

**VISVESVARAYA TECHNOLOGICAL
UNIVERSITY**
“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT
on**

Machine Learning (23CS6PCMAL)

Submitted by

Rushi Hundiwala (1BM22CS224)

in partial fulfillment for the award of the degree of

**BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025**

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “Machine Learning (23CS6PCMAL)” carried out by **Rushi Hundiwala(1BM22CS224)**, who is bonafide student of **B.M.S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Machine Learning (23CS6PCMAL) work prescribed for the said degree.

Lab Faculty Incharge	
Name: Ms. Saritha A N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	21-2-2025	Write a python program to import and export data using Pandas library functions	
2	3-3-2025	Demonstrate various data pre-processing techniques for a given dataset	
3	10-3-2025	Implement Linear and Multi-Linear Regression algorithm using appropriate dataset	
4	17-3-2025	Build Logistic Regression Model for a given dataset	
5	24-3-2025	Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample	
6	7-4-2025	Build KNN Classification model for a given dataset	
7	21-4-2025	Build Support vector machine model for a given dataset	
8	5-5-2025	Implement Random forest ensemble method on a given dataset	
9	5-5-2025	Implement Boosting ensemble method on a given dataset	
10	12-5-2025	Build k-Means algorithm to cluster a set of data stored in a .CSV file	
11	12-5-2025	Implement Dimensionality reduction using Principal Component Analysis (PCA) method	

Github Link:<https://github.com/RishiHundiwala344/ML-lab/tree/main>

Program 1

Write a python program to import and export data using Pandas library functions

Code:

```
import pandas as pd
```

```
# ---- IMPORTING DATA ----
```

```
# Read data from a CSV file
input_file = 'input_data.csv' # Make sure this file exists in your working directory
try:
    data = pd.read_csv(input_file)
    print("Imported Data:")
    print(data)
except FileNotFoundError:
    print(f"File '{input_file}' not found.")
```

```
# ---- EXPORTING DATA ----
```

```
# Modify or process the data (optional) - Example: Add a new column
if 'data' in locals():
    data['Processed'] = data.iloc[:, 0].apply(lambda x: x) # Just copying the first column for example

# Export the modified data to a new CSV file
output_file = 'output_data.csv'
data.to_csv(output_file, index=False)
print(f"\nData exported successfully to '{output_file}'")
```

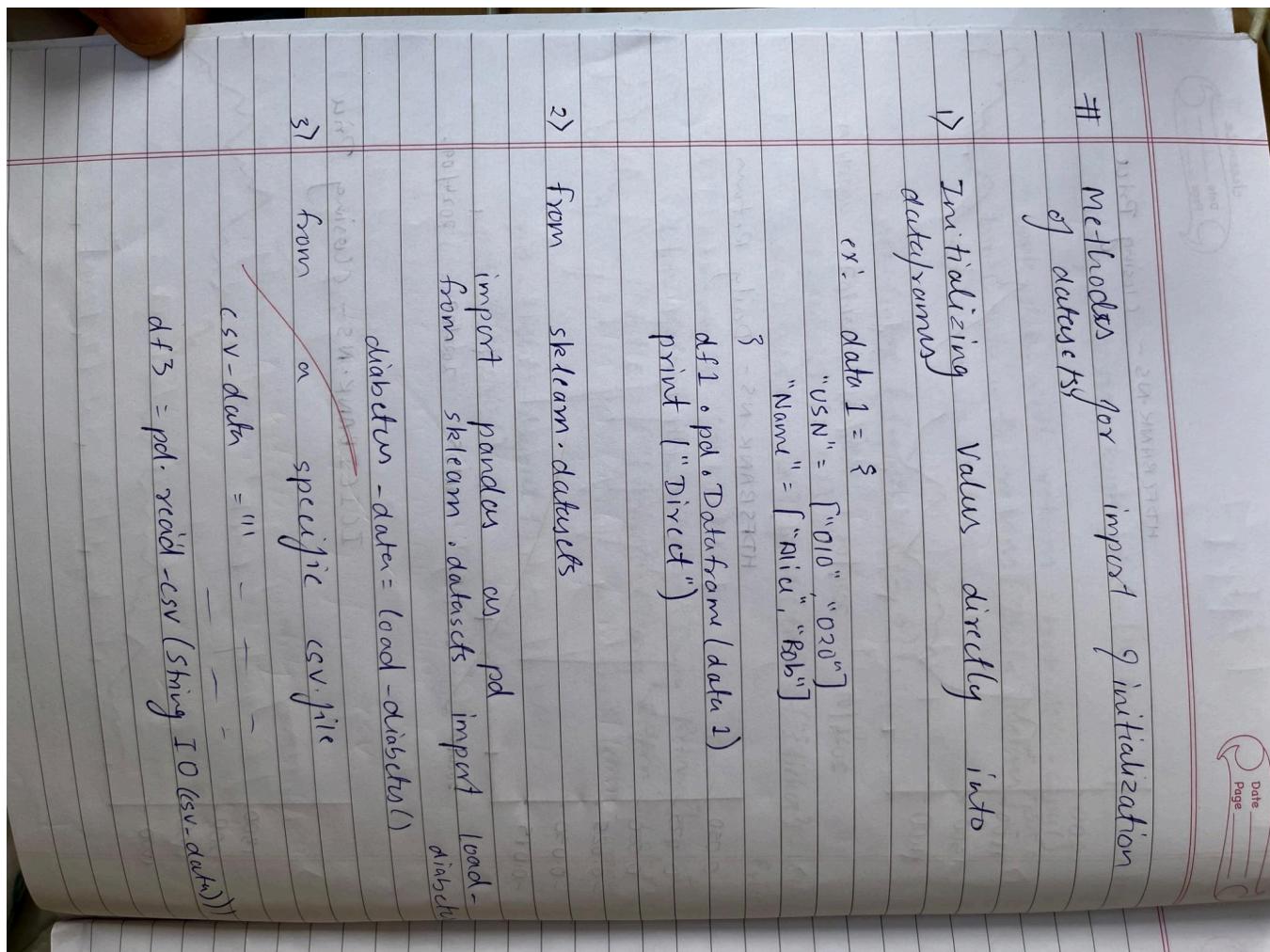
OUTPUT:

```
Name,Age,Country
Alice,30,USA
Bob,25,Canada
Charlie,35,UK
```

Program 2

Demonstrate various data pre-processing techniques for a given dataset

SCREENSHOT:



Mt fluid - 4

Passenger Id	Survived	Pclass
0	6	3
1	1	1
2	1	3
3	1	1
4	0	3

10
3/3/25

10
3/3/25

4) Downloading ? Using online databases.

```
try :  
    url = "https://raw.githubusercontent.com"
```

try	:	from	urllib	urllib
-	-	-	import	urllib
except :	as	data	as	data
1.	-	-	for	for
2.	-	-	in	in
like Kaggle etc.				

Code :

S - button

```
csv_data = """Product ID,Quantity  
1001,10  
1002,5
```

1003,20

1004,15

1005,8

1006,10

1007,12

1008,18

1009,10

1010,15

1011,20

1012,10

1013,15

1014,20

1015,10

1016,12

```
df3 = pd.read_csv(StringIO(csv_data))
```

method 4

```
method 4  
def func(url):  
    url = "https://raw.githubusercontent.com"
```

```
    try :  
        response = requests.get(url)
```

response.raise_for_status()

except:

requests.exceptions.RequestException as e:

print(f"\nError downloading database : {e}")

Output:

[[{"CustomerID": 1, "Gender": "Male", "Age": 25, "Salary": 50000}, {"CustomerID": 2, "Gender": "Female", "Age": 30, "Salary": 60000}, {"CustomerID": 3, "Gender": "Female", "Age": 35, "Salary": 52000}, {"CustomerID": 4, "Gender": "Male", "Age": 40, "Salary": None}, {"CustomerID": 5, "Gender": "Female", "Age": 45, "Salary": 58000}]]

Method - 1: Using `None` = `NaN`

USN	Name	Marks
0	Alice	85
1	Bob	92
2	Cecile	78
3	David	88
4	Eve	95
5	John	70

Method - 2

Importing `Diabetes` dataset from `sklearn`

age	sex	bmi
0	0.38076	0.0501
1	-0.601882	-0.00464
2	0.083299	0.050686

Method - 3: Importing from CSV

product ID	Quantity	Unit Price
0	1001	10
1	1002	5
2	1003	20

CODE:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Sample dataset
data = {
    'CustomerID': [1, 2, 3, 4, 5],
    'Gender': ['Male', 'Female', 'Female', 'Male', 'Female'],
    'Age': [25, 30, None, 35, 40],
    'Salary': [50000, 60000, 52000, None, 58000],
}
```

```
'Purchased': ['No', 'Yes', 'No', 'Yes', 'Yes']  
}  
  
# Create DataFrame  
df = pd.DataFrame(data)  
  
# -----  
# 1. Handling Missing Values  
# -----  
  
# Fill missing 'Age' with mean age  
df['Age'].fillna(df['Age'].mean(), inplace=True)  
  
# Fill missing 'Salary' with mean salary  
df['Salary'].fillna(df['Salary'].mean(), inplace=True)  
  
# -----  
# 2. Encoding Categorical Variables  
# -----  
  
# Convert 'Gender' to numeric (Male: 0, Female: 1)  
df['Gender'] = df['Gender'].map({'Male': 0, 'Female': 1})  
  
# Encode 'Purchased' column (No: 0, Yes: 1)
```

```
le = LabelEncoder()
df['Purchased'] = le.fit_transform(df['Purchased'])

# -----
# 3. Feature Scaling
# -----

# Scale 'Age' and 'Salary' using StandardScaler
scaler = StandardScaler()
df[['Age', 'Salary']] = scaler.fit_transform(df[['Age', 'Salary']])

# -----
# 4. Feature Engineering
# -----

# Create a new feature: Age to Salary ratio
df['Age_Salary_Ratio'] = df['Age'] / (df['Salary'] + 1e-6) # add epsilon to avoid division by zero

# -----
# 5. Drop Irrelevant Columns
# -----

# Drop 'CustomerID' as it's just an identifier
df.drop(columns=['CustomerID'], inplace=True)
```

```
# -----
```

```
# Final Output
```

```
# -----
```

```
print(df)
```

Program 3

Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

Screenshot:

LAB - 4

classmate
Date _____
Page _____

1) Linear Regression:

```
def fit(x,y):
    n = len(x)
    if n != len(y):
        raise ValueError("The length of X and Y must be the same")
    mean_x = sum(x)/n
    mean_y = sum(y)/n
    numerator = sum((x[i]-mean_x)*(y[i]-mean_y))
    denominator = sum((x[i]-mean_x)**2 for i in range(n))
    slope = numerator/denominator
    intercept = mean_y - (slope * mean_x)
    return slope, intercept
if __name__ == "__main__":
    x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    y = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

2) Multiple Linear Regression:

```

import random

class MultipleLinearRegression:
    def __init__(self):
        self.coefficients = [0]
        self.learning_rate = 0.01
        self.n_iterations = 1000
        self.n_samples, self.n_features = len(x), len(x[0])
        self.coefficients = [0.0] * self.n_features
        self.error = 0.0

    def fit(self, x, y):
        for i in range(self.n_iterations):
            predictions = self.predict(x)
            error = np.sum((y - predictions) ** 2) / self.n_samples
            gradients = [0.0] * self.n_features
            for j in range(self.n_features):
                for i in range(self.n_samples):
                    gradients[j] += (-2 / self.n_samples) * (y[i] - predictions[i]) * x[i][j]
            for j in range(self.n_features):
                self.coefficients[j] -= self.learning_rate * gradients[j]
        return self

```

Code:

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# --- Simulated Weather Dataset ---
data = {
    'Temperature': [30, 35, 40, 45, 50, 55, 60, 65, 70, 75],
    'Humidity': [90, 85, 80, 78, 70, 65, 60, 55, 50, 45],
    'WindSpeed': [5, 7, 6, 8, 10, 9, 11, 12, 13, 15],
    'Rainfall': [20, 18, 15, 12, 10, 8, 5, 3, 2, 1]
}
df = pd.DataFrame(data)

```

```

# --- Helper: Normal Equation ---
def normal_equation(X, y):
    XTX = X.T @ X
    XTy = X.T @ y

```

```

beta = np.linalg.inv(XTX) @ XTy
return beta

# === LINEAR REGRESSION: Rainfall ~ Temperature ===
X_lin = df[['Temperature']].values
y = df['Rainfall'].values.reshape(-1, 1)

# Add intercept term (bias) to X
X_lin_b = np.hstack((np.ones((X_lin.shape[0], 1)), X_lin)) # [1, Temperature]

# Compute coefficients
theta_lin = normal_equation(X_lin_b, y)

# Predictions
y_pred_lin = X_lin_b @ theta_lin

# === MULTIPLE LINEAR REGRESSION: Rainfall ~ Temperature + Humidity + WindSpeed ===
X_multi = df[['Temperature', 'Humidity', 'WindSpeed']].values
X_multi_b = np.hstack((np.ones((X_multi.shape[0], 1)), X_multi)) # [1, T, H, W]

# Compute coefficients
theta_multi = normal_equation(X_multi_b, y)

# Predictions
y_pred_multi = X_multi_b @ theta_multi

# --- Print Results ---
print("== Linear Regression ==")
print(f"Intercept: {theta_lin[0][0]:.2f}")
print(f"Coefficient (Temperature): {theta_lin[1][0]:.2f}")
print()

print("== Multiple Linear Regression ==")
print(f"Intercept: {theta_multi[0][0]:.2f}")
print(f"Coefficients:")
print(f" Temperature: {theta_multi[1][0]:.2f}")
print(f" Humidity: {theta_multi[2][0]:.2f}")
print(f" WindSpeed: {theta_multi[3][0]:.2f}")
print()

# --- Visualization: Linear Regression ---
plt.figure(figsize=(10, 5))
plt.scatter(X_lin, y, color='blue', label='Actual')
plt.plot(X_lin, y_pred_lin, color='red', label='Predicted')
plt.xlabel('Temperature (°F)')
plt.ylabel('Rainfall (mm)')
plt.title('Linear Regression: Rainfall vs Temperature')
plt.legend()

```

```
plt.grid(True)  
plt.show()
```

OUTPUT:

== Linear Regression ==

Intercept: 36.00

Coefficient (Temperature): -0.48

== Multiple Linear Regression ==

Intercept: 27.93

Coefficients:

Temperature: -0.32

Humidity: 0.16

WindSpeed: -0.66

Program 4

Build Logistic Regression Model for a given dataset

SCREENSHOT:

The image shows handwritten code on lined paper. The code defines a class named 'Logistic Regression' with various methods and annotations. The code is as follows:

```
class LogisticRegression:
    def __init__(self):
        self.coefficients = [0.0] * n_features
    def sigmoid(self, z):
        return 1 / (1 + math.exp(-z))
    def fit(self, X, y, learning_rate=0.01,
            iteration=1000):
        n_samples, n_features = len(X), len(X[0])
        self.coefficients = [0.0] * n_features
        for i in range(iteration):
            for j in range(n_samples):
                linear_combination = sum([coefficient * X[j][i] for coefficient in self.coefficients])
                prediction = self.sigmoid(linear_combination)
                error = y[i] - prediction
                for coefficient_index in range(len(self.coefficients)):
                    self.coefficients[c] += learning_rate * (y[i] - prediction) * X[j][c]
```

The code includes several annotations in blue ink, such as 'linear-combination = sum(coef * X[i])' and 'for coefficient in self.coefficients'. There are also some crossed-out parts and a circled '1'.

```

def predict(self, x):
    predictions = []

    for n in x:
        linear_combination = sum([w_i * n[i] for i in range(len(n))])
        if linear_combination + self.b >= 0:
            predictions.append(1)
        else:
            predictions.append(0)

    return predictions

def create_dataset(num_samples = 100):
    df = pd.DataFrame({
        'Temperature': [30, 35, 40, 45, 50, 55, 60, 65, 70, 75],
        'Humidity': [90, 85, 80, 78, 70, 65, 60, 55, 50, 45],
        'RainToday': [1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
    })
    df = pd.DataFrame(df)

```

Sample Predictions : [1, 1, 1, 1, 1, 1]

Actual Labels [0, 1, 1, 1, 1] not

CODE:

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# -----
# 1. Simulated Dataset
# -----
# Binary classification: Predict if it will rain (1) or not (0) based on weather conditions
data = {
    'Temperature': [30, 35, 40, 45, 50, 55, 60, 65, 70, 75],
    'Humidity': [90, 85, 80, 78, 70, 65, 60, 55, 50, 45],
    'RainToday': [1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
}
df = pd.DataFrame(data)

# Features and target
X = df[['Temperature', 'Humidity']].values
y = df['RainToday'].values.reshape(-1, 1)

# -----
# 2. Preprocessing: Add Bias Term

```

```

# -----
X = np.hstack((np.ones((X.shape[0], 1)), X)) # Add column of 1s

# -----
# 3. Define Logistic Functions
# -----
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def compute_cost(X, y, theta):
    m = len(y)
    h = sigmoid(X @ theta)
    cost = -(1/m) * (y.T @ np.log(h + 1e-10) + (1 - y).T @ np.log(1 - h + 1e-10))
    return cost.item()

def gradient_descent(X, y, theta, lr, iterations):
    m = len(y)
    cost_history = []

    for i in range(iterations):
        h = sigmoid(X @ theta)
        gradient = (1/m) * (X.T @ (h - y))
        theta -= lr * gradient
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)

    return theta, cost_history

# -----
# 4. Training the Model
# -----
theta_init = np.zeros((X.shape[1], 1)) # [bias, temp, humidity]
learning_rate = 0.01
iterations = 1000

theta_final, cost_history = gradient_descent(X, y, theta_init, learning_rate, iterations)

# -----
# 5. Prediction
# -----
def predict(X, theta):
    return (sigmoid(X @ theta) >= 0.5).astype(int)

y_pred = predict(X, theta_final)

# -----
# 6. Print Results
# -----
print("== Logistic Regression ==")
print("Final Coefficients (theta):")
print(f" Intercept : {theta_final[0][0]:.2f}")
print(f" Temperature: {theta_final[1][0]:.2f}")

```

```
print(f" Humidity : {theta_final[2][0]:.2f}")
print()
print("Predictions:", y_pred.ravel())
print("Actual   :", y.ravel())
print(f"Accuracy : {np.mean(y_pred == y) * 100:.2f}%")

# -----
# 7. Visualization (Cost over Time)
# -----
plt.plot(cost_history)
plt.title('Cost Function over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.grid(True)
plt.show()
```

Program 5

Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample

SCREENSHOT:

1/3/25
Lab - 3
CLASSENAME _____
Date _____
Page _____

```
import math

dataset = [
    ('Sunny', 'Hot', 'High', 'Weak', 'No'),
    ('Sunny', 'Hot', 'High', 'Strong', 'No'),
    ('Overcast', 'Hot', 'High', 'Weak', 'Yes'),
    ('Rain', 'Mild', 'High', 'Weak', 'Yes'),
    ('Rain', 'Cool', 'Normal', 'Weak', 'Yes'),
    ('Overcast', 'Cool', 'Normal', 'Strong', 'Yes'),
    ('Sunny', 'Mild', 'High', 'Weak', 'No'),
    ('Sunny', 'Hot', 'Normal', 'Weak', 'Yes'),
    ('Rain', 'Mild', 'Normal', 'Weak', 'Yes'),
    ('Overcast', 'Normal', 'Weak', 'Yes'),
    ('Sunny', 'Hot', 'High', 'Strong', 'Yes')
]

def entropy(data):
    total = len(data)
    no_values = set([entry[-1] for entry in data])
    if len(no_values) == 1:
        entropy_value = 0.0
    else:
        entropy_value = 0.0
        for value in values:
            prob = sum(1 for entry in data
                       if entry[-1] == value) / total
            entropy_value -= prob * math.log2(prob)

    return entropy_value
```

Date _____
Page _____

```

def info_gain(data, attribute_index):
    total_entropy = entropy(data)
    attribute_values = set([entry[attribute_index] for entry in data])
    weighted_entropy = 0.0
    for value in attribute_values:
        subset = [entry for entry in data if entry[attribute_index] == value]
        weighted_entropy += len(subset)/len(data) * entropy(subset)
    return total_entropy - weighted_entropy

```

~~def id3(data, attributes):
 if len(attributes) == 1:
 return {'label': data[0][0]}
 else:
 tree = {}
 attribute_index = attributes[-1]
 for entry in data:
 if entry[attribute_index] == 1:
 tree['attribute'] = 'but-attr', 'branch1'
 else:
 tree['attribute'] = 'but-attr', 'branch2'
 attribute_values = set([entry[attribute_index] for entry in data])
 for value in attribute_values:
 tree[value] = id3([entry for entry in data if entry[attribute_index] == value], attributes[:-1])
 return tree~~

CODE:

```

import pandas as pd
import math
from collections import Counter

# -----
# 1. Sample Weather Dataset
# -----
data = [
    ['Sunny', 'Hot', 'High', False, 'No'],
    ['Sunny', 'Hot', 'High', True, 'No'],
    ['Overcast', 'Hot', 'High', False, 'Yes'],
    ['Rain', 'Mild', 'High', False, 'Yes'],
    ['Rain', 'Cool', 'Normal', False, 'Yes'],
    ['Rain', 'Cool', 'Normal', True, 'No'],
    ['Overcast', 'Cool', 'Normal', True, 'Yes'],
    ['Sunny', 'Mild', 'High', False, 'No'],
    ['Sunny', 'Cool', 'Normal', False, 'Yes'],
    ['Rain', 'Mild', 'Normal', False, 'Yes'],
    ['Sunny', 'Mild', 'Normal', True, 'Yes'],
]

```

```

['Overcast', 'Mild', 'High', True, 'Yes'],
['Overcast', 'Hot', 'Normal', False, 'Yes'],
['Rain', 'Mild', 'High', True, 'No']
]

columns = ['Outlook', 'Temperature', 'Humidity', 'Windy', 'PlayTennis']
df = pd.DataFrame(data, columns=columns)

# -----
# 2. Entropy and Information Gain
# -----
def entropy(target_col):
    counts = Counter(target_col)
    total = len(target_col)
    return -sum((count/total) * math.log2(count/total) for count in counts.values())

def info_gain(df, split_attr, target_attr):
    total_entropy = entropy(df[target_attr])
    vals = df[split_attr].unique()
    weighted_entropy = 0

    for val in vals:
        subset = df[df[split_attr] == val]
        weighted_entropy += (len(subset)/len(df)) * entropy(subset[target_attr])

    return total_entropy - weighted_entropy

# -----
# 3. ID3 Algorithm (Recursive Tree Builder)
# -----
def id3(df, target_attr, attributes):
    labels = df[target_attr]
    if len(labels.unique()) == 1:
        return labels.iloc[0]
    if len(attributes) == 0:
        return labels.mode()[0]

    gains = {attr: info_gain(df, attr, target_attr) for attr in attributes}
    best_attr = max(gains, key=gains.get)

    tree = {best_attr: {}}
    for val in df[best_attr].unique():
        subset = df[df[best_attr] == val]
        subtree = id3(subset, target_attr, [a for a in attributes if a != best_attr])
        tree[best_attr][val] = subtree

    return tree

# -----
# 4. Train the Tree
# -----
attributes = ['Outlook', 'Temperature', 'Humidity', 'Windy']
tree = id3(df, 'PlayTennis', attributes)

```

```

# -----
# 5. Classification Function
# -----
def classify(tree, sample):
    if not isinstance(tree, dict):
        return tree
    attr = next(iter(tree))
    value = sample[attr]
    if value in tree[attr]:
        return classify(tree[attr][value], sample)
    else:
        return 'Unknown' # For unseen feature values

# -----
# 6. Predict on New Sample
# -----
new_sample = {
    'Outlook': 'Sunny',
    'Temperature': 'Cool',
    'Humidity': 'High',
    'Windy': True
}
prediction = classify(tree, new_sample)

# -----
# 7. Output
# -----
import pprint
print("== ID3 Decision Tree ==")
pprint.pprint(tree)
print("\n== New Sample Classification ==")
print("Input Sample:", new_sample)
print("Predicted Class:", prediction)

```

OUTPUT:

```

== ID3 Decision Tree ==
{'Outlook': {
    'Overcast': 'Yes',
    'Rain': {'Windy': {'False': 'Yes', 'True': 'No'}},
    'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}
}}
== New Sample Classification ==
Input Sample: {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Windy': True}
Predicted Class: No

```

Program 6

Build KNN Classification model for a given dataset

SCREENSHOT:

ML - Lab - 5

classmate
Date _____
Page _____

```
KNN Algorithm:-
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
X, y = make_classification(n_samples=200, n_features=2, n_informative=2, n_redundant=0, n_clusters=2, n_classes=2)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
h = 0.02
```

```
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
arrange(y_min, y_max, n))
```

```
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

```
plt.figure(figsize=(10,6))
```

```
plt.contour(x,y,z, alpha = 0.8)
```

```
plt.scatter(X_train[:10], X_train[1:10], c=y_train[:10], alpha = 0.8)
```

```
plt.scatter(X_test[:10], X_test[1:10], c=y_test[:10], alpha = 0.8)
```

```
plt.scatter(X_test[:10], X_test[1:10], c=y_test[:10], color='k', marker='s', s=100, label='Test Data', cmap=plt.cm.coolwarm)
```

```
plt.show()
```

K Nearest Neighbor Decision Bound

```
with plt.style.context('fivethirtyone')
```

```
plt.xlabel('feature 1')
```

```
plt.ylabel('feature 2')
```

```
plt.legend()
```

```
plt.show()
```

CODE:

```
import numpy as np
import pandas as pd
from collections import Counter

# -----
# 1. Sample Dataset
# -----
data = {
    'Temperature': [30, 35, 40, 45, 50, 55, 60, 65, 70, 75],
    'Humidity': [90, 85, 80, 78, 70, 65, 60, 55, 50, 45],
    'RainToday': ['Yes', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No']
}

df = pd.DataFrame(data)

# Encode target labels
df['RainToday'] = df['RainToday'].map({'Yes': 1, 'No': 0})
```

```

# 2. Euclidean Distance Function
# -----
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))

# -----
# 3. KNN Classifier Function
# -----
def knn_predict(X_train, y_train, x_test, k=3):
    distances = []
    for i in range(len(X_train)):
        dist = euclidean_distance(X_train[i], x_test)
        distances.append((dist, y_train[i]))

    # Sort by distance and get k nearest labels
    distances.sort(key=lambda x: x[0])
    k_labels = [label for _, label in distances[:k]]

    # Majority vote
    vote = Counter(k_labels).most_common(1)[0][0]
    return vote

# -----
# 4. Train and Predict
# -----
X = df[['Temperature', 'Humidity']].values
y = df['RainToday'].values

# New sample to classify
new_sample = np.array([48, 66]) # 48°F and 66% humidity

# Predict with k=3
prediction = knn_predict(X, y, new_sample, k=3)

# -----
# 5. Output
# -----
print("== KNN Classification ==")
print("New Sample:", new_sample)
print(f"Predicted Class (RainToday): {'Yes' if prediction == 1 else 'No'}")

```

OUTPUT:

```

== KNN Classification ==

New Sample: [48 66]
Predicted Class (RainToday): No

```

Program 7

Build Support vector machine model for a given dataset

SCREENSHOT:

27 SVM. (Support Vector Machine).

```
from sklearn import datasets
import train
from sklearn.model_selection import train
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

iris = datasets.load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model = SVC('kernel': 'linear')
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy", accuracy)
```

CODE:

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

# -----
# 1. Sample Dataset
# -----
data = {
    'Temperature': [30, 35, 40, 45, 50, 55, 60, 65, 70, 75],
    'Humidity': [90, 85, 80, 78, 70, 65, 60, 55, 50, 45],
    'RainToday': [1, 1, 1, 1, 0, 0, 0, 0, 0, 0] # 1 = Yes, 0 = No
}
```

```

df = pd.DataFrame(data)

# Features and labels
X = df[['Temperature', 'Humidity']].values
y = df['RainToday'].values

# Convert labels to -1 and 1 for SVM
y = np.where(y == 1, 1, -1)

# -----
# 2. SVM Training Function
# -----
class SVM:
    def __init__(self, lr=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = lr
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y[idx] * (np.dot(x_i, self.w) + self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y[idx]))
                    self.b -= self.lr * y[idx]

    def predict(self, X):
        linear_output = np.dot(X, self.w) + self.b
        return np.where(linear_output >= 0, 1, 0)

# -----
# 3. Train the SVM Model
# -----
svm = SVM()
svm.fit(X, y)

# -----
# 4. Predict New Sample
# -----
new_sample = np.array([[48, 66]]) # Example: Temp=48°F, Humidity=66%
prediction = svm.predict(new_sample)[0]

# -----
# 5. Output

```

```

# -----
print("== SVM Classification ==")
print("New Sample:", new_sample[0])
print("Predicted Class (RainToday):", 'Yes' if prediction == 1 else 'No')

# -----
# 6. Visualization
# -----
def plot_svm(X, y, model):
    def get_line(w, b, x):
        return (-w[0]*x - b) / w[1]

    plt.figure(figsize=(8,6))
    for idx, label in enumerate(y):
        plt.scatter(X[idx][0], X[idx][1], color='blue' if label == 1 else 'red')

    ax = plt.gca()
    x_vals = np.linspace(np.min(X[:,0]), np.max(X[:,0]), 100)
    y_vals = get_line(model.w, model.b, x_vals)

    plt.plot(x_vals, y_vals, 'k--', label='Decision boundary')
    plt.xlabel('Temperature')
    plt.ylabel('Humidity')
    plt.title('SVM Decision Boundary')
    plt.grid(True)
    plt.legend()
    plt.show()

plot_svm(X, y, svm)

```

OUTPUT:

== SVM Classification ==
 New Sample: [48 66]
 Predicted Class (RainToday): No

Program 8

Implement Random forest ensemble method on a given dataset

SCREENSHOT:

```

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Random forest accuracy : ", accuracy)

```

CODE:

```

import numpy as np
import pandas as pd
import math
from collections import Counter
import random

# -----
# 1. Dataset (Play Tennis)
# -----
data = [
    ['Sunny', 'Hot', 'High', False, 'No'],
    ['Sunny', 'Hot', 'High', True, 'No'],
    ['Overcast', 'Hot', 'High', False, 'Yes'],
    ['Rain', 'Mild', 'High', False, 'Yes'],
    ['Rain', 'Cool', 'Normal', True, 'Yes'],
    ['Overcast', 'Cool', 'Normal', True, 'Yes'],
    ['Sunny', 'Hot', 'Normal', False, 'No'],
    ['Overcast', 'Hot', 'Normal', False, 'Yes'],
    ['Overcast', 'Normal', 'Normal', True, 'Yes'],
    ['Rain', 'Mild', 'Normal', True, 'Yes']
]

```

```

['Rain', 'Cool', 'Normal', False, 'Yes'],
['Rain', 'Cool', 'Normal', True, 'No'],
['Overcast', 'Cool', 'Normal', True, 'Yes'],
['Sunny', 'Mild', 'High', False, 'No'],
['Sunny', 'Cool', 'Normal', False, 'Yes'],
['Rain', 'Mild', 'Normal', False, 'Yes'],
['Sunny', 'Mild', 'Normal', True, 'Yes'],
['Overcast', 'Mild', 'High', True, 'Yes'],
['Overcast', 'Hot', 'Normal', False, 'Yes'],
['Rain', 'Mild', 'High', True, 'No']
]
columns = ['Outlook', 'Temperature', 'Humidity', 'Windy', 'PlayTennis']
df = pd.DataFrame(data, columns=columns)

# -----
# 2. Entropy & Info Gain Functions
# -----
def entropy(target_col):
    counts = Counter(target_col)
    total = len(target_col)
    return -sum((count/total) * math.log2(count/total) for count in counts.values())

def info_gain(df, split_attr, target_attr):
    total_entropy = entropy(df[target_attr])
    vals = df[split_attr].unique()
    weighted_entropy = 0
    for val in vals:
        subset = df[df[split_attr] == val]
        weighted_entropy += (len(subset)/len(df)) * entropy(subset[target_attr])
    return total_entropy - weighted_entropy

# -----
# 3. ID3 Tree Builder with Feature Bagging
# -----
def id3(df, target_attr, attributes):
    labels = df[target_attr]
    if len(labels.unique()) == 1:
        return labels.iloc[0]
    if len(attributes) == 0:
        return labels.mode()[0]

    # Select subset of attributes randomly (sqrt of total)
    subset_size = max(1, int(math.sqrt(len(attributes))))
    selectedAttrs = random.sample(attributes, subset_size)

    gains = {attr: info_gain(df, attr, target_attr) for attr in selectedAttrs}
    best_attr = max(gains, key=gains.get)

    tree = {best_attr: {}}
    for val in df[best_attr].unique():
        subset = df[df[best_attr] == val]

```

```

subtree = id3(subset, target_attr, [a for a in attributes if a != best_attr])
tree[best_attr][val] = subtree
return tree

# -----
# 4. Bootstrap Sampling
# -----
def bootstrap_sample(df):
    n_samples = len(df)
    indices = np.random.choice(n_samples, size=n_samples, replace=True)
    return df.iloc[indices]

# -----
# 5. Random Forest Class
# -----
class RandomForest:
    def __init__(self, n_trees=5):
        self.n_trees = n_trees
        self.trees = []
        self.attributes = None
        self.target = None

    def fit(self, df, target_attr, attributes):
        self.target = target_attr
        self.attributes = attributes
        for _ in range(self.n_trees):
            sample = bootstrap_sample(df)
            tree = id3(sample, target_attr, attributes)
            self.trees.append(tree)

    def classify_tree(self, tree, sample):
        if not isinstance(tree, dict):
            return tree
        attr = next(iter(tree))
        value = sample.get(attr)
        if value not in tree[attr]:
            # If unseen feature value, return mode of subtree (or 'Unknown')
            return 'Unknown'
        return self.classify_tree(tree[attr][value], sample)

    def predict(self, samples):
        results = []
        for sample in samples:
            votes = []
            for tree in self.trees:
                pred = self.classify_tree(tree, sample)
                votes.append(pred)
            # Majority vote ignoring 'Unknown'
            votes = [v for v in votes if v != 'Unknown']
            if votes:
                final_pred = Counter(votes).most_common(1)[0][0]
            else:

```

```

        final_pred = 'Unknown'
        results.append(final_pred)
    return results

# -----
# 6. Train Random Forest
# -----
attributes = ['Outlook', 'Temperature', 'Humidity', 'Windy']
target = 'PlayTennis'

rf = RandomForest(n_trees=10)
rf.fit(df, target, attributes)

# -----
# 7. Predict New Samples
# -----
new_samples = [
    {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Windy': True},
    {'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'Normal', 'Windy': False},
]
predictions = rf.predict(new_samples)

# -----
# 8. Output
# -----
print("== Random Forest Predictions ==")
for sample, pred in zip(new_samples, predictions):
    print(f'Input: {sample} => Predicted: {pred}')

```

OUTPUT:

```

== Random Forest Predictions ==
Input: {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Windy': True} => Predicted: No
Input: {'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'Normal', 'Windy': False} => Predicted: Yes

```

Program 9

Implement Boosting ensemble (Adaboost) method on a given dataset

SCREENSHOT:

```
for polarity in [1, -1] :
    for predictions = np.array ((decision
        -stamp (n, features, threshold, polarity))
```

```
for n in)
```

```
error = np.sum (w * (predictions != y))
```

```
if error < min_error :
```

```
min_error = error
bst_stump = (features, threshold
, polarity, predictions)
```

```
epsilon = 1e-10
alpha = 0.5 * np.log ((1 - min_error +
epsilon) / (min_error + epsilon))
```

```
w = w + np.exp (-alpha * y * bst_stump (3))
w = w / np.sum (w)
```

~~classifications.append (bst_stump)~~

~~alpha.append (alpha)~~

~~for i in range (n)~~

CODE:

```
import numpy as np
import pandas as pd

# -----
# 1. Sample Dataset (Binary Classification)
# -----
data = {
    'Temperature': [30, 35, 40, 45, 50, 55, 60, 65],
    'Humidity': [90, 85, 80, 78, 70, 65, 60, 55],
    'RainToday': [1, 1, 1, 1, 0, 0, 0, 0] # 1 = Yes, 0 = No
}

df = pd.DataFrame(data)
X = df[['Temperature', 'Humidity']].values
y = df['RainToday'].values
y = np.where(y == 1, 1, -1) # Convert labels to -1 and 1

# -----
# 2. Decision Stump Weak Learner
# -----
```

```

class DecisionStump:
    def __init__(self):
        self.feature_index = None
        self.threshold = None
        self.polarity = 1

    def predict(self, X):
        n_samples = X.shape[0]
        X_column = X[:, self.feature_index]
        preds = np.ones(n_samples)
        if self.polarity == 1:
            preds[X_column < self.threshold] = -1
        else:
            preds[X_column > self.threshold] = -1
        return preds

# -----
# 3. AdaBoost Implementation
# -----
class AdaBoost:
    def __init__(self, n_clf=5):
        self.n_clf = n_clf

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Initialize weights to 1/N
        w = np.full(n_samples, (1 / n_samples))

        self.clfs = []
        self.alphas = []

        for _ in range(self.n_clf):
            clf = DecisionStump()
            min_error = float('inf')

            # Find best decision stump
            for feature_i in range(n_features):
                X_column = X[:, feature_i]
                thresholds = np.unique(X_column)
                for threshold in thresholds:
                    for polarity in [1, -1]:
                        preds = np.ones(n_samples)
                        if polarity == 1:
                            preds[X_column < threshold] = -1
                        else:
                            preds[X_column > threshold] = -1

                        # Calculate weighted error
                        misclassified = w[y != preds]
                        error = sum(misclassified)

                        if error < min_error:
                            min_error = error
                            clf.feature_index = feature_i
                            clf.threshold = threshold
                            clf.polarity = polarity
                            self.alphas.append(w.sum())
                            self.clfs.append(clf)

        return self

```

```

        if error < min_error:
            min_error = error
            clf.polarity = polarity
            clf.threshold = threshold
            clf.feature_index = feature_i

    # Compute alpha (model weight)
    EPS = 1e-10
    alpha = 0.5 * np.log((1.0 - min_error) / (min_error + EPS))

    # Update weights
    preds = clf.predict(X)
    w *= np.exp(-alpha * y * preds)
    w /= np.sum(w) # Normalize

    # Save classifier and alpha
    self.clfs.append(clf)
    self.alphas.append(alpha)

def predict(self, X):
    clf_preds = [alpha * clf.predict(X) for clf, alpha in zip(self.clfs, self.alphas)]
    y_pred = np.sum(clf_preds, axis=0)
    return np.where(y_pred >= 0, 1, 0)

# -----
# 4. Train and Predict
# -----
ada = AdaBoost(n_clf=5)
ada.fit(X, y)

new_samples = np.array([
    [48, 66], # New sample to predict
    [60, 55],
])
predictions = ada.predict(new_samples)

# -----
# 5. Output
# -----
print("== AdaBoost Predictions ==")
for sample, pred in zip(new_samples, predictions):
    print(f'Input: {sample} => Predicted Class (RainToday): {'Yes' if pred == 1 else 'No'}')

```

OUTPUT:

```

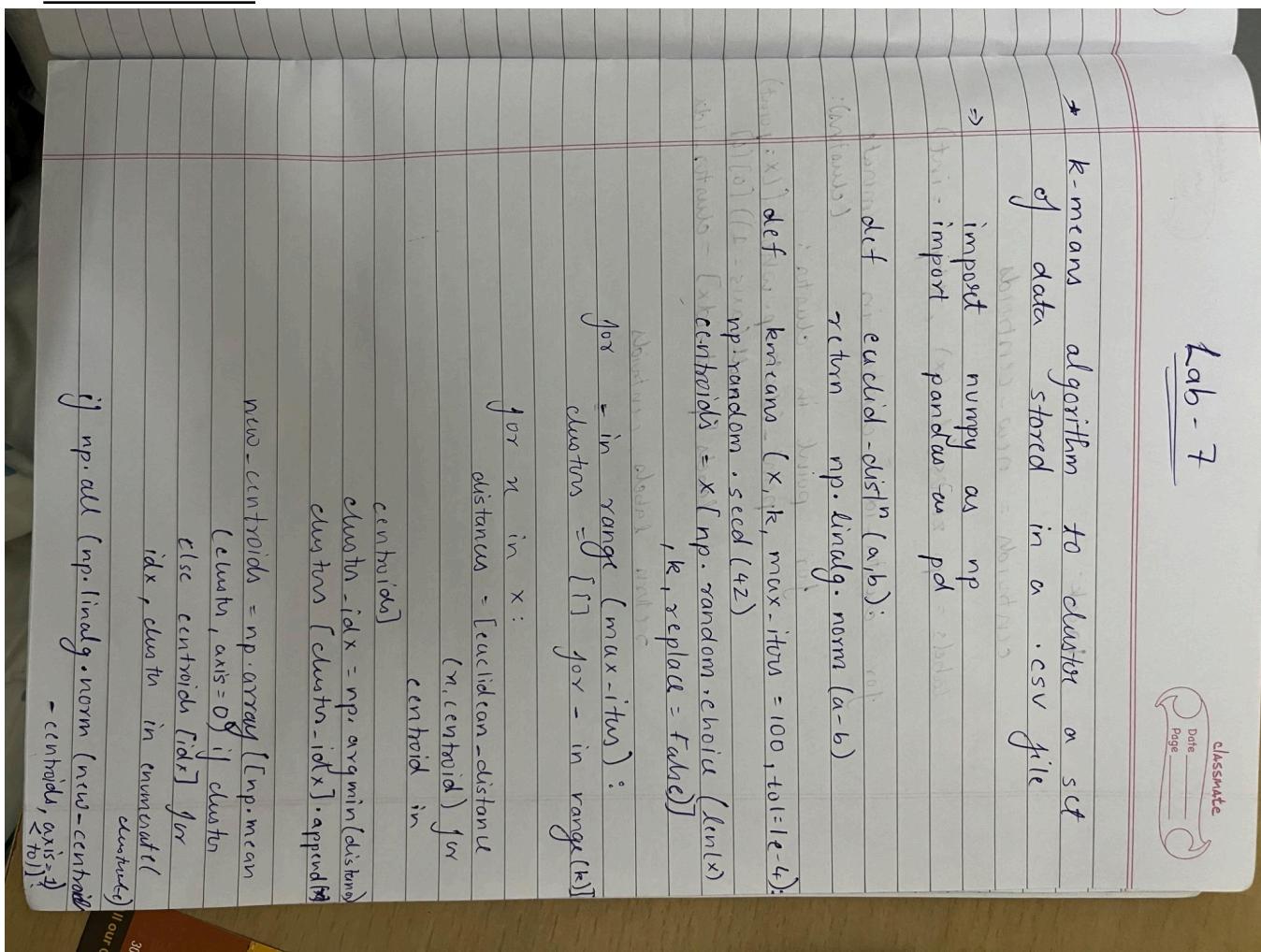
== AdaBoost Predictions ==
Input: [48 66] => Predicted Class (RainToday): No
Input: [60 55] => Predicted Class (RainToday): No

```

Program 10

Build k-Means algorithm to cluster a set of data stored in a .CSV file

SCREENSHOT:



CODE:

```
import numpy as np
import pandas as pd

def euclidean_distance(a, b):
    return np.linalg.norm(a - b)

def kmeans(X, k, max_iters=100, tol=1e-4):
    # Randomly initialize centroids by selecting k random points from data
    np.random.seed(42)
    centroids = X[np.random.choice(len(X), k, replace=False)]
    for _ in range(max_iters):
        distances = [euclidean_distance(centroids[i], x) for i in range(k) for x in X]
        cluster_idx = np.argmin(distances, axis=0)
        for i in range(k):
            centroids[i] = np.mean(X[cluster_idx == i], axis=0)
    return centroids
```

```

clusters = [[] for _ in range(k)]

# Assign points to nearest centroid
for x in X:
    distances = [euclidean_distance(x, centroid) for centroid in centroids]
    cluster_idx = np.argmin(distances)
    clusters[cluster_idx].append(x)

new_centroids = np.array([np.mean(cluster, axis=0) if cluster else centroids[idx]
                         for idx, cluster in enumerate(clusters)])

# Check convergence (if centroids do not change more than tol)
if np.all(np.linalg.norm(new_centroids - centroids, axis=1) < tol):
    break

centroids = new_centroids

# Prepare cluster labels
labels = np.zeros(len(X), dtype=int)
for cluster_idx, cluster in enumerate(clusters):
    for point in cluster:
        point_idx = np.where((X == point).all(axis=1))[0][0]
        labels[point_idx] = cluster_idx

return labels, centroids

# -----
# Load dataset from CSV
# Replace 'data.csv' with your CSV filename
# -----
df = pd.read_csv('data.csv') # Make sure CSV has only numerical columns or select subset
X = df.values

# -----
# Run k-Means
# -----
k = 3 # number of clusters
labels, centroids = kmeans(X, k)

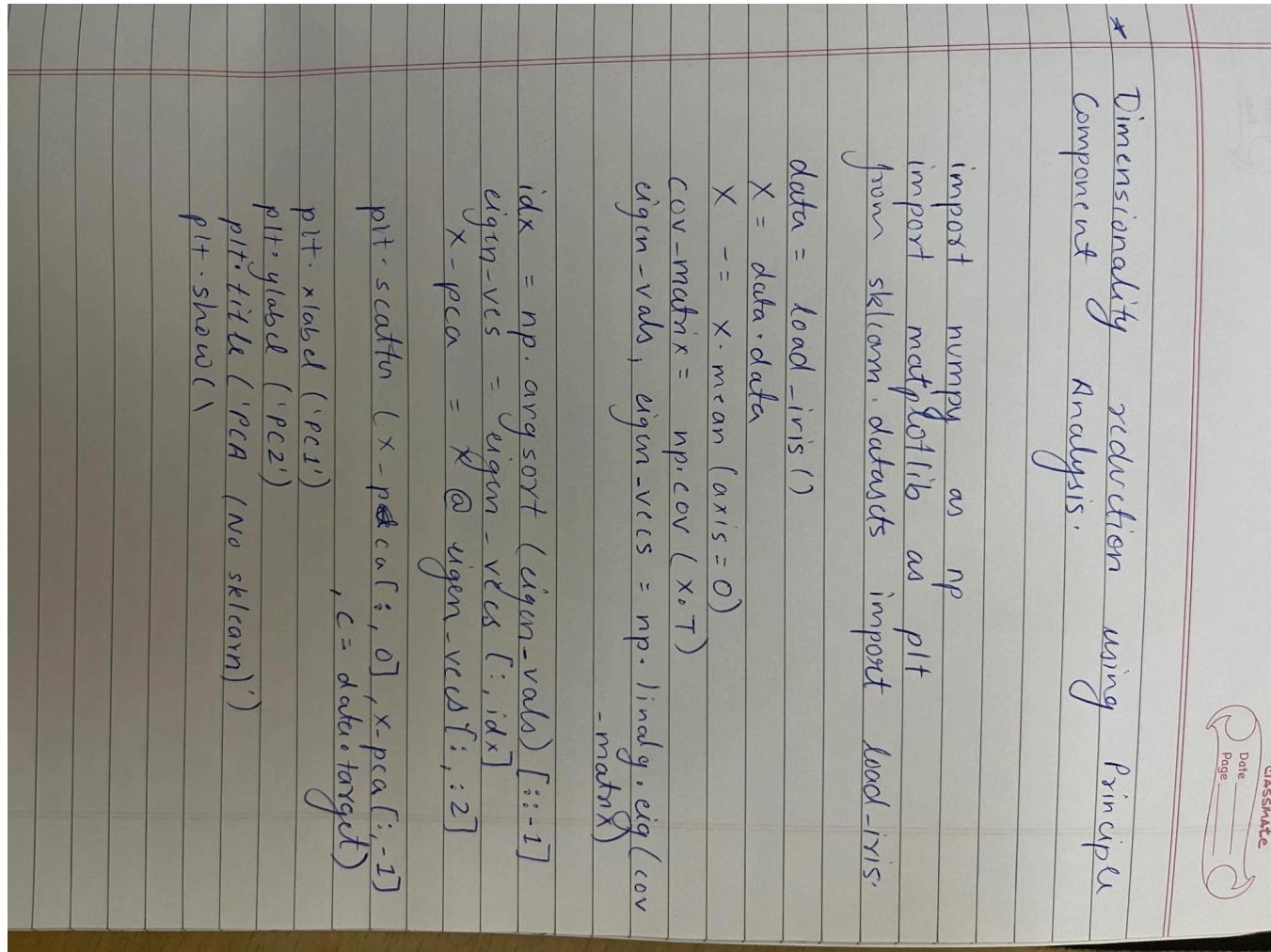
# -----
# Output results
# -----
print("Cluster assignments for each data point:")
print(labels)
print("\nCentroids:")
print(centroids)

```

Program 11

Implement Dimensionality reduction using Principal Component Analysis (PCA) method

SCREENSHOT:



CODE:

```
import numpy as np  
import pandas as pd  
  
def standardize(X):  
    """Standardize features to zero mean and unit variance""""  
    return (X - np.mean(X, axis=0)) / np.std(X, axis=0)  
  
def pca(X, n_components):  
    # Step 1: Standardize data  
    X_std = standardize(X)  
  
    # Step 2: Compute covariance matrix  
    cov_matrix = np.cov(X_std, rowvar=False)
```

```

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

# Step 4: Sort eigenvectors by descending eigenvalues
sorted_idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sorted_idx]
eigenvectors = eigenvectors[:, sorted_idx]

# Step 5: Select top n_components eigenvectors (principal components)
components = eigenvectors[:, :n_components]

# Step 6: Project data
X_pca = np.dot(X_std, components)

return X_pca, eigenvalues, components

# -----
# Example Usage with Sample Data
# -----
# Create example dataset or load your own CSV file
data = {
    'Feature1': [2.5, 0.5, 2.2, 1.9, 3.1, 2.3, 2, 1, 1.5, 1.1],
    'Feature2': [2.4, 0.7, 2.9, 2.2, 3, 2.7, 1.6, 1.1, 1.6, 0.9],
    'Feature3': [1, 3, 2, 4, 5, 1, 2, 1, 3, 2]
}
df = pd.DataFrame(data)

X = df.values
n_components = 2

X_reduced, eigenvalues, components = pca(X, n_components)

print("Reduced data (first 2 principal components):")
print(X_reduced)

print("\nEigenvalues:")
print(eigenvalues)

print("\nPrincipal Components (eigenvectors):")
print(components)

```

OUTPUT:

```

Reduced data (first 2 principal components):
[[-1.33464313  0.18236219]
 [-2.74540499 -0.03668303]
 [-0.80719059 -0.27982202]
 [-0.60117914 -0.24580249]
 [-1.51377843  0.45088336]]

```

```
[-1.08458099 -0.01252943]
[-1.5563464 -0.09316944]
[-2.28299081 -0.37639029]
[-1.71419074 -0.19499621]
[-2.3510216 -0.38885353]]
```

Eigenvalues:
[2.22135779 1.45080258 0.32619848]

Principal Components (eigenvectors):
[[0.6764725 0.65370205 0.33974892]
 [0.73517866 -0.67392404 -0.07898289]
 [0.02637013 0.34399291 -0.93884852]]