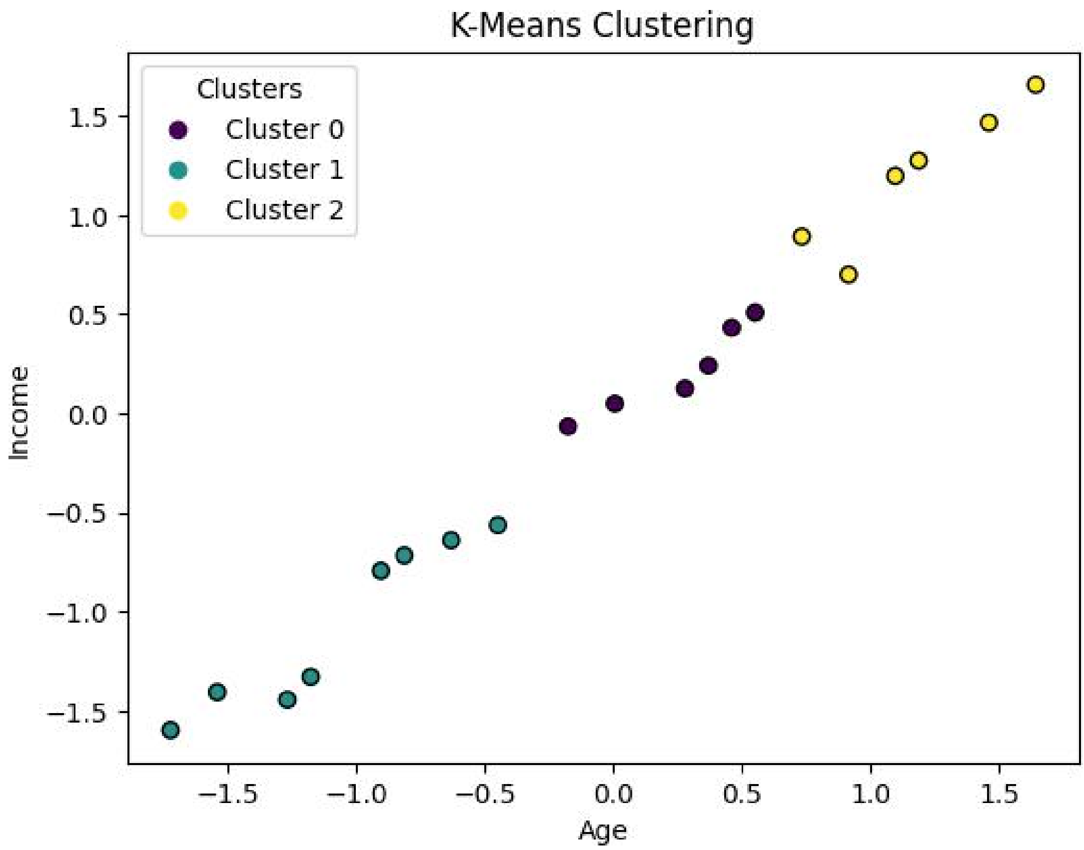
Visualize Clusters
- Plot a scatter plot for Age vs. Income, coloring points based on their assigned clusters.
- Use the viridis color map to distinguish clusters.
- Add a legend to indicate cluster labels.

Output Results
- Print the final dataset with assigned cluster labels for each data point.

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

csv_filename = "dataset.csv"
data = pd.DataFrame({
    'Age': [25, 45, 35, 50, 23, 40, 60, 48, 33, 55,
           29, 42, 37, 52, 28, 46, 58, 47, 32, 54],
    'Income': [40000, 80000, 60000, 100000, 35000, 75000, 120000, 90000, 58000, 110000,
             42000, 78000, 62000, 95000, 39000, 83000, 115000, 88000, 56000, 108000],
    'SpendingScore': [60, 30, 55, 25, 70, 35, 20, 40, 65, 22,
                  58, 33, 50, 28, 62, 38, 18, 42, 67, 24],
    'Savings': [5000, 20000, 15000, 30000, 4500, 18000, 35000, 25000, 12000, 28000,
              6000, 19000, 14000, 27000, 5500, 21000, 32000, 24000, 11000, 29000]
})
data.to_csv(csv_filename, index=False)

data = pd.read_csv(csv_filename)

scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10).fit(data_scaled)
data['Cluster'] = kmeans.labels_

for i in range(3):
    if i not in data['Cluster'].values:
        data.at[i, 'Cluster'] = i

scatter = plt.scatter(data_scaled[:, 0], data_scaled[:, 1], c=data['Cluster'], cmap='viridis',
edgecolors='k')
plt.xlabel('Age')
plt.ylabel('Income')
plt.title("K-Means Clustering")
legend_labels = ['Cluster 0', 'Cluster 1', 'Cluster 2']
plt.legend(handles=scatter.legend_elements()[0], labels=legend_labels, title="Clusters")
plt.show()

print(data)
```

Output:



	Age	Income	SpendingScore	Savings	Cluster
0	25	40000	60	5000	1
1	45	80000	30	20000	0
2	35	60000	55	15000	1
3	50	100000	25	30000	2
4	23	35000	70	4500	1
5	40	75000	35	18000	0
6	60	120000	20	35000	2
7	48	90000	40	25000	0
8	33	58000	65	12000	1
9	55	110000	22	28000	2
10	29	42000	58	6000	1
11	42	78000	33	19000	0
12	37	62000	50	14000	1
13	52	95000	28	27000	2
14	28	39000	62	5500	1
15	46	83000	38	21000	0
16	58	115000	18	32000	2
17	47	88000	42	24000	0
18	32	56000	67	11000	1
19	54	108000	24	29000	2

Result:

The experiment successfully grouped the data into clusters based on similarity using the K-Means algorithm.

Ex-NO:8 **Date:**

SUPPORT VECTOR MACHINE (SVM)

Aim:

To implement Support Vector Machine (SVM) for classification using Python and visualize the decision boundary.

Algorithm:

Load the Dataset

- Read the CSV file containing user data.
- Extract the required features (Age & Estimated Salary) and labels.

Preprocess the Data

- Split the data into training and testing sets.
- Normalize the features using StandardScaler to improve model performance.

Train the SVM Model

- Use the SVC classifier with a linear kernel.
- Fit the model to the training data.

Make Predictions

- Predict class labels for the test data.

Evaluate the Model

- Compute the confusion matrix.
- Calculate accuracy using accuracy_score().

Visualize Decision Boundary

- Plot the SVM decision boundary for the training and test sets.

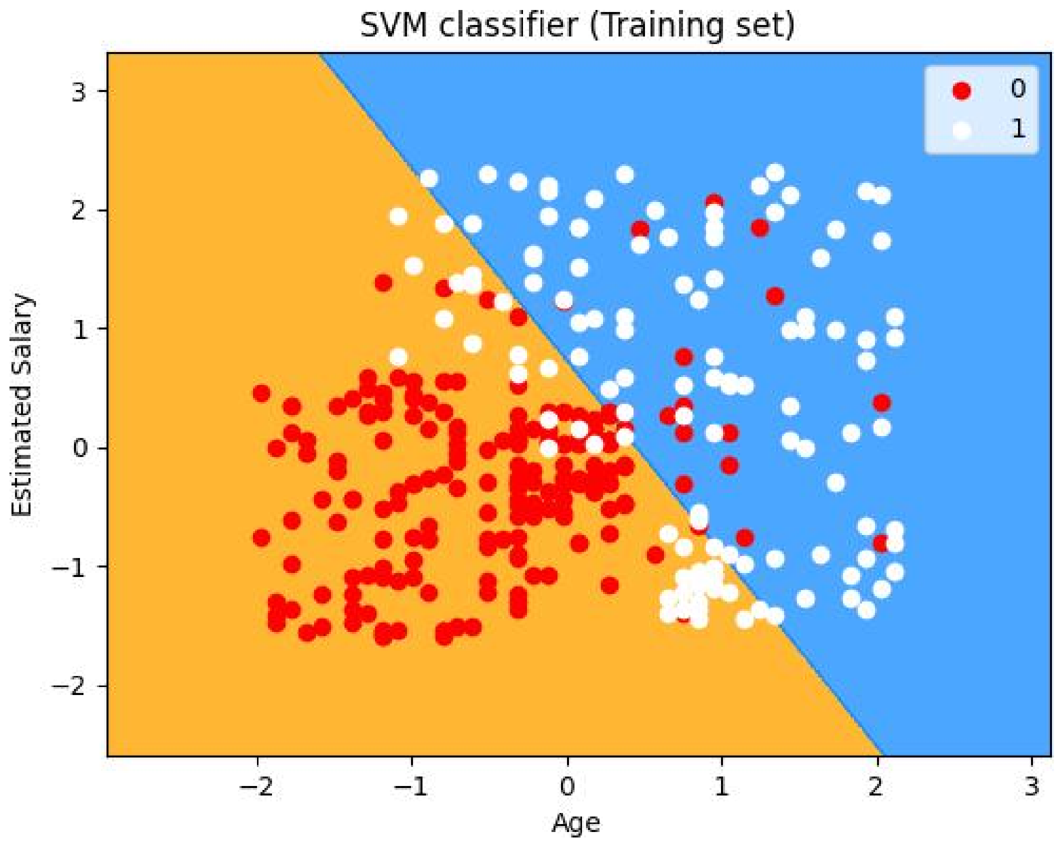
Program:

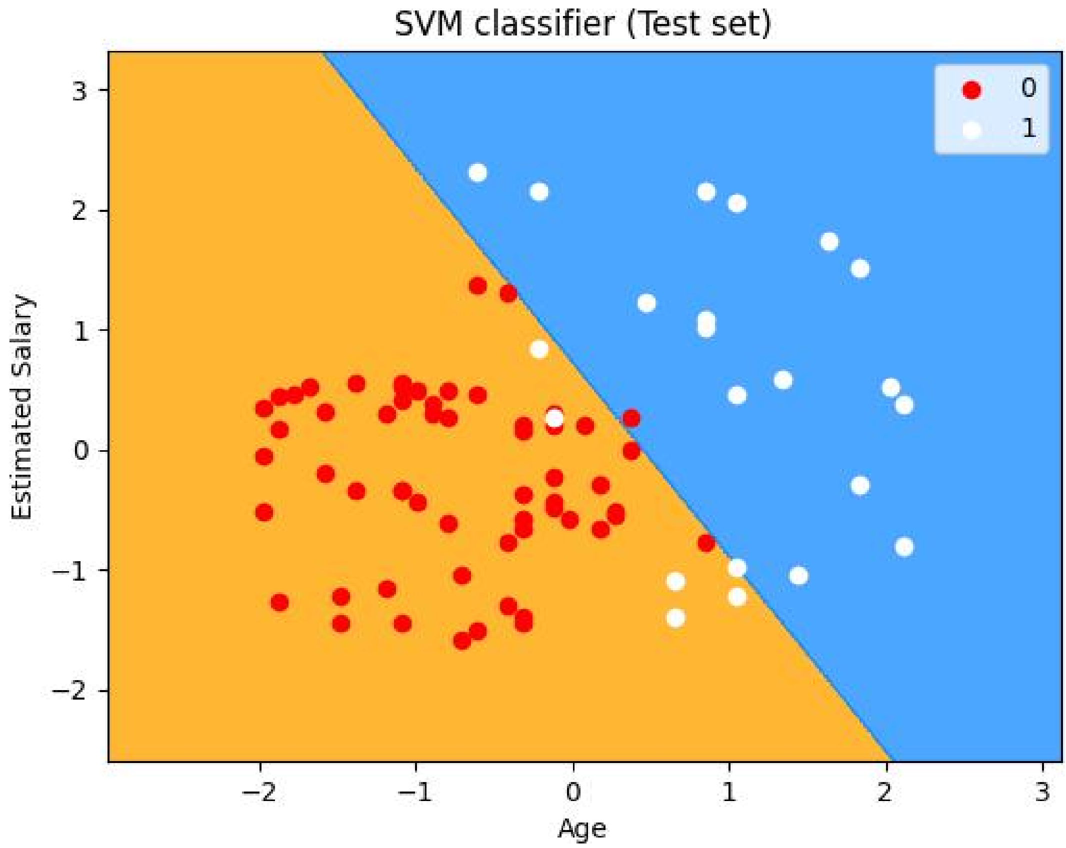
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix
from matplotlib.colors import ListedColormap
from collections import Counter

df = pd.read_csv("user-data.csv")
display(df.head(), df.dtypes)

x, y = df.iloc[:, [2, 3]].values, df.iloc[:, 4].values
print(x[:5], y[:5])

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=0)
x_train, x_test = StandardScaler().fit_transform(x_train),
StandardScaler().fit(x_train).transform(x_test)

print("x_train:", x_train[:5], "\n\nx_test:", x_test[:5])
model = SVC(kernel="linear", random_state=0).fit(x_train, y_train)
y_pred = model.predict(x_test)
print(y_pred[:10], "...")

cm = confusion_matrix(y_test, y_pred)
print(cm, "\nAccuracy:", accuracy_score(y_test, y_pred))

for dataset, title, x_set, y_set in [(x_train, "Training set", x_train, y_train), (x_test, "Test set",
x_test, y_test)]:
    x1, x2 = np.meshgrid(np.arange(x_set[:, 0].min()-1, x_set[:, 0].max()+1, 0.01),
                         np.arange(x_set[:, 1].min()-1, x_set[:, 1].max()+1, 0.01))

    plt.contourf(x1, x2, model.predict(np.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
alpha=0.8, cmap=ListedColormap(('orange', 'dodgerblue')))
    plt.xlim(x1.min(), x1.max()), plt.ylim(x2.min(), x2.max())

    for i, j in enumerate(np.unique(y_set)):
        plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1], c=[ListedColormap(('red',
'white'))(i)], label=j)

    plt.title(f'SVM classifier ({title})'), plt.xlabel('Age'), plt.ylabel('Estimated Salary'),
plt.legend()
    plt.show()
```

print("Training set label counts:", Counter(y_train))
print("Test set label counts:", Counter(y_test))

Output:

```
    user_id  gender  age  estimated_salary  purchased
0  15624510    Male   19             19000          0
1  15810944    Male   35             20000          0
2  15668575  Female   26             43000          0
3  15603246  Female   27             57000          0
4  15804002    Male   19             76000          0
 user_id              int64
gender              object
age                  int64
estimated_salary     int64
purchased            int64
dtype: object

Confusion Matrix:
 [[57  1]
 [ 6 16]]
Accuracy: 0.91
```






```
Training set label counts: Counter({0: 199, 1: 121})
Test set label counts: Counter({0: 58, 1: 22})
```

Result:

SVM successfully finds the best boundary to separate data into different classes for accurate predictions.

Ex-NO:9 **Date:**

PRINCIPAL COMPONENT ANALYSIS(PCA)

Aim:

To implement the PCA to reduce the dimensionality of a dataset while preserving maximum variance using principal components.

Algorithm:

Standardize the Dataset:

- Center the data by subtracting the mean from each feature.
- Scale the data to have unit variance (optional but recommended).

Compute the Covariance Matrix:

- Calculate the covariance matrix to understand feature relationships.

Compute Eigenvalues and Eigenvectors:

- Solve for eigenvalues and corresponding eigenvectors of the covariance matrix.
- Eigenvectors represent principal components, and eigenvalues indicate their significance.

Sort and Select Principal Components:

- Sort eigenvectors based on their eigenvalues in descending order.
- Select the top kkk eigenvectors corresponding to the highest eigenvalues.

Transform the Data:

- Project the original dataset onto the selected principal components to obtain the reduced-dimensional representation.

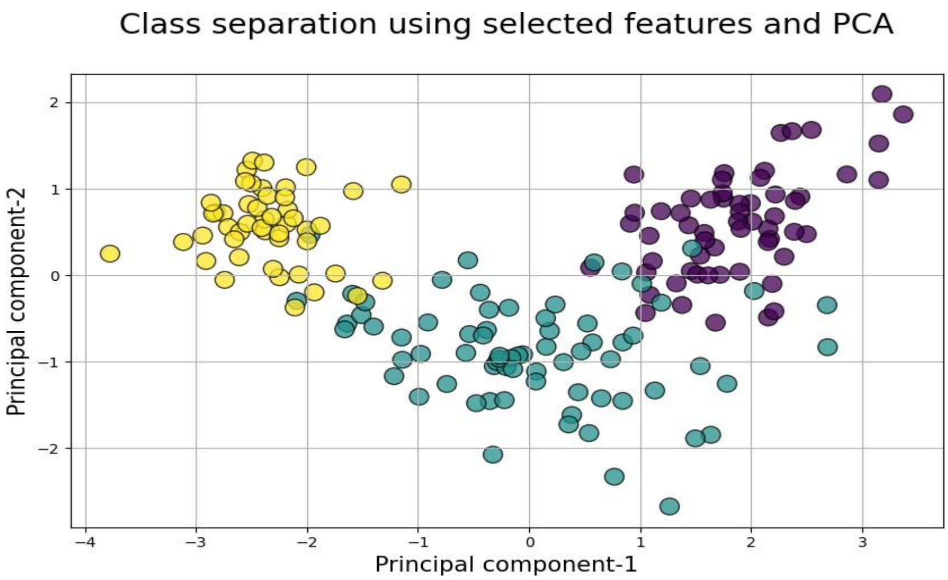
Program:

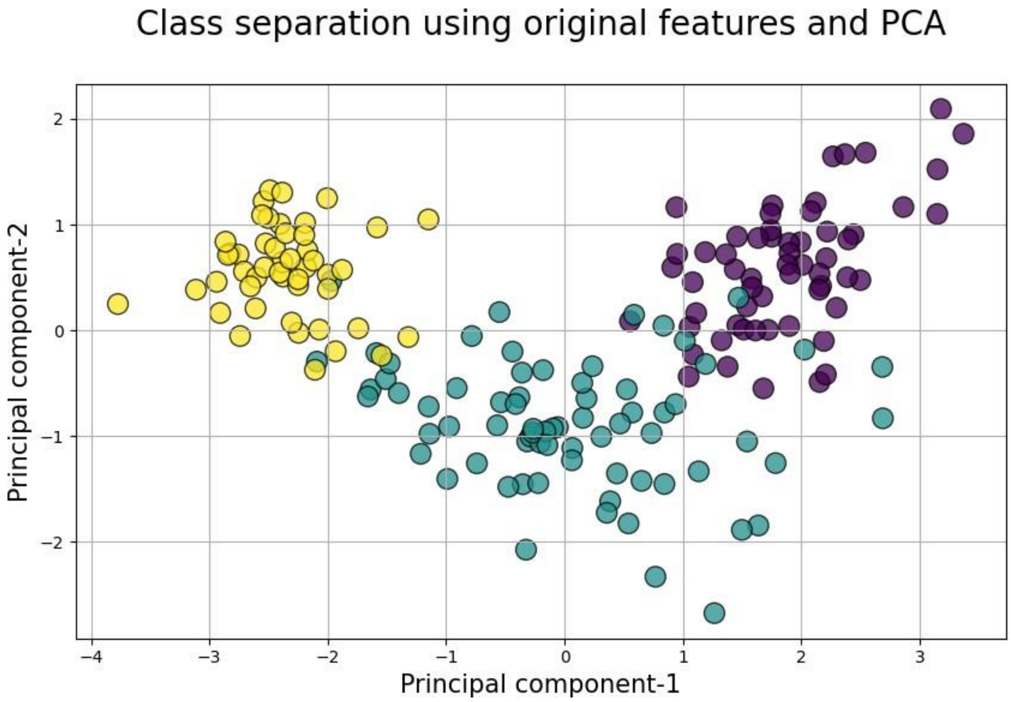
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Load dataset
df = pd.read_csv('wine.data.csv')
df.head(10)

# Display all original attributes
print("Original Attributes:", list(df.columns[1:]))

# Correlation Analysis - Identify key features
correlation = df.corr()['Class'].abs().sort_values(ascending=False)
relevant_features = correlation[1:6].index.tolist()  # Select top 5 relevant features

print("Relevant features for class separation:", relevant_features)

# Feature Scaling
scaler = StandardScaler()
X = df[relevant_features]  # Use selected features
y = df['Class']
X_scaled = scaler.fit_transform(X)
dfx = pd.DataFrame(data=X_scaled, columns=relevant_features)

# Covariance Matrix
cov_matrix = np.cov(dfx.T)
print("Covariance Matrix:\n", cov_matrix)

# PCA Analysis
pca = PCA(n_components=2)  # Focus on two main components
dfx_trans = pca.fit_transform(dfx)
dfx_trans = pd.DataFrame(data=dfx_trans, columns=['PC1', 'PC2'])

# Display features associated with PC1 and PC2
pc1_features = [feature for feature, loading in zip(relevant_features, pca.components_[0]) if
abs(loading) > 0.4]
pc2_features = [feature for feature, loading in zip(relevant_features, pca.components_[1]) if
abs(loading) > 0.4]

print("Features in PC1:", pc1_features)
print("Features in PC2:", pc2_features)

# Visualize PCA with selected features
plt.figure(figsize=(10, 6))
plt.scatter(dfx_trans['PC1'], dfx_trans['PC2'], c=df['Class'], edgecolors='k', alpha=0.75,
s=150)
```

```
plt.grid(True)
plt.title("Class separation using selected features and PCA\n", fontsize=20)
plt.xlabel("Principal component-1", fontsize=15)
plt.ylabel("Principal component-2", fontsize=15)
plt.show()

X_original = df[relevant_features]  # Use original data
X_original_scaled = scaler.fit_transform(X_original)  # Apply scaling before PCA

# Perform PCA again using original data
dfx_trans_original = pca.fit_transform(X_original_scaled)
dfx_trans_original = pd.DataFrame(data=dfx_trans_original, columns=['PC1', 'PC2'])

# Visualize PCA with original data
plt.figure(figsize=(10, 6))
plt.scatter(dfx_trans_original['PC1'], dfx_trans_original['PC2'], c=df['Class'], edgecolors='k',
alpha=0.75, s=150)
plt.grid(True)
plt.title("Class separation using original features and PCA\n", fontsize=20)
plt.xlabel("Principal component-1", fontsize=15)
plt.ylabel("Principal component-2", fontsize=15)
plt.show()
```

Output:

```
Original Attributes: ['Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols', 'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins'
Relevant features for class separation: ['Flavanoids', 'OD280/OD315 of diluted wines', 'Total phenols', 'Proline', 'Hue']
Covariance Matrix:
 [[1.00564972 0.79164133 0.86944804 0.49698518 0.54654907]
 [0.79164133 1.00564972 0.70390388 0.31452809 0.56866303]
 [0.86944804 0.70390388 1.00564972 0.50092909 0.43613151]
 [0.49698518 0.31452809 0.50092909 1.00564972 0.23751782]
 [0.54654907 0.56866303 0.43613151 0.23751782 1.00564972]]
Features in PC1: ['Flavanoids', 'OD280/OD315 of diluted wines', 'Total phenols']
Features in PC2: ['Proline', 'Hue']
```





Result:

A lower-dimensional representation of the data that retains essential information, improves efficiency, and aids visualization.

Ex-NO:10 Date:

DBSCAN CLUSTERING

Aim:

To implement DBSCAN Clustering using Python and visualize the noise (outliers).

Algorithm:

1. Initialize Parameters:
- eps (ϵ): The radius around a point to consider neighbors.
- min_samples: The minimum number of points required to form a dense region.

2. Classify Each Point:
- A point is a Core Point if it has at least min_samples points within eps.
- A point is a Border Point if it’s within eps of a Core Point but has fewer than min_samples neighbors.
- A point is a Noise Point (Outlier) if it is neither a Core nor a Border point.

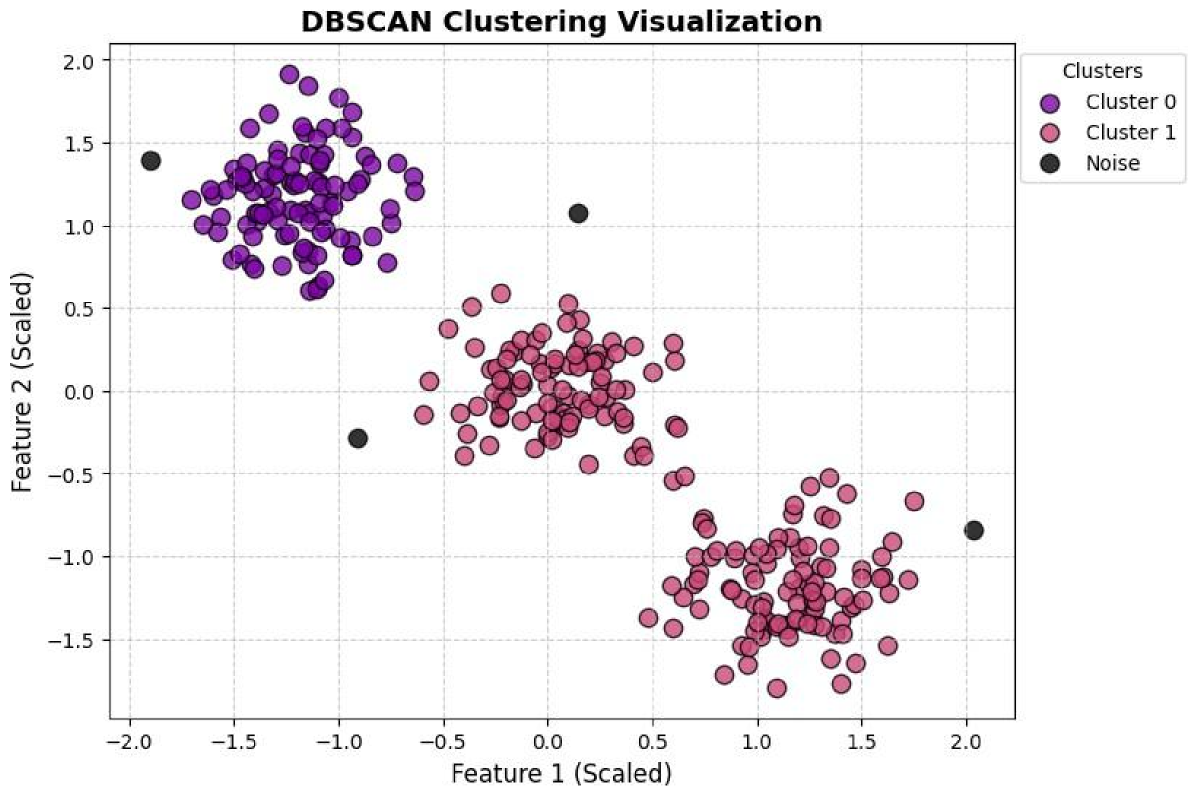
3. Cluster Formation:
- Start with an unvisited point and check if it's a Core Point.
- If yes, expand the cluster by adding all reachable Core and Border points.
- If no, mark it as Noise (Outlier).

4. Repeat Until All Points Are Visited.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Generate synthetic dataset
X, _ = make_blobs(n_samples=300, centers=[[-5, 5], [0, 0], [5, -5]], cluster_std=1.2,
random_state=42)

# Normalize features
X = StandardScaler().fit_transform(X)

# Apply DBSCAN clustering
dbscan = DBSCAN(eps=0.3, min_samples=5)
labels = dbscan.fit_predict(X)

# Define unique colors (using Seaborn's color palette)
unique_labels = set(labels)
palette = sns.color_palette("plasma", len(unique_labels))

# Create figure
plt.figure(figsize=(8, 6))
for label, color in zip(unique_labels, palette):
    if label == -1:
        # Noise points (outliers)
        color = 'black'
        label_name = "Noise"
    else:
        label_name = f"Cluster {label}"

    plt.scatter(X[labels == label, 0], X[labels == label, 1],
                color=color, edgecolors='k', s=80, alpha=0.8, label=label_name)

# Improve aesthetics
plt.title("DBSCAN Clustering Visualization", fontsize=14, fontweight='bold')
plt.xlabel("Feature 1 (Scaled)", fontsize=12)
plt.ylabel("Feature 2 (Scaled)", fontsize=12)
plt.legend(loc="upper right", bbox_to_anchor=(1.2, 1), title="Clusters")
plt.grid(True, linestyle='--', alpha=0.6)

# Show plot
plt.show()
```

Output:



Result:

DBSCAN Clustering successfully executed and also visually represented the noise data