MAR ATHANASIUS COLLEGE OF ENGINEERING

(Govt. Aided & Autonomous)

KOTHAMANGALAM

M24CS1L107 ADVANCED MACHINE LEARNING LAB

Submitted in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

APJ Abdul Kalam Technological University

Submitted by

PAUL JOSE

Reg.No : MAC24CSCE07

Department of Computer Science and Engineering

Mar Athanasius College of Engineering

(Govt. Aided & Autonomous)

Kothamangalam

December 2024

Mar Athanasius College of Engineering (Govt. Aided & Autonomous)

KOTHAMANGALAM

M24CS1L107
ADVANCED MACHINE LEARNING LAB

Bonafide record of work done by

PAUL JOSE

Register Number: MAC24CSCE07

Submitted in partial fulfillment of the requirements
for the Degree of

Master of Technology
in
Computer Science and Engineering
of
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

…………………	…………………
External Examiner	***Internal Examiner***

Place: Kothamangalam
Date: ....

CONTENTS

EXPERIMENT NO: 1
DATE:

SIMPLE AND MULTIPLE LINEAR REGRESSION

AIM:

To write a program to implement the Simple and Multiple Linear Regression for a sample training data set stored as a .CSV file. Compute Mean Square Error by considering few test data sets.

ALGORITHM:

Notations:
The following notations are used in the algorithm:

X	Feature matrix (independent variables).
y	Target vector (dependent variable).
Xtrain	Training data for features (subset of X).
Xtest	Testing data for features (subset of X).
ytrain	Training data for the target variable (subset of y).
ytest	Testing data for the target variable (subset of y).
Y	Predicted values of the target variable (y) .
n	Number of samples in the dataset.
MSE	Mean Squared Error

Algorithm :
1. Load the dataset into a DataFrame.
2. If there are categorical variables, apply label encoding to convert them into numerical values.
3. Separate the features X (independent variables) and target y (dependent variable).
4. Split the dataset into training and testing sets using an 80%-20% split.
5. Create a linear regression model.
6. Fit the model to the training data Xtrain,ytrain.
7. Predict the target variable Y using the test data Xtest.
8. Compute the Mean Squared Error (MSE) by comparing the predicted values Y with the actual values ytest.
9. Print the MSE for model performance.
10. Accept manual input for feature values.
11. Use the trained model to predict the target variable for the manual input.
12. Display the predicted value for the manual input.

DESCRIPTION:

Data Set Description:

Simple Linear Regression:
The dataset contains 30 entries with two attributes: YearsExperience and Salary. The target variable is Salary, predicted based on YearsExperience.

Multiple Linear Regression:
The dataset includes attributes like R&D Spend, Administration, Marketing Spend and State. The target variable is Index.

Linear regression:
Linear regression models predict a continuous target variable y by establishing a linear relationship with one or more input features X. The relationship is represented by the equation:

$$y = \theta 1X1 + \theta 2X2 + \cdots + \theta nXn + b$$

For **Simple Linear Regression**, the model uses a single feature (X) and simplifies to:

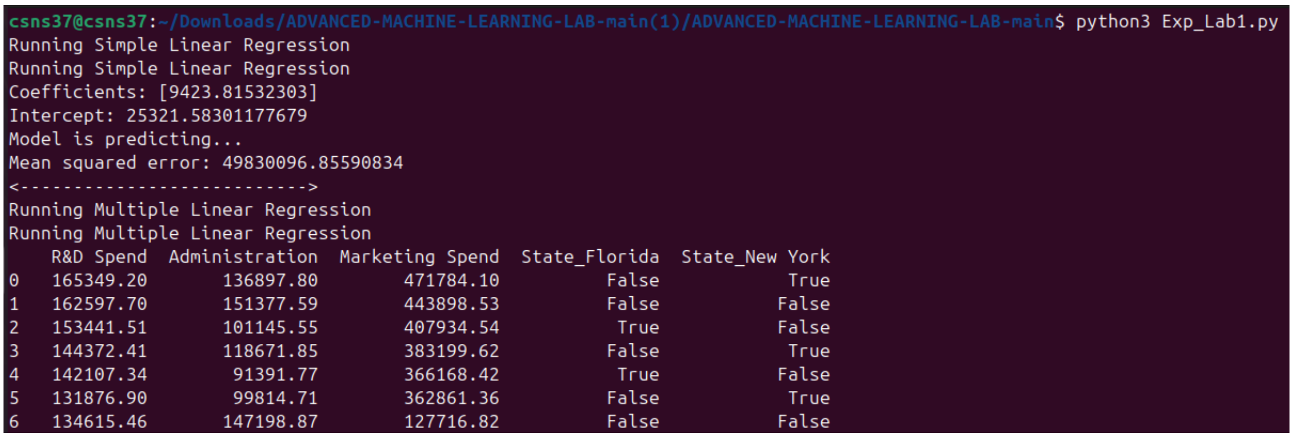
$$y = \theta X + b$$

For **Multiple Linear Regression**, the model incorporates multiple features (X1,X2,…,Xn) to make predictions.

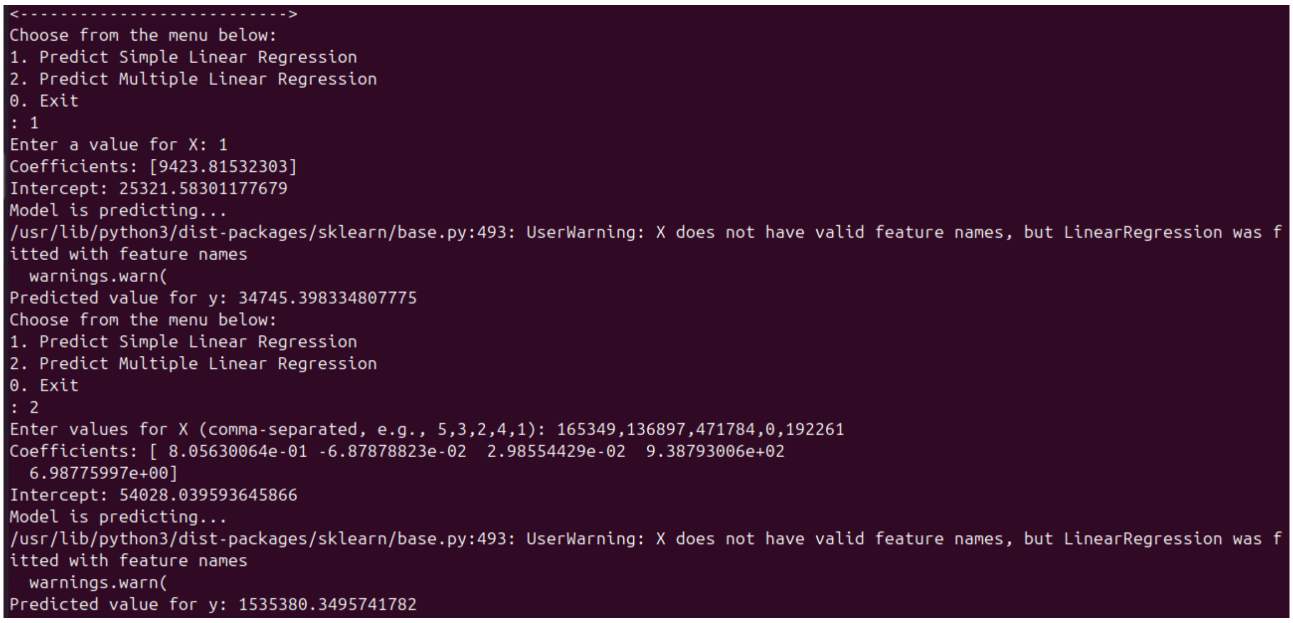
IMPLEMENTATION:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
import time

def readFile_asDF(dataset):
    file_path = f'./Dataset/{dataset}.csv'
    return pd.read_csv(file_path)

def separateDataset(df):
    y = df.iloc[:, -1]
    X = df.iloc[:, :-1]
    return X, y

def model_predict(model, X):
    print(f"Coefficients: {model.coef_}")
    print(f"Intercept: {model.intercept_}")
    print("Model is predicting...")
    time.sleep(2)
    return model.predict(X)

def MeanSquaredError(y_obtained, y_target):
    mse = np.mean((y_target - y_obtained) ** 2)
    print(f'Mean squared error: {mse}')

def simple_LR():
    print("Running Simple Linear Regression")
    df = readFile_asDF('simpleLR')
    X, y = separateDataset(df)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    model_simple = LinearRegression()
    model_simple.fit(X_train, y_train)
    y_pred = model_predict(model_simple, X_test)
    MeanSquaredError(y_pred, y_test)
    return model_simple

def multiple_LR():
    print("Running Multiple Linear Regression")
    df = readFile_asDF('multipleLR')
    X, y = separateDataset(df)
    if 'State' in X.columns:
        X['State'] = X['State'].str.strip()  # Remove extra whitespaces
            X = pd.get_dummies(X, columns=['State'], drop_first=True)   # One-hot encode the 'State'
column
    print(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    model_multi = LinearRegression()
    model_multi.fit(X_train, y_train)
    y_pred = model_predict(model_multi, X_test)
    MeanSquaredError(y_pred, y_test)
    return model_multi

def main():
    print("Running Simple Linear Regression")
    model_simple = simple_LR()
    print('<--------------------------->')

    print("Running Multiple Linear Regression")
    model_multi = multiple_LR()
    print('<--------------------------->')

    userInput = 1
    while userInput:
        userInput = int(input("Choose from the menu below:\n1. Predict Simple Linear Regression\n2.
Predict Multiple Linear Regression\n0. Exit\n: "))

    if userInput == 1:
      X = float(input("Enter a value for X: "))
      X = np.array([[X]])
      y_pred = model_predict(model_simple, X)
      print("Predicted value for y:", y_pred[0])

    elif userInput == 2:
      X_values = input("Enter values for X (comma-separated, e.g., 5,3,2,4,1): ")
      # Map categorical variables before creating the input array
      X_list = X_values.split(",")
      X_numeric = []

      for value in X_list:
        value = value.strip()
        if value in ['True', 'False']:
          X_numeric.append(int(value == 'True'))  # Convert boolean strings to integers
        else:
          X_numeric.append(float(value))  # Convert other numeric inputs

      X = np.array(X_numeric).reshape(1, -1)
      y_pred = model_predict(model_multi, X)
      print("Predicted value for y:", y_pred[0])
    elif userInput == 0:
      print("Exiting the program.")

    else:
      print(f'Invalid option: {userInput}.')

if __name__ == "__main__":
  main()
```

OUTPUT:



```
csns37@csns37:~/Downloads/ADVANCED-MACHINE-LEARNING-LAB-main(1)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 Exp_Lab1.py
Running Simple Linear Regression
Running Simple Linear Regression
Coefficients: [9423.81532303]
Intercept: 25321.58301177679
Model is predicting...
Mean squared error: 49830096.85590834
<------------------------->
Running Multiple Linear Regression
Running Multiple Linear Regression
   R&D Spend  Administration  Marketing Spend  State_Florida  State_New York
0  165349.20       136897.80        471784.10          False            True
1  162597.70       151377.59        443898.53          False           False
2  153441.51       101145.55        407934.54           True           False
3  144372.41       118671.85        383199.62          False            True
4  142107.34        91391.77        366168.42           True           False
5  131876.90        99814.71        362861.36          False            True
6  134615.46       147198.87        127716.82          False           False
```



```
<-------------------------->
Choose from the menu below:
1. Predict Simple Linear Regression
2. Predict Multiple Linear Regression
0. Exit
: 1
Enter a value for X: 1
Coefficients: [9423.81532303]
Intercept: 25321.58301177679
Model is predicting...
/usr/lib/python3/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but LinearRegression was f
itted with feature names
  warnings.warn(
Predicted value for y: 34745.398334807775
Choose from the menu below:
1. Predict Simple Linear Regression
2. Predict Multiple Linear Regression
0. Exit
: 2
Enter values for X (comma-separated, e.g., 5,3,2,4,1): 165349,136897,471784,0,192261
Coefficients: [ 8.05630064e-01 -6.87878823e-02  2.98554429e-02  9.38793006e+02
  6.98775997e+00]
Intercept: 54028.039593645866
Model is predicting...
/usr/lib/python3/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but LinearRegression was f
itted with feature names
  warnings.warn(
Predicted value for y: 1535380.3495741782
```

RESULT:

Program implementing simple and multilinear regression executed successfully, Used the model to predict for a new sample and the output is obtained.

EXPERIMENT NO: 2
DATE:

NAIVE BAYESIAN CLASSIFIER

AIM:

Write a program to implement the naïve bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

ALGORITHM:

1. Input:
   1. A training dataset with features (e.g., sepal length, sepal width, petal length, petal width in the Iris dataset).
   2. Corresponding target labels , representing the class for each instance (e.g., Species: Setosa, Versicolor, Virginica).
   3. A test instance , representing the feature values of an instance to be classified.
2. Compute Prior Probabilities:
   1. Calculate , the prior probability of each class :
3. Compute Conditional Probabilities:
   1. For each feature and class , calculate the conditional probability assuming a Gaussian distribution:
      where:
      1. : Mean of feature for class .
      2. : Standard deviation of feature for class .
4. Compute Class Scores:
   1. For the test instance , calculate the product of probabilities for each class :
5. Classify the Test Instance:
   1. Identify the class with the highest score :
6. Output:
   1. Assign the test instance the class label .

DESCRIPTION:

Naive Bayes theorem is a collection of classification algorithms based on Bayes Theorem. It is stated as:

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

Data Set Description:

The dataset used in this program is the Iris dataset, which is one of the most popular datasets in machine learning. It contains the following features:

- Sepal Length: The length of the sepal in centimeters (numeric).
- Sepal Width: The width of the sepal in centimeters (numeric).
- Petal Length: The length of the petal in centimeters (numeric).
- Petal Width: The width of the petal in centimeters (numeric).

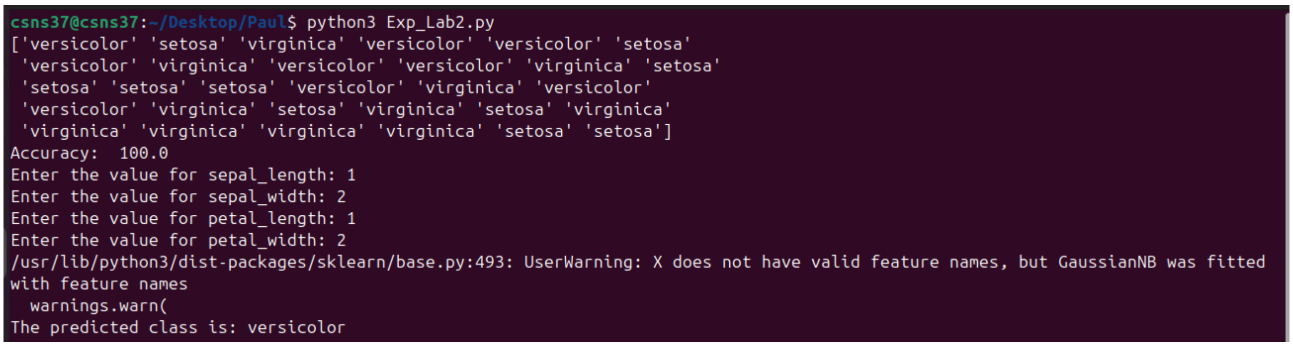
- Species: The target variable indicating the class of the iris flower. It has three possible values:
  1. Setosa
  2. Versicolor
  3. Virginica

The dataset is used to classify iris flowers into one of the three species based on their sepal and petal measurements.

IMPLEMENTATION:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

file_path = './Dataset/iris.csv'
df = pd.read_csv(file_path)

X = df.drop('species', axis=1)
y = df['species']

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

model = GaussianNB()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print(y_pred)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: ", accuracy * 100)

user_input_1 = float(input("Enter the value for sepal_length: "))
user_input_2 = float(input("Enter the value for sepal_width: "))
user_input_3 = float(input("Enter the value for petal_length: "))
user_input_4 = float(input("Enter the value for petal_width: "))

feature_columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

user_input = pd.DataFrame([[user_input_1, user_input_2, user_input_3, user_input_4]])

pred = model.predict(user_input)

print(f"The predicted class is: {pred[0]}”)
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul$ python3 Exp_Lab2.py
['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor' 'setosa'
 'versicolor' 'virginica' 'versicolor' 'versicolor' 'virginica' 'setosa'
 'setosa' 'setosa' 'setosa' 'versicolor' 'virginica' 'versicolor'
 'versicolor' 'virginica' 'setosa' 'virginica' 'setosa' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'setosa' 'setosa']
Accuracy:  100.0
Enter the value for sepal_length: 1
Enter the value for sepal_width: 2
Enter the value for petal_length: 1
Enter the value for petal_width: 2
/usr/lib/python3/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but GaussianNB was fitted
with feature names
  warnings.warn(
The predicted class is: versicolor
```

RESULT:

Program for implementing Naive bayes classifier for a given dataset is executed successfully and new tuple is classified using the same.

EXPERIMENT NO: 3
DATE:

ID3 Decision Tree

AIM:

Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

ALGORITHM:

Notations:
The following notations are used in the algorithm:

S	The set of examples
C	The set of class labels
F	The set of features
A	An arbitrary feature (attribute) Values(A) The set of values of the feature A
v	An arbitrary value of A
Sv	The set of examples with A = v
Root	The root node of a tree

ID3 Decision Tree:
ID3 stands for Iterative Dichotomiser 3 and is named such because the algorithm iteratively (repeatedly) dichotomizes (divides) features into two or more groups at each step. ID3 algorithm selects the best feature at each step while building a Decision tree. ID3 uses Information Gain or just Gain to find the best feature.

Dataset is denoted as S, entropy is calculated as:

$$\mathbf{Entropy(S) = - \sum p_i * \log_2(p_i) \; ; \; i = 1 \text{ to } n}$$

$$\mathbf{IG(S, A) = Entropy(S) - \sum((|S_v| / |S|) * Entropy(S_v))}$$

Algorithm ID3(S, F, C):

```
1. Create a root node for the tree.
2. if (all examples in S are positive) then
3. return single node tree Root with label
4. end if
5. if (all examples are negative) then
6. return single node tree Root with label
7. end if
8. if (number of features is 0) then
9. return single node tree Root with label equal to the most common class label. 10. else
10. Let A be the feature in F with the highest information gain.
11. Assign A to the Root node in decision tree.
12. for all (values v of A) do
13. Add a new tree branch below Root corresponding to v.
```

```
14. if (Sv is empty) then
15. Below this branch add a leaf node with label equal to the most common class label in
the set S.
16. else
17. Below this branch add the subtree formed by applying the same algorithm ID3 with the
18. end if
19. end for
20. end if
```

DESCRIPTION:

Data Set Description:
The dataset has 51 tuples and four attributes: outlook, temperature, humidity and wind. The target variable is given as answer “Yes” or “No”.

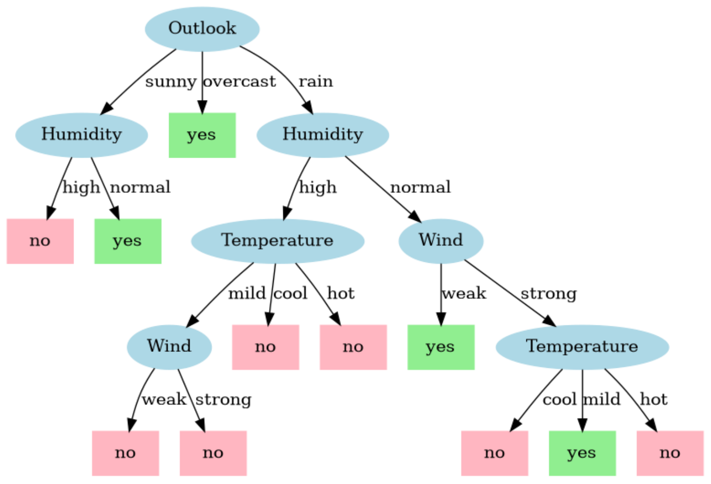
IMPLEMENTATION:

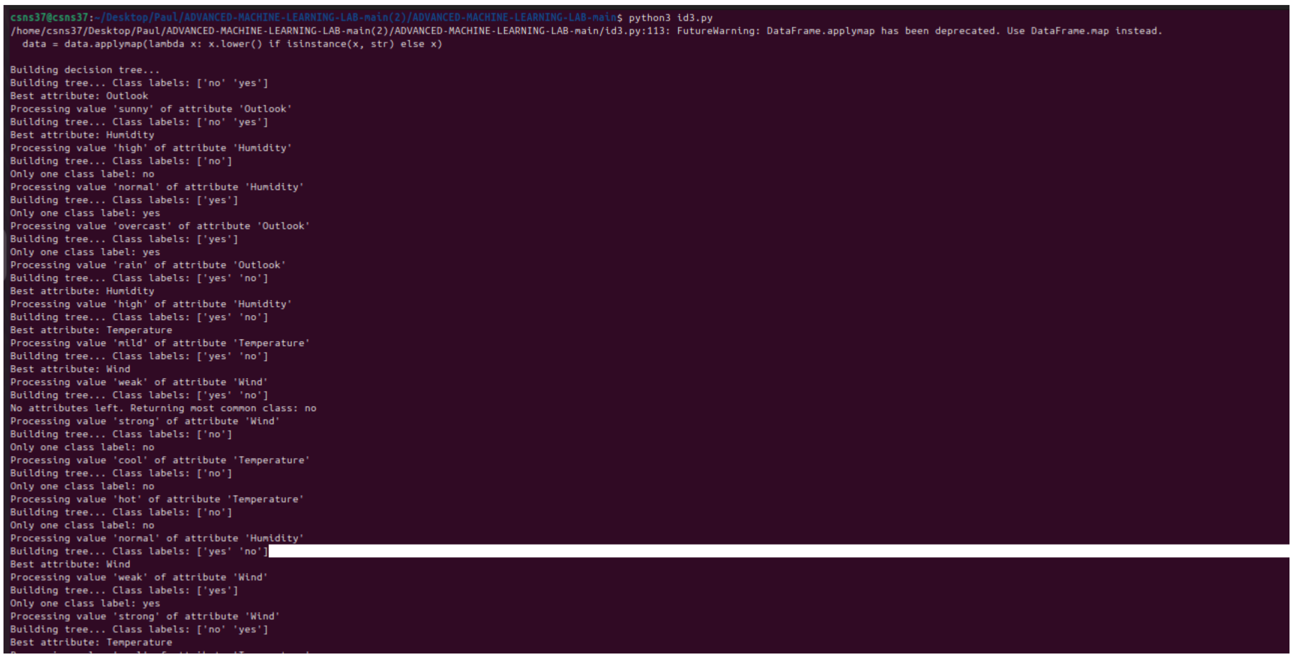
```python
import pandas as pd
import math
from graphviz import Digraph

# Calculate entropy
def entropy(data):
    total = len(data)
    class_counts = data.iloc[:, -1].value_counts()
    ent = 0
    for count in class_counts:
        p = count / total
        ent -= p * math.log2(p)
    return ent

# Calculate information gain
def information_gain(data, attribute):
    total_entropy = entropy(data)
    total = len(data)
    values = data[attribute].unique()
    weighted_entropy = 0
    for value in values:
        subset = data[data[attribute] == value]
        weighted_entropy += (len(subset) / total) * entropy(subset)
    return total_entropy - weighted_entropy

# Find the best attribute
def best_attribute(data, attributes):
    gains = {attr: information_gain(data, attr) for attr in attributes}
    return max(gains, key=gains.get)

# Build the decision tree
```

```
def build_tree(data, attributes):
    class_labels = data.iloc[:, -1].unique()
    print(f"Building tree... Class labels: {class_labels}")
    if len(class_labels) == 1:
        print(f"Only one class label: {class_labels[0]}")
        return class_labels[0]
    if not attributes:
        mode_class = data.iloc[:, -1].mode()[0]
        print(f"No attributes left. Returning most common class: {mode_class}")
        return mode_class
    best_attr = best_attribute(data, attributes)
    print(f"Best attribute: {best_attr}")
    tree = {best_attr: {}}
    remaining_attributes = [attr for attr in attributes if attr != best_attr]
    for value in data[best_attr].unique():
        print(f"Processing value '{value.strip()}' of attribute '{best_attr}'")
        subset = data[data[best_attr] == value]
        if subset.empty:
            mode_class = data.iloc[:, -1].mode()[0]
            print(f"Subset is empty for '{value}'. Returning most common class: {mode_class}")
            tree[best_attr][value.strip()] = mode_class
        else:
            tree[best_attr][value.strip()] = build_tree(subset, remaining_attributes)
    return tree

# Classify a new sample
def classify(tree, sample, data):
    print(f"Classifying sample: {sample}")
    if not isinstance(tree, dict):
        print(f"Reached leaf node: {tree}")
        return tree
    root = next(iter(tree))
    value = sample[root]
    print(f"Root: {root}, Sample value: {value}")

    # Check if the value exists in the tree path
    if value.strip() not in tree[root]:  # Strip spaces from both the tree and input
        print(f"Value '{value.strip()}' not found in tree. Returning most common class.")
        return data.iloc[:, -1].mode()[0]  # Fallback to the most common class

    return classify(tree[root][value.strip()], sample, data)
def plot_tree_simple(tree, dot=None, parent=None, label=None, node_counter=None):
    if node_counter is None:
        node_counter = [0]  # List to maintain a counter between recursive calls

    if dot is None:
        dot = Digraph(format='png', graph_attr={'rankdir': 'TB'})
    if isinstance(tree, dict):  # Non-leaf node
```

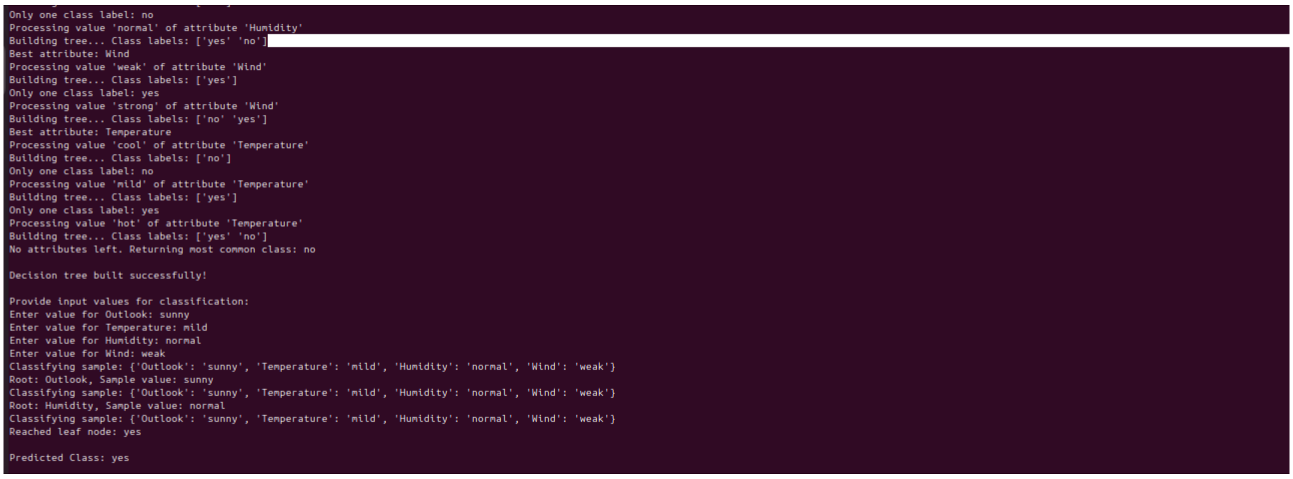
```
        root = next(iter(tree))  # Get the root attribute of this subtree
        node_id = f"{root}_{node_counter[0]}"  # Unique ID based on the root attribute and a counter
        dot.node(node_id, root, shape='ellipse', style='filled', color='lightblue')
        if parent:
            dot.edge(parent, node_id, label=label)

        node_counter[0] += 1  # Increment the node counter for the next node

        for value, subtree in tree[root].items():
            plot_tree_simple(subtree, dot, parent=node_id, label=str(value), node_counter=node_counter)
    else:  # Leaf node
        leaf_id = f"leaf_{node_counter[0]}"  # Unique leaf ID based on the counter
            dot.node(leaf_id, str(tree), shape='box', style='filled', color='lightgreen' if tree == 'yes' else
'lightpink')

        if parent:
            dot.edge(parent, leaf_id, label=str(label))

        node_counter[0] += 1  # Increment the node counter

    return dot
file_path = 'data.csv'
data = pd.read_csv(file_path)
data.columns = data.columns.str.strip()
data = data.applymap(lambda x: x.lower() if isinstance(x, str) else x)
attributes = list(data.columns[:-1])
print("\nBuilding decision tree...")
tree = build_tree(data, attributes)
print("\nDecision tree built successfully!")
dot = plot_tree_simple(tree)
dot.render("decision_tree", format="png", cleanup=True)
dot.view()
while True:
    # Classify a new sample
    print("\nProvide input values for classification:")
    new_sample = {}
    for attr in attributes:
        new_sample[attr] = input(f"Enter value for {attr}: ").strip().lower()  # Strip extra spaces and
convert to lower

    # Predict class
    predicted_class = classify(tree, new_sample, data)
    print(f"\nPredicted Class: {predicted_class}”)
```

OUTPUT:





```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 id3.py
/home/csns37/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main/id3.py:113: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
  data = data.applymap(lambda x: x.lower() if isinstance(x, str) else x)

Building decision tree...
Building tree... Class labels: ['no' 'yes']
Best attribute: Outlook
Processing value 'sunny' of attribute 'Outlook'
Building tree... Class labels: ['no' 'yes']
Best attribute: Humidity
Processing value 'high' of attribute 'Humidity'
Building tree... Class labels: ['no']
Only one class label: no
Processing value 'normal' of attribute 'Humidity'
Building tree... Class labels: ['yes']
Only one class label: yes
Processing value 'overcast' of attribute 'Outlook'
Building tree... Class labels: ['yes']
Only one class label: yes
Processing value 'rain' of attribute 'Outlook'
Building tree... Class labels: ['yes' 'no']
Best attribute: Humidity
Processing value 'high' of attribute 'Humidity'
Building tree... Class labels: ['yes' 'no']
Best attribute: Temperature
Processing value 'mild' of attribute 'Temperature'
Building tree... Class labels: ['yes' 'no']
Best attribute: Wind
Processing value 'weak' of attribute 'Wind'
Building tree... Class labels: ['yes' 'no']
No attributes left. Returning most common class: no
Processing value 'strong' of attribute 'Wind'
Building tree... Class labels: ['no']
Only one class label: no
Processing value 'cool' of attribute 'Temperature'
Building tree... Class labels: ['no']
Only one class label: no
Processing value 'hot' of attribute 'Temperature'
Building tree... Class labels: ['no']
Only one class label: no
Processing value 'normal' of attribute 'Humidity'
Building tree... Class labels: ['yes' 'no']
Best attribute: Wind
Processing value 'weak' of attribute 'Wind'
Building tree... Class labels: ['yes']
Only one class label: yes
Processing value 'strong' of attribute 'Wind'
Building tree... Class labels: ['no' 'yes']
Best attribute: Temperature
```



```
Only one class label: no
Processing value 'normal' of attribute 'Humidity'
Building tree... Class labels: ['yes' 'no']
Best attribute: Wind
Processing value 'weak' of attribute 'Wind'
Building tree... Class labels: ['yes']
Only one class label: yes
Processing value 'strong' of attribute 'Wind'
Building tree... Class labels: ['no' 'yes']
Best attribute: Temperature
Processing value 'cool' of attribute 'Temperature'
Building tree... Class labels: ['no']
Only one class label: no
Processing value 'mild' of attribute 'Temperature'
Building tree... Class labels: ['yes']
Only one class label: yes
Processing value 'hot' of attribute 'Temperature'
Building tree... Class labels: ['yes' 'no']
No attributes left. Returning most common class: no

Decision tree built successfully!

Provide input values for classification:
Enter value for Outlook: sunny
Enter value for Temperature: mild
Enter value for Humidity: normal
Enter value for Wind: weak
Classifying sample: {'Outlook': 'sunny', 'Temperature': 'mild', 'Humidity': 'normal', 'Wind': 'weak'}
Root: Outlook, Sample value: sunny
Classifying sample: {'Outlook': 'sunny', 'Temperature': 'mild', 'Humidity': 'normal', 'Wind': 'weak'}
Root: Humidity, Sample value: normal
Classifying sample: {'Outlook': 'sunny', 'Temperature': 'mild', 'Humidity': 'normal', 'Wind': 'weak'}
Reached leaf node: yes

Predicted Class: yes
```

RESULT:

Program implementing decision tree based on ID3 algorithm executed successfully, Used the decision tree to classify a new sample and the output is obtained.

EXPERIMENT NO: 4
DATE:

K-NEAREST NEIGHBOUR ALGORITHM

AIM:

To write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions.

ALGORITHM:

```
1.Load the dataset containing features and corresponding labels.
2.Define the value of k (number of nearest neighbors).
3.For a given test instance x:
        3.1. Compute the distance between x and all training data points using a distance metric (e.g.,
        Euclidean distance).
        3.2. Sort the training instances based on their distances to x.
        3.3. Select the top k closest training instances.
        3.4. Identify the most common label among these k instances.
        3.5. Assign the most common label to x as the predicted class.
4.Repeat Step 3 for all test instances.
5.Evaluate the model's performance using metrics such as accuracy, confusion matrix, etc.
6.Accept user input for features and predict the class using the trained model.
```

DESCRIPTION:

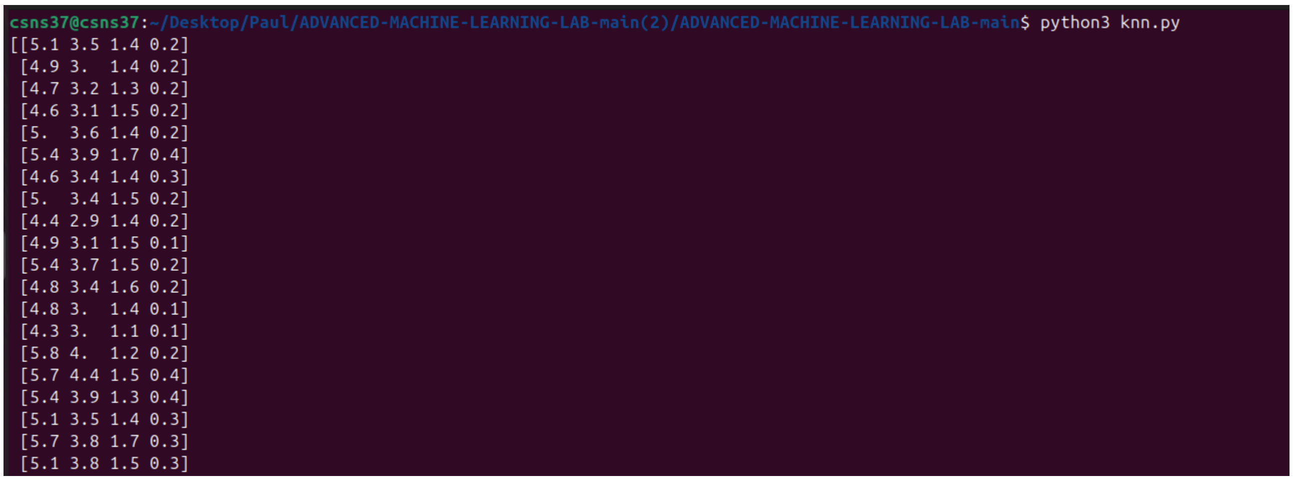
k-Nearest Neighbors Classifier:
k-Nearest Neighbors (k-NN) is a simple, non-parametric, and instance-based machine learning algorithm used for classification and regression tasks. It predicts the class of a sample based on the majority class among its kkk nearest neighbors in the feature space.

Data Set Description:
The Iris dataset consists of 150 samples from three species of Iris flowers: setosa, versicolor, and virginica. Each sample is described by four features:
1. Sepal length (cm)
2. Sepal width (cm)
3. Petal length (cm)
4. Petal width (cm)

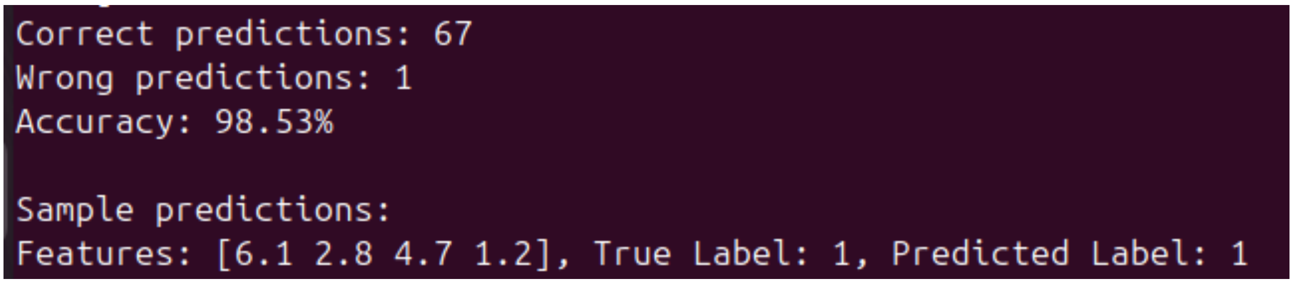
IMPLEMENTATION:

```
import numpy as np
import pandas as pd
from collections import Counter
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels
print(X, y)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.45, random_state=42)

# k-NN Algorithm Implementation
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))

def knn(X_train, y_train, X_test, k):
    predictions = []
    for test_point in X_test:
        # Calculate distances from the test point to all training points
        distances = [euclidean_distance(test_point, train_point) for train_point in X_train]
        # Get indices of the k smallest distances
        k_indices = np.argsort(distances)[:k]
        # Get the labels of the k nearest neighbors
        k_nearest_labels = [y_train[i] for i in k_indices]
        # Get the most common class label among the k nearest neighbors
        most_common = Counter(k_nearest_labels).most_common(1)
        predictions.append(most_common[0][0])
    return predictions

# Setting k value
k = 8

# Make predictions using k-NN
predictions = knn(X_train, y_train, X_test, k)

# Compare predictions with actual labels and count correct/wrong predictions
correct_predictions = 0
wrong_predictions = 0
for true, predicted in zip(y_test, predictions):
    if true == predicted:
        correct_predictions += 1
    else:
        wrong_predictions += 1

# Calculate accuracy
accuracy = (correct_predictions / len(y_test)) * 100

# Print results
print(f"Correct predictions: {correct_predictions}")
print(f"Wrong predictions: {wrong_predictions}")
print(f"Accuracy: {accuracy:.2f}%")

# Print sample predictions and their corresponding true labels
print("\nSample predictions:")
for true, predicted, features in zip(y_test, predictions, X_test):
    print(f"Features: {features}, True Label: {true}, Predicted Label: {predicted}")
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 knn.py
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]
 [5.1 3.5 1.4 0.3]
 [5.7 3.8 1.7 0.3]
 [5.1 3.8 1.5 0.3]
```



```
Correct predictions: 67
Wrong predictions: 1
Accuracy: 98.53%

Sample predictions:
Features: [6.1 2.8 4.7 1.2], True Label: 1, Predicted Label: 1
```

RESULT:

Program implementing k-Nearest Neighbors (k-NN) algorithm executed successfully. Used the K-NN classifier to classify a new sample, and the output is obtained.

EXPERIMENT NO: 5
DATE:

SINGLE LAYER NEURAL NETWORK

AIM:

Implement a single layer neural network and for different logic gates.

ALGORITHM:

Notations:
The following notations are used in the algorithm:

X	Feature matrix (independent variables).
y	Target vector (dependent variable).
Xtrain	Training data for features (subset of X).
Xtest	Testing data for features (subset of X).
ytrain	Training data for the target variable (subset of y).
ytest	Testing data for the target variable (subset of y).
Y	Predicted values of the target variable (y) .
n	Number of samples in the dataset.
MSE	Mean Squared Error

Algorithm :

```
1. Define the Step Function (Activation Function) As:
    1.  step(x) = {1 if x >= 0; 0 if x < 0}
2. Initialize the input data for the AND and OR gates:
        2.1. For the AND gate:
                Inputs: and_inputs=[[0,0],[0,1],[1,0],[1,1]]
                Expected output: and_expected_output=[0,0,0,1]
        2.2. For the OR gate:
                Inputs: or_inputs=[[0,0],[0,1],[1,0],[1,1]]
                Expected output: or_expected_output=[0,1,1,1]
3. Initialize the weights and bias for both AND and OR gates:
        3.1. weights=[0.5,0.5]
        3.2. bias=0.0
4. Train the neural network using the train_gate function:
        4.1. For each epoch, calculate the weighted sum of inputs and apply the activation function.
        4.2. Compare the predicted output with the expected output.
        4.3. Update the weights and bias based on the error:
```

$$\text{weights} += \text{learning\_rate} \times \text{error} \times \text{inputs}[i]$$

$$\text{bias} += \text{learning\_rate} \times \text{error}$$

4.4. Repeat the process for a specified number of epochs or until the error is zero (convergence).

5. Test the trained model using the test_gate function:
   5.1. Calculate the weighted sum of inputs and apply the step function to make predictions.
6. Print the final weights and bias after training for both AND and OR gates.

DESCRIPTION:

Single Layer Neural Network:

A single-layer neural network consists of input nodes connected to an output node, with each connection having an associated weight and a bias term. The network uses a step function as the activation function to produce binary outputs (0 or 1).

The general equation for a single-layer neural network is:

$$y=f(\sum_{i=1}^{n} w_i x_i + b)$$

Where:

- y: Output of the network
- f: Activation function (step function)
- w_i: Weights associated with the inputs
- x_i: Input values
- b: Bias

Logic Gates:

AND GATE

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR GATE

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

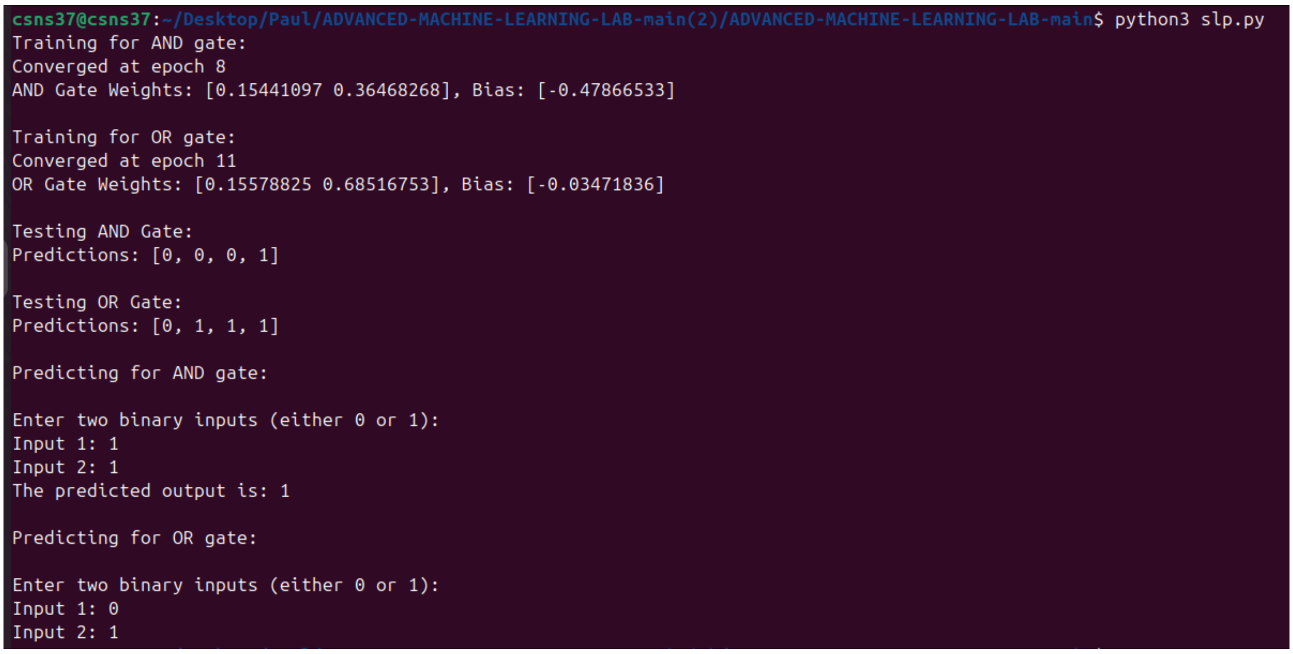
IMPLEMENTATION:

```
import numpy as np

# Define the step function (activation function)
def step_function(x):
    return 1 if x >= 0 else 0
```

```
# Define a function to simulate a single-layer neural network with convergence tracking
def train_logic_gate(inputs, expected_output, learning_rate=0.1, epochs=10000):
    # Initialize weights and bias
    weights = np.random.rand(2)  # Two inputs
    bias = np.random.rand(1)     # One bias

    # Training loop
    for epoch in range(epochs):
        total_error = 0  # To track if the model has no errors
        for i in range(len(inputs)):
            # Calculate the weighted sum (input * weights + bias)
            weighted_sum = np.dot(inputs[i], weights) + bias
            # Apply the activation function (step function)
            output = step_function(weighted_sum)
            # Calculate the error (difference between expected and actual output)
            error = expected_output[i] - output
            total_error += abs(error)

            # Update weights and bias if there's an error
            weights += learning_rate * error * inputs[i]
            bias += learning_rate * error

        # If there is no error after this epoch, then stop training
        if total_error == 0:
            print(f"Converged at epoch {epoch+1}")
            break  # Stop training as the model has converged

    return weights, bias

# Define the dataset for AND and OR gates
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
# Expected output for AND gate
and_output = np.array([0, 0, 0, 1])
# Expected output for OR gate
or_output = np.array([0, 1, 1, 1])

# Train for AND gate
print("Training for AND gate:")
and_weights, and_bias = train_logic_gate(inputs, and_output)
print(f"AND Gate Weights: {and_weights}, Bias: {and_bias}")

# Train for OR gate
print("\nTraining for OR gate:")
or_weights, or_bias = train_logic_gate(inputs, or_output)
print(f"OR Gate Weights: {or_weights}, Bias: {or_bias}")

# Define a function to make predictions
```

```
def predict(inputs, weights, bias):
    predictions = []
    for i in range(len(inputs)):
        weighted_sum = np.dot(inputs[i], weights) + bias
        output = step_function(weighted_sum)
        predictions.append(output)
    return predictions

# Test the model with the trained weights and biases for both gates
print("\nTesting AND Gate:")
and_predictions = predict(inputs, and_weights, and_bias)
print(f"Predictions: {and_predictions}")

print("\nTesting OR Gate:")
or_predictions = predict(inputs, or_weights, or_bias)
print(f"Predictions: {or_predictions}")

# User prediction function
def user_prediction(weights, bias):
    print("\nEnter two binary inputs (either 0 or 1):")
    try:
        input1 = int(input("Input 1: "))
        input2 = int(input("Input 2: "))
        if input1 not in [0, 1] or input2 not in [0, 1]:
            print("Invalid input. Please enter 0 or 1.")
            return
        input_data = np.array([input1, input2])
        prediction = predict([input_data], weights, bias)
        print(f"The predicted output is: {prediction[0]}")
    except ValueError:
        print("Invalid input. Please enter integers 0 or 1.")

# Ask user for a prediction for AND Gate
print("\nPredicting for AND gate:")
user_prediction(and_weights, and_bias)

# Ask user for a prediction for OR Gate
print("\nPredicting for OR gate:")
user_prediction(or_weights, or_bias)
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 slp.py
Training for AND gate:
Converged at epoch 8
AND Gate Weights: [0.15441097 0.36468268], Bias: [-0.47866533]

Training for OR gate:
Converged at epoch 11
OR Gate Weights: [0.15578825 0.68516753], Bias: [-0.03471836]

Testing AND Gate:
Predictions: [0, 0, 0, 1]

Testing OR Gate:
Predictions: [0, 1, 1, 1]

Predicting for AND gate:

Enter two binary inputs (either 0 or 1):
Input 1: 1
Input 2: 1
The predicted output is: 1

Predicting for OR gate:

Enter two binary inputs (either 0 or 1):
Input 1: 0
Input 2: 1
```

RESULT:

Program implementing single-layer neural network for AND and OR gates executed successfully. Used the trained model to classify the gate outputs for all input combinations and the output is obtained.

EXPERIMENT NO: 6
DATE:

BACKPROPAGATION ALGORITHM

AIM:

Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

DESCRIPTION:

Back Propagation:
Backpropagation is a supervised learning algorithm used to train neural networks. It involves two main steps:

1. Forward Propagation: Input data is passed through the network, and activations are calculated at each layer using a weighted sum and an activation function.
2. Backward Propagation: The error (difference between predicted and actual output) is propagated back through the network. Gradients of the error with respect to weights are calculated using the chain rule.
3. Weight Update: Weights and biases are updated by subtracting the gradient multiplied by the learning rate.

ALGORITHM:

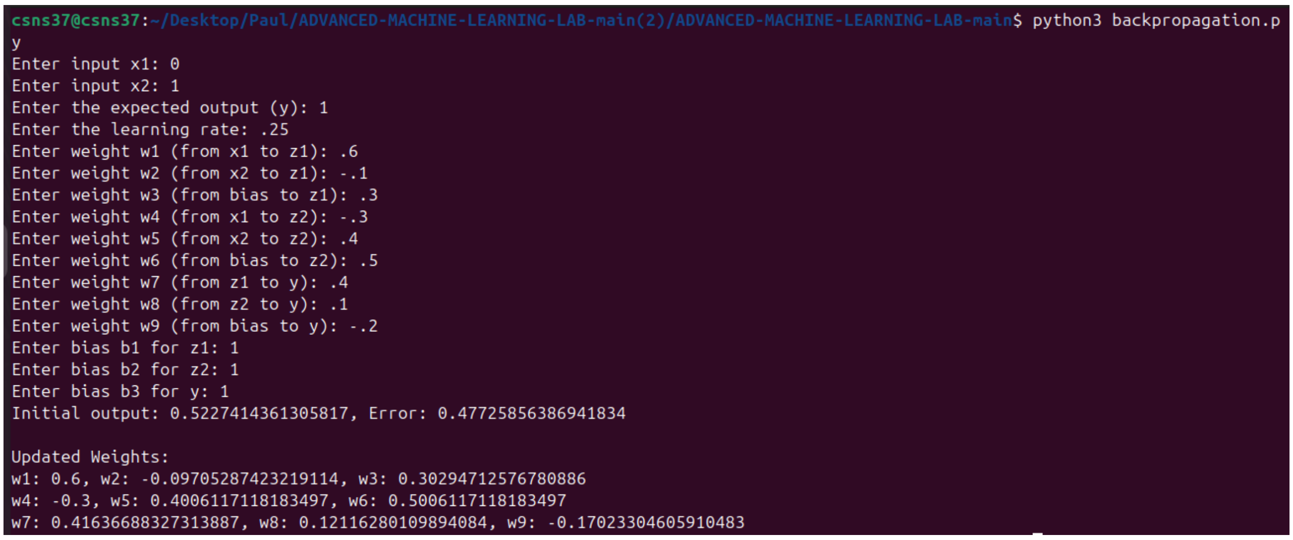
```
1. Initialize the network with random weights and biases for each layer.
2. For each input in the training dataset,
        2.1. calculate the weighted sum for each node in the hidden layers and apply the activation
                function to get activations
        2.2. then compute the output using the activations from the last hidden layer.
3. Calculate the difference between the predicted output and the actual target output.
4. Compute the gradient of the error with respect to the output layer weights.
5. Propagate the error back through each hidden layer, calculating gradients for each node.
6. Update the weights and biases using the calculated gradients and the learning rate.
7. Repeat the process for a specified number of epochs or until the error converges.
```

IMPLEMENTATION:

```
import numpy as np

# Sigmoid Activation Function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Function to initialize parameters based on user input
```

```
def initialize_parameters():
    # Get user inputs for parameters
    X = np.array([float(input("Enter input x1: ")), float(input("Enter input x2: "))])  # Inputs [x1, x2]
    y_true = float(input("Enter the expected output (y): "))  # Expected output

    learning_rate = float(input("Enter the learning rate: "))

    # Weights from input to hidden layer
    w1 = float(input("Enter weight w1 (from x1 to z1): "))
    w2 = float(input("Enter weight w2 (from x2 to z1): "))
    w3 = float(input("Enter weight w3 (from bias to z1): "))

    w4 = float(input("Enter weight w4 (from x1 to z2): "))
    w5 = float(input("Enter weight w5 (from x2 to z2): "))
    w6 = float(input("Enter weight w6 (from bias to z2): "))

    # Weights from hidden layer to output layer
    w7 = float(input("Enter weight w7 (from z1 to y): "))
    w8 = float(input("Enter weight w8 (from z2 to y): "))
    w9 = float(input("Enter weight w9 (from bias to y): "))

    # Biases
    b1 = float(input("Enter bias b1 for z1: "))
    b2 = float(input("Enter bias b2 for z2: "))
    b3 = float(input("Enter bias b3 for y: "))

    return X, y_true, learning_rate, w1, w2, w3, w4, w5, w6, w7, w8, w9, b1, b2, b3

# Main function
def main():
    # Initialize parameters
    X, y_true, learning_rate, w1, w2, w3, w4, w5, w6, w7, w8, w9, b1, b2, b3 = initialize_parameters()

    # Forward propagation
    z1_input = w1 * X[0] + w2 * X[1] + w3 * b1  # Input to z1
    z1 = sigmoid(z1_input)

    z2_input = w4 * X[0] + w5 * X[1] + w6 * b2  # Input to z2
    z2 = sigmoid(z2_input)

    y_input = w7 * z1 + w8 * z2 + w9 * b3  # Input to y
    y_pred = sigmoid(y_input)

    # Compute the error (Mean Squared Error)
    error = y_true - y_pred
    print(f"Initial output: {y_pred}, Error: {error}")

    # Backpropagation
    delta_y = error * sigmoid_derivative(y_pred)

    # Gradients for weights from hidden layer to output layer
    grad_w7 = delta_y * z1
    grad_w8 = delta_y * z2
    grad_w9 = delta_y * b3

    # Calculate the gradient of the hidden layer
    delta_z1 = delta_y * w7 * sigmoid_derivative(z1)
    delta_z2 = delta_y * w8 * sigmoid_derivative(z2)

    # Gradients for weights from input layer to hidden layer
    grad_w1 = delta_z1 * X[0]
    grad_w2 = delta_z1 * X[1]
    grad_w3 = delta_z1 * b1

    grad_w4 = delta_z2 * X[0]
    grad_w5 = delta_z2 * X[1]
    grad_w6 = delta_z2 * b2

    # Update weights with the gradients
    w1 += learning_rate * grad_w1
    w2 += learning_rate * grad_w2
    w3 += learning_rate * grad_w3
    w4 += learning_rate * grad_w4
    w5 += learning_rate * grad_w5
    w6 += learning_rate * grad_w6
    w7 += learning_rate * grad_w7
    w8 += learning_rate * grad_w8
    w9 += learning_rate * grad_w9

    # Output updated weights after one backpropagation iteration
    print("\nUpdated Weights:")
    print(f"w1: {w1}, w2: {w2}, w3: {w3}")
    print(f"w4: {w4}, w5: {w5}, w6: {w6}")
    print(f"w7: {w7}, w8: {w8}, w9: {w9}")

# Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 backpropagation.p
y
Enter input x1: 0
Enter input x2: 1
Enter the expected output (y): 1
Enter the learning rate: .25
Enter weight w1 (from x1 to z1): .6
Enter weight w2 (from x2 to z1): -.1
Enter weight w3 (from bias to z1): .3
Enter weight w4 (from x1 to z2): -.3
Enter weight w5 (from x2 to z2): .4
Enter weight w6 (from bias to z2): .5
Enter weight w7 (from z1 to y): .4
Enter weight w8 (from z2 to y): .1
Enter weight w9 (from bias to y): -.2
Enter bias b1 for z1: 1
Enter bias b2 for z2: 1
Enter bias b3 for y: 1
Initial output: 0.5227414361305817, Error: 0.47725856386941834

Updated Weights:
w1: 0.6, w2: -0.09705287423219114, w3: 0.30294712576780886
w4: -0.3, w5: 0.4006117118183497, w6: 0.5006117118183497
w7: 0.41636688327313887, w8: 0.12116280109894084, w9: -0.17023304605910483
```

RESULT:

Program implementing backpropagation in a neural network executed successfully. The weights and biases were updated during training, and the final updated weights and biases were displayed for each epoch.

EXPERIMENT NO: 7
DATE:

EM AND K-MEANS ALGORITHM

AIM:

Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering.

DESCRIPTION :

EM Algorithm :
The primary aim of the EM algorithm is to estimate the missing data in the latent variables through observed data in datasets. The EM algorithm or latent variable model has a broad range of real-life applications in machine learning.

K-Means Algorithm:
K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.

ALGORITHM:

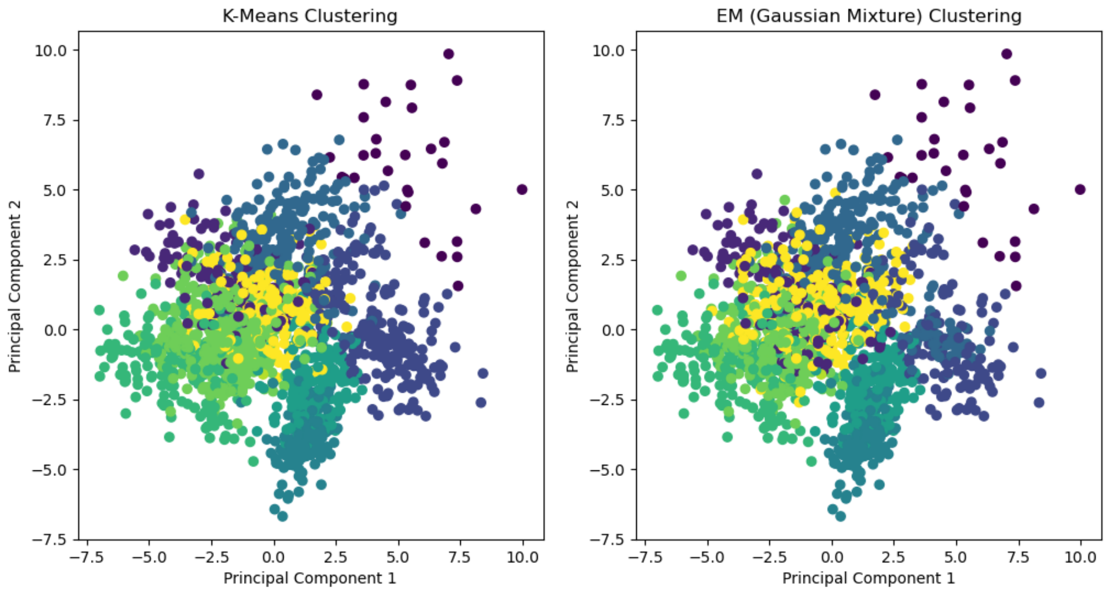
1. Import necessary modules
2. Read the iris dataset
3. Build K-Means cluster model and Gaussian Mixture model using this dataset, with number of clusters as 10
4. Draw the real, k-Means and Gaussian Mixture plots
5. Compute the Silhouette score of both the models.
6. Print Silhouette score for the models

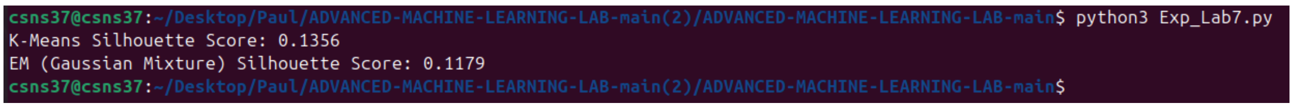
IMPLEMENTATION:

```
from sklearn.datasets import load_digits
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler

# Load Digits dataset
```

```
data = load_digits()
df = pd.DataFrame(data.data, columns=data.feature_names)  # Extract the features (64 pixels)

# Preprocess data: Scaling for better clustering performance
scaler = StandardScaler()
data_scaled = scaler.fit_transform(df)

# Apply K-Means clustering
kmeans = KMeans(n_clusters=10, random_state=42)  # 10 clusters for 10 digits
kmeans_labels = kmeans.fit_predict(data_scaled)

# Apply EM (Gaussian Mixture Model) clustering
gmm = GaussianMixture(n_components=10, random_state=42)  # 10 components for 10 digits
gmm_labels = gmm.fit_predict(data_scaled)

# Compare clustering results using silhouette score
kmeans_silhouette = silhouette_score(data_scaled, kmeans_labels)
gmm_silhouette = silhouette_score(data_scaled, gmm_labels)

# Visualize the clusters: We'll only visualize the first two principal components
# to better understand the clustering in a 2D space
from sklearn.decomposition import PCA

# Apply PCA for dimensionality reduction to 2D for visualization
pca = PCA(n_components=2)
data_pca = pca.fit_transform(data_scaled)

# Visualize the clusters
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(data_pca[:, 0], data_pca[:, 1], c=kmeans_labels, cmap='viridis')
plt.title('K-Means Clustering')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')

plt.subplot(1, 2, 2)
plt.scatter(data_pca[:, 0], data_pca[:, 1], c=gmm_labels, cmap='viridis')
plt.title('EM (Gaussian Mixture) Clustering')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')

plt.show()

# Print the comparison results
print(f'K-Means Silhouette Score: {kmeans_silhouette:.4f}')
print(f'EM (Gaussian Mixture) Silhouette Score: {gmm_silhouette:.4f}')
```

OUTPUT:





```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 Exp_Lab7.py
K-Means Silhouette Score: 0.1356
EM (Gaussian Mixture) Silhouette Score: 0.1179
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$
```

RESULT:

Implemented EM Algorithm to cluster a set of data points and compared the accuracy of the model with k-Means clustering using the same dataset.

EXPERIMENT NO: 8
DATE:

DBSCAN CLUSTERING ALGORITHM

AIM:

Write a program to implement the DBSCAN clustering algorithm.

ALGORITHM:

```
1. Accept the inputs: epsilon , minPts, Coordinates of n points in the form (x,y).
2. Initialize an empty dictionary to store neighbors for each point.
3. For each point iii in the dataset:
        1. Compute the Euclidean distance between i and all other points j.
        2. Add j to the neighbor list
4. Classify points into the following categories:
        1. Core points: Points with at least minpts-1
        2. Border points: Points with fewer than minpts -1, but within the neighbor of a core point
        3. Noise points: Points that do not belong to any cluster.
5. Return the list of core points, border points, and noise points.
```

DESCRIPTION:

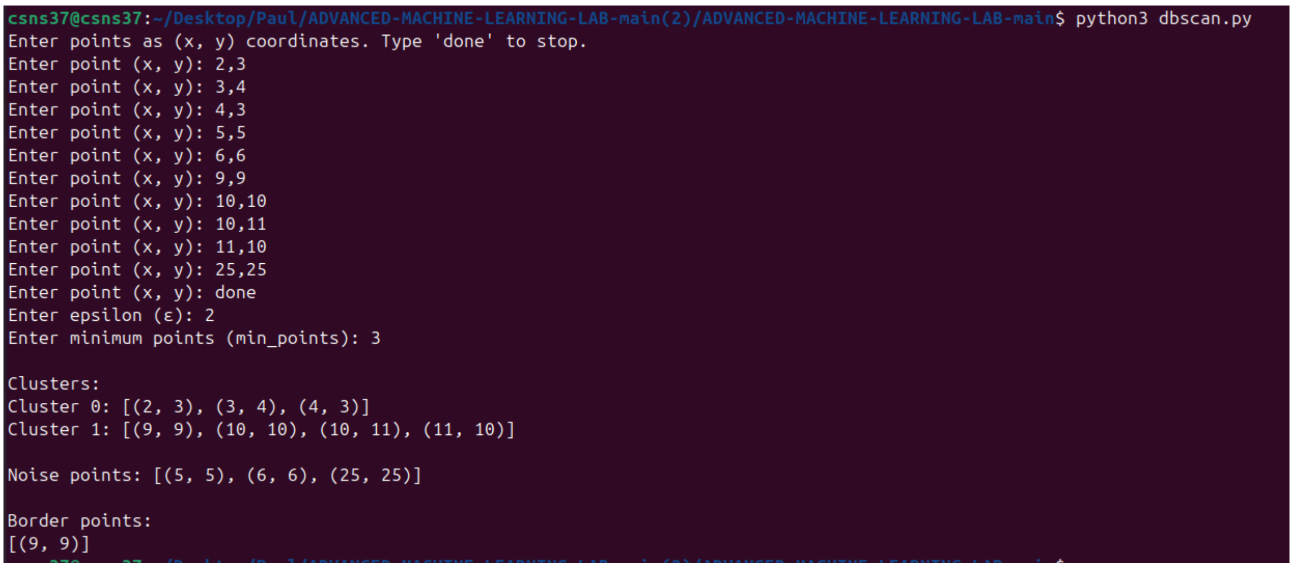
DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups points into clusters based on their spatial proximity. It identifies clusters as dense regions of points separated by sparser regions, making it suitable for datasets with noise and non-linear shapes.

The algorithm works as follows:
1. A point is classified as a core point if it has at least min points within the radius of epsilon
2. A point is classified as a border point if it has fewer than minPts neighbors but is within the neighborhood of a core point.
3. Points that do not fall into the above categories are labeled as noise points.

IMPLEMENTATION:

```
import math

# Function to calculate Euclidean distance between two points
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# DBSCAN Algorithm
def dbscan(points, epsilon, min_points):
    labels = [-1] * len(points)  # -1 denotes noise
    cluster_id = 0
    border_points = set()
    def region_query(point_idx):
        neighbors = []
        for i, p in enumerate(points):
            if euclidean_distance(points[point_idx], p) <= epsilon:
                neighbors.append(i)
        return neighbors

    def expand_cluster(point_idx, neighbors, cluster_id):
        labels[point_idx] = cluster_id
        i = 0
        while i < len(neighbors):
            neighbor_idx = neighbors[i]
            if labels[neighbor_idx] == -1:  # Mark it as part of the cluster
                labels[neighbor_idx] = cluster_id
            elif labels[neighbor_idx] == -1:
                labels[neighbor_idx] = cluster_id
                new_neighbors = region_query(neighbor_idx)
                if len(new_neighbors) >= min_points:
                    neighbors.extend(new_neighbors)
            i += 1

    for i in range(len(points)):
        if labels[i] != -1:  # Skip if already processed
            continue

        neighbors = region_query(i)
        if len(neighbors) >= min_points:
            expand_cluster(i, neighbors, cluster_id)
            cluster_id += 1
        else:
            labels[i] = -1  # Mark as noise

    # Identifying border points
    for i in range(len(points)):
        if labels[i] != -1:  # If not noise
            neighbors = region_query(i)
            if len(neighbors) >= 1 and len(neighbors) < min_points:
                border_points.add(i)

    return labels, border_points

# Get points from the user
def get_points():
    points = []
    print("Enter points as (x, y) coordinates. Type 'done' to stop.")
    while True:
        point = input("Enter point (x, y): ")
        if point.lower() == "done":
            break
        try:
            x, y = map(int, point.split(","))
            points.append((x, y))
        except ValueError:
            print("Invalid input, please enter in the format x,y.")
    return points

# Main function
def main():
    points = get_points()
    epsilon = float(input("Enter epsilon (ε): "))
    min_points = int(input("Enter minimum points (min_points): "))

    # Perform DBSCAN clustering
    labels, border_points = dbscan(points, epsilon, min_points)

    # Display results
    clusters = {}
    for idx, label in enumerate(labels):
        if label != -1:
            if label not in clusters:
                clusters[label] = []
            clusters[label].append(points[idx])

    # Show clusters and noise points
    print("\nClusters:")
    for cluster_id, cluster_points in clusters.items():
        print(f"Cluster {cluster_id}: {cluster_points}")

    noise_points = [points[i] for i in range(len(points)) if labels[i] == -1]
    if noise_points:
        print(f"\nNoise points: {noise_points}")
    else:
        print("\nNo noise points detected.")

    # Show border points
    print("\nBorder points:")
    border_points_list = [points[i] for i in border_points]
    if border_points_list:
        print(border_points_list)
    else:
        print("No border points detected.")

# Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 dbscan.py
Enter points as (x, y) coordinates. Type 'done' to stop.
Enter point (x, y): 2,3
Enter point (x, y): 3,4
Enter point (x, y): 4,3
Enter point (x, y): 5,5
Enter point (x, y): 6,6
Enter point (x, y): 9,9
Enter point (x, y): 10,10
Enter point (x, y): 10,11
Enter point (x, y): 11,10
Enter point (x, y): 25,25
Enter point (x, y): done
Enter epsilon (ε): 2
Enter minimum points (min_points): 3

Clusters:
Cluster 0: [(2, 3), (3, 4), (4, 3)]
Cluster 1: [(9, 9), (10, 10), (10, 11), (11, 10)]

Noise points: [(5, 5), (6, 6), (25, 25)]

Border points:
[(9, 9)]
```

RESULT:

The DBSCAN classification executed successfully. Points were classified into Core Points, Border Points, and Noise Points.

EXPERIMENT NO: 9
DATE:

PRINCIPLE COMPONENT ANALYSIS

AIM:

Write a program to implement Dimensionality reduction using Principle Component Analysis (PCA) method.

ALGORITHM:

1. Input the dataset with n samples and m features.
2. Compute the mean of each feature and center the data by subtracting the mean from each feature value.
3. Calculate the covariance matrix of the centered data.
4. Perform eigen decomposition on the covariance matrix to obtain eigenvalues and eigenvectors.
5. Sort the eigenvalues in descending order and select the top k eigenvectors corresponding to the largest eigenvalues.
6. Project the data onto the selected k eigenvectors to obtain the reduced-dimensional data.
7. Output the reduced data, covariance matrix, eigenvalues, and variance explained by each principal component.

DESCRIPTION:

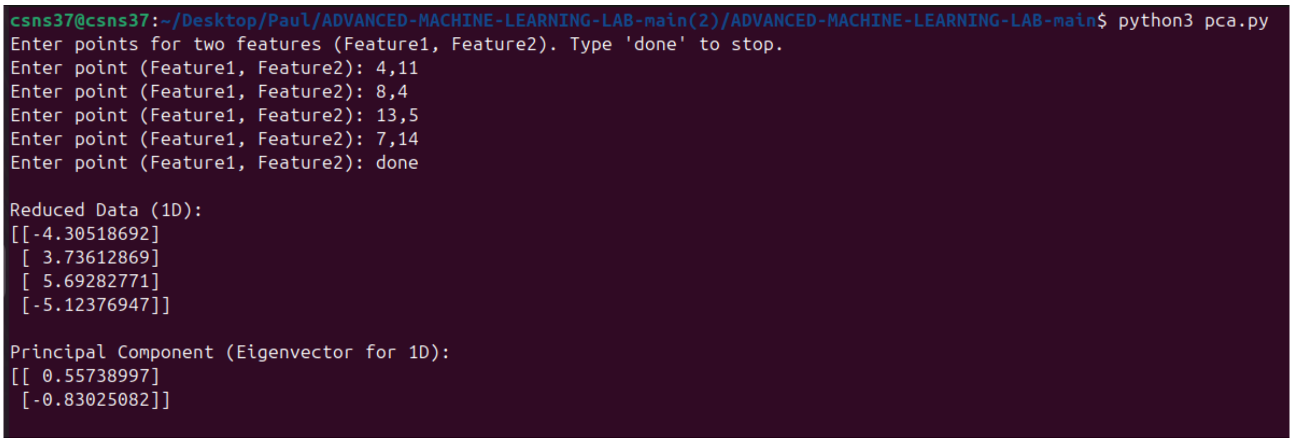
Principal Component Analysis (PCA):
Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms a high-dimensional dataset into a lower-dimensional space while preserving as much variance as possible. It identifies the directions (principal components) in which the data varies the most.
Steps:

1. Center the data by subtracting the mean.
2. Compute the covariance matrix.
3. Perform eigen decomposition to obtain eigenvalues (variance) and eigenvectors (principal components).
4. Select top k components based on eigenvalues.
5. Project the data onto the selected components.

IMPLEMENTATION:

```
import numpy as np

# Step 1: Function to perform PCA
def pca(data, num_components=1):
    # Mean centering the data
    mean = np.mean(data, axis=0)
    centered_data = data - mean
    # Step 2: Calculate the covariance matrix
    cov_matrix = np.cov(centered_data.T)

    # Step 3: Compute the eigenvalues and eigenvectors of the covariance matrix
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

    # Step 4: Sort the eigenvectors by eigenvalues in descending order
    sorted_indices = np.argsort(eigenvalues)[::-1]
    sorted_eigenvectors = eigenvectors[:, sorted_indices]

    # Step 5: Select the top eigenvectors (principal components)
    top_eigenvectors = sorted_eigenvectors[:, :num_components]

    # Step 6: Ensure the eigenvector has a consistent sign
    for i in range(top_eigenvectors.shape[1]):
        # Make sure the largest value in the eigenvector is positive
        if top_eigenvectors[0, i] < 0:
            top_eigenvectors[:, i] *= -1

    # Step 7: Project the data onto the top eigenvectors
    reduced_data = centered_data.dot(top_eigenvectors)

    return reduced_data, top_eigenvectors

# Step 2: Function to get input points from the user
def get_input_points():
    points = []
    print("Enter points for two features (Feature1, Feature2). Type 'done' to stop.")
    while True:
        point = input("Enter point (Feature1, Feature2): ")
        if point.lower() == 'done':
            break
        try:
            feature1, feature2 = map(int, point.split(','))
            points.append([feature1, feature2])
        except ValueError:
            print("Invalid input. Please enter two integers separated by a comma.")
    return np.array(points)

# Step 3: Main function to run the PCA
def main():
    # Get input points
    data = get_input_points()

    # Perform PCA to reduce from 2D to 1D
    reduced_data, principal_components = pca(data, num_components=1)

    # Display the results
    print("\nReduced Data (1D):")
    print(reduced_data)

    print("\nPrincipal Component (Eigenvector for 1D):")
    print(principal_components)

    print("\nMean of the original data:")
    print(np.mean(data, axis=0))

# Step 4: Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 pca.py
Enter points for two features (Feature1, Feature2). Type 'done' to stop.
Enter point (Feature1, Feature2): 4,11
Enter point (Feature1, Feature2): 8,4
Enter point (Feature1, Feature2): 13,5
Enter point (Feature1, Feature2): 7,14
Enter point (Feature1, Feature2): done

Reduced Data (1D):
[[-4.30518692]
 [ 3.73612869]
 [ 5.69282771]
 [-5.12376947]]

Principal Component (Eigenvector for 1D):
[[ 0.55738997]
 [-0.83025082]]
```

RESULT:

Program executed successfully. Principal Component Analysis (PCA) was performed on the input dataset.

EXPERIMENT NO: 10
DATE:

SUPPORT VECTOR MACHINE

AIM:

Write a program to implement Support Vector Machine algorithm to classify the iris data set. Print both correct and wrong predictions.

ALGORITHM:

Algorithm :

1. Load the Iris dataset into a variable.
2. Split the dataset into features X (sepal length, sepal width, petal length, petal width) and target y (species).
3. Split the data into training and testing sets using an 80%-20% split.
4. Create a Support Vector Machine (SVM) model with a linear kernel.
5. Train the SVM model using the training data Xtrain,ytrain.
6. Use the trained model to predict the target variable ypred using the test data Xtest.
7. Calculate the accuracy score by comparing the predicted values ypred with the actual values ytest.
8. Print the accuracy score.
9. Identify and print the correct and wrong predictions by comparing ytest and ypred.
10. Accept user input for flower features (sepal length, sepal width, petal length, petal width).
11. Use the trained model to predict the class for the user input.
12. Display the predicted class for the user input.

DESCRIPTION:

Support Vector Machine:
SVM aims to find a hyperplane that best separates the data into classes. The equation of the hyperplane in a 2-dimensional space is:

$$w*x+b = 0$$

where w is the weight vector, x is the feature vector, and b is the bias term. The SVM algorithm maximizes the margin between the classes to achieve optimal classification.

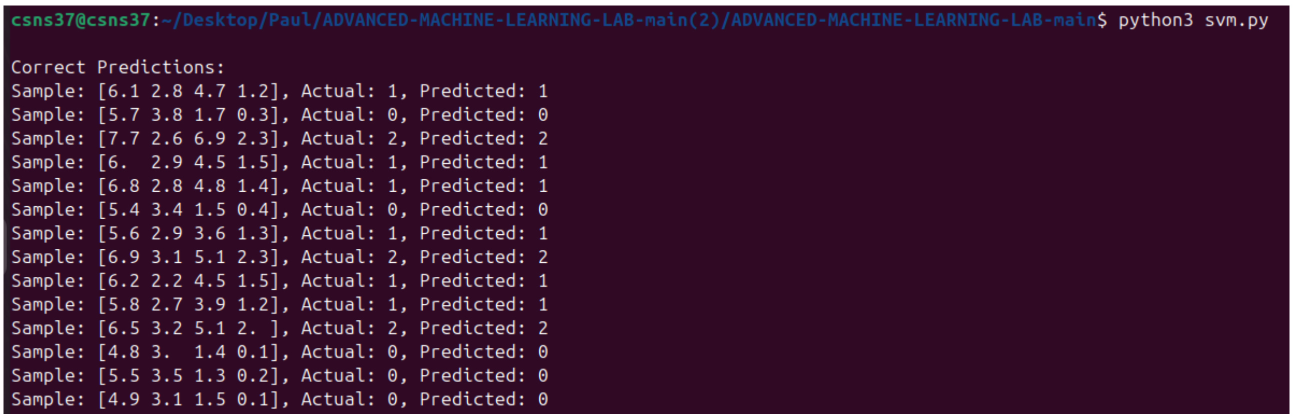
Data Set Description:

The Iris dataset consists of 150 samples of iris flowers, categorized into three species: setosa, versicolor, and virginica. Each sample has four features: sepal length, sepal width, petal length, and petal width. The task is to classify the flowers into one of the three species based on these features.

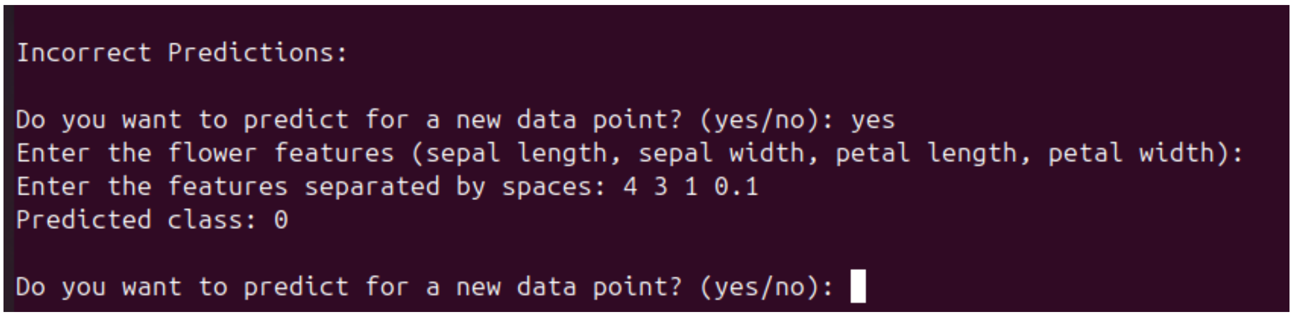
IMPLEMENTATION:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import numpy as np

# Step 1: Load the Iris dataset
def load_iris_data():
    iris = datasets.load_iris()
    X = iris.data  # Features (sepal length, sepal width, petal length, petal width)
    y = iris.target  # Labels (Setosa, Versicolour, Virginica)
    return X, y

# Step 2: Train the SVM classifier
def train_svm(X_train, y_train):
    clf = SVC(kernel='linear')  # Use linear kernel for simplicity
    clf.fit(X_train, y_train)
    return clf

# Step 3: Evaluate the SVM model
def evaluate_svm(clf, X_test, y_test):
    y_pred = clf.predict(X_test)

    correct_predictions = []
    incorrect_predictions = []

    for i in range(len(y_pred)):
        if y_pred[i] == y_test[i]:
            correct_predictions.append((X_test[i], y_test[i], y_pred[i]))
        else:
            incorrect_predictions.append((X_test[i], y_test[i], y_pred[i]))

    return correct_predictions, incorrect_predictions

# Step 4: Function to make predictions for new data points
def predict_svm(clf):
    print("Enter the flower features (sepal length, sepal width, petal length, petal width):")
    try:
        features = list(map(float, input("Enter the features separated by spaces: ").split()))
        prediction = clf.predict([features])
        print(f"Predicted class: {prediction[0]}")
    except ValueError:
        print("Invalid input. Please enter numeric values.")

# Step 5: Main function to execute the program
def main():
    # Load data
    X, y = load_iris_data()

    # Step 1: Split the 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)

    # Step 2: Train the SVM model
    clf = train_svm(X_train, y_train)

    # Step 3: Evaluate the model and print correct and incorrect predictions
    correct_predictions, incorrect_predictions = evaluate_svm(clf, X_test, y_test)

    print("\nCorrect Predictions:")
    for sample, actual, predicted in correct_predictions:
        print(f"Sample: {sample}, Actual: {actual}, Predicted: {predicted}")

    print("\nIncorrect Predictions:")
    for sample, actual, predicted in incorrect_predictions:
        print(f"Sample: {sample}, Actual: {actual}, Predicted: {predicted}")

    # Step 4: Allow the user to input their own data and make a prediction
    while True:
        user_input = input("\nDo you want to predict for a new data point? (yes/no): ").lower()
        if user_input == 'yes':
            predict_svm(clf)
        else:
            print("Exiting the program.")
            break

# Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 svm.py

Correct Predictions:
Sample: [6.1 2.8 4.7 1.2], Actual: 1, Predicted: 1
Sample: [5.7 3.8 1.7 0.3], Actual: 0, Predicted: 0
Sample: [7.7 2.6 6.9 2.3], Actual: 2, Predicted: 2
Sample: [6.  2.9 4.5 1.5], Actual: 1, Predicted: 1
Sample: [6.8 2.8 4.8 1.4], Actual: 1, Predicted: 1
Sample: [5.4 3.4 1.5 0.4], Actual: 0, Predicted: 0
Sample: [5.6 2.9 3.6 1.3], Actual: 1, Predicted: 1
Sample: [6.9 3.1 5.1 2.3], Actual: 2, Predicted: 2
Sample: [6.2 2.2 4.5 1.5], Actual: 1, Predicted: 1
Sample: [5.8 2.7 3.9 1.2], Actual: 1, Predicted: 1
Sample: [6.5 3.2 5.1 2. ], Actual: 2, Predicted: 2
Sample: [4.8 3.  1.4 0.1], Actual: 0, Predicted: 0
Sample: [5.5 3.5 1.3 0.2], Actual: 0, Predicted: 0
Sample: [4.9 3.1 1.5 0.1], Actual: 0, Predicted: 0
```



```
Incorrect Predictions:

Do you want to predict for a new data point? (yes/no): yes
Enter the flower features (sepal length, sepal width, petal length, petal width):
Enter the features separated by spaces: 4 3 1 0.1
Predicted class: 0

Do you want to predict for a new data point? (yes/no):
```

RESULT:

Program for implementing SVM Classifier executed successfully. Accuracy and predictions for test data and user input obtained.

EXPERIMENT NO: 11
DATE:

5-FOLD CROSS VALIDATION

AIM:

Write a program to implement 5-fold cross validation on a learning problem using real time dataset. Compare the accuracy, precision, recall, and F-score for different folds.

ALGORITHM:

```
1. Load the Iris dataset.
2. Split the dataset into features X and target labels y.
3. Initialize the Decision Tree classifier.
4. Set up 5-fold cross-validation using KFold with 5 splits and shuffling.
5. For each fold:
        5.1. Split the data into training and testing sets based on the current fold indices.
        5.2. Train the classifier on the training set.
        5.3. Make predictions on the test set.
        5.4. Calculate accuracy, precision, recall, and F1-score for the current fold.
        5.5. Store the metrics for each fold.
6. Print the metrics for each fold.
7. Calculate and print the average of all metrics across the 5 fold.
```

DESCRIPTION:

Data Set Description:

The Iris dataset consists of 150 samples with 4 features: sepal length, sepal width, petal length, and petal width. It includes 3 species of Iris flowers: Setosa, Versicolor, and Virginica.

Cross-Validation:
In k-fold cross-validation, the dataset is split into k parts. The model is trained on k-1 parts and tested on the remaining part. This process repeats k times, and the average performance metrics (accuracy, precision, recall, F1-score) are calculated.

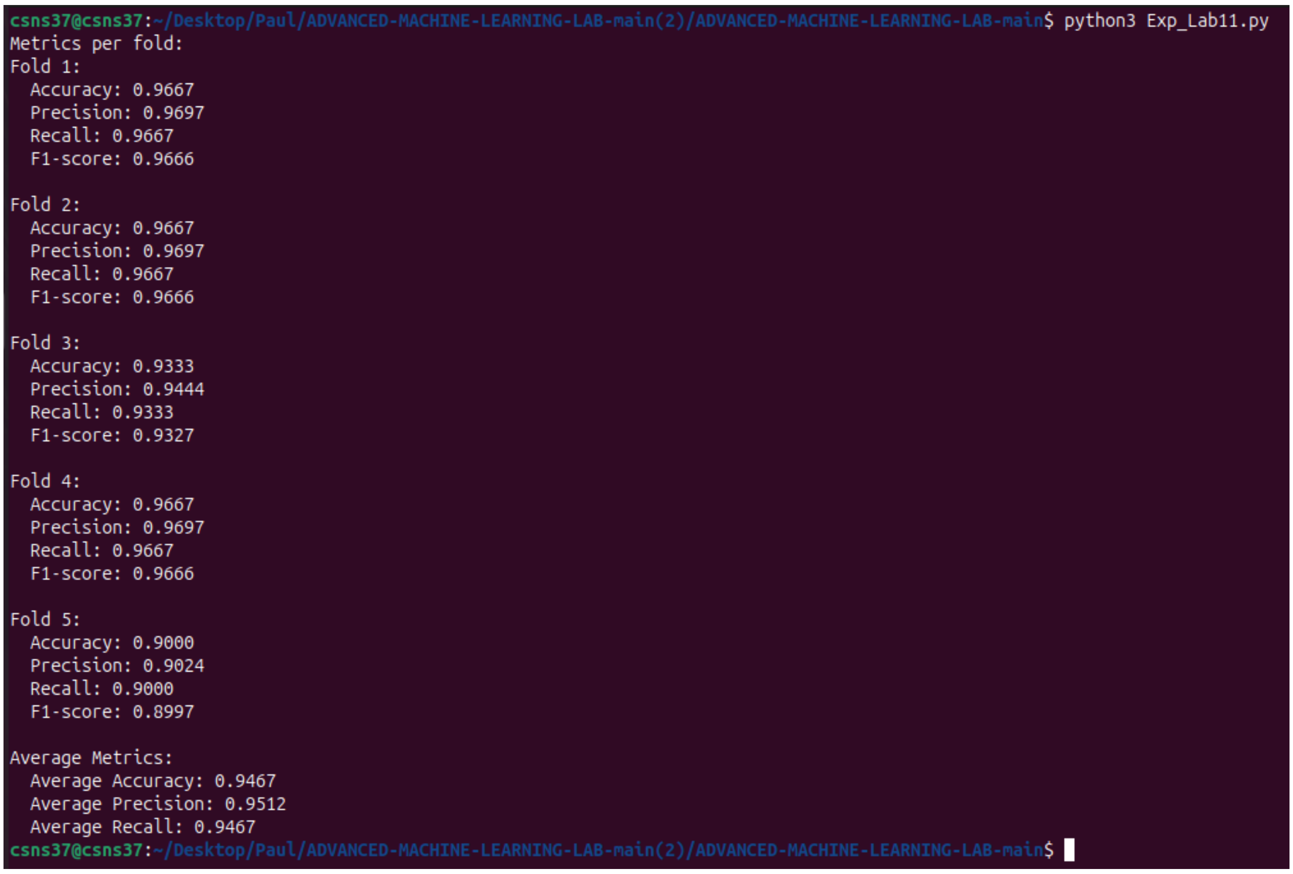
IMPLEMENTATION:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Load the Iris dataset (substitute with your own dataset if needed)
data = load_iris()
```

```
X = data.data
y = data.target

# Initialize the model (Random Forest as an example)
model = RandomForestClassifier(n_estimators=100, random_state=42)

# Initialize StratifiedKFold (ensures class distribution is maintained across folds)
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics for each fold
accuracies = []
precisions = []
recalls = []
f1_scores = []

# Loop over each fold
for train_index, test_index in kf.split(X, y):
    # Split the data into train and test sets for this fold
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the model on the training set
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='macro', zero_division=0)
    recall = recall_score(y_test, y_pred, average='macro', zero_division=0)
    f1 = f1_score(y_test, y_pred, average='macro', zero_division=0)

    # Store the metrics for this fold
    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)

# Print out the metrics for each fold
print("Metrics per fold:")
for i in range(5):
    print(f"Fold {i+1}:")
    print(f"  Accuracy: {accuracies[i]:.4f}")
    print(f"  Precision: {precisions[i]:.4f}")
    print(f"  Recall: {recalls[i]:.4f}")
    print(f"  F1-score: {f1_scores[i]:.4f}")
    print()
```

```
# Compute average of each metric across all folds
avg_accuracy = np.mean(accuracies)
avg_precision = np.mean(precisions)
avg_recall = np.mean(recalls)
avg_f1_score = np.mean(f1_scores)

# Print out the average metrics
print("Average Metrics:")
print(f"  Average Accuracy: {avg_accuracy:.4f}")
print(f"  Average Precision: {avg_precision:.4f}")
print(f"  Average Recall: {avg_recall:.4f}")
print(f"  Average F1-score: {avg_f1_score:.4f}")
```

OUTPUT:



```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 Exp_Lab11.py
Metrics per fold:
Fold 1:
  Accuracy: 0.9667
  Precision: 0.9697
  Recall: 0.9667
  F1-score: 0.9666

Fold 2:
  Accuracy: 0.9667
  Precision: 0.9697
  Recall: 0.9667
  F1-score: 0.9666

Fold 3:
  Accuracy: 0.9333
  Precision: 0.9444
  Recall: 0.9333
  F1-score: 0.9327

Fold 4:
  Accuracy: 0.9667
  Precision: 0.9697
  Recall: 0.9667
  F1-score: 0.9666

Fold 5:
  Accuracy: 0.9000
  Precision: 0.9024
  Recall: 0.9000
  F1-score: 0.8997

Average Metrics:
  Average Accuracy: 0.9467
  Average Precision: 0.9512
  Average Recall: 0.9467
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$
```

RESULT:

Executed the program for 5-fold cross validation on iris dataset and printed the accuracy, precision and recall of each fold.

EXPERIMENT NO: 12
DATE:

BOOSTING ENSEMBLE METHOD

AIM:

Write a program to implement Boosting ensemble method on a given dataset.

ALGORITHM:

Algorithm :

1. Load Dataset
   Load the Iris dataset using load_iris() from sklearn.datasets.
2. Feature and Target Separation
   Extract features X and target labels y from the dataset.
3. Data Splitting
   Split the data into training and testing sets using train_test_split(), with 20% of the data allocated for testing.
4. Initialize Gradient Boosting Classifier
   Set up a GradientBoostingClassifier with default parameters or customized settings as per requirements.
5. Train the Model
   Train the GradientBoostingClassifier on the training data using fit().
6. Make Predictions
   Use the trained model to predict the class labels for the test dataset (Xtest).
7. Model Evaluation
   Evaluate the performance of the model using accuracy_score() and confusion_matrix().
8. Display Results
   Print the confusion matrix and the accuracy score of the model.
9. User Input for Prediction
   Accept user input for feature values and predict the class of the input sample using the trained model.
10. Output the Prediction
    Display the predicted species based on the user-provided feature values.

DESCRIPTION:

Ensemble Methods: Ensemble methods combine multiple models (usually weak learners) to improve prediction accuracy. The key idea is that the collective performance of many models is better than a single model, as they help to reduce errors and improve robustness.

Boosting: Boosting is a type of ensemble method where models are trained sequentially, with each new model attempting to correct the errors made by the previous ones. The final prediction is based on a weighted combination of all the models.

Gradient Boosting: Gradient Boosting is a boosting technique where models are trained to minimize the loss function of the previous model using gradient descent. It builds models sequentially, correcting errors at each step, and combines the predictions of all models for final output. Gradient Boosting is effective for both classification and regression tasks and often performs well on structured datasets.

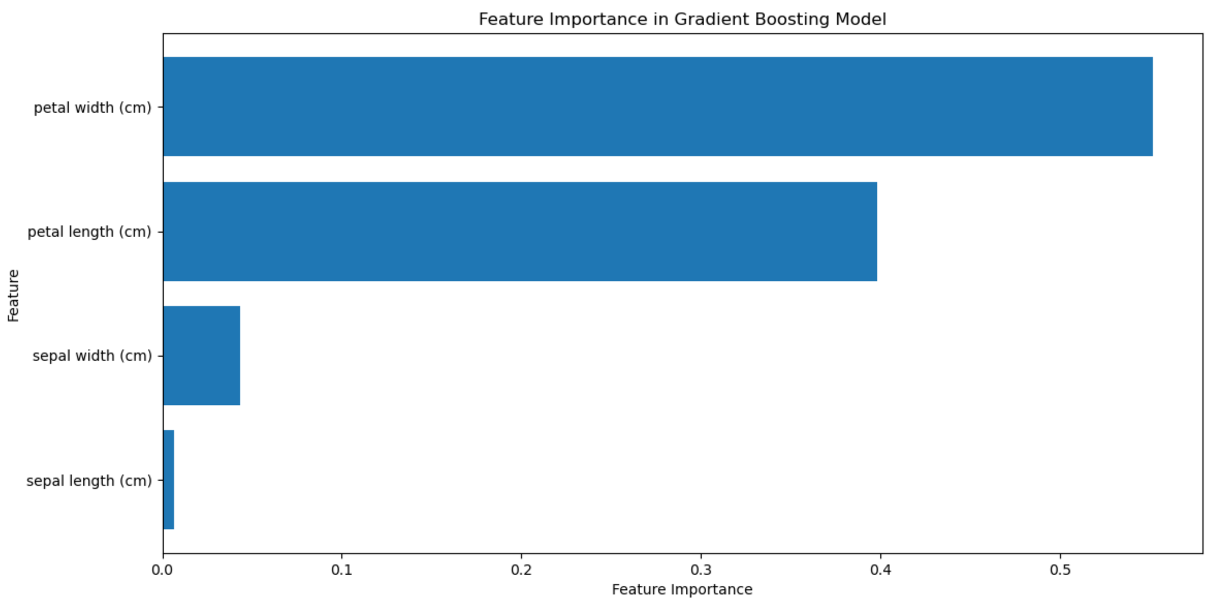
Data Set Description:

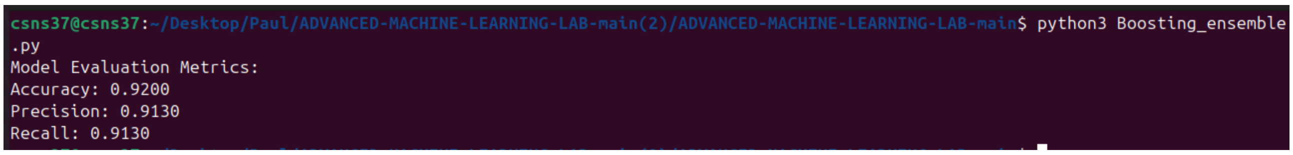
The Iris dataset consists of 150 samples of iris flowers, each having four features: sepal length, sepal width, petal length, and petal width. These samples are classified into three species: Setosa, Versicolor, and Virginica, with 50 samples for each species.

IMPLEMENTATION:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Load the Iris dataset (or any other dataset)
data = load_iris()
X = data.data
y = data.target

# 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)

# Initialize the Gradient Boosting Classifier
model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3,
random_state=42)

# Train the model
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model performance
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro', zero_division=0)
recall = recall_score(y_test, y_pred, average='macro', zero_division=0)
f1 = f1_score(y_test, y_pred, average='macro', zero_division=0)

# Print the evaluation metrics
print("Model Evaluation Metrics:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-score: {f1:.4f}")

# Optional: Plot the feature importance (which features are most important in decision making)
feature_importance = model.feature_importances_
features = data.feature_names

plt.barh(features, feature_importance)
plt.xlabel("Feature Importance")
plt.ylabel("Feature")
plt.title("Feature Importance in Gradient Boosting Model")
plt.show()
```

OUTPUT:





```
csns37@csns37:~/Desktop/Paul/ADVANCED-MACHINE-LEARNING-LAB-main(2)/ADVANCED-MACHINE-LEARNING-LAB-main$ python3 Boosting_ensemble
.py
Model Evaluation Metrics:
Accuracy: 0.9200
Precision: 0.9130
Recall: 0.9130
```

RESULT:

Program implementing Gradient Boosting Classifier executed successfully. The output is obtained.