VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum- 590014, Karnataka.

LAB REPORT

on

Machine Learning (23CS6PCMAL)

Submitted by

Pradeep P T (1BM22CS197)

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
February 2025 – June 25

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 **Pradeep P T (1BM22CS197),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Laboratory report has been approved as it satisfies the academic requirements in respect of a 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

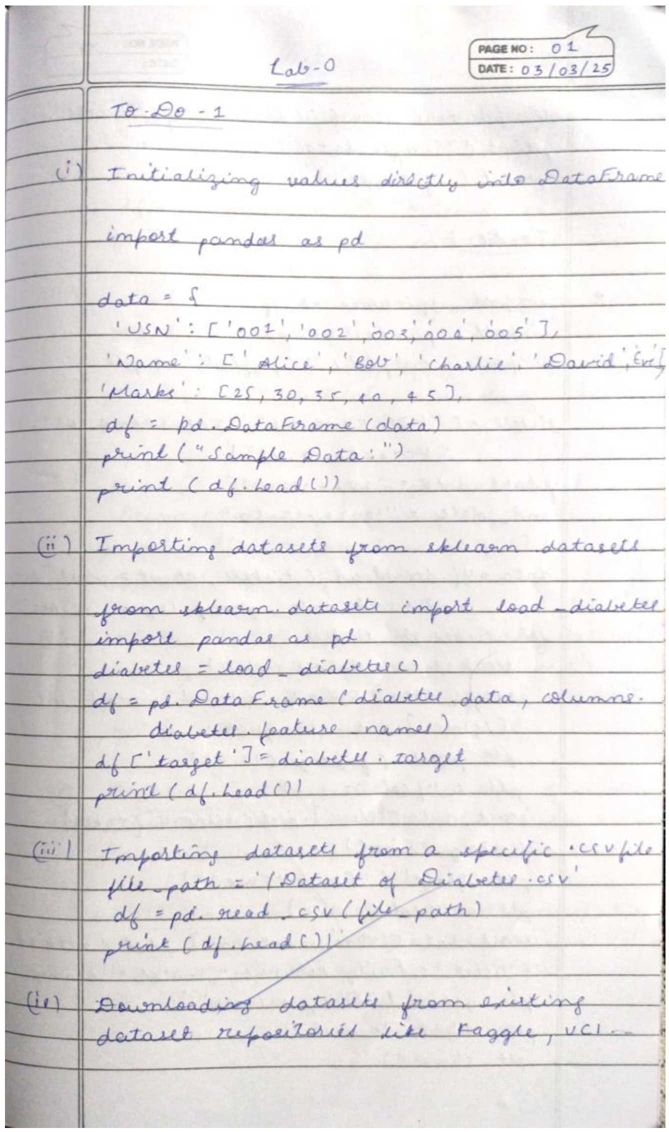
INDEX

Github Link: https://github.com/Pradeep-P-T/ML

Program 1

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

Screenshot:



PAGE NO: 01
DATE: 03/03/25

Lab-0

To-Do-1

(i) Initializing values directly into DataFrame

```
import pandas as pd

data = {
 'USN' : ['001', '002', '003', '004', '005'],
 'Name' : ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
 'Marks' : [25, 30, 35, 40, 45],
 df = pd.DataFrame (data)
print ("Sample Data:")
print ( df.head())
```

(ii) Importing datasets from sklearn datasets

```
from sklearn.datasets import load_diabetes
import pandas as pd
diabetes = load_diabetes()
df = pd.DataFrame (diabetes.data, columns=
     diabetes.feature_names)
df ['target'] = diabetes.target
print (df.head())
```

(iii) Importing datasets from a specific .csv file

```
file_path = '/Dataset of Diabetes.csv'
 df = pd.read_csv(file_path)
 print (df.head())
```

(iv) Downloading datasets from existing dataset repositories like Kaggle, UCI...

PAGE NO: 2
DATE:

```
df=pd.read_csv('/Dataset of Diabetes.csv')
print("Sample data")
print(df.head())
```

To-Do-2

```
import yfinance as yf
import pandas as pd
import matplotlib.pyplot as plt

tickers = ["HDFCBANK.NS", "ICICIBANK.NS",
           "KOTAKBANK.NS"]
start_date = "2024-01-01"
end_date = "2024-12-30"

data = yf.download(tickers, start=start_date,
        end=end_date, group_by='ticker')
for ticker in tickers:
    stock_data = data[ticker]
    stock_data = ['Daily Return'] = stock_data
    ['Close'].pct_change()
    plt.figure(figsize(12,6))
    plt.subplot(2,1,1)
    stock_data['Close'].plot(title=f"{ticker}-
    Closing Price")
    plt.ylabel("Price(INR)")
    plt.subplot(2,1,2)
    stock_data['Daily Return'].plot(title=f
    "{ticker}-Daily Returns", color='orange')
    plt.ylabel("Daily Return")
    plt.tight_layout()
    plt.show().
```

Code:

```
from sklearn.datasets import load_iris

import pandas as pd

iris = load_iris()

df = pd.DataFrame(iris.data, columns=iris.feature_names)

df.head()


df['target'] = iris.target

df

import kagglehub



# Download latest version

path = kagglehub.dataset_download("abdulmalik1518/mobiles-dataset-2025")



print("Path to dataset files:", path)



df = pd.read_csv("/content/Mobiles_Dataset_(2025).csv", encoding='latin-1') # or 'ISO-8859-1', or
'cp1252'

df.head()

df['Company Name']



data = {"USN" : ['1', "2", "3"], "Name" : ["A", "B", "C"]}

df = pd.DataFrame(data)

df
```

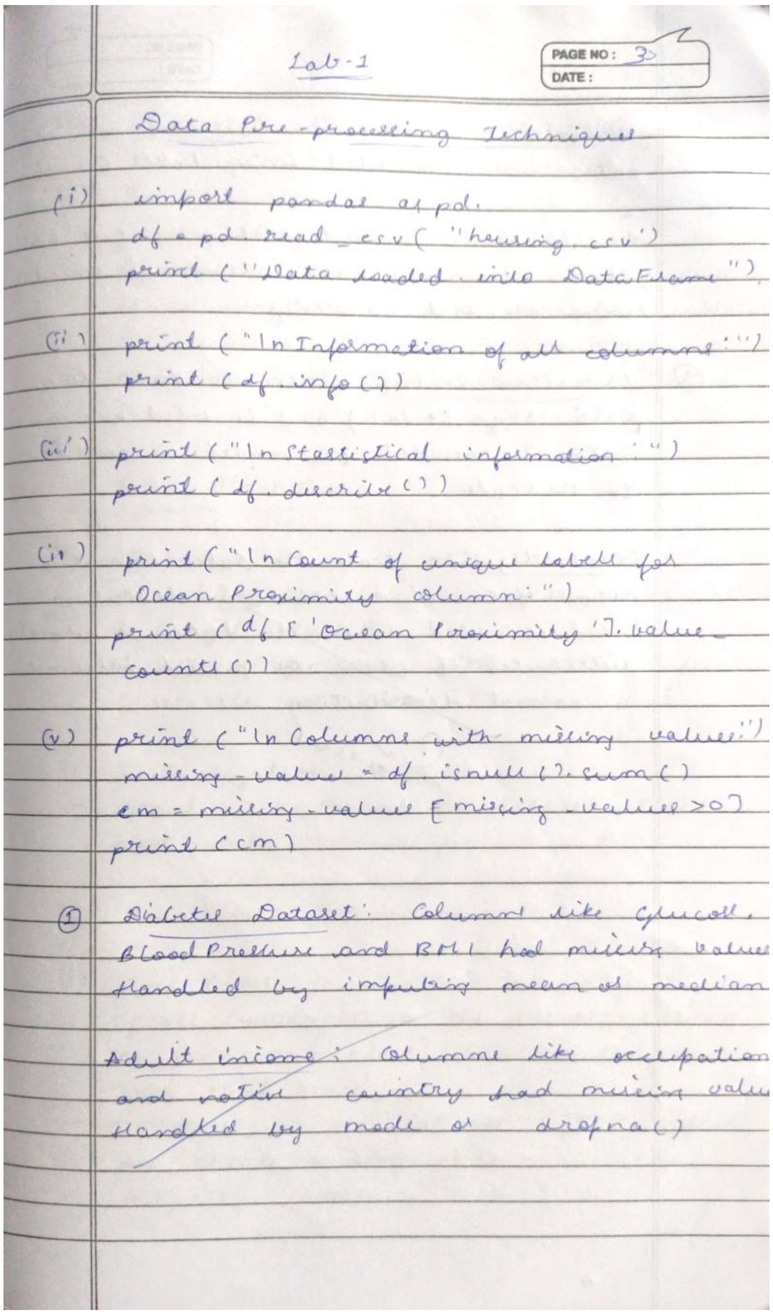
```
from sklearn.datasets import load_diabetes

diabetes = load_diabetes()

df = pd.DataFrame(diabetes.data, columns=diabetes.feature_names)

df.head()

df.columns

df = pd.read_csv("/content/Dataset_of_Diabetes .csv")

df.head()

import yfinance as yf import pandas as pd

import matplotlib.pyplot as plt



tickers = ["RELIANCE.NS", "TCS.NS", "INFY.NS"]

# Fetch historical data for the last 1 year

data = yf.download(tickers, start="2022-10-01", end="2023-10-01", group_by='ticker')


# Display the first 5 rows of the dataset

print("First 5 rows of the dataset:")


print(data.head())

print("\nShape of the dataset:")

print(data.shape)
```

```
# Summary statistics for a specific stock (e.g., Reliance)

reliance_data = data['RELIANCE.NS']

print("\nSummary statistics for Reliance Industries:")

print(reliance_data.describe())

# Calculate daily returns

reliance_data['Daily Return'] = reliance_data['Close'].pct_change()


# Plot the closing price and daily returns

plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)

reliance_data['Close'].plot(title="Reliance Industries - Closing Price")

plt.subplot(2, 1, 2)

reliance_data['Daily Return'].plot(title="Reliance Industries - Daily Returns", color='orange')
plt.tight_layout()

plt.show()
```

Program 2

Demonstrate various data pre-processing techniques for a given dataset

Screenshot:


Lab-1

PAGE NO: 3

DATE:

Data Pre-processing Techniques

(i)
```
import pandas as pd
df = pd.read_csv("housing.csv')
print ("Data loaded into DataFrame")
```

(ii)
```
print ("\n Information of all columns:")
print (df.info())
```

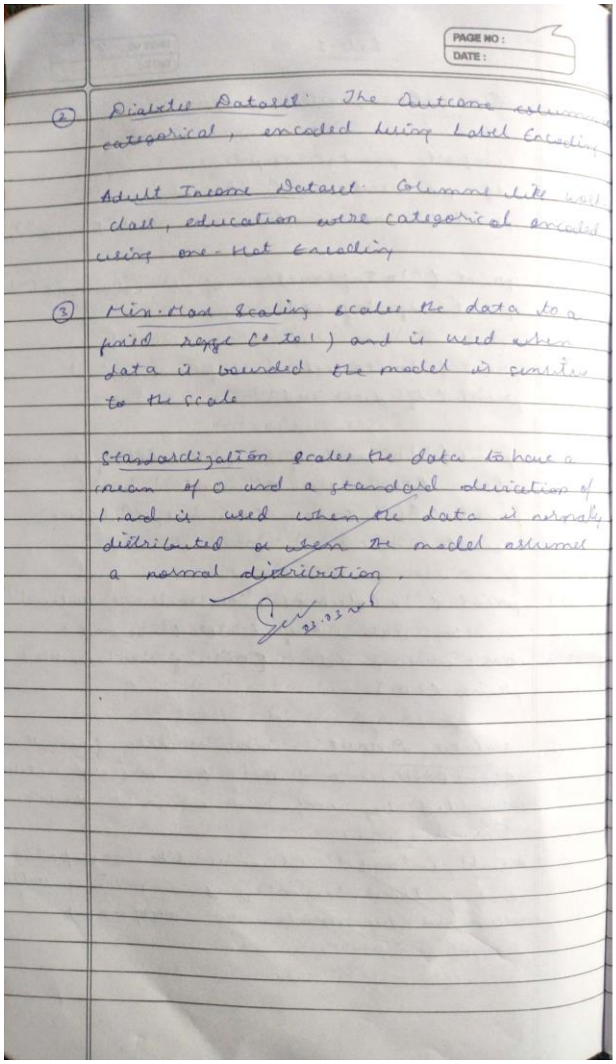
(iii)
```
print ("\n Statistical information :")
print (df.describe())
```

(iv)
```
print ("\n Count of unique labels for
 Ocean Proximity column:")
print (df ['Ocean Proximity'].value_
counts())
```

(v)
```
print ("\n Columns with missing values:")
missing_values = df.isnull().sum()
cm = missing_values [missing_values >0]
print (cm)
```

(1) Diabetes Dataset: Columns like Glucose, Blood Pressure and BMI had missing values Handled by imputing mean or median

Adult income: Columns like occupation and native country had missing values Handled by mode or dropna()



PAGE NO :
DATE :

(2) Diabetes Dataset: The Outcome column categorical, encoded using Label Encoding

Adult Income Dataset: Columns like work class, education were categorical encoded using One-Hot Encoding

(3) Min-Max Scaling scales the data to a fixed range (0 to 1) and is used when data is bounded the model is sensitive to the scale

Standardization scales the data to have a mean of 0 and a standard deviation of 1, and is used when the data is normally distributed or when the model assumes a normal distribution.

Code:

```
import pandas as pd

import numpy as np


# Load dataset

df = pd.read_csv("data.csv")

print(df.head())
```

```python
# Check missing values
print(df.isnull().sum())

# Drop rows with missing values
df_cleaned = df.dropna()

# Or fill missing values with mean/median
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].median(), inplace=True)

# For nominal categories
df = pd.get_dummies(df, columns=['Gender', 'Country'], drop_first=True)

# For ordinal categories
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder()
df[['Education_Level']] = encoder.fit_transform(df[['Education_Level']])


from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Standardization (Z-score)
scaler = StandardScaler()
```

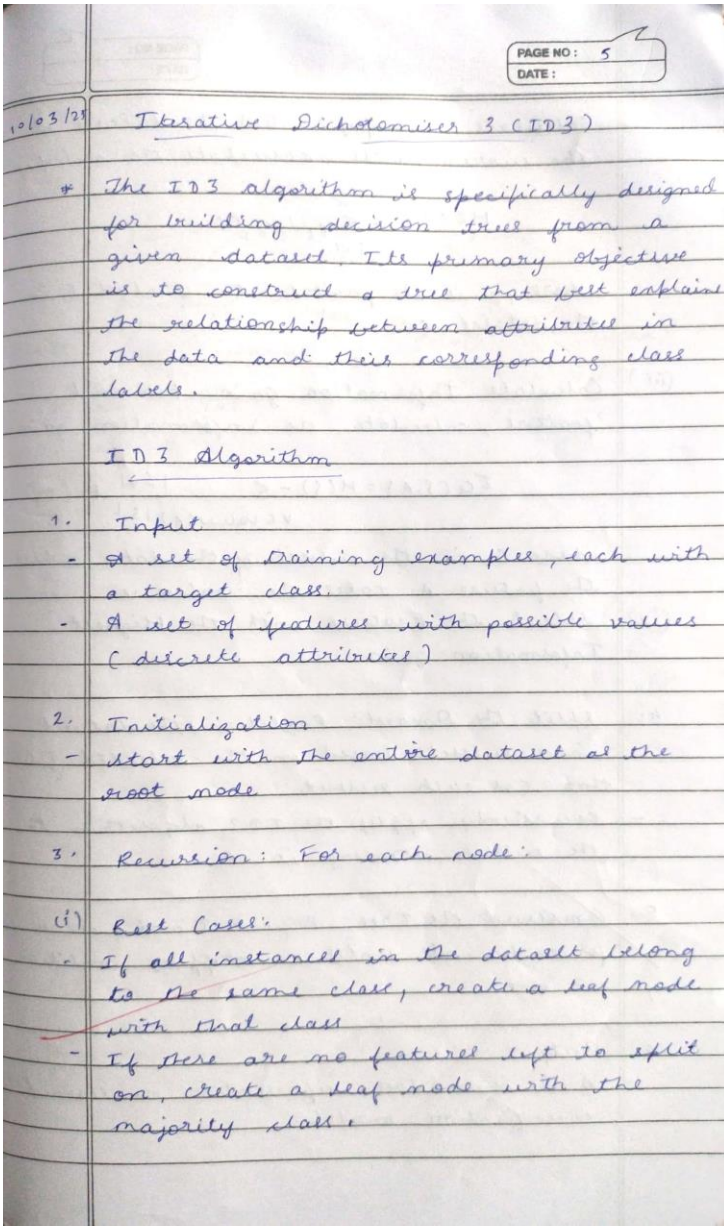
```
df[['Age', 'Salary']] = scaler.fit_transform(df[['Age', 'Salary']])


# Min-Max Normalization
minmax = MinMaxScaler()
df[['Age', 'Salary']] = minmax.fit_transform(df[['Age', 'Salary']])




# Using IQR method
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1
df = df[(df['Salary'] >= Q1 - 1.5*IQR) & (df['Salary'] <= Q3 + 1.5*IQR)]



df['Age_Salary_Ratio'] = df['Age'] / df['Salary']




# Drop irrelevant columns
df.drop(['User_ID', 'Name'], axis=1, inplace=True)


# Correlation-based filtering
correlation_matrix = df.corr()
```

```python
print(correlation_matrix)

from sklearn.model_selection import train_test_split


X = df.drop('Purchased', axis=1)

y = df['Purchased']


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

Program 3

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

Screenshot:



PAGE NO: 5
DATE:

10/03/25 Iterative Dichotomiser 3 (ID3)

* The ID3 algorithm is specifically designed for building decision trees from a given dataset. Its primary objective is to construct a tree that best explains the relationship between attributes in the data and their corresponding class labels.

ID3 Algorithm

1. Input
- A set of training examples, each with a target class.
- A set of features with possible values (discrete attributes)

2. Initialization
- start with the entire dataset as the root node

3. Recursion: For each node:

(i) Base Cases:
- If all instances in the dataset belong to the same class, create a leaf node with that class
- If there are no features left to split on, create a leaf node with the majority class.

PAGE NO :
DATE :

(ii) Calculate Entropy: For the dataset of the current node, calculate the entropy.

$$H(S) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

where p_i is the probability of class i in the dataset S.

(iii) Calculate Information Gain: For each feature, calculate the information gain

$$IG(S,A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v)$$

where S_v is the subset of the data where the feature A takes the value v.

(iv) Select the Feature with the Highest Information gain

4. Split the Dataset: Partition the dataset into subsets based on the selected feature that. For each subset:

- Recursively apply the ID3 algorithm to the subset, treating it as

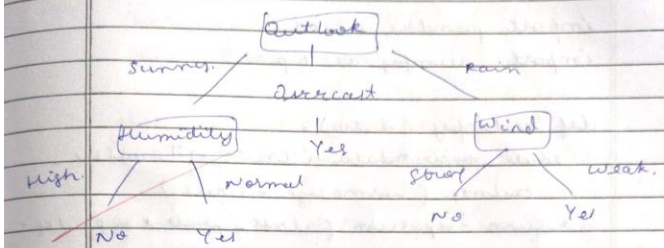
5. Construct the Tree: Repeat 3 and 4 recursively for each node until the stopping condition is met.

6. Output:

- A decision tree representing the learned classification model.

PAGE NO :
DATE :

Decision tree is:



10/3/25

PAGE NO : 8
DATE :

17|03|24 ID3 code :

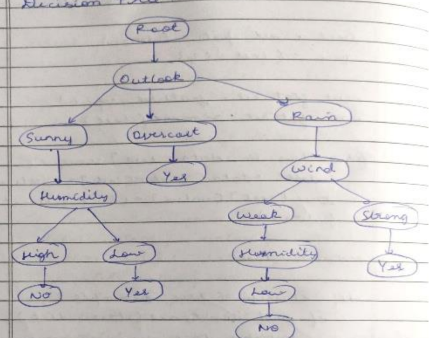
```
import pandas as pd
import numpy as np

def entropy (data):
  class_prob = data.iloc[:, -1].value_
     counts (normalize =True)
  return -np.sum (class_prob * np.log2
  (class_prob))

def information_gain (data, feature):
  Total_entropy = entropy (data)
  feature_values = data[feature].unique()
  weighted_entropy = 0
  for value in feature_values:
    subset = data[data[feature] == value]
    weighted_entropy += (len(subset)/
    len(data)) * entropy (subset)
  return total_entropy - weighted_entropy

def best_feature (data):
  features = data.columns[:-1]
  gains = {feature : information_gain(
   data, feature) for feature in features}
  return max(gains, key = gains.get)

def id3 (data, features = None):
  if len(data.iloc[:, -1].unique()) == 1
    return data.iloc[:, -1].iloc[0]

  if len(features) == 0 :
```

PAGE NO :
DATE :

```
return data.iloc[:,1].mode()[0]

best = best_feature(data)
tree = {best: {}}

new_features = features.copy()
new_features.remove(best)

for value in data[best].unique
    subset = data[data[best] == value]
    tree[best][value] = id3(subset, new_
    features)

return tree

def classify(tree, example):
    if not isinstance(tree, dict):
        return tree
    feature = list(tree.keys())[0]
    value = example[feature]
    return classify(tree[feature][value], examp

data = pd.read_csv("/3-dataset.csv")

tree = id3(data, features = list(data.colum
print("Decision tree:", tree)                [:-1]))

example = {'outlook': 'Sunny', 'Temperature':
'cool', 'Humidity': 'Low', 'Wind': 'Strong'}
prediction = classify(tree, example)
print("Prediction for the example:",
        prediction)
```

PAGE NO :
DATE :

Decision tree:



Prediction for the example: no

Code:

```
import pandas as pd

import numpy as np

from graphviz import Digraph


# Calculate Entropy

def entropy(data):

    class_probabilities = data.iloc[:, -1].value_counts(normalize=True)

    return -np.sum(class_probabilities * np.log2(class_probabilities))


# Calculate Information Gain

def information_gain(data, feature):

    total_entropy = entropy(data)

    feature_values = data[feature].unique()

    weighted_entropy = 0

    for value in feature_values:

        subset = data[data[feature] == value]

        weighted_entropy += (len(subset) / len(data)) * entropy(subset)

    return total_entropy - weighted_entropy


# Find the best feature to split the data

def best_feature(data):

    features = data.columns[:-1]  # Exclude the target column

    gains = {feature: information_gain(data, feature) for feature in features}
```

```
    return max(gains, key=gains.get)


# Create the decision tree
def id3(data, features=None):
    if len(data.iloc[:, -1].unique()) == 1:  # All data points belong to the same class
        return data.iloc[:, -1].iloc[0]

    if len(features) == 0:  # No more features to split on
        return data.iloc[:, -1].mode()[0]

    best = best_feature(data)
    tree = {best: {}}

    new_features = features.copy()
    new_features.remove(best)

    for value in data[best].unique():
        subset = data[data[best] == value]
        tree[best][value] = id3(subset, new_features)

    return tree


# Function to classify new examples based on the decision tree
def classify(tree, example):
```

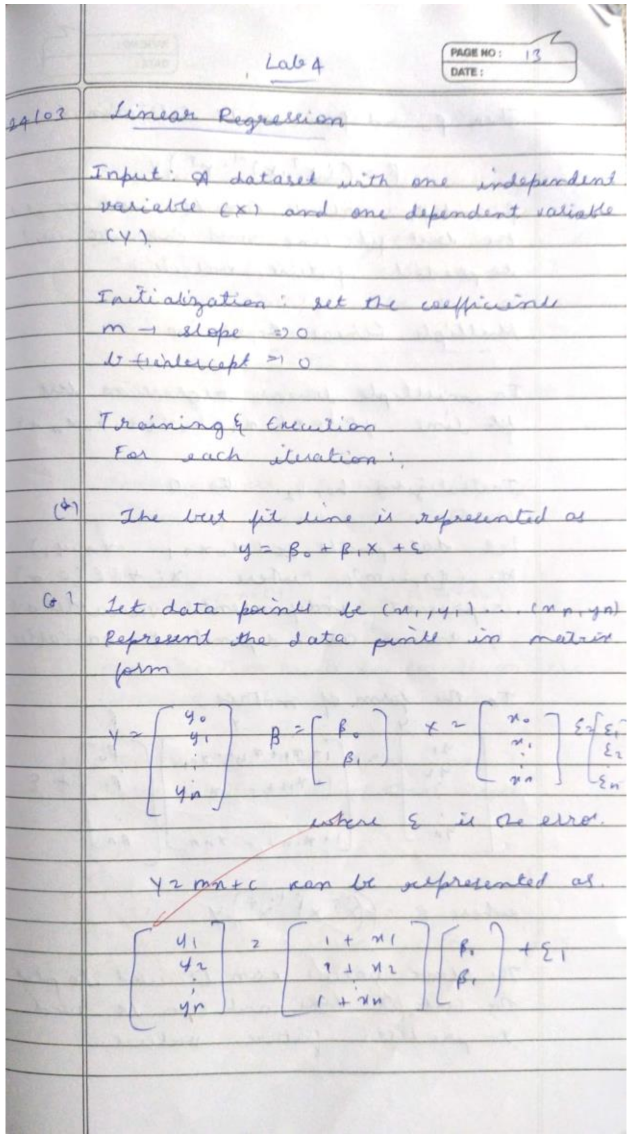
```
    if not isinstance(tree, dict):
        return tree
    feature = list(tree.keys())[0]
    value = example[feature]
    return classify(tree[feature][value], example)

# Function to visualize the decision tree using Graphviz
def create_tree_diagram(tree, dot=None, parent_name="Root", parent_value=""):
    if dot is None:
        dot = Digraph(format="png", engine="dot")

    if isinstance(tree, dict):  # Tree node
        for feature, branches in tree.items():
            feature_name = f"{parent_name}_{feature}"
            dot.node(feature_name, feature)
            dot.edge(parent_name, feature_name, label=parent_value)

            for value, subtree in branches.items():
                value_name = f"{feature_name}_{value}"
                dot.node(value_name, f"{feature}: {value}")
                dot.edge(feature_name, value_name, label=str(value))
                # Recurse for each subtree
                create_tree_diagram(subtree, dot, value_name, str(value))
    else:  # Leaf node
```

```
        dot.node(parent_name + "_class", f"Class: {tree}")

        dot.ede(parent_name, parent_name + "_class", label="Leaf")

    return dot

# Example usage

data = pd.DataFrame({

    'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain', 'Overcast', 'Sunny', 'Sunny', 'Rain',
'Sunny', 'Overcast', 'Overcast', 'Rain'],

    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot',
'Mild'],

    'Humidity': ['High', 'High', 'High', 'High', 'High', 'Low', 'Low', 'High', 'Low', 'Low', 'Low', 'High', 'Low',
'High'],

    'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Weak', 'Strong', 'Weak', 'Weak', 'Strong', 'Strong',
'Weak', 'Strong', 'Weak'],

    'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']

})

# Train the decision tree

tree = id3(data, features=list(data.columns[:-1]))

print("Decision Tree:", tree)

# Classify a new example

example = {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'Low', 'Wind': 'Strong'}

prediction = classify(tree, example)

print("Prediction for the example:", prediction)



# Visualize the decision tree

dot = create_tree_diagram(tree)

dot.render("decision_tree", view=True)  # This will generate and open the tree diagram
```

Program 4

Implement Linear and Multi-Linear Regression algorithm for appropriate dataset

Screenshot:



PAGE NO: 13
DATE:

Lab 4

24/03 Linear Regression

Input: A dataset with one independent variable (X) and one dependent variable (Y).

Initialization: set the coefficients
m → slope ⇒ 0
b → intercept ⇒ 0

Training & Execution
For each iteration:

(i) The best fit line is represented as

$$y = \beta_0 + \beta_1 X + \varepsilon$$

(ii) Let data points be $(x_1, y_1) \ldots (x_n, y_n)$
Represent the data points in matrix form

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} \quad X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

where ε is the error.

$Y = mx + c$ can be represented as

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 + x_1 \\ 1 + x_2 \\ \vdots \\ 1 + x_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \varepsilon_1$$

PAGE NO :
DATE :

Then β_0 and β_1 can be determined by

$$\beta = ((X^T \cdot X)^{-1} \cdot X^T) Y$$

β_0 and β_1 values can be used to plot the best fit line and can be used to predict future values.

<u>Multiple Linear Regression</u>

In multiple linear regression best fit line $y = b_0 + b_1 x_1 + b_2 x_2 + \cdots b_n x_n + \epsilon$

Initializing $b_0, b_2, \ldots$ to 0

Let datapoints be $(x_1, x_2, \ldots x_n, y_i)$ $\forall i \in \{0, \ldots m\}$ where $x_i \; \forall i \in \{0, n\}$ represents independent variables and y values are dependent variables.

In the form of matrix,

$$\begin{array}{ccccc} Y & & X & \beta & \\ \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} & = & \begin{bmatrix} 1 + x_{11} + x_{21} + \cdots x_{n1} \\ 1 + x_{12} + \cdots x_{n2} \\ \vdots \\ 1 + x_{1n} + \cdots x_{nn} \end{bmatrix} & \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} & + \epsilon \end{array}$$

where $\beta = ((X^T X)^{-1} \cdot X^T) Y$

The above values can be used to plot the best fit line and can be used to predict future values.

Code:

Linear Regression

```
import pandas as pd

df = pd.read_csv("/content/tvmarketing.csv")

df



# Visualise the relationship between the features and the response using scatterplots

df.plot(x='TV',y='Sales',kind='scatter')



from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(df['TV'], df['Sales'], test_size=0.2, random_state=42)

from sklearn.linear_model import LinearRegression model = LinearRegression()
model.fit(x_train.values.reshape(-1, 1), y_train) y_train
model.coef_

model.intercept_
```

MultiLinearRegression

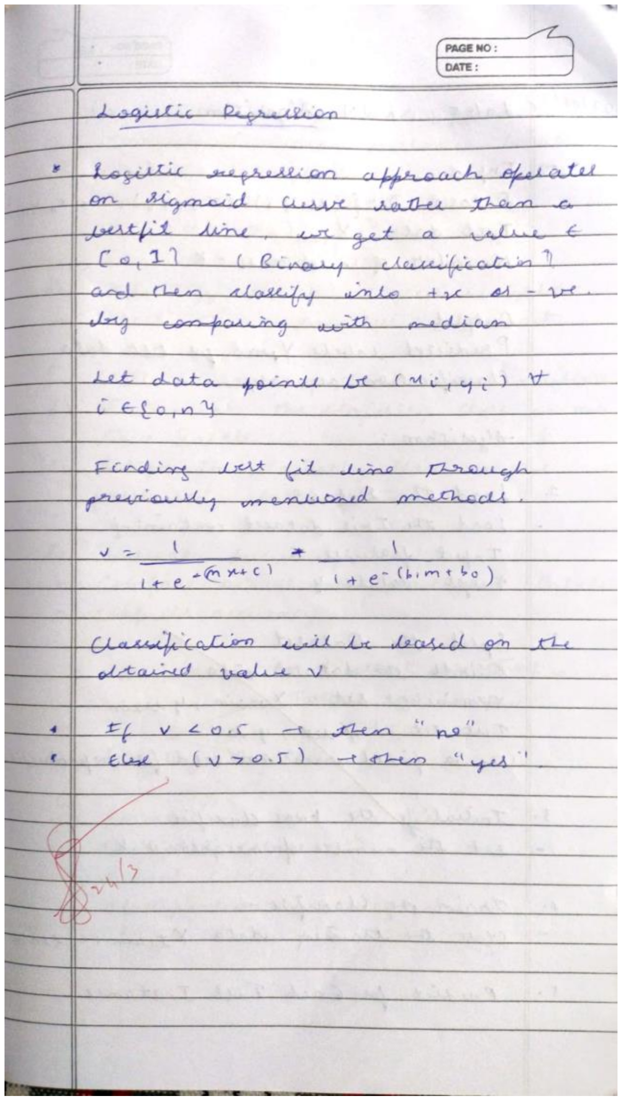
```
import pandas as pd

# Step 2 : import data

house = pd.read_csv('https://github.com/YBIFoundation/Dataset/raw/main/Boston.csv')



# display first 5 rows
```

```
house.head()

y = house['MEDV']

X = house.drop(['MEDV'],axis=1)

# Step 4 : train test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, train_size=0.7, random_state=2529)
# Step 5 : select model
from sklearn.linear_model import LinearRegression
model = LinearRegression()
# Step 6 : train or fit model
model.fit(X_train,y_train)
model.intercept_

model.coef_
```

Program 5

Build Logistic Regression Model for a given dataset

Screenshot:



PAGE NO :
DATE :

Logistic Regression

* Logistic regression approach operates on sigmoid curve rather than a bestfit line. we get a value $\in [0, 1]$ (Binary classification) and then classify into +ve or -ve. by comparing with median

Let data points be (x_i, y_i) $\forall$ $i \in \{0, n\}$

Finding best fit line through previously mentioned methods.

$$v = \frac{1}{1+e^{-(mx+c)}} * \frac{1}{1+e^{-(b_1 m + b_0)}}$$

Classification will be based on the obtained value v

* If $v < 0.5$ → then "no"
* Else $(v > 0.5)$ → then "yes".

24/3

Code:

```
from sklearn.linear_model import LogisticRegression

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Load sample dataset (binary classification - Iris with only 2 classes)

iris = load_iris()

X = iris.data[iris.target != 2]

y = iris.target[iris.target != 2]


# Train/Test split

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


# Logistic Regression model

model = LogisticRegression()

model.fit(X_train, y_train)


# Predict and evaluate

y_pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
```

Program 6

Build KNN Classification model for a given dataset

Screenshot:

PAGE NO : 16
DATE :

07/04/25 Lab 5: KNN Algorithm

→ Input:
Dataset $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
Test data X_{test}
Number of neighbours : K

→ Output:
Predicted labels Y_{pred} for test data
Classification accuracy.

Algorithm:

1. Load the Dataset
- Load the Iris dataset containing
  Input features X
  Target labels y

2. Split the Dataset
- Divide the dataset into:
  Training set: X train, y train
  Test set: X test, y test
- Use a fixed random seed for reproducibility

3. Initialize the KNN classifier
- set the number of neighbours K

4. Train the Classifier.
- Store the training data X train, y train

5. Predict for Each Test Instance

For each test sample $x \in X_{test}$:

a. Compute Euclidean Distance to all training samples:

$$d(x, x_i) = \sqrt{\sum_{j=1}^{n} (x_j - x_{ij})^2}$$

b. Identify the k Nearest Neighbors.

- Select the k smallest distances

c. Extract labels of the k nearest neighbors.

d. Determine the Majority class among these labels.

e. Assign the Majority class as the predicted label for x.

6. Evaluate Accuracy.

- Compare actual and predicted labels
- Compute accuracy

$$\text{Accuracy} = \frac{\text{No of correct predictions}}{\text{Total test samples}} \times 100$$

7. Display Results

- Print:

Predicted labels

Actual labels

Classification accuracy.

Code:

KNN

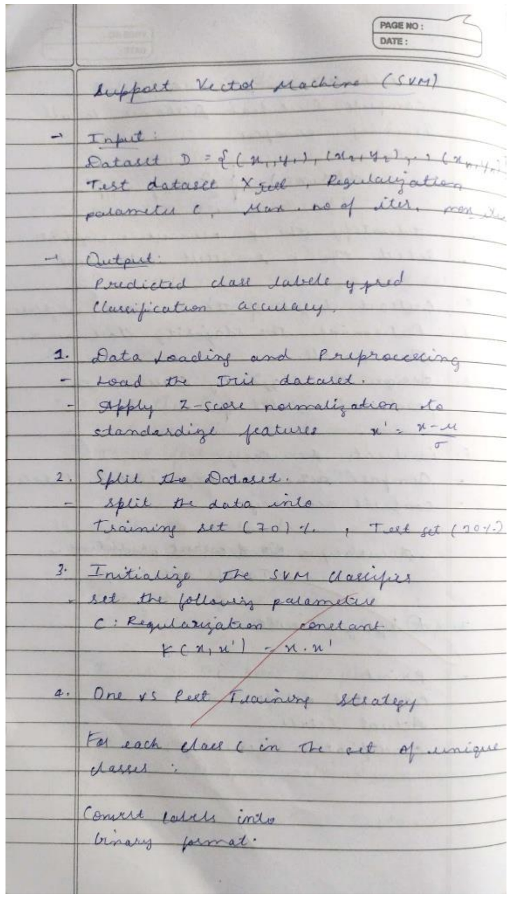
```
import numpy as np

from collections import Counter


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 euclidean_distance(self, x1, x2):

        return np.sqrt(np.sum((x1 - x2) ** 2))


    def predict(self, X):

        predictions = [self._predict(x) for x in X]

        return np.array(predictions)


    def _predict(self, x):

        # Compute distances to all training points

        distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]


        # Get indices of k nearest neighbors
```

```
        k_indices = np.argsort(distances)[:self.k]

        # Get the labels of those neighbors
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        # Return the most common label
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

# Sample dataset (like a mini version of Iris)
X_train = [[1, 2], [2, 3], [3, 1], [6, 5], [7, 7], [8, 6]]
y_train = [0, 0, 0, 1, 1, 1]

# Test data
X_test = [[5, 5], [1, 1]]

# Using the KNN modelh
knn = KNN(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)

print("Predictions:", predictions)
```

Program 7

Build Support vector machine model for a given dataset

Screenshot:



PAGE NO :
DATE :

Support Vector Machine (SVM)

→ Input:
Dataset $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
Test dataset X_{test}, Regularization parameter C, Max. no of iter. max_iter

→ Output:
Predicted class labels y_pred
Classification accuracy.

1. Data Loading and Preprocessing
- Load the Iris dataset.
- Apply Z-score normalization to standardize features $x' = \frac{x - \mu}{\sigma}$

2. Split the Dataset.
- split the data into Training set (70)%, Test set (30%)

3. Initialize the SVM classifier
- set the following parameters
C : Regularization constant
$K(x, x') = x \cdot x'$

4. One vs Rest Training Strategy

For each class c in the set of unique classes :

Convert labels into binary format.

PAGE NO :
DATE :

$$y_{binary} = \begin{cases} 1 & \text{if } y = c \\ -1 & \text{otherwise} \end{cases}$$

Train a binary SVM classifier using the simplified SMO algorithm.

5. Binary SVM Training

Initialize:
$\alpha = \vec{0}$: Lagrange multipliers
$b = 0$: Bias term.

Repeat for max iter iterations:
For each training sample i:
(i) Randomly select another index $j \neq i$
(ii) Compute prediction errors E_i & E_j
(iii) Save old values α_i, α_j
(iv) Compute bounds L, H

if L = H, continue

(v) Compute
$$\eta = 2K(x_i, x_j) - K(x_i, x_i) - K(x_j, x_j)$$
If $\eta \geq 0$, skip update.

vi) Update α_j:
$$\alpha_j = \alpha_j + \frac{y_j(E_i - E_j)}{\eta}$$

vii) Clip $\alpha_j \in [L, H]$.
viii) Update α_j

(ix) Compute:

PAGE NO :
DATE :

$b_2 = b - E_i - y_i(\alpha_i - \alpha_i^{old}) K(x_i, x_i) - y_j(\alpha_j - \alpha_j^{old}) K(x_i, x_j)$, Compute b_2

4) Update the bias term b
Check for convergence. $\|\alpha - \alpha_{prev}\| < 10^{-5}$

6. Prediction Phase.
- For each test sample x:
- For each trained binary classifier
Compute decision score:

$$f(x) = \sum_j a_j y_j K(x, x_j) + b$$

Store the score., predict the class with the maximum decision score.

7. Evaluation
- Compare predicted labels. y pred with true labels y test.

- Calculate accuracy.

$$\text{Accuracy} = \frac{\text{No of correct predictions}}{\text{Total test samples}} \times 100$$

Code:

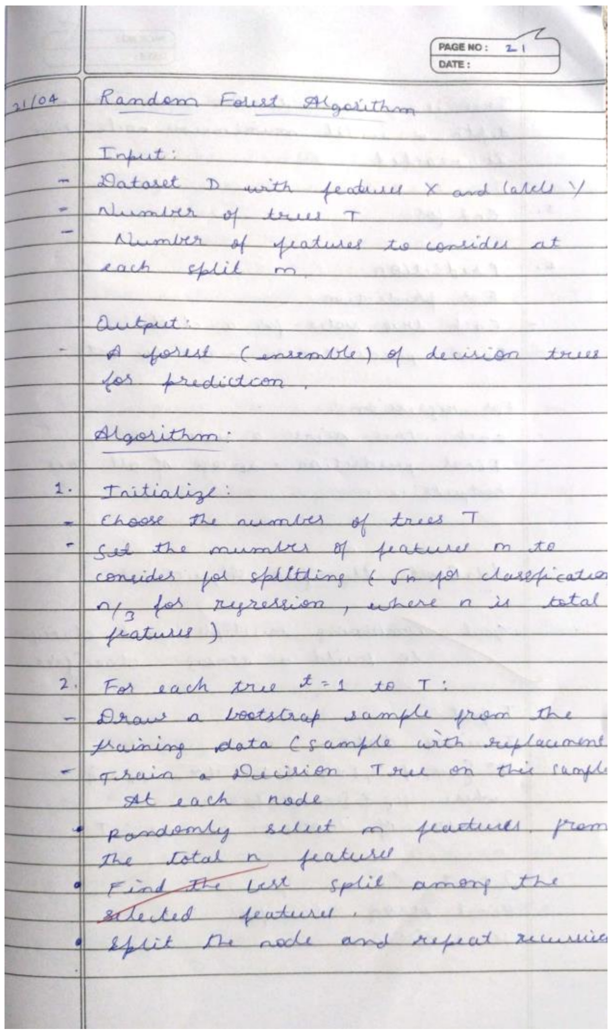
```
from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

import matplotlib.pyplot as plt
```

```
from sklearn.decomposition import PCA


# Load dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target


# For binary classification (class 0 vs 1)
X = X[y != 2]
y = y[y != 2]


# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)


# Train SVM
clf = SVC(kernel='linear')  # Try 'rbf', 'poly', etc.
clf.fit(X_train, y_train)


# Accuracy
print("Test Accuracy:", clf.score(X_test, y_test))
```

Program 8

Implement Random forest ensemble method on a given dataset

Screenshot:



PAGE NO: 21
DATE:

21/04 Random Forest Algorithm

Input:
- Dataset D with features X and labels Y
- Number of trees T
- Number of features to consider at each split m.

Output:
- A forest (ensemble) of decision trees for prediction.

Algorithm:

1. Initialize:
- Choose the number of trees T
- Set the number of features m to consider for splitting ($\sqrt{n}$ for classification $n/3$ for regression, where n is total features)

2. For each tree t = 1 to T:
- Draw a bootstrap sample from the training data (sample with replacement
- Train a Decision Tree on this sample
  At each node
  • Randomly select m features from the total n features
  • Find the best split among the selected features.
  • Split the node and repeat recursive

PAGE NO :
DATE :

Tree is grown to the maximum depth or until minimum node size is reached

3. End for

4. Prediction :
For prediction
- Each tree votes for a class
- Final prediction = majority vote.

For regression :
- each tree gives a value
- Final prediction = average of all tree outputs.

AdaBoost Classifier Algorithm

Goal : combining multiple weak classifier to build a strong classifier

Input :
Training data
$D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
where $y_i \in \{-1, +1\}$
Numbers of boosting rounds : T

Output :
Final strong classifier :
$H(x)$ ~~s/s~~

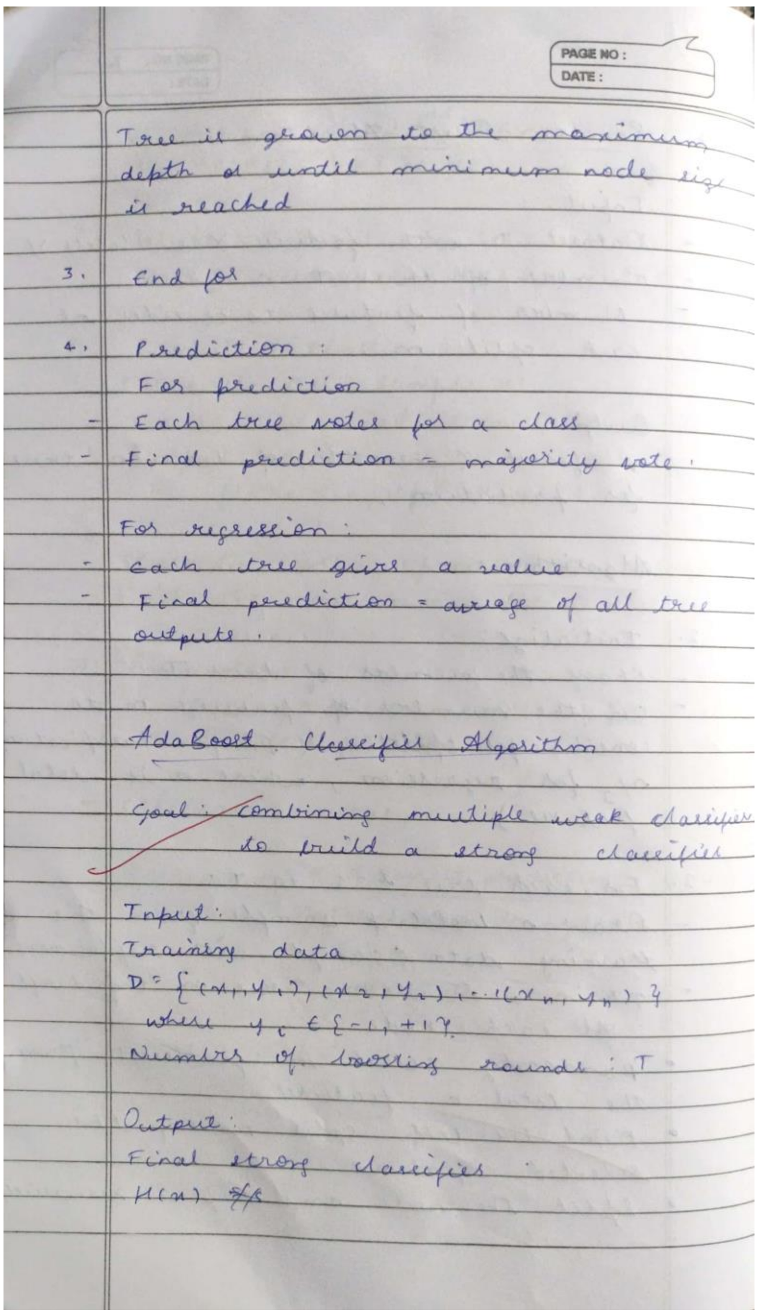
Code:

```
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

# Load sample dataset

iris = load_iris()

X, y = iris.data, iris.target


# Train/test split

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


# Initialize Random Forest

rf = RandomForestClassifier(n_estimators=100, random_state=42)

rf.fit(X_train, y_train)


# Predict and evaluate

y_pred = rf.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
```

Program 9

Implement Boosting ensemble method on a given dataset

Screenshot:



PAGE NO :
DATE :

Tree is grown to the maximum depth or until minimum node size is reached

3. End for

4. Prediction :
For prediction
- Each tree votes for a class
- Final prediction = majority vote.

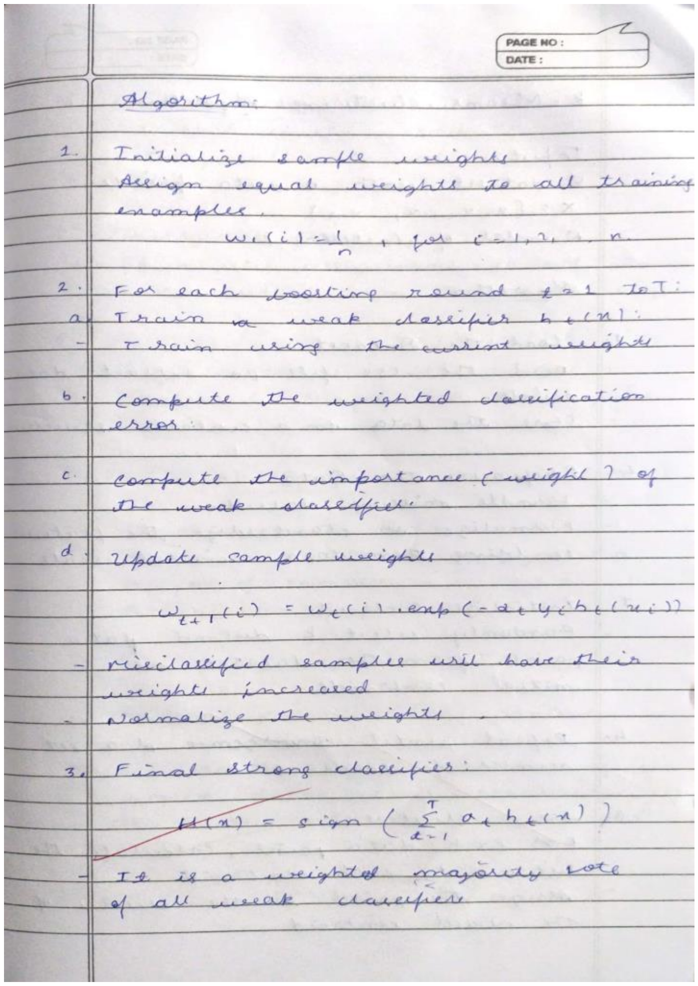
For regression :
- Each tree gives a value
- Final prediction = average of all tree outputs.

AdaBoost Classifier Algorithm

Goal : combining multiple weak classifiers to build a strong classifier

Input :
Training data
$D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
where $y_i \in \{-1, +1\}$
Number of boosting rounds : T

Output :
Final strong classifier
$H(x)$



PAGE NO :
DATE :

Algorithm:

1. Initialize sample weights
Assign equal weights to all training examples.

$$w_1(i) = \frac{1}{n}, \text{ for } i = 1, 2, \ldots, n.$$

2. For each boosting round $t = 1$ to T:

a. Train a weak classifier $h_t(x)$:
- Train using the current weights

b. Compute the weighted classification error

c. Compute the importance (weight) of the weak classifier.

d. Update sample weights

$$w_{t+1}(i) = w_t(i) \cdot \exp(-\alpha_t y_i h_t(x_i))$$

- misclassified samples will have their weights increased
- Normalize the weights.

3. Final strong classifier:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

- It is a weighted majority vote of all weak classifiers

Code:

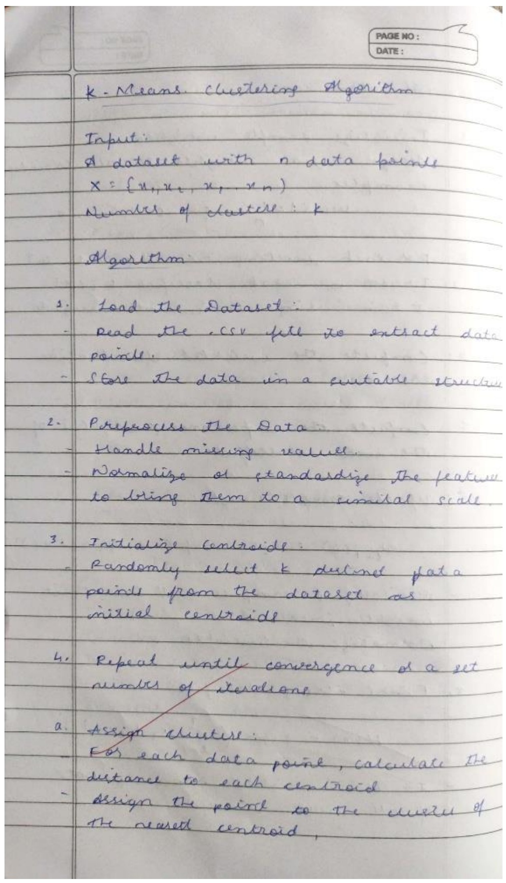
```
from sklearn.ensemble import AdaBoostClassifier

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Load Iris dataset

iris = load_iris()

X, y = iris.data, iris.target
```

```
# For AdaBoost, we'll use binary classification
# Convert to binary (setosa vs. not-setosa)
y = (y == 0).astype(int)

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

# Train AdaBoost
model = AdaBoostClassifier(n_estimators=50, learning_rate=1.0, random_state=42)
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
print("AdaBoost Accuracy (sklearn):", accuracy_score(y_test, y_pred))
```

Program 10

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

Screenshot:



PAGE NO :
DATE :

K-Means Clustering Algorithm

Input:
A dataset with n data points
$X = \{x_1, x_2, x, \dots x_n)$
Number of clusters : k

Algorithm

1. Load the Dataset:
- Read the .csv file to extract data points.
- Store the data in a suitable structure

2. Preprocess the Data
- Handle missing values.
- Normalize or standardize the features to bring them to a similar scale.

3. Initialize Centroids:
- Randomly select k distinct data points from the dataset as initial centroids

4. Repeat until convergence or a set number of iterations

a. Assign clusters:
- For each data point, calculate the distance to each centroid
- Assign the point to the cluster of the nearest centroid,

PAGE NO :
DATE :

4. Update centroids:
- For each cluster, compute the mean of the points assigned to it.
- Update the centroid with this mean

5. Convergence check:
- If cluster assignments do not change or centroids remain the same, stop

6. Output:
- Final centroids of all clusters
- Cluster labels for each data point.

<u>Principal Component Analysis (PCA)</u>

1. Standardize the Data
- Given a dataset X of size $n \times d$,
$n \rightarrow$ no of samples.
$d \rightarrow$ number of features,

$$X_{centered} = X - \mu.$$

2. Compute the covariance Matrix:
- Calculate the covariance matrix of the centered data. This matrix captures the relationship between different features

$$C = \frac{1}{n-1} X^T_{centered} X_{centered}.$$

Code:

```
import pandas as pd

from sklearn.cluster import KMeans
```

```
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris # Import load_iris


# Step 1: Load the Iris dataset directly

iris = load_iris()

# Create a DataFrame from the data and target

data = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Add the target column for potential reference, though not used for clustering

data['target'] = iris.target




# Step 2: Extract only numeric columns (or select required features)

# All features in the Iris dataset are numeric

X = data[iris.feature_names].values # Use the feature names to select columns


# Step 3: Apply KMeans

# Adjust n_clusters based on the expected number of clusters in your data (3 for Iris)

kmeans = KMeans(n_clusters=3, random_state=42, n_init=10) # Added n_init to suppress future
warnings

data['Cluster'] = kmeans.fit_predict(X)


# Step 4: Plot clusters (for 2D data)

# Iris data has 4 features. We will plot the first two features for visualization.

if X.shape[1] >= 2:
```

```python
    plt.scatter(X[:, 0], X[:, 1], c=data['Cluster'], cmap='viridis')
    plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], color='red', marker='x', s=200)
    plt.title("K-Means Clustering of Iris Dataset")
    plt.xlabel(iris.feature_names[0]) # Label with actual feature name
    plt.ylabel(iris.feature_names[1]) # Label with actual feature name
    plt.show()
else:
    print("Cannot plot clustering results directly for data with less than 2 features.")
```

Program 11

Implement Dimensionality reduction using Principal Component Analysis (PCA) method

Screenshot:

PAGE NO :
DATE :

4. Update Centroids:
- For each cluster, compute the mean of the points assigned to it.
- Update the centroid with this mean

5. Convergence check:
- If cluster assignments do not change or centroids remain the same, stop

6. Output:
- Final centroids of all clusters
- Cluster labels for each data point.

Principal Component Analysis (PCA)

1. Standardize the Data:
- Given a dataset X of size $n \times d$,
  n → no of samples.
  d → number of features.

$$X_{centered} = X - \mu$$

2. Compute the covariance Matrix:

- Calculate the covariance matrix of the centered data. This matrix captures the relationship between different features

$$C = \frac{1}{n-1} X^T_{centered} X_{centered}$$

PAGE NO :
DATE :

3. Compute the Eigenvalues & Eigenvectors
- Find the eigenvalues and eigenvectors
Eigenvectors (directions of the new feature space)

Eigenvalues (variance)

4. Sort Eigenvalues and Eigenvectors:
- Sort the eigenvalues in descending order.

$$\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d.$$

5. Select the Top K Eigenvectors.
- Choose the top K eigenvectors corresponding to the K largest eigenvalues. These eigenvectors form a new basis for the reduced feature space.

6. Construction The Projection Matrix:
- Form a matrix W

$$W = [v_1, v_2, v_3 \ldots v_k]$$

7. Project the Data:
- Multiply the centered data $X_{centered}$ by the projection matrix W to obtain reduced dataset in the new K-dimensional space

21/4/25

$$X_{reduced} = X_{centered} W$$

Code:

```
import pandas as pd

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler
```

```python
import matplotlib.pyplot as plt


# Load dataset
data = pd.read_csv("your_data.csv")  # Replace with your file
X = data.select_dtypes(include=['float64', 'int64'])


# Step 1: Standardize
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# Step 2: Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)


# Print explained variance ratio
print("Explained variance ratio:", pca.explained_variance_ratio_)


# Visualize
plt.scatter(X_pca[:, 0], X_pca[:, 1], c='blue', alpha=0.5)
plt.title("PCA - 2D Projection")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```

```
📋 Accuracy Before PCA:
Logistic Regression: 0.9016
SVM: 0.8525
Random Forest: 0.8361

📉 Accuracy After PCA (n_components=5):
Logistic Regression: 0.8689
SVM: 0.8689
Random Forest: 0.8852
```