VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Machine Learning (23CS6PCMAL)

Submitted by

Rani Aishwarya H S (1BM22CS217)

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 **Rani Aishwarya H S (1BM22CS217),** 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

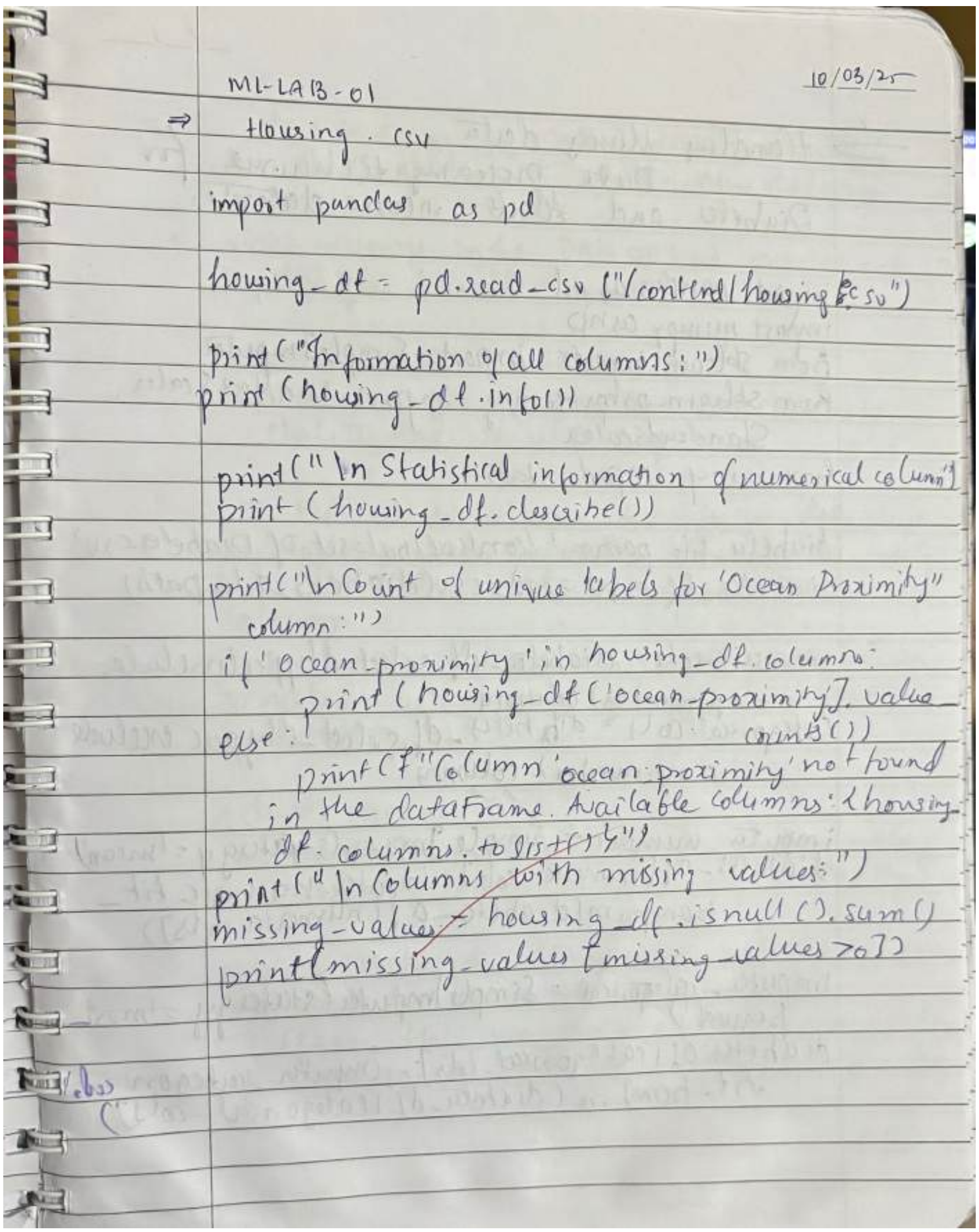
Github Link:

https://github.com/RaniAishwarya/Machine-learning/tree/main

Program 1

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

Observation:



ML-LAB-01 10/03/25

⇒ Housing . csv

```
import pandas as pd

housing_df = pd.read_csv ("/content/housing.csv")

print ("Information of all columns:")
print (housing_df.info())

print ("\n Statistical information of numerical column")
print (housing_df.describe())

print ("\n Count of unique labels for 'Ocean Proximity" column:")
if 'ocean_proximity' in housing_df.columns:
    print (housing_df['ocean_proximity'].value_counts())
else:
    print (f"Column 'ocean proximity' not found in the dataFrame. Available columns: {housing_df.columns.tolist()}")
print ("\n Columns with missing values:")
missing_values = housing_df.isnull().sum()
print(missing_values[missing_values > 0])
```

_/_/_

⟹ Handling Missing data,
- Data processing technique for Diabetes and Adult income datasets.

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler,
    StandardScaler
from scipy import stats

diabetes_file_path = '/content/Dataset of Diabetes.csv'
diabetes_df = pd.read_csv(diabetes_file_path)

numeric_cols = diabetes_df.select_dtypes(include=
    np.number).columns
categorical_cols = diabetes_df.select_dtypes(exclude
    = np.number).columns

imputer_numeric = SimpleImputer(strategy='mean')
diabetes_df[numeric_cols] = imputer_numeric.fit_
    transform(diabetes_df[numeric_cols])

imputer_categorical = SimpleImputer(strategy='most_
    frequent')
diabetes_df[categorical_cols] = imputer_categorical.
    fit_transform(diabetes_df[categorical_cols])
```

__/__/__

=> observations on Data processing techniques for the Diabetes & Adult income datasets

1. which columns in the Data set had missing values? How did you handle them?

<u>Diabetes dataset</u>:

→ columns with missing value: Urea, Cr, HbA1c, Chol, TG, HDL, LDL, VLDL, BMI.

→ For numeric columns for missing ~~dat~~ values, we applied mean imputation. The missing values were replaced with the mean of each respective column.

→ For categorical columns missing values were handled using most frequent value (mode) imputation.

<u>Adult income Dataset</u>:

→ ~~columns~~: In this dataset, missing values were represented by ? in the og dataset

→ we replaced all occurences of ? with NaN to make the missing values identifiable

→ After this, we applied mode imputation for categorical columns to replace missing values

—/—/—

2. which categorical ~~&~~ columns did you identify in the dataset? How did you encode them?

Diabetes:

→ columns: Gender, Races, Ches

→ For encoding categorical variables, one-hot encoding or label encoding could be applied to convert into binary columns.

Adult Income Dataset:

→ columns: work class, education, marital-s occupation, relationship, race, sex, native-country

→ one-hot encoding was applied to convert the categorical columns into multiple binary columns, For instance, work class would be converted into ~~ma~~ multiple binary column

→ Using pd.get_dummies() with drop_first allowed us to avoid multicollinearity by removing the first column from each category variable set.

3. What is the difference b/w Min-Max scaling and standardization? when would you use one over the other?

→ Min-Max Scaling:

or normalization transforms data to a specific range usually b/w 0&

- using the formula

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

when to use: Min-Max scaling is prefered when the dataset contains features with varying units or scales & you want to scale them to a fixed range.

Standardization:

or Z-Score normalization transforms data to ha a mean of 0 & a standard deviation of 1. It is calculated using the formula:

$$X_{scaled} = \frac{X - \mu}{\sigma}$$

μ - mean & σ - standard deviation.

→ standardisation is typically used when the data is assumed to follow a Gaussian distribution, especially when working with algorithms ~~that~~ assume normally distributed data regression, logistic regression.

10/3/25

Code:

1) <u>Data Cleaning:</u>

```
import pandas as pd

import numpy as np

from sklearn.impute import SimpleImputer


# --- Diabetes Dataset ---


# Load the diabetes dataset

diabetes_file_path = '/content/Dataset of Diabetes .csv'  # Replace with your actual file path

diabetes_df = pd.read_csv(diabetes_file_path)


# 1. Handling Missing Values:

# a. Identify numeric and categorical columns

numeric_cols = diabetes_df.select_dtypes(include=np.number).columns

categorical_cols = diabetes_df.select_dtypes(exclude=np.number).columns


# b. Impute missing values using the mean for numeric columns only

imputer_numeric = SimpleImputer(strategy='mean')

diabetes_df[numeric_cols] = imputer_numeric.fit_transform(diabetes_df[numeric_cols])


# c. Impute missing values using the most frequent value for categorical columns

imputer_categorical = SimpleImputer(strategy='most_frequent')
```

```
diabetes_df[categorical_cols] = imputer_categorical.fit_transform(diabetes_df[categorical_cols])


# --- Adult Income Dataset ---

# Load the adult income dataset
adult_file_path = '/content/adult.csv'  # Replace with your actual file path
adult_df = pd.read_csv(adult_file_path)

# 1. Handling Missing Values: Replace '?' with NaN and then impute
adult_df.replace('?', np.nan, inplace=True)
imputer = SimpleImputer(strategy='most_frequent')  # Use most frequent for categorical features

# Fit and transform, but keep column names using columns=adult_df.columns
adult_df_imputed = pd.DataFrame(imputer.fit_transform(adult_df), columns=adult_df.columns)

# 2. Handling Categorical Data: One-hot encoding for adult income dataset
# Ensure categorical_cols are present in adult_df_imputed
categorical_cols = adult_df_imputed.select_dtypes(include=['object']).columns.tolist()

adult_df_encoded = pd.get_dummies(adult_df_imputed, columns=categorical_cols, drop_first=True)

# --- Handling Outliers (for both datasets) ---
```

```
# (Example using Z-score - adjust as needed)
# from scipy import stats
# z = np.abs(stats.zscore(diabetes_df_imputed))  # For diabetes dataset
# diabetes_df_no_outliers = diabetes_df_imputed[(z < 3).all(axis=1)]
# z = np.abs(stats.zscore(adult_df_encoded.select_dtypes(include=np.number)))  # For adult dataset
# adult_df_no_outliers = adult_df_encoded[(z < 3).all(axis=1)]


# Print the preprocessed dataframes (example)
print("Diabetes data with imputed missing values:\n", diabetes_df.head())
print("\nAdult income data with imputed values and encoding:\n", adult_df_encoded.head())
```

2) Data Transformations:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.impute import SimpleImputer


# --- Diabetes Dataset ---


# Load the diabetes dataset
diabetes_file_path = '/content/Dataset of Diabetes .csv'
diabetes_df = pd.read_csv(diabetes_file_path)


# 1. Handling Missing Values:
```

```
# a. Identify numeric and categorical columns
numeric_cols = diabetes_df.select_dtypes(include=np.number).columns
categorical_cols = diabetes_df.select_dtypes(exclude=np.number).columns

# b. Impute missing values using the mean for numeric columns only
imputer_numeric = SimpleImputer(strategy='mean')
diabetes_df[numeric_cols] = imputer_numeric.fit_transform(diabetes_df[numeric_cols])

# c. Impute missing values using the most frequent value for categorical columns
imputer_categorical = SimpleImputer(strategy='most_frequent')
diabetes_df[categorical_cols] = imputer_categorical.fit_transform(diabetes_df[categorical_cols])



# 2. Data Transformations:
# a. Min-Max Scaling
scaler_minmax = MinMaxScaler()
diabetes_df[numeric_cols] = scaler_minmax.fit_transform(diabetes_df[numeric_cols])

# b. Standard Scaling (create a separate copy)
diabetes_df_std = diabetes_df.copy()
scaler_std = StandardScaler()
diabetes_df_std[numeric_cols] = scaler_std.fit_transform(diabetes_df_std[numeric_cols])
```

```
# --- Adult Income Dataset ---

# Load the adult income dataset
adult_file_path = '/content/adult.csv'
adult_df = pd.read_csv(adult_file_path)

# 1. Handling Missing Values: Replace '?' with NaN and then impute
adult_df.replace('?', np.nan, inplace=True)
imputer = SimpleImputer(strategy='most_frequent')
adult_df_imputed = pd.DataFrame(imputer.fit_transform(adult_df), columns=adult_df.columns)

# 2. Handling Categorical Data: One-hot encoding
categorical_cols = adult_df_imputed.select_dtypes(include=['object']).columns.tolist()
adult_df_encoded = pd.get_dummies(adult_df_imputed, columns=categorical_cols, drop_first=True)

# ... (previous code)

# 3. Data Transformations:
# Get numeric columns after encoding.
# Check if any numeric columns exist to avoid error
numeric_cols_adult = adult_df_encoded.select_dtypes(include=np.number).columns
```

```
# Initialize adult_df_encoded_std before the if statement

# to an empty DataFrame if no numeric columns are found

adult_df_encoded_std = pd.DataFrame()

if len(numeric_cols_adult) > 0:

   # a. Min-Max Scaling

   # ... (code for Min-Max scaling remains the same) ...



   # b. Standard Scaling (create a separate copy)

   adult_df_encoded_std = adult_df_encoded.copy()

   scaler_std_adult = StandardScaler()

   adult_df_encoded_std[numeric_cols_adult] =
scaler_std_adult.fit_transform(adult_df_encoded_std[numeric_cols_adult])

else:

   print("No numeric columns found for scaling in adult income dataset.")

# ... (rest of the code)





# Print the preprocessed dataframes (examples)

print("Diabetes data with Min-Max scaling:\n", diabetes_df.head())

print("\nDiabetes data with Standard scaling:\n", diabetes_df_std.head())

print("\nAdult income data with Min-Max scaling:\n", adult_df_encoded.head())

print("\nAdult income data with Standard scaling:\n", adult_df_encoded_std.head())
```

Program 2

Demonstrate various data pre-processing techniques for a given dataset

Observation:

LAB - 02

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

dataset = pd.read_csv('/content/weather.csv')
X = dataset.iloc[:,:].values
X

output:
array([['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'],
['Rain', 'Cool', 'Normal', 'Strong', 'No'],
['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
['Sunny', 'Mild', 'High', 'weak', 'No'],
['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
['Rain', 'Mild', 'Normal', 'weak', 'Yes'],
['Sunny', 'Mild', 'Normal', ~~'weak'~~ Strong', 'Yes'],
['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
['Overcast', 'Hot', 'Normal', 'weak', 'Yes'],
['Rain', 'Mild', 'High', 'Strong', 'No']], dtype=object)

attribute = ['Outlook', 'Temp', 'Humidity', 'Wind']
```

_/_/_

```
→ class Node(object):
      def __init__(self):
          self.value = None
          self.decision = None
          self.child = None

→ def findEntropy(data, rows):
      yes = 0
      no = 0
      ans = -1
      idx = len(data[0]) - 1
      entropy = 0
      for i in rows:
          if data[i][idx] == 'Yes':
              yes = yes + 1
          else:
              no = no + 1

      x = yes / (yes + no)
      y = no / (yes + no)
      if x != 0 and y != 0:
          entropy = -1 * (x * math.log2(x) + y * math.
                                                log2(y))
      if x == 1:
          ans = 1
      if y == 1:
          ans = 0
      return entropy, ans
```

```
→ def findMaxGain(data, rows, columns):
      maxGain = 0
      retidx = -1
      entropy, ans = findEntropy(data, rows)
      if entropy == 0:
          return maxGain, retidx, ans
      for j in columns:
          mydict = {}
          idx = j
          for i in rows:
              key = data[i][idx]
              if key not in mydict:
                  mydict[Key] = 1
              else:
                  mydict[Key] = mydict[Key
          gain = entropy
          for key in mydict:
              yes = 0
              no = 0
              for k in rows:
                  if data[k][j] == key:
                      if data[k][-1] == 'yes':
                          yes = yes + 1
                      else:
                          no = no + 1
              x = yes/(yes + no)
              y = no/(yes + no)
```

```
if x!=0 and y!=0:
    gain += (mydict[key] * (x* math.log2(x)+ y* math.log2(y)))/14

if gain > maxGain:
    maxGain = gain
    retidx = j
return maxGain, retidx, ans

def buildTree(data, rows, columns):
    maxGain, idx, ans = findMaxGain(X, rows, columns)
    root = Node()
    root.childs = []

    if maxGain == 0:
        if ans == 1:
            root.value = 'yes'
        else:
            root.value = 'No'
        return root
    root.value = attribute[idx]
    mydict = {}
    for i in rows:
        key = data[i][idx]
        if key not in mydict:
            mydict[key] = 1
        else:
            mydict[key] += 1
```

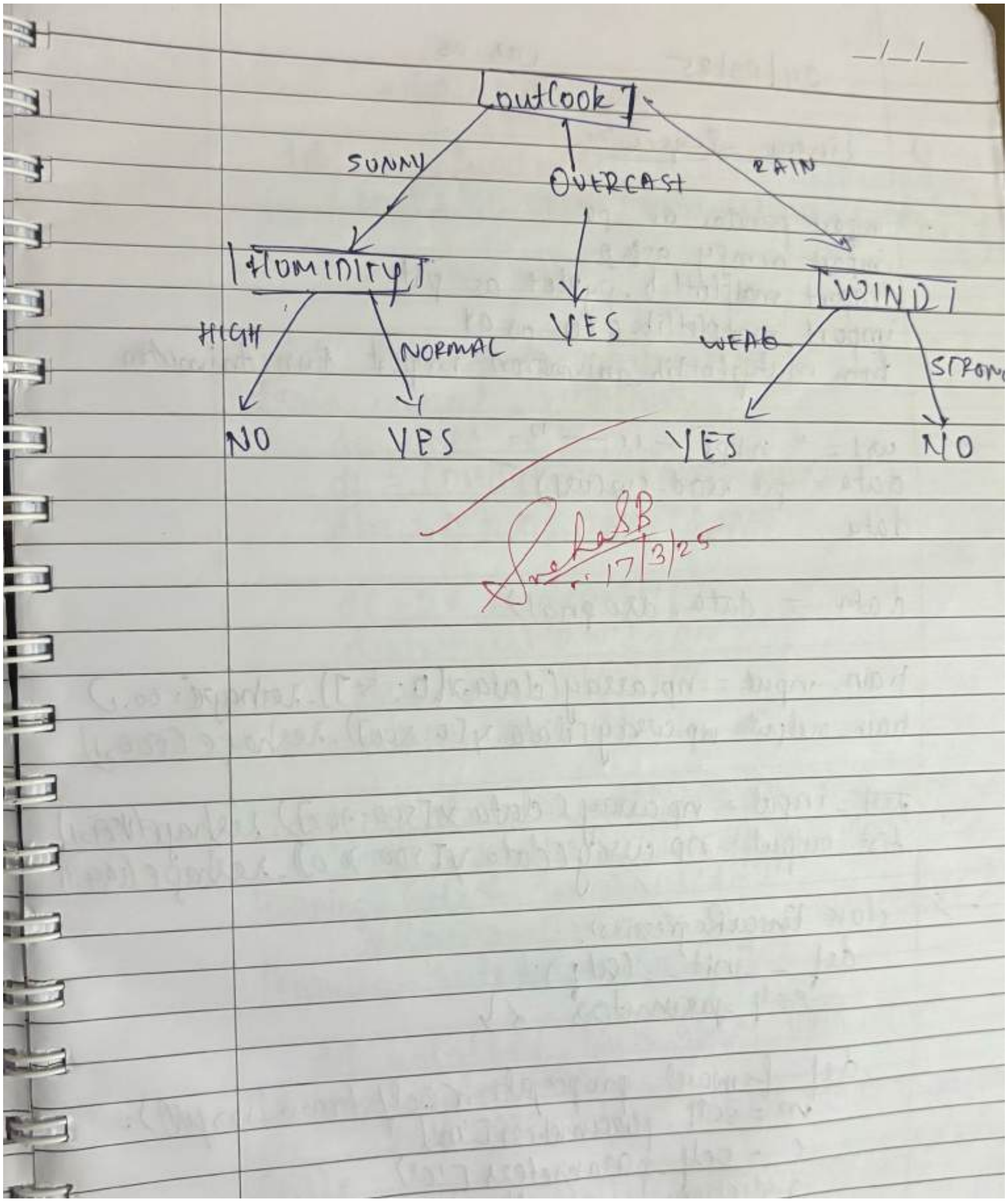
__/__/__

```
newcolumns = copy.deepcopy(columns)
new columns.remove(idx)
for key in mydict:
    newrows = []
    for i in rows:
        if data[i][idx] == key:
            newrows.append(i)
    temp = buildtree(data, newrows, newcolumns)
    temp.decision = key
    root.childs.append(temp)
return root

→ def traverse(root):
    print(root.decision)
    print(root.value)

    n = len(root.childs)
    if n>0:
        for i in range(0,n):
            traverse(root.childs[i])

→ def calculate():
    rows = [i for i in range(0,14)]
    columns = [i for i in range(0,4)]
    root = build.tree(X, rows, columns)
    root.decision = 'Start'
    traverse(root)
```

→ calculate()

output: Start
outlook
Sunny
Humidity
High
No
Normal
Yes
Overcast
Yes
Rain
wind
Weak
Yes
Strong
No

→
```
from graphviz import Digraph
from Ipython.display import Image
dot = Digraph()
dot.node('image', label='', image='/content/
   TennisDTFinal (1).jpg')

dot.format = 'png'
dot.render('image_graph', view=False)
display(Image('image_graph.png'))
```



OUTLOOK
SUNNY
OVERCAST
RAIN
HUMIDITY
YES
WIND
HIGH
NORMAL
WEAK
STRONG
NO
YES
YES
NO
17/3/25

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
import copy
dataset = pd.read_csv('D:\Datasets\Tennis.csv')
X = dataset.iloc[:,:].values
X

attribute = ['Outlook', 'Temp', 'Humidity', ‘Wind']

def findEntropy(data, rows):
    yes=0
    no=0
    ans=-1
    idx=len(data[0])-1
    entropy=0

    for i in rows:
        if data[i][idx]=='Yes':
            yes=yes+1
        else:
            no=no+1

    x=yes/(yes+no)
    y=no/(yes+no)
    if x!=0 and y!=0:
        entropy= -1*(x*math.log2(x)+y*math.log2(y))
    if x==1:
        ans = 1
    if y==1:
        ans = 0
    return entropy, ans



def findMaxGain(data, rows, columns):
    maxGain = 0
    retidx = -1
    entropy, ans = findEntropy(data, rows)
```

```
    if entropy == 0:
        """if ans == 1:
            print("Yes")
        else:
            print("No")"""
        return maxGain, retidx, ans
    for j in columns:
        mydict = {}
        idx = j
        for i in rows:
            key = data[i][idx]
            if key not in mydict:
                mydict[key] = 1
            else:
                mydict[key] = mydict[key] + 1
        gain = entropy

        # print(mydict)
        for key in mydict:
            yes = 0
            no = 0
            for k in rows:
                if data[k][j] == key:
                    if data[k][-1] == 'Yes':
                        yes = yes + 1
                    else:
                        no = no + 1
            # print(yes, no)
            x = yes/(yes+no)
            y = no/(yes+no)
            # print(x, y)
            if x != 0 and y != 0:
              gain += (mydict[key] * (x*math.log2(x) +
y*math.log2(y)))/14
        # print(gain)
        if gain > maxGain:
            # print("hello")
            maxGain = gain
            retidx = j

    return maxGain, retidx, ans
```

```python
def buildTree(data, rows, columns):

    maxGain, idx, ans = findMaxGain(X, rows, columns)
    root = Node()
    root.childs = []
    # print(maxGain)

    if maxGain == 0:
        if ans == 1:
            root.value = 'Yes'
        else:
            root.value = 'No'
        return root

    root.value = attribute[idx]
    mydict = {}
    for i in rows:
        key = data[i][idx]
        if key not in mydict:
            mydict[key] = 1
        else:
            mydict[key] += 1

    newcolumns = copy.deepcopy(columns)
    newcolumns.remove(idx)
    for key in mydict:
        newrows = []
        for i in rows:
            if data[i][idx] == key:
                newrows.append(i)
        # print(newrows)
        temp = buildTree(data, newrows, newcolumns)
        temp.decision = key
        root.childs.append(temp)
    return root

def traverse(root):
    print(root.decision)
    print(root.value)

    n = len(root.childs)
```

```
    if n > 0:
        for i in range(0, n):
            traverse(root.childs[i])

def calculate():
    rows = [i for i in range(0, 14)]
    columns = [i for i in range(0, 4)]
    root = buildTree(X, rows, columns)
    root.decision = 'Start'
    traverse(root)

calculate()
```

```
from graphviz import Digraph
from IPython.display import Image
dot = Digraph()
dot.node('image', label='', image='/content/TennisDTFinal
(1).jpg')
# Replace /path/to/your/image.jpg with the actual path to your
image
dot.format = 'png'
dot.render('image_graph', view=False)
# Render to a file named 'image_graph.png' and display the image
display(Image('image_graph.png'))
```

Program 3

Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

Observation:

24|03|25 Lab-03

1) Linear Regression:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from matplotlib.animation import FuncAnimation

url = "https:-//-"
data = pd.read_csv(url)
data

data = data.dropna()

train_input = np.array(data.x[0:500]).reshape(500,1)
train_output = np.array(data.y[0:500]).reshape(500,1)

test_input = np.array(data.x[500:700]).reshape(199,1)
test_output = np.array(data.y[500:700]).reshape(199,1)

class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propogation(self, train_input):
        m = self.parameters['m']
        c = self.parameters['c']
        predictions = np.multiply(m, train_input) + c
```

```
         return predictions

    def cost_function(self, predictions, train_output):
        cost = np.mean((train_output - predictions)
                                                  ** 2)
        return cost

    def backward_propagation(self, train_input,
train_output, predictions):
        derivatives = {}
        df = (predictions - train_output)
        dm = 2 * np.mean(np.multiply(train_input,
                                              df))
        dc = 2 * np.mean(df)
        derivatives['dm'] = dm
        derivatives['dc'] = dc
        return derivatives

    def update_parameters(self, derivatives, learning_rate):
        self.parameters['m'] = self.parameters['m'] -
learning_rate * derivatives['dm']
        self.parameters['c'] = self.parameters['c'] -
learning_rate * derivatives['dc']

    def train(self, train_input, train_output,
learning_rate, iters):
        self.parameters['m'] = np.random.uniform
(0,1) * -1
        self.parameters['c'] = np.random.uniform(0,1) *
```

__/__/__

```
ani = FuncAnimation(fig, update, frames=iter,
interval=200, blit=True)

ani.save('Linear_regression_A.gif', writer='ffmpeg')

plt.xlabel('Input')
plt.ylabel('Output')
plt.title('Linear Regression')
plt.legend()
plt.show()

return self.parameters, self.loss
```

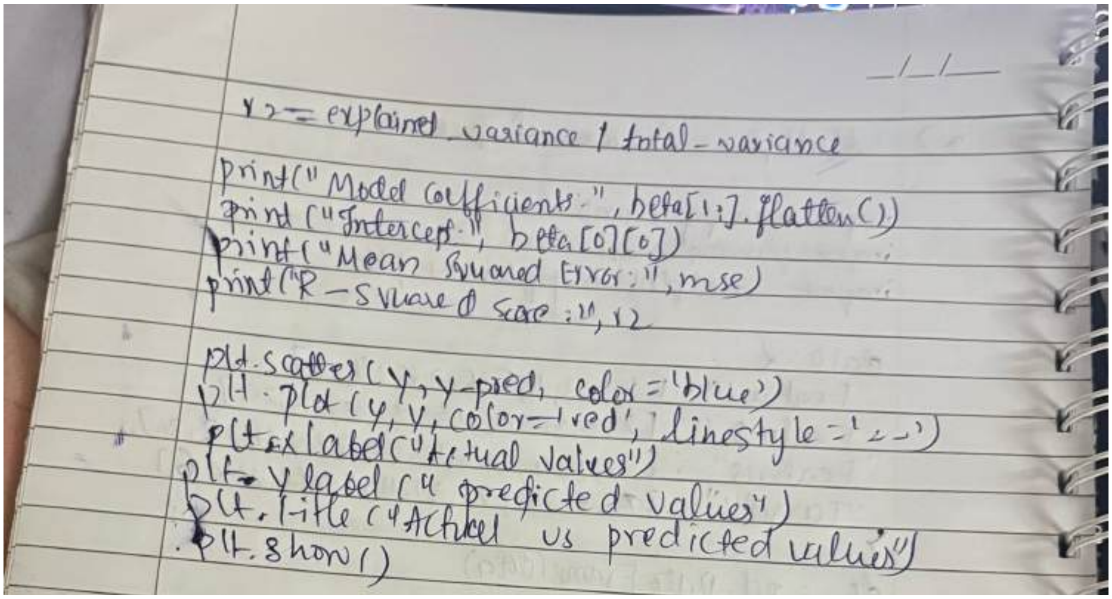
output

```
Iteration 1, Loss = 12238.85901767012
Iteration 2, Loss = 10.40095211078788527
Iteration 3, Loss = 8.13213234195354
```

__/__/__

2) Multiple Linear Regression:

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

data = {
 "Feature1": [1,2,3,4,5,6,7,8,9,10],
 "Feature2": [2,3,5,7,11,13,17,19,23,29],
 "Feature3": [3,6,9,12,15,18,21,24,27,30],
 "Target": [5,9,15,22,31,41,53,66,80,96]
}
df = pd.DataFrame(data)

X = df.drop(columns=["Target"]).values
y = df["Target"].values.reshape(-1,1)

~~X = np.linalg.sol~~
X = np.hstack((np.ones((X.shape[0],1)), X))

beta = np.linalg.solve(X.T @ X + 0.01 * np.identity
 (X.shape[1]), X.T @ y)

y_pred = X @ beta

mse = np.mean((y - y_pred) ** 2)
total_variance = np.sum((y - np.mean(y)) ** 2)
explained_variance = np.sum((y_pred - np.mean(y)
 ** 2)
```


r2 = explained_variance / total_variance
print("Model coefficients:", beta[1:].flatten())
print("Intercept:", beta[0][0])
print("Mean Squared Error:", mse)
print("R-Squared Score:", r2
plt.scatter(y, y_pred, color='blue')
plt.plot(y, y, color='red', linestyle='--')
plt.xlabel("Actual values")
plt.ylabel("predicted values")
plt.title("Actual vs predicted values")
plt.show()

Code:

```python
#multiple linear regression
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Sample dataset
data = {
    "Feature1": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    "Feature2": [2, 3, 5, 7, 11, 13, 17, 19, 23, 29],
    "Feature3": [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
    "Target": [5, 9, 15, 22, 31, 41, 53, 66, 80, 96]
}

df = pd.DataFrame(data)

# Split dataset into features (X) and target variable (y)
X = df.drop(columns=["Target"]).values
y = df["Target"].values.reshape(-1, 1)

# Add intercept column (bias term)
X = np.hstack((np.ones((X.shape[0], 1)), X))

# Compute the coefficients using the Normal Equation
beta = np.linalg.solve(X.T @ X + 0.01 * np.identity(X.shape[1]),
X.T @ y)
```

```
# Make predictions
y_pred = X @ beta

# Evaluate the model
mse = np.mean((y - y_pred) ** 2)
total_variance = np.sum((y - np.mean(y)) ** 2)
explained_variance = np.sum((y_pred - np.mean(y)) ** 2)
r2 = explained_variance / total_variance

# Display results
print("Model Coefficients:", beta[1:].flatten())
print("Intercept:", beta[0][0])
print("Mean Squared Error:", mse)
print("R-squared Score:", r2)

# Plot actual vs predicted values
plt.scatter(y, y_pred, color='blue')
plt.plot(y, y, color='red', linestyle='--')
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs Predicted Values")
plt.show()
```

Linear Regression:

```
from google.colab import drive
drive.mount('/content/drive')

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from matplotlib.animation import FuncAnimation


url = 'https://media.geeksforgeeks.org/wp-content/uploads/
20240320114716/data_for_lr.csv'
data = pd.read_csv(url)
data

# Drop the missing values
```

```
data = data.dropna()

# training dataset and labels
train_input = np.array(data.x[0:500]).reshape(500, 1)
train_output = np.array(data.y[0:500]).reshape(500, 1)

# valid dataset and labels
test_input = np.array(data.x[500:700]).reshape(199, 1)
test_output = np.array(data.y[500:700]).reshape(199, 1)


class LinearRegression:
    def __init__(self):
        self.parameters = {}

    def forward_propagation(self, train_input):
        m = self.parameters['m']
        c = self.parameters['c']
        predictions = np.multiply(m, train_input) + c
        return predictions

    def cost_function(self, predictions, train_output):
        cost = np.mean((train_output - predictions) ** 2)
        return cost

    def backward_propagation(self, train_input, train_output,
predictions):
        derivatives = {}
        df = (predictions-train_output)
        # dm= 2/n * mean of (predictions-actual) * input
        dm = 2 * np.mean(np.multiply(train_input, df))
        # dc = 2/n * mean of (predictions-actual)
        dc = 2 * np.mean(df)
        derivatives['dm'] = dm
        derivatives['dc'] = dc
        return derivatives

    def update_parameters(self, derivatives, learning_rate):
        self.parameters['m'] = self.parameters['m'] -
learning_rate * derivatives['dm']
```

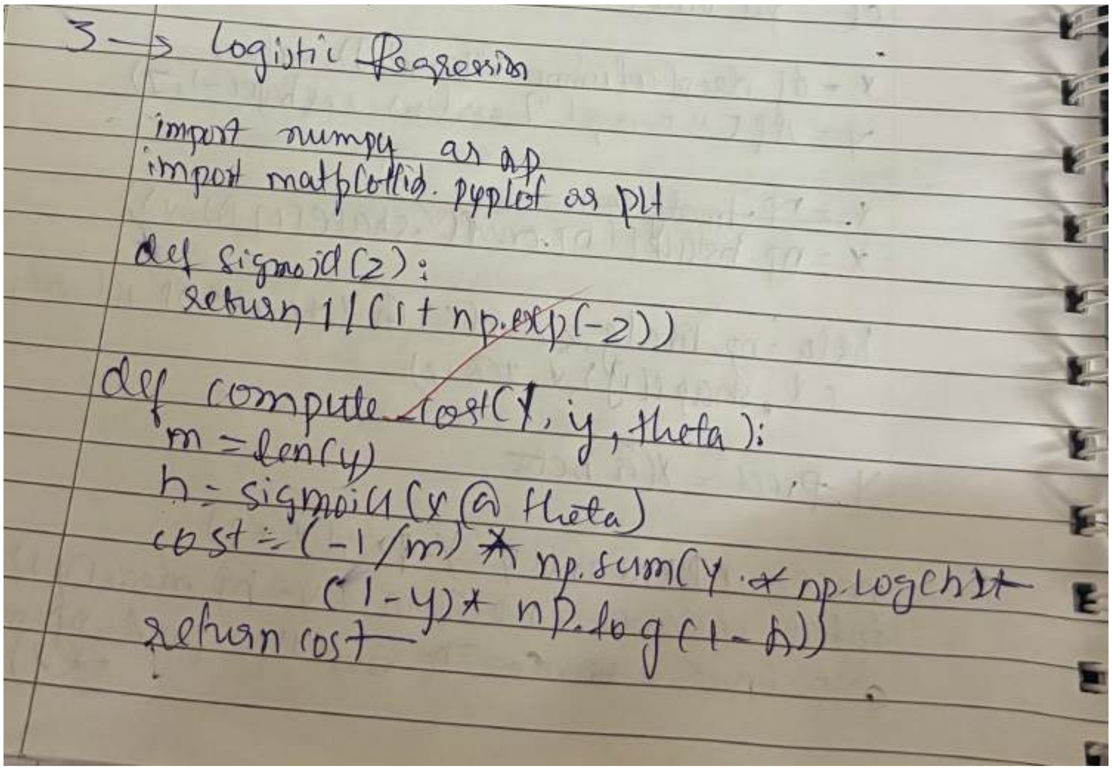
```
        self.parameters['c'] = self.parameters['c'] - 
learning_rate * derivatives['dc']

    def train(self, train_input, train_output, learning_rate, 
iters):
        # Initialize random parameters
        self.parameters['m'] = np.random.uniform(0, 1) * -1
        self.parameters['c'] = np.random.uniform(0, 1) * -1

        # Initialize loss
        self.loss = []

        # Initialize figure and axis for animation
        fig, ax = plt.subplots()
        x_vals = np.linspace(min(train_input), max(train_input), 
100)
        line, = ax.plot(x_vals, self.parameters['m'] * x_vals +
                        self.parameters['c'], color='red', 
label='Regression Line')
        ax.scatter(train_input, train_output, marker='o',
                color='green', label='Training Data')

        # Set y-axis limits to exclude negative values
        ax.set_ylim(0, max(train_output) + 1)

        def update(frame):
            # Forward propagation
            predictions = self.forward_propagation(train_input)

            # Cost function
            cost = self.cost_function(predictions, train_output)

            # Back propagation
            derivatives = self.backward_propagation(
                train_input, train_output, predictions)

            # Update parameters
            self.update_parameters(derivatives, learning_rate)

            # Update the regression line
            line.set_ydata(self.parameters['m']
```

```
                        * x_vals + self.parameters['c'])

            # Append loss and print
            self.loss.append(cost)
            print("Iteration = {}, Loss = {}".format(frame + 1,
cost))

            return line,
        # Create animation
        ani = FuncAnimation(fig, update, frames=iters,
interval=200, blit=True)

        # Save the animation as a video file (e.g., MP4)
        ani.save('linear_regression_A.gif', writer='ffmpeg')

        plt.xlabel('Input')
        plt.ylabel('Output')
        plt.title('Linear Regression')
        plt.legend()
        plt.show()

        return self.parameters, self.loss


#Example usage
from matplotlib.animation import FuncAnimation #import
FuncAnimation
linear_reg = LinearRegression()
parameters, loss = linear_reg.train(train_input, train_output,
0.0001, 20)
```

Program 4

Build Logistic Regression Model for a given dataset

Observation:



3 → Logistic Regression

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

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) * np.sum(y * np.log(h) +
           (1-y) * np.log(1-h))
    return cost
```

_/_/_

```
def gradient_descent(x, y, theta, alpha,
  iterations):
    m = len(y)
    cost_history = []

    for _ in range(iteration):
        gradient = (1/m) * X.T @ (sigmoid (X
                    @ theta) - y)
        theta -= alpha * gradient
        cost_history.append(compute_cost (X,
            y, theta))
    return theta, cost_history

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

np.random.seed(42)
x = np.random.rand(100, 1) * 10
y = (X > 5).astype(int).ravel()

X_b = np.c_[np.ones((X.shape[0], 1)), X]
theta = np.zeros(X_b.shape[1])
alpha = 0.1
iteration = 1000

theta, cost_history = gradient_descent(X_b, y,
  theta, alpha, iteration)

y_pred = predict(X_b, theta)
```

```
accuracy = np.mean(y_pred == y)
print(f"Accuracy: {accuracy:.2f}")

plt.scatter(x, y, color='blue', label='Actual Data')

plt-scatter(x, y_pred, color='red', marker='x',
   label='predicted labels')

plt.xlabel("Feature(x)")
plt.ylabel("class(0 or 1)")
plt.legend()
plt.show()
```

output:
Accuracy: 0.97

24/3/25

Code:

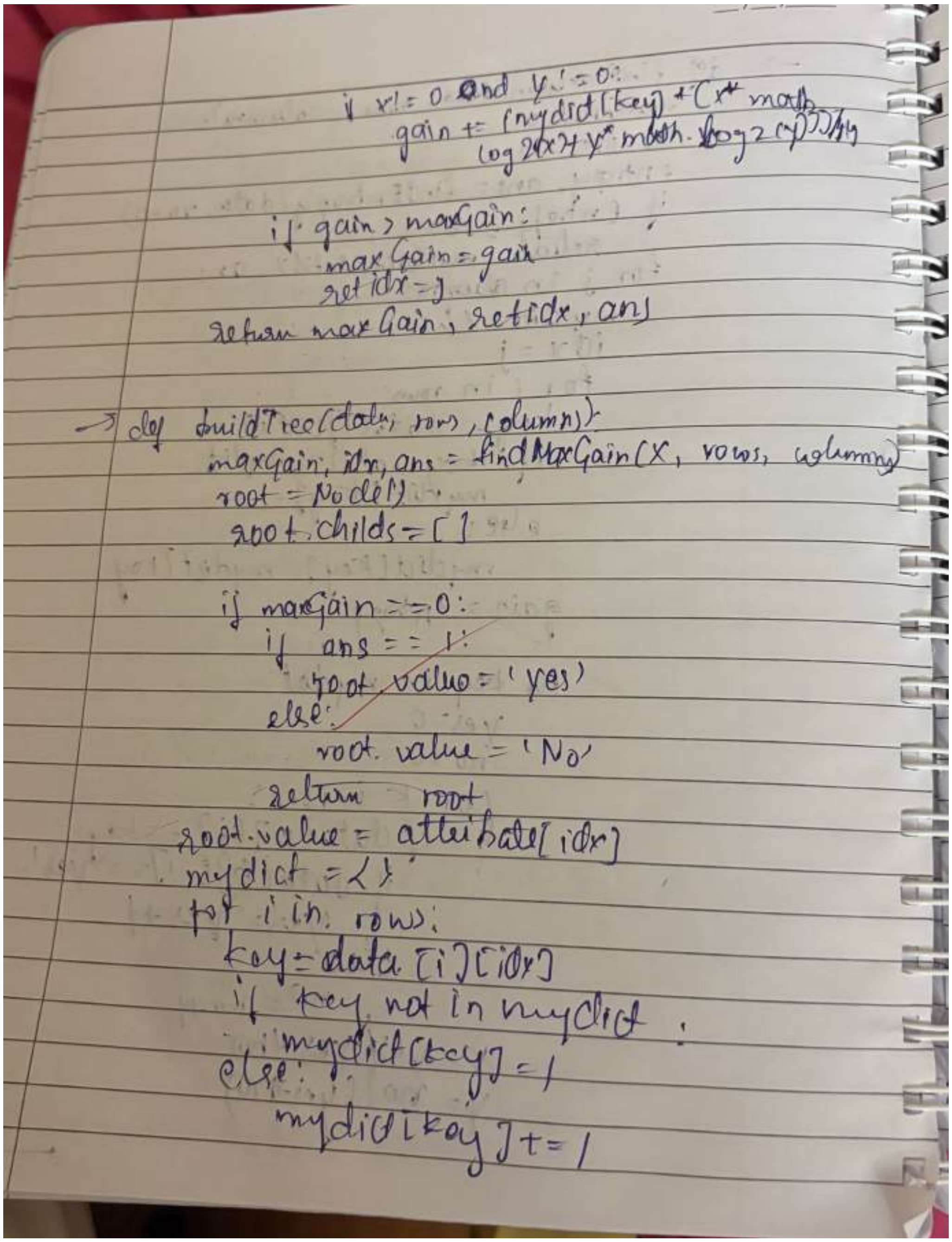
```
import numpy as np
import matplotlib.pyplot as plt

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) * np.sum(y * np.log(h) + (1 - y) * np.log(1 -
h))
    return cost

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

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

    return theta, cost_history

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

# Generate synthetic binary classification data
np.random.seed(42)
X = np.random.rand(100, 1) * 10  # Feature values between 0 and
10
y = (X > 5).astype(int).ravel()  # Label: 1 if X > 5, else 0

# Add intercept term
X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Initialize parameters
theta = np.zeros(X_b.shape[1])
alpha = 0.1
```

```
iterations = 1000

# Train logistic regression using gradient descent
theta, cost_history = gradient_descent(X_b, y, theta, alpha,
iterations)

# Make predictions
y_pred = predict(X_b, theta)

# Compute accuracy
accuracy = np.mean(y_pred == y)
print(f"Accuracy: {accuracy:.2f}")

# Plot the decision boundary
plt.scatter(X, y, color='blue', label='Actual Data')
plt.scatter(X, y_pred, color='red', marker='x', label='Predicted
Labels')
plt.xlabel("Feature X")
plt.ylabel("Class (0 or 1)")
plt.legend()
plt.title("Logistic Regression Model (Without Scikit-learn)")
plt.show()
```

Program 5

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

Observation:



```
if x != 0 and y != 0:
    gain += (mydict[key] * (x* math.log2(x)+ y* math.log2(y)))/14

if gain > maxGain:
    maxGain = gain
    retidx = j
return maxGain, retidx, ans

def buildTree(data, rows, columns):
    maxGain, idx, ans = findMaxGain(X, rows, columns)
    root = Node()
    root.childs = []

    if maxGain == 0:
        if ans == 1:
            root.value = 'yes'
        else:
            root.value = 'No'
        return root
    root.value = attribute[idx]
    mydict = {}
    for i in rows:
        key = data[i][idx]
        if key not in mydict:
            mydict[key] = 1
        else:
            mydict[key] += 1
```

__/__/__

```
newcolumns = copy.deepcopy(columns)
newcolumns.remove(idx)
for key in mydict:
    newrows = []
    for i in rows:
        if data[i][idx] == key:
            newrows.append(i)
    temp = buildtree(data, newrows, newcolu
    temp.decision = key
    root.childs.append(temp)
return root
```

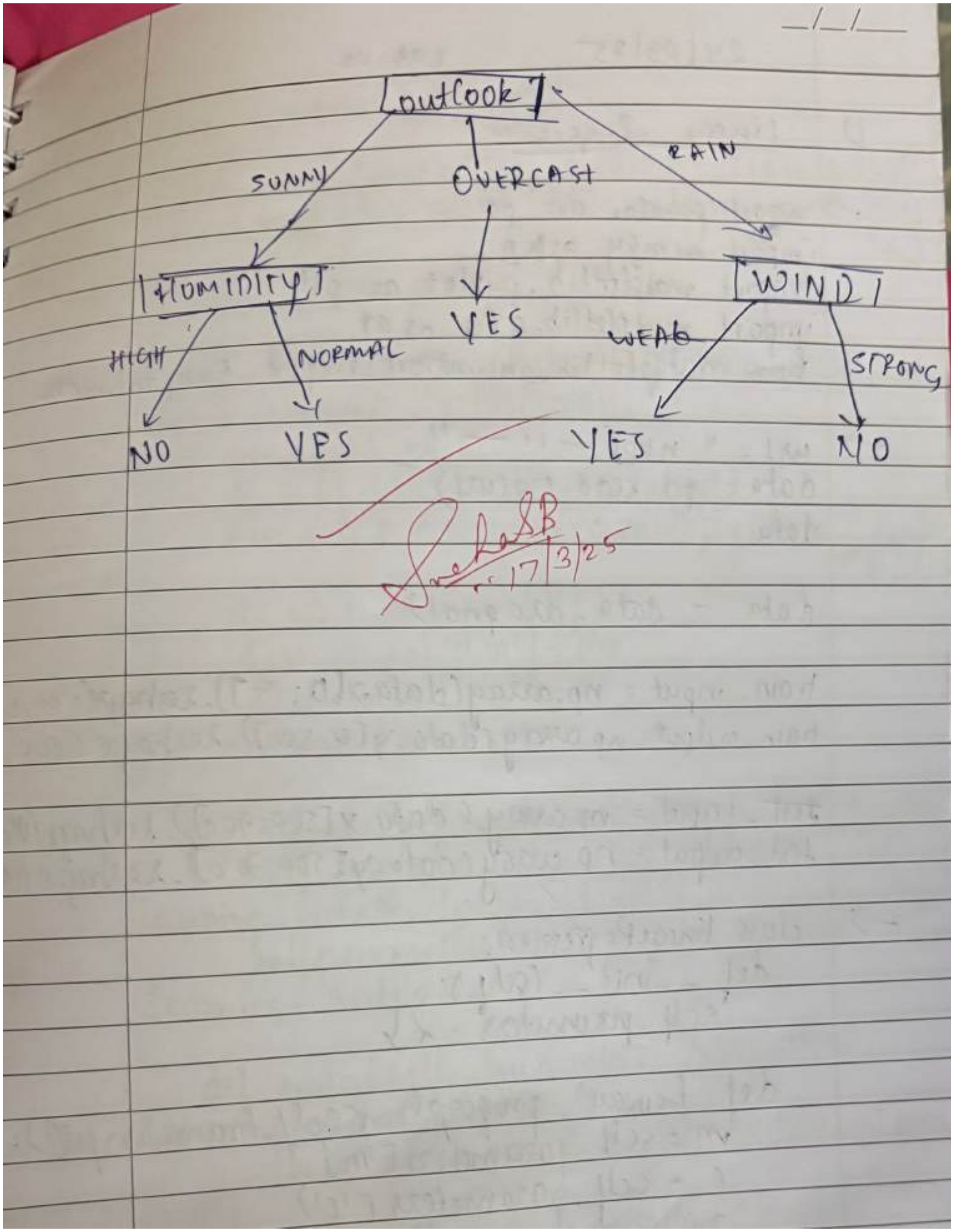
```
→ def traverse(root):
    print(root.decision)
    print(root.value)

    n = len(root.childs)
    if n>0:
        for i in range(0,n):
            traverse(root.childs[i])
```

```
→ def calculate():
    rows = [i for i in range(0,14)]
    columns = [i for i in range(0,4)]
    root = build.tree(X, rows, columns)
    root.decision = 'Start'
    traverse(root)
```

→ calculate()

output: Start
outlook
Sunny
Humidity
High
No
Normal
Yes
Overcast
Yes
Rain
wind
Weak
Yes
Strong
No

```
→ from graphviz import Digraph
from Ipython.display import Image
dot = Digraph()
dot.node('image', label='', image='/content/TennisDTFinal (1).jpg')

dot.format = 'png'
dot.render('image_graph', view=False)
display(Image('image_graph.png'))
```



outlook
SUNNY
OVERCAST
RAIN
HUMIDITY
YES
WIND
HIGH
NORMAL
WEAK
STRONG
NO
YES
YES
NO
17/3/25

Code:

```
def buildTree(data, rows, columns):


    maxGain, idx, ans = findMaxGain(X, rows, columns)

    root = Node()

    root.childs = []

    # print(maxGain)


    if maxGain == 0:

        if ans == 1:

            root.value = 'Yes'

        else:

            root.value = 'No'

        return root


    root.value = attribute[idx]

    mydict = {}

    for i in rows:

        key = data[i][idx]

        if key not in mydict:

            mydict[key] = 1

        else:

            mydict[key] += 1
```

```
    newcolumns = copy.deepcopy(columns)
    newcolumns.remove(idx)
    for key in mydict:
        newrows = []
        for i in rows:
            if data[i][idx] == key:
                newrows.append(i)
        # print(newrows)
        temp = buildTree(data, newrows, newcolumns)
        temp.decision = key
        root.childs.append(temp)
    return root

def traverse(root):
    print(root.decision)
    print(root.value)


    n = len(root.childs)
    if n > 0:
        for i in range(0, n):
            traverse(root.childs[I])
```

```
def calculate():
    rows = [i for i in range(0, 14)]
    columns = [i for i in range(0, 4)]
    root = buildTree(X, rows, columns)
    root.decision = 'Start'
    traverse(root)


calculate()


from graphviz import Digraph
from IPython.display import Image
dot = Digraph()
dot.node('image', label='', image='/content/TennisDTFinal (1).jpg')
# Replace /path/to/your/image.jpg with the actual path to your image
dot.format = 'png'
dot.render('image_graph', view=False)
# Render to a file named 'image_graph.png' and display the image
display(Image('image_graph.png'))
```

Program 6

Build KNN Classification model for a given dataset

Observation:

07/04/25 LAB-04. __/__/__

K Nearest Number:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

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

class KNN:
    def init_(self,k=3):
        self.k=k

    def fit(self,x,y):
        self.x_train= np.array(x)
        self.y_train =np.array(y)

    def predict(self, x):
        return [self._predict(x) for x in X]

    def _predict(self,x):
        distances= [euclidean_distance(x,x_train) for
x_train in self.x_train]
        k_indices=np.argsort(distances)[:self.k]
        k_nearest_labels= [self.y_train[i] for i in k_indices
        most_common= (Counter(k_nearest_labels).
most_common())
        return most_common[0][0]
```

___/___/___

```
def score(self, x, y):
    predictions = self.predict(x)
    return np.mean(predictions == y)

x_train = np.array([[1,2],[2,3],[3,1],[6,5],[7,7],
[8,6]])
y_train = np.array([[0,0,0,1,1,1]])
x_test = np.array([[5,5]])

knn = KNN(k=3)
knn.fit(x_train, y_train)
prediction = knn.predic(x_test)

plt.figure(figsize=(8,6))

for i in range(len(x_train)):
    plt.scatter(x_train[i][0], x_train[i][1],
                color='red' if y_train[i]==0 else 'blue',
                label=f"class{y_train[i]}" if f"class
{y_train[i]}" not in plt.gca().get_legend_handles_
labels()[1] else "")

plt.scatter(x_test[0][0], x_test[0][1], color='green',
  s=200, marker='X', label='Test point')

plt.scatter(x_test[0][0], x_test[0][1], color=
  'green', s=200, marker='X', label='Test point'
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
```

___/___/___

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

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

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

# KNN Class
class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = np.array(X)
        self.y_train = np.array(y)

    def predict(self, X):
        return [self._predict(x) for x in X]

    def _predict(self, x):
        distances = [euclidean_distance(x, x_train) for x_train
in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

    def score(self, X, y):
        predictions = self.predict(X)
        return np.mean(predictions == y)

# Sample data
X_train = np.array([[1, 2], [2, 3], [3, 1], [6, 5], [7, 7], [8,
6]])
y_train = np.array([0, 0, 0, 1, 1, 1])
X_test = np.array([[5, 5]])

# Instantiate and fit KNN
```

```
knn = KNN(k=3)
knn.fit(X_train, y_train)
prediction = knn.predict(X_test)

# Plotting
plt.figure(figsize=(8, 6))

# Training data
for i in range(len(X_train)):
    plt.scatter(X_train[i][0], X_train[i][1],
                color='red' if y_train[i] == 0 else 'blue',
                label=f"Class {y_train[i]}" if f"Class
{y_train[i]}" not in plt.gca().get_legend_handles_labels()[1]
else "")

# Test point
plt.scatter(X_test[0][0], X_test[0][1], color='green', s=200,
marker='X', label='Test Point')

plt.title(f"KNN Classification (Predicted Class:
{prediction[0]})")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.grid(True)
plt.show()
```

Program 7

Build Support vector machine model for a given dataset

Observation:

⟹ SVM.

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

class SVM:
  def __init__(self, learning_rate=0.001,
lambda_param=0.01, n_iters=1000):
      self.lr = learning_rate
      self.lambdaparam = lambda_param
      self.n_iters = n_iters
      self.w = None
      self.b = None

  def fit(self, X, y):
      y = np.where(y <= 0, -1, 1)
      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)
            self.b += self.lr * y[idx]

    def predict (self, X):
        approx = np.dot (X, self.w + self.b
        return np.sign(approx)

    def visualize (self, X, Y, new_point = None,
prediction = None):
        def get_hyperplane(x, w, b, offset):
            return(-w[0] * x+b+ offset) / w[1]

        fig = plt.figure()
        ax = fig.add_subplot(1,1,1)

        for i, sample in enumerate(X):
            if y[i] == 1:
                plt.scatter(sample[0], sample[1],
marker ='o', color = 'blue', label = 'Class +1' if
i == 0 else "")
            else:
                plt.scatter(sample[0], sample[1],
marker='x', color='red', label='class -1' if
i == 0 else "")
```

```
        x0 = np.linspace(np.min(X[:, 0])-1,
np.max(X[:, 0])+1, 100)
        x1 = get_hyperplane(x0, self.w, self.b,
        x1_m = get_hyperplane(x0, self.w, self.b,
        x1_p = get_Hyperplane(x0, self.w, self.b,

        ax.plot(x0, x1, 'k-', label='Decision Bounda
        ax.plot(x0, x1_m, 'k--', label='Margin
        ax.plot(x0, x1_p, 'k--')

        if new_point is not None:
            color = 'green' if prediction == 1 else
'orange'
            label = f'New Point: Class {"1" if prediction
== 1 else "0"}'
            plt.scatter(new_point[0], new_point[1],
c=color, s=100, edgecolors='black', label=
label, marker='*')

        ax.legend()
        plt.xlabel("Feature 1")
        plt.ylabel("Feature 2")
        plt.title("SVM with New Point
Prediction")
        plt.grid(True)
        plt.show().

if __name__ == "__main__":
    X = np.array([
```

```
        [1,7],
        [2,8],
        [3,8],
        [8,1],
        [9,2],
        [10,2]
    ])
    y = np.array([0,0,0,1,1,1])

    new_point = np.array([[5,5]])

    svm = SVM()
    svm.fit(X, y)
    prediction = svm.predict(new_point)[0]

    svm.visualize(X, y, new_point = new_point[0],
prediction=prediction)

    print(f"New point {new_point[0]} classified
as: {'Class 1' if prediction == 1 else 'Class 0'}")
```

Code:

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

class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        y = np.where(y <= 0, -1, 1)  # Convert labels to -1 and 1
        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):
        approx = np.dot(X, self.w) + self.b
        return np.sign(approx)

    def visualize(self, X, y, new_point=None, prediction=None):
        def get_hyperplane(x, w, b, offset):
            return (-w[0] * x + b + offset) / w[1]

        fig = plt.figure()
```

```
        ax = fig.add_subplot(1, 1, 1)

        # Plot existing data points
        for i, sample in enumerate(X):
            if y[i] == 1:
                plt.scatter(sample[0], sample[1], marker='o', 
color='blue', label='Class +1' if i == 0 else "")
            else:
                plt.scatter(sample[0], sample[1], marker='x', 
color='red', label='Class -1' if i == 0 else "")

        # Plot decision boundary
        x0 = np.linspace(np.min(X[:, 0])-1, np.max(X[:, 0])+1, 
100)
        x1 = get_hyperplane(x0, self.w, self.b, 0)
        x1_m = get_hyperplane(x0, self.w, self.b, -1)
        x1_p = get_hyperplane(x0, self.w, self.b, 1)

        ax.plot(x0, x1, 'k-', label='Decision Boundary')
        ax.plot(x0, x1_m, 'k--', label='Margins')
        ax.plot(x0, x1_p, 'k--')

        # Plot the new point
        if new_point is not None:
            color = 'green' if prediction == 1 else 'orange'
            label = f'New Point: Class {"1" if prediction == 1 
else "0"}'
            plt.scatter(new_point[0], new_point[1], c=color, 
s=100, edgecolors='black', label=label, marker='*')

        ax.legend()
        plt.xlabel("Feature 1")
        plt.ylabel("Feature 2")
        plt.title("SVM with New Point Prediction")
        plt.grid(True)
        plt.show()


# 🚀 Example usage
if __name__ == "__main__":
    # Training data
```

```
    X = np.array([
        [1, 7],
        [2, 8],
        [3, 8],
        [8, 1],
        [9, 2],
        [10, 2]
    ])
    y = np.array([0, 0, 0, 1, 1, 1])  # 0 -> -1, 1 -> +1

    # New point to classify
    new_point = np.array([[5, 5]])

    # Train and predict
    svm = SVM()
    svm.fit(X, y)
    prediction = svm.predict(new_point)[0]

    # Visualize
    svm.visualize(X, y, new_point=new_point[0], 
prediction=prediction)

    # Print prediction
    print(f"New point {new_point[0]} classified as: {'Class 1' 
if prediction == 1 else 'Class 0'}")
```

Program 8

Implement Random forest ensemble method on a given dataset

Observation:

=> RANDOM FOREST

```
import pandas as pd
from sklearn.model_selection import train_
 test_split
from sklearn.ensemble import RandomForestC
lassifier
from sklearn.metrics import accuracy_score,
    classification_report
from google.colab import files

uploaded = files.upload()

for filename in uploaded.keys():
    df=pd.read_csv(filename)
    print(f"Data loaded from:{filename}")
    display(df.head())

X = df.iloc[:, :-1]
y = df.iloc[:, -1]

x_train, X_test, y_train, y_test = train_
    test split(X, y, test_size=0.2, 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(f"Accuracy on Random Forest Model:
   {accuracy * 100:.2f}%")

print("Classification Report:")
print(classification_report(y_test, y_pred))
```

Code:

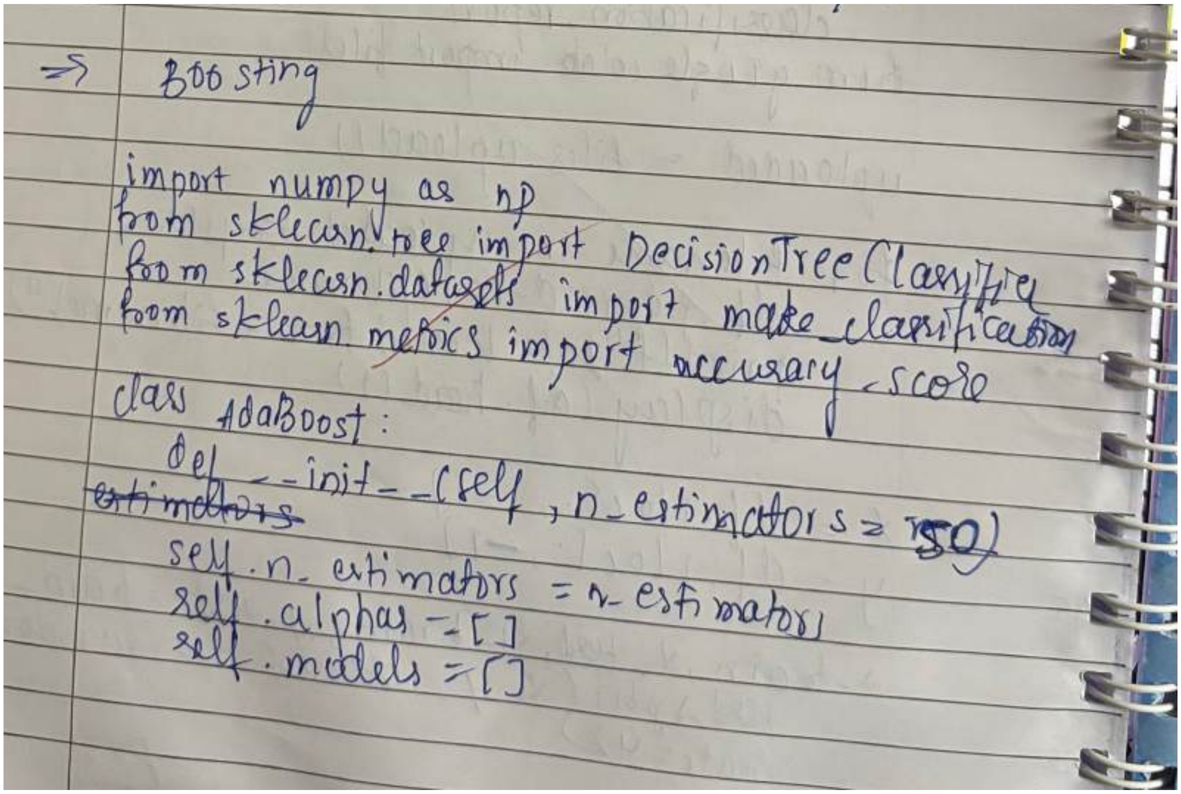
```
# STEP 1: Import required libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,
classification_report
from google.colab import files

# STEP 2: Upload your dataset
uploaded = files.upload()

# STEP 3: Load the dataset (assuming it's a CSV)
for filename in uploaded.keys():
    df = pd.read_csv(filename)
    print(f"Data loaded from: {filename}")
    display(df.head())  # Display first 5 rows of data

# STEP 4: Preprocessing
# Assume the last column is the target variable (label)
X = df.iloc[:, :-1]  # Features (all rows, all columns except
last)
y = df.iloc[:, -1]   # Target (last column)

# STEP 5: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# STEP 6: Initialize and train the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100,
random_state=42)  # 100 trees in the forest
rf_model.fit(X_train, y_train)

# STEP 7: Make predictions on the test set
y_pred = rf_model.predict(X_test)

# STEP 8: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Random Forest Model: {accuracy * 100:.2f}%")

# STEP 9: Print classification report
```

```
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

<u>**Program 9**</u>

Implement Boosting ensemble method on a given dataset

Observation:



⇒ Boosting

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
class AdaBoost:
    def __init__(self, n_estimators = 50)
        self.n_estimators = n_estimators
        self.alphas = []
        self.models = []
```

_/_/__

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    w = np.ones(n_samples) / n_samples

    for estimator in range(self.n_estimators):
        model = DecisionTreeClassifier(max_
depth = 1)
        model.fit(X, y, sample_weight=w)
        y_pred = model.predict(X)

        err = np.sum(w * (y_pred != y))
/np.sum(w)

        alpha = 0.5 * np.log((1-err)/err)
if err < 1 else 0
        self.alphas.append(alpha)
        self.models.append(model)

        w = w * np.exp(-alpha * y * y_pred)
        w = w / np.sum(w)

def predict(self, X):
    final_pred = np.zeros(X.shape[0])

    for model, alpha in zip(self.models,
self.alphas):
        final_pred += alpha * model.predict(X)
```

_/_/__

```
        return np.sign(final_pred)
```

Sneha SB
16/5/25

Code:

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

class AdaBoost:
    def __init__(self, n_estimators=50):
        self.n_estimators = n_estimators
        self.alphas = []  # Weights of each weak classifier
        self.models = []  # Weak classifiers (e.g., decision
stumps)

    def fit(self, X, y):
        # Initialize weights for each data point
        n_samples, n_features = X.shape
        w = np.ones(n_samples) / n_samples  # Equal weights
initially

        for estimator in range(self.n_estimators):
            # Train weak classifier (decision stump)
            model = DecisionTreeClassifier(max_depth=1)  #
Decision stump
            model.fit(X, y, sample_weight=w)
            y_pred = model.predict(X)

            # Calculate error rate
            err = np.sum(w * (y_pred != y)) / np.sum(w)

            # Compute alpha (weight for the classifier)
            alpha = 0.5 * np.log((1 - err) / err) if err < 1
else 0
            self.alphas.append(alpha)
            self.models.append(model)

            # Update weights for misclassified samples
            w = w * np.exp(-alpha * y * y_pred)  # Update
weights based on classifier performance
            w = w / np.sum(w)  # Normalize the weights
```

```python
    def predict(self, X):
        # Initialize the final prediction
        final_pred = np.zeros(X.shape[0])

        for model, alpha in zip(self.models, self.alphas):
            final_pred += alpha * model.predict(X)

        # Return the sign of the final prediction
        return np.sign(final_pred)

    def score(self, X, y):
        # Return accuracy of the model
        return accuracy_score(y, self.predict(X))


# Generate a synthetic binary classification dataset
X, y = make_classification(n_samples=500, n_features=20,
n_classes=2, random_state=42)

# Convert labels to -1 and 1 for AdaBoost
y = 2 * y - 1

# Create and train AdaBoost model
adaboost = AdaBoost(n_estimators=50)
adaboost.fit(X, y)

# Evaluate the model
accuracy = adaboost.score(X, y)
print(f"Model accuracy: {accuracy:.4f}")
```

Program 10

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

Observation:

⇒ K-MEANS:

```
class KMeans:
    def __init__(self, k=2, tolerance=0.001, max_
            iter=500):
        self.max_iterations = max_iter
        self.tolerance = tolerance

    def eucliclean_distance(self, point1, point2):
        return np.linalg.norm(point1 - point2, axis=0)

    def predict(self, data):
        distances = [np.linalg.norm(data - self.centroids
            [centroid]) for centroid in self.centroids]
    return classification
```

__/__/___

```
def main():
    K = 3
    Center_1 = np.array([1,1])
    center_2 = np.array([5,5])
    center_3 = np.array([8,1])

    cluster_1 = np.random.randn(100,2) + Center_1
    cluster_2 = np.random.randn(100,2) + Center_2
    cluster_3 = np.random.randn(100,2) + Center_3

    data = np.concatenate((cluster_1, cluster_2,
cluster_3), axis=0)

    K_means = K_means(K)
    K_means.fit(data

    colors = 10*["r", "g", "c", "b", "k"]

    for centroid in K_means.centroids
        plt.scatter(K_means.centroids[centroid][0],
K_means.centroids[centroid][1], s=130, marker="x")

    for cluster_index in K_means.classes:
        color = colors[cluster_index]
        for features in K_means.classes[cluster_
index]:
            plt.scatter(features[0], features[1],
color=color, s=30)
```

__/__/__

```
~~def~~
def fit(self, data):
    self.centroids = {}
    for i in range(self.k):
        self.centroids[i] = data[i]

    for i in range(self.max_iteration):
        self.classes = {}
        for j in range(self.k):
            self.classes[j] = []

        for point in data:
            distances = []
            for index in self.centroids:
                distances.append(self.euclidean_
distance(point, self.centroids[index]))
            cluster_index = distances.index(min(
distances))
            self.classes[cluster_index].append(po
int)
        previous = dict(self.centroids)
        for cluster_index in self.classes:
            self.centroids[cluster_index] =
np.average(self.classes[cluster_index], axis=0)

        isOptimal = True
        break
```

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

Code:

```
class K_Means:

    def __init__(self, k=2, tolerance = 0.001, max_iter = 500):
        self.k = k
        self.max_iterations = max_iter
        self.tolerance = tolerance

    def euclidean_distance(self, point1, point2):
        #return math.sqrt((point1[0]-point2[0])**2 + (point1[1]-
point2[1])**2 + (point1[2]-point2[2])**2)   #sqrt((x1-x2)^2 +
(y1-y2)^2)
        return np.linalg.norm(point1-point2, axis=0)

    def predict(self,data):
        distances = [np.linalg.norm(data-
self.centroids[centroid]) for centroid in self.centroids]
        classification = distances.index(min(distances))
        return classification

    def fit(self, data):
        self.centroids = {}
        for i in range(self.k):
            self.centroids[i] = data[i]


        for i in range(self.max_iterations):
            self.classes = {}
            for j in range(self.k):


                self.classes[j] = []



            for point in data:
                distances = []
                for index in self.centroids:

distances.append(self.euclidean_distance(point,self.centroids[in
dex]))
```

```
                cluster_index = distances.index(min(distances))
                self.classes[cluster_index].append(point)

            previous = dict(self.centroids)
            for cluster_index in self.classes:
                self.centroids[cluster_index] = 
np.average(self.classes[cluster_index], axis = 0)



            isOptimal = True

            for centroid in self.centroids:
                original_centroid = previous[centroid]
                curr = self.centroids[centroid]
                if np.sum((curr - original_centroid)/
original_centroid * 100.0) > self.tolerance:
                    isOptimal = False
            if isOptimal:
                break



def main():
    K=3
    center_1 = np.array([1,1])
    center_2 = np.array([5,5])
    center_3 = np.array([8,1])

    # Generate random data and center it to the three centers
    cluster_1 = np.random.randn(100, 2) + center_1
    cluster_2 = np.random.randn(100,2) + center_2
    cluster_3 = np.random.randn(100,2) + center_3

    data = np.concatenate((cluster_1, cluster_2, cluster_3), 
axis = 0)

    k_means = K_Means(K)
    k_means.fit(data)
```

```
    # Plotting starts here
    colors = 10*["r", "g", "c", "b", "k"]

    for centroid in k_means.centroids:
        plt.scatter(k_means.centroids[centroid][0],
k_means.centroids[centroid][1], s = 130, marker = "x")

    for cluster_index in k_means.classes:
        color = colors[cluster_index]
        for features in k_means.classes[cluster_index]:
            plt.scatter(features[0], features[1], color =
color,s = 30)

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

Program 11

Implement Dimensionality reduction using Principal Component Analysis (PCA) method

Observation:

21/04/25

⟹ PRINCIPAL COMPONENT ANALYSIS.

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

uploaded = files.upload()

for filename in uploaded.keys():
    df = pd.read_csv('/content/pizza.csv')
    print(f"uploaded: {'/content/pizza.csv'}")
    display(df.head())

numeric_df = df.select_dtypes(include=[np.number])
print("Numerical features found:", list(numeric_df.columns

selected_features=numeric_df.columns

X = numeric_df[selected_features].dropna()
X_scaled= StandardScaler().fit_transform(X)

pca = PCA(n_components=2)
principle_components = pca.fit_transform(X_scaled)

pca_df= pd.DataFrame(data=principle_components,
        columns=['PC1','PC2'])
```

__/__/___

```
plt.figure(figsize = (8,6))
plt.scatter(pca_df['PC1'], pca_df['PC2'], alpha=0.7)
plt.xlabel('Principal component 1')
plt.ylabel('principal component 2')
plt.title('2D PCA visualisation')
plt.grid(True)
plt.show()

print("Explained variance ratio:", pca.explained_
    variance_ratio)

print(f"Model accuracy: {accuracy:.4f}")
```

⇒ K-MEANS:

```
class KMeans:
    def __init__(self, k=2, tolerance=0.001, max_
        iter =500):
        self.max_iterations = max_iter
        self.tolerance = tolerance

    def euclidean_distance(self, point1, point2):
        return np.linalg.norm(point1-point2,axis=0)

    def predict(self,data):
        distances= [np.linalg.norm(data - self.centroids
            [centroid]) for centroid in self.centroids]
        return classification
```

Code:

```
# STEP 1: Import packages
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from google.colab import files

# STEP 2: Upload the CSV file
uploaded = files.upload()

# STEP 3: Load the dataset
for filename in uploaded.keys():
    df = pd.read_csv('/content/Pizza.csv')
    print(f"✅ Uploaded: {'/content/Pizza.csv'}")
    display(df.head())

# STEP 4: Select numerical columns
numeric_df = df.select_dtypes(include=[np.number])
print("📊 Numerical features found:", list(numeric_df.columns))

# OPTIONAL: Manually select columns if needed
# selected_features = ['feature1', 'feature2', ...]
selected_features = numeric_df.columns  # use all numeric
features for now

# STEP 5: Standardize data
X = numeric_df[selected_features].dropna()
X_scaled = StandardScaler().fit_transform(X)

# STEP 6: Apply PCA
pca = PCA(n_components=2)
principal_components = pca.fit_transform(X_scaled)

# STEP 7: Create DataFrame for components
pca_df = pd.DataFrame(data=principal_components, columns=['PC1',
'PC2'])

# STEP 8: Visualize the first two principal components
```

```
plt.figure(figsize=(8,6))
plt.scatter(pca_df['PC1'], pca_df['PC2'], alpha=0.7)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('2D PCA Visualization')
plt.grid(True)
plt.show()

# STEP 9: Explained variance ratio
print("📈 Explained Variance Ratio:",
pca.explained_variance_ratio_)




print(f"Model accuracy: {accuracy:.4f}")
```