VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Machine Learning (23CS6PCMAL)

Submitted by

PRIYANSHU KUMAR (1BM22CS210)

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 **Priyanshu Kumar (1BM22CS210),** who is a 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 Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	 **Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/pkcs210/6thSem-ML-Lab/tree/main

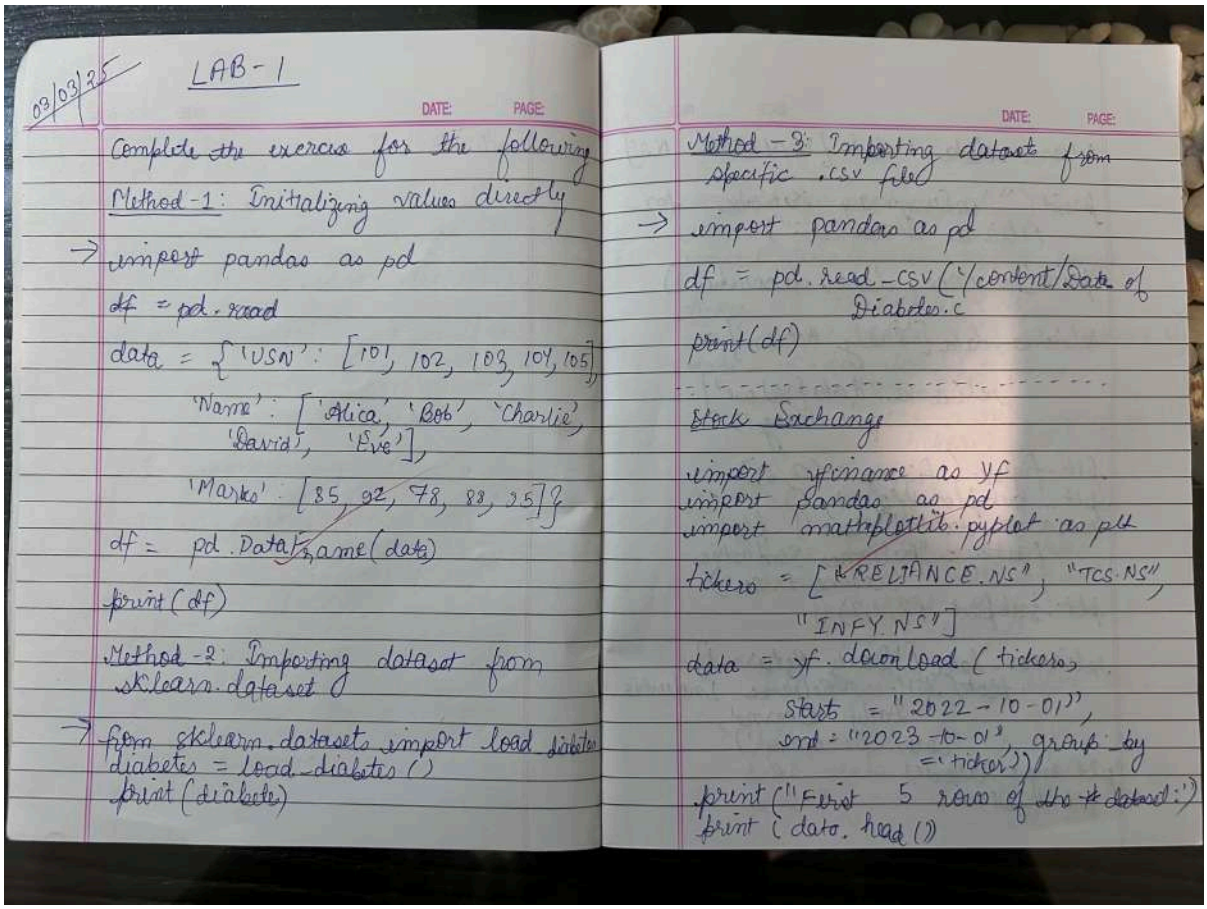
▼ LAB 1: Using Pandas

Code

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

# Assuming 'data' is a DataFrame with stock data
reliance_data = data["RELIANCE.NS"]

print("Summary Statistics for Reliance Industries:")
print(reliance_data.describe())

# Calculate daily returns
reliance_data["Daily Return"] = reliance_data["Close"].pct_change()

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

# Plot Closing Price
plt.subplot(2, 1, 1)
reliance_data["Close"].plot(title="Reliance Industries Closing Price")

# Plot Daily Return
plt.subplot(2, 1, 2)
reliance_data["Daily Return"].plot(
    title="Reliance Industries Daily Return", color="orange"
)

plt.tight_layout()
plt.show()



import pandas as pd
import yfinance as yf
```

```
import matplotlib.pyplot as plt

tickers = ["HDFCBANK.NS", "ICICIBANK.NS", "KOTAKBANK.NS"]
data = yf.download(
    tickers,
    start="2021-01-01",
    end="2021-12-30",
    group_by="ticker"
)

for ticker in tickers:
    try:
        bank_data = data[ticker]
        bank_data["Daily Return"] = bank_data["Close"].pct_change()

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

        # Plot Closing Price
        plt.subplot(2, 1, 1)
        bank_data["Close"].plot(title=f"{ticker} - Closing Price")

        # Plot Daily Return
        plt.subplot(2, 1, 2)
        bank_data["Daily Return"].plot(
            title=f"{ticker} - Daily Return", color="orange"
        )

        plt.tight_layout()
        plt.show()
    except KeyError:
        print(f"Data not found for {ticker}")
```

Record Book



03/03/25 LAB-1

Complete the exercise for the following

Method-1: Initializing values directly

→ import pandas as pd

df = pd.read

data = {'USN': [101, 102, 103, 104, 105],
'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
'Marks': [85, 92, 78, 88, 95]}

df = pd.DataFrame(data)

print(df)

Method-2: Importing dataset from sklearn.dataset ()

→ from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
print(diabetes)

Method-3: Importing dataset from specific .csv file

→ import pandas as pd

df = pd.read_csv('/content/Data of Diabetes.c

print(df)

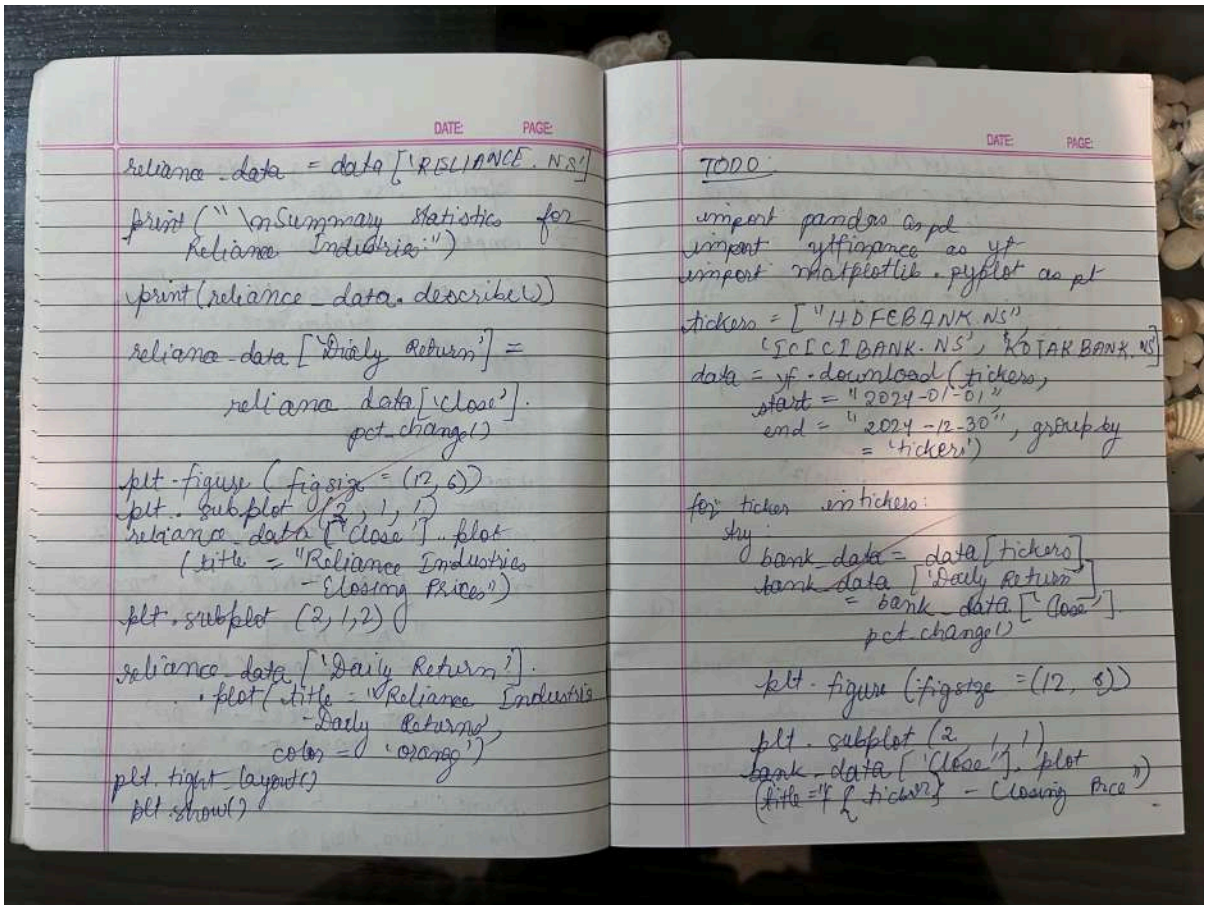
Stock Exchange

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

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

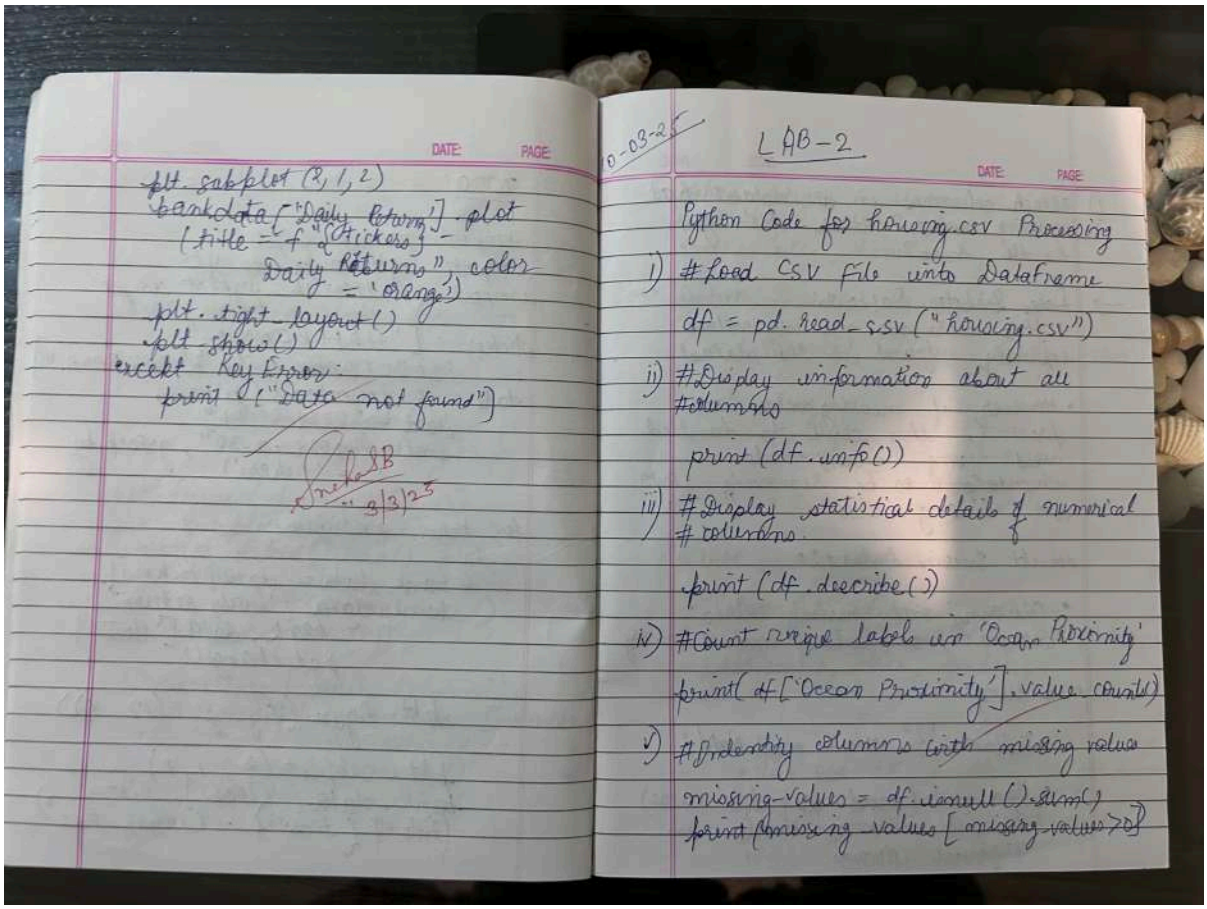
data = yf.download(tickers, start = "2022-10-01", end = "2023-10-01", group_by = 'ticker')

print("First 5 rows of the dataset:")
print(data.head())



```
reliance_data = data['RELIANCE.NS']
print("\nSummary Statistics for Reliance Industries:")
print(reliance_data.describe())
reliance_data['Daily Return'] =
    reliance_data['Close'].
        pct_change()
plt.figure(figsize=(12,6))
plt.subplot(2,1,1)
reliance_data['Close'].plot
(title="Reliance Industries
    -Closing Prices")
plt.subplot(2,1,2)
reliance_data['Daily Return'].
    .plot(title="Reliance Industries
        -Daily Returns",
        color='orange')
plt.tight_layout()
plt.show()
```

```
TODO:
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
tickers = ["HDFCBANK.NS"
    'ICICIBANK.NS', 'KOTAKBANK.NS']
data = yf.download(tickers,
    start="2024-01-01"
    end="2024-12-30", groupby
        ='tickers')

for ticker in tickers:
    try:
        bank_data = data[tickers]
        bank_data['Daily Return']
            = bank_data['Close'].
                pct_change()

        plt.figure(figsize=(12,8))

        plt.subplot(2,1,1)
        bank_data['Close'].plot
        (title=f"{ticker} - Closing Price")
```



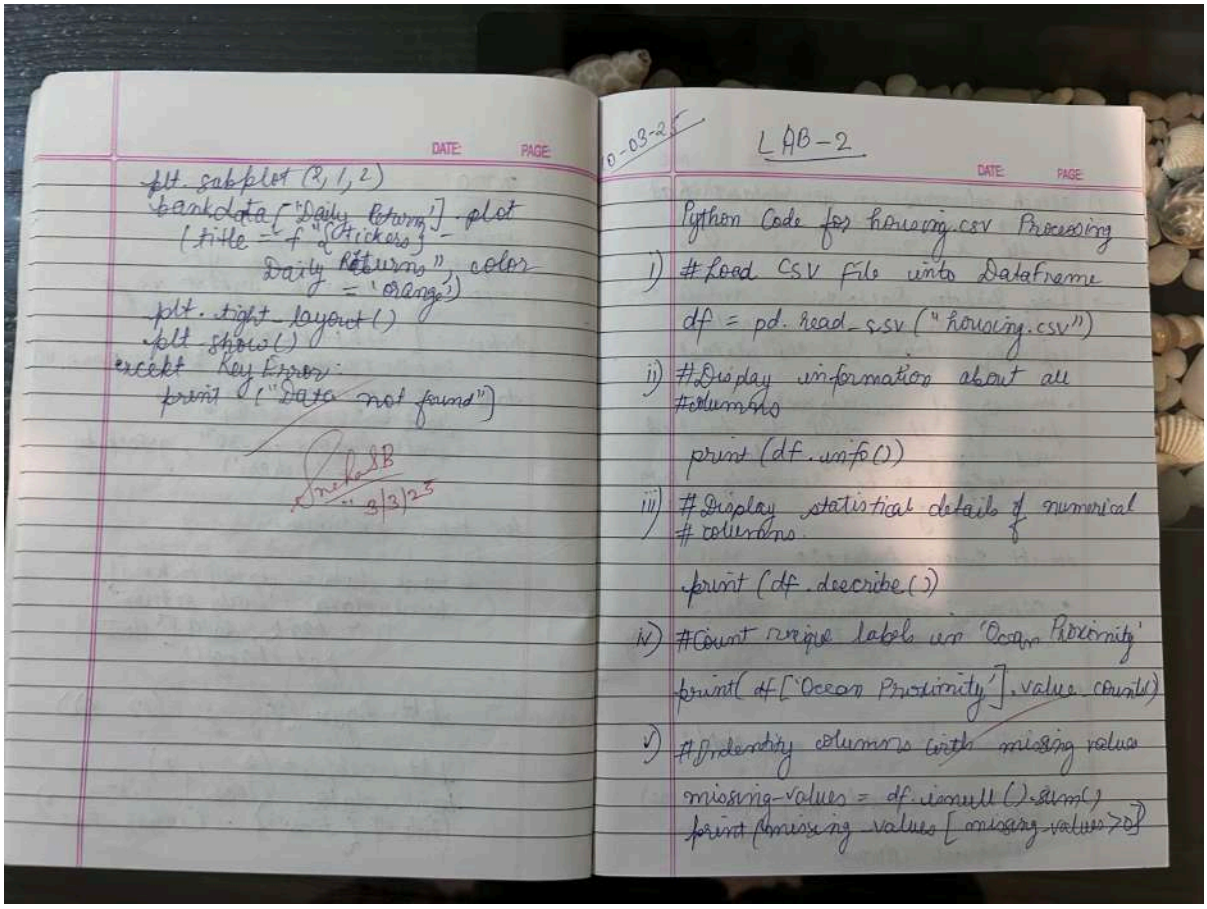
▼ LAB 2: Data Preprocessing

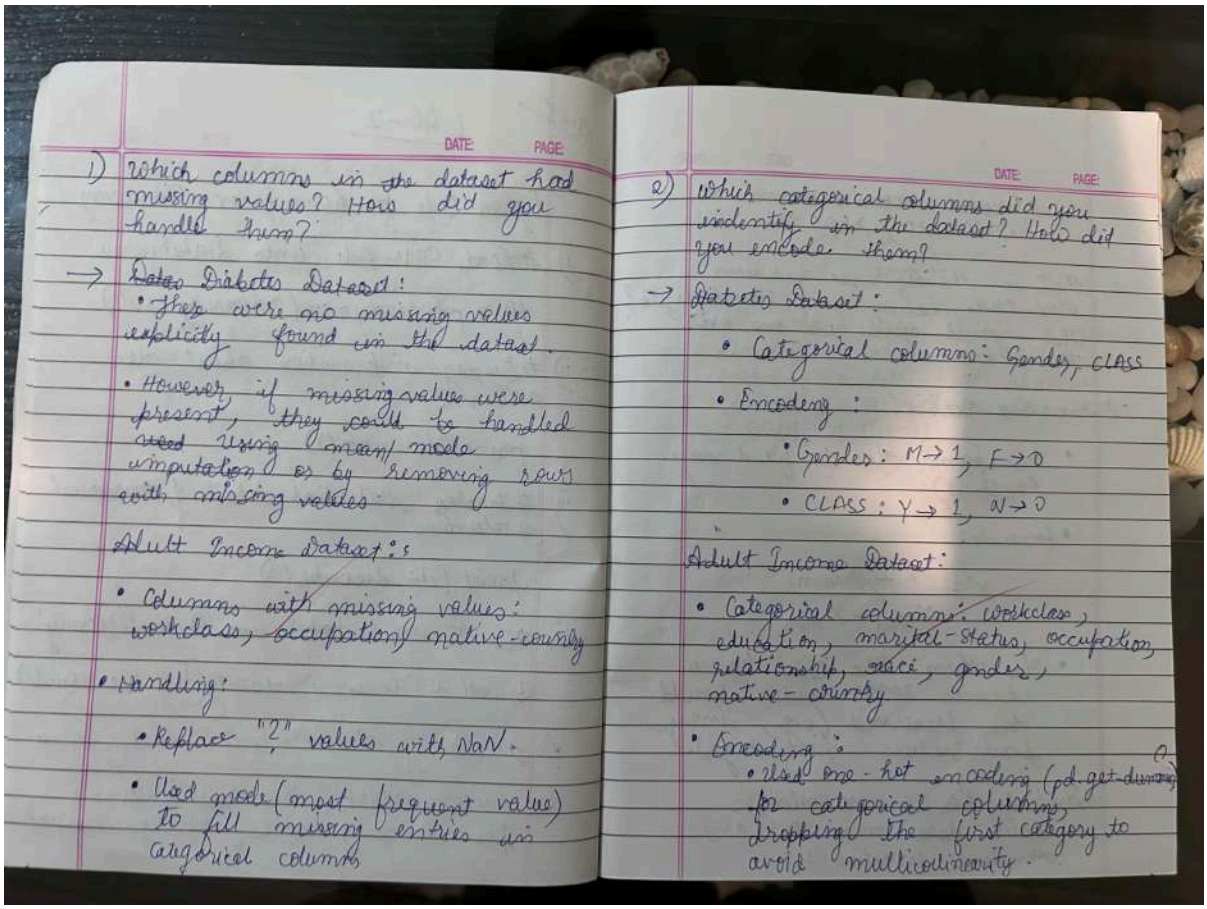
Code

```
import pandas as pd

# i) Read CSV File into DataFrame
df = pd.read_csv("housing.csv")

# ii) Display information about all columns
print(df.info())

# iii) Display statistical details of numerical columns
print(df.describe())

# iv) Count unique labels in 'Ocean Proximity'
print(df["Ocean Proximity"].value_counts())
```

```
# v) Find columns with missing values
missing_values = df.isnull().sum()
print("Missing values:\n", missing_values[missing_values > 0])
```

Record Book





1) Which columns in the dataset had missing values? How did you handle them?

→ ~~Data~~ Diabetes Dataset:

- There were no missing values explicitly found in the dataset.
- However, if missing values were present, they could be handled ~~used~~ using mean/mode imputation or by removing rows with missing values.

Adult Income Dataset:

- Columns with missing values: workclass, occupation, native-country
- Handling:
  - Replace "?" values with NaN.
  - Used mode (most frequent value) to fill missing entries in categorical columns

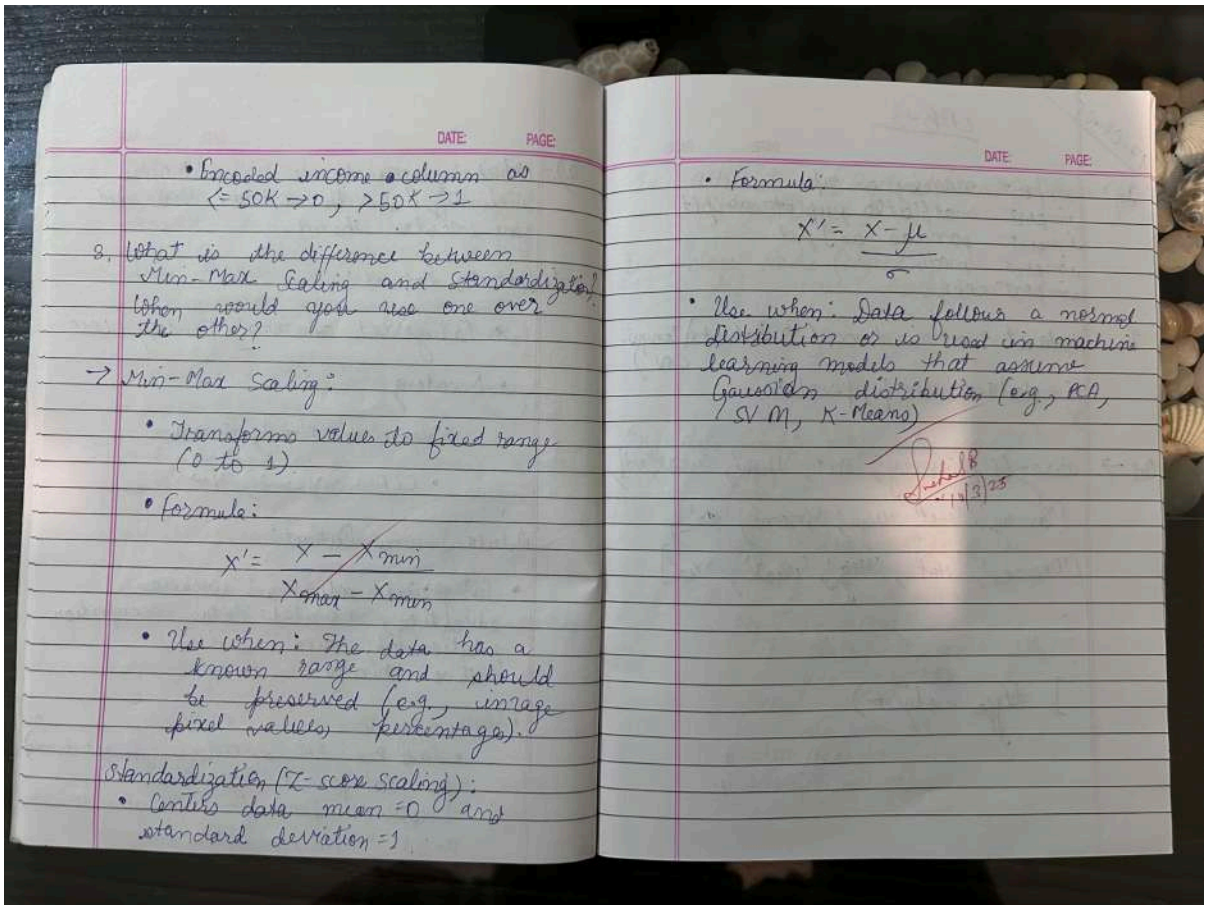
2) Which categorical columns did you identify in the dataset? How did you encode them?

→ Diabetes Dataset:

- Categorical columns: Gender, CLASS
- Encoding:
  - Gender: M → 1, F → 0
  - CLASS: Y → 1, N → 0

Adult Income Dataset:

- Categorical columns: workclass, education, marital-status, occupation, relationship, race, gender, native-country
- Encoding:
  - Used one-hot encoding (pd.get_dummies) for categorical columns, dropping the first category to avoid multicollinearity.



▼ LAB 3: ID3

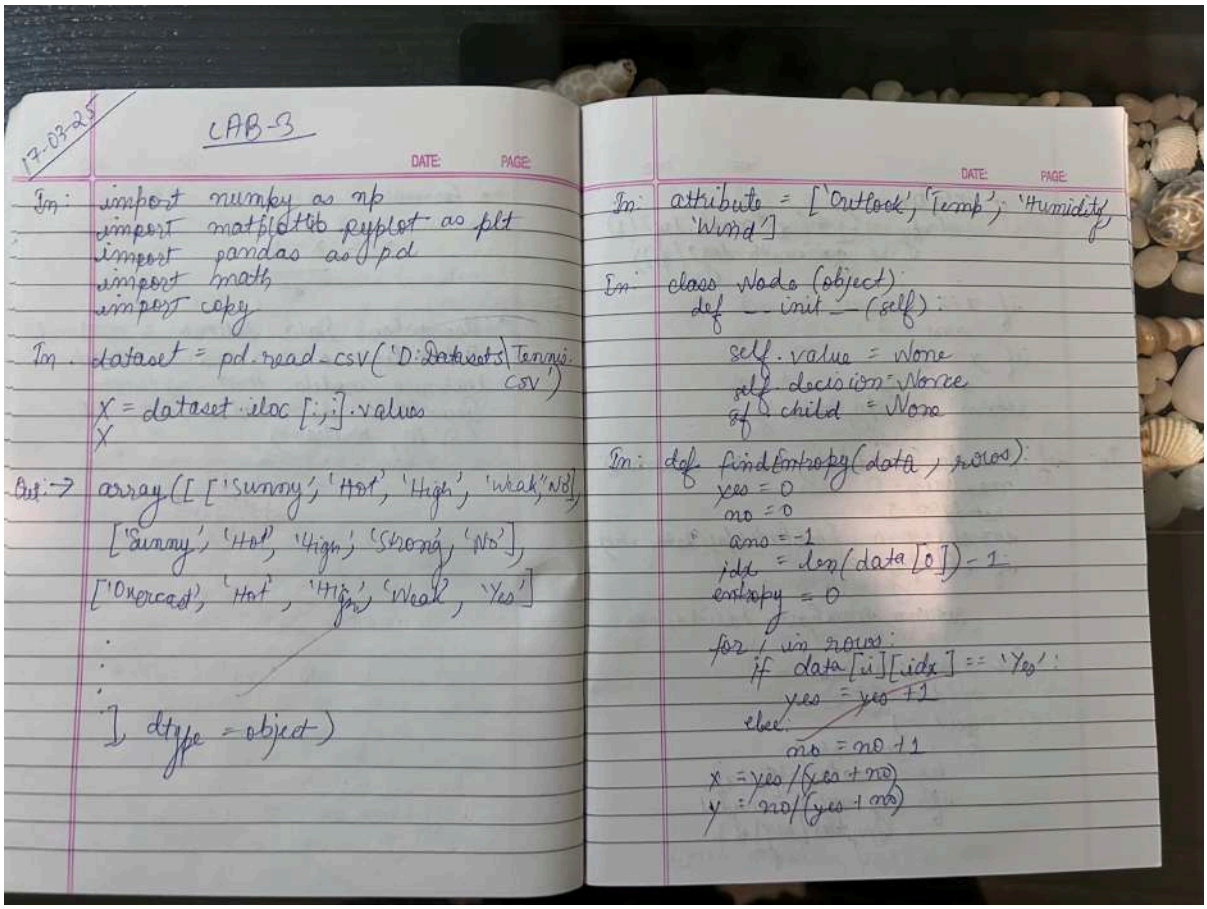
Code

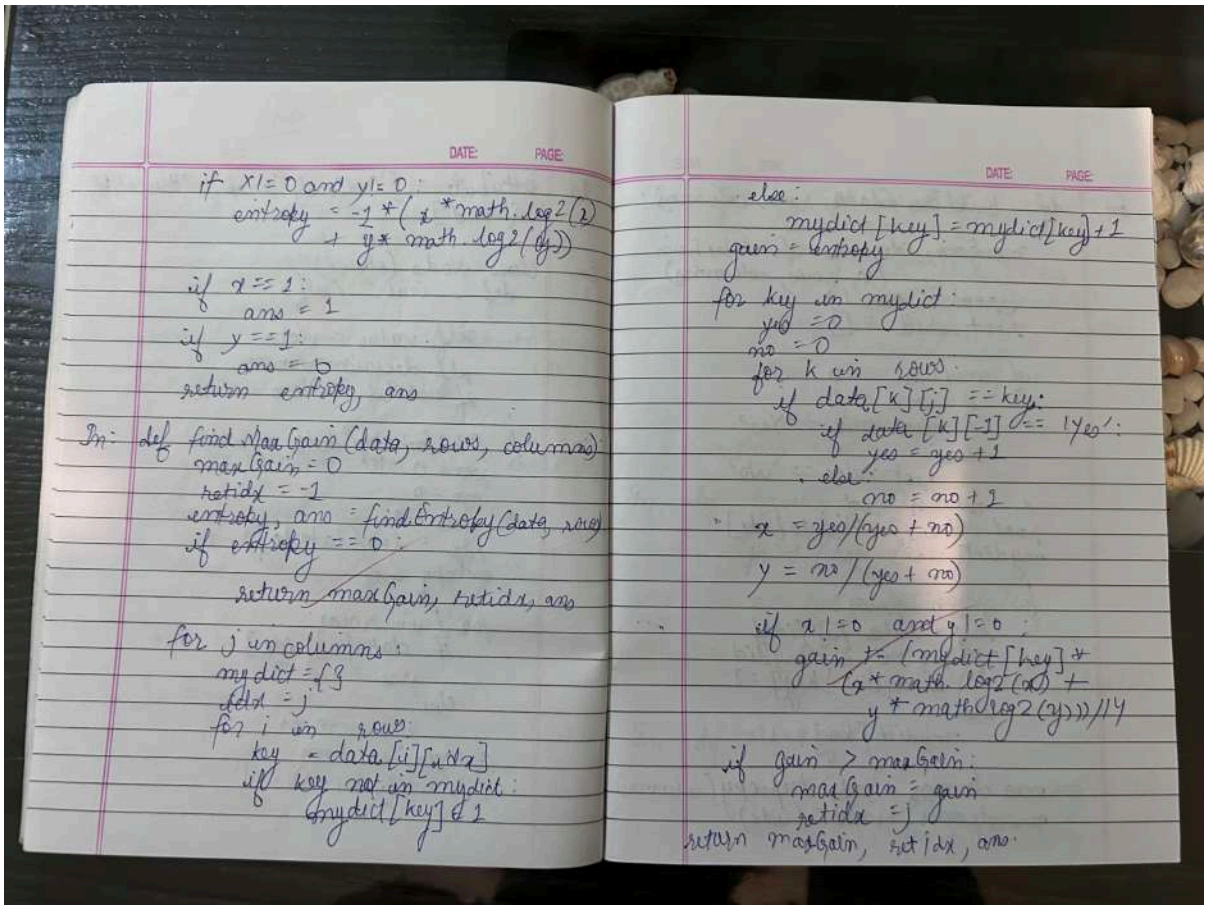
```
import math

def find_entropy(x, y):
    if x != 0 and y != 0:
        entropy = -1 * (x * math.log2(x) + y * math.log2(y))
        return entropy
    if x == 1:
        return 1
    if y == 1:
        return 0
    return 0

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

```
    max_gain = 0
    retidx = -1
    entropy_ans = find_entropy(data, rows)
    if entropy_ans == 0:
        return max_gain, retidx, entropy_ans

    for j in columns:
        mydict = {}
        ddd = 1
        for i in rows:
            key = data[i][j]
            if key not in mydict:
                mydict[key] = 1
            else:
                mydict[key] += 1

        gain = entropy_ans
        for key in mydict:
            yes = 0
            no = 0
            for k in rows:
                if data[k][j] == key:
                    if data[k][-1] == 1:  # Assuming last column is the label
                        yes += 1
                    else:
                        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  # 1

        if gain > max_gain:
            max_gain = gain
            retidx = j

    return max_gain, retidx, entropy_ans
```

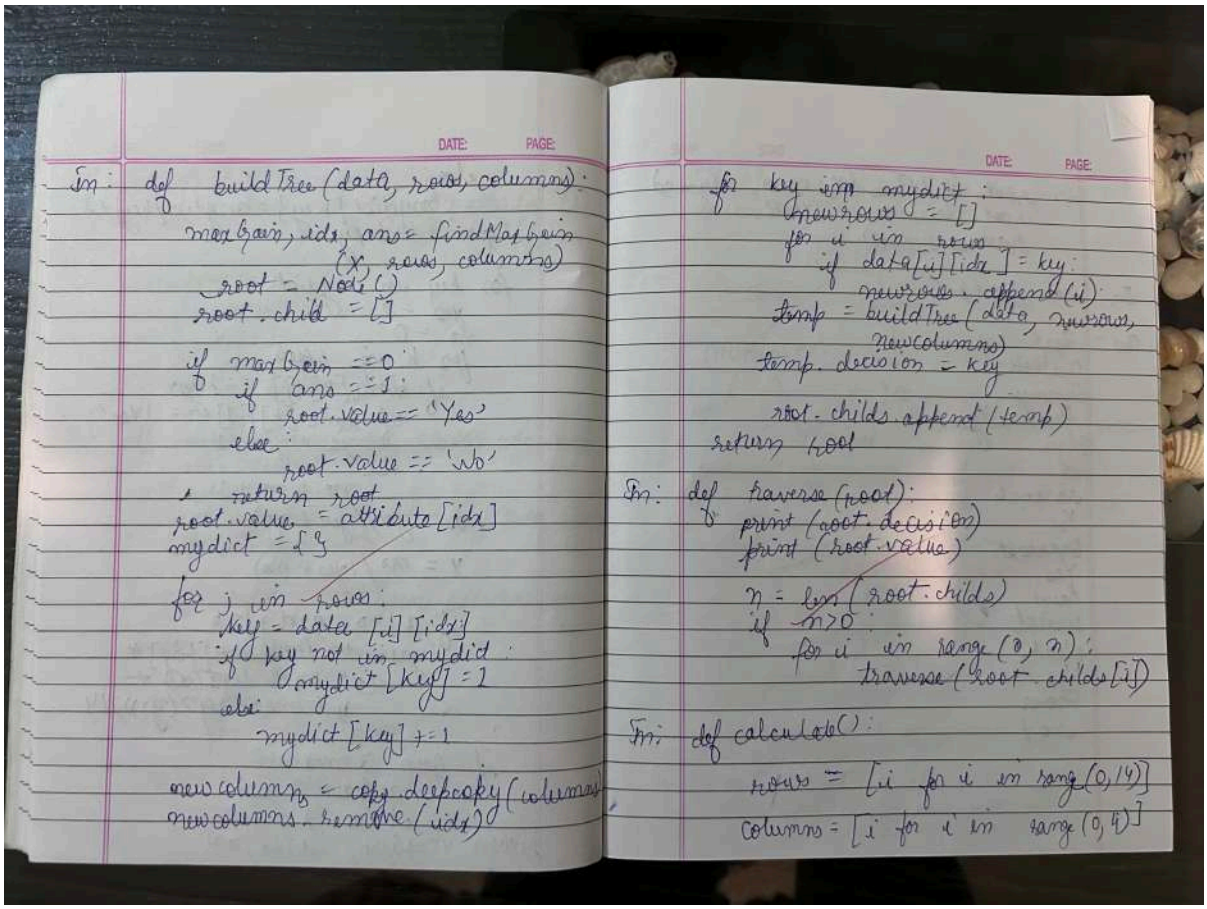
```
def build_tree(X, rows, columns):
    max_gain, best_col, entropy = find_max_gain(X, rows, columns)
    if best_col == -1 or entropy == 0:
        return
```

Record Book

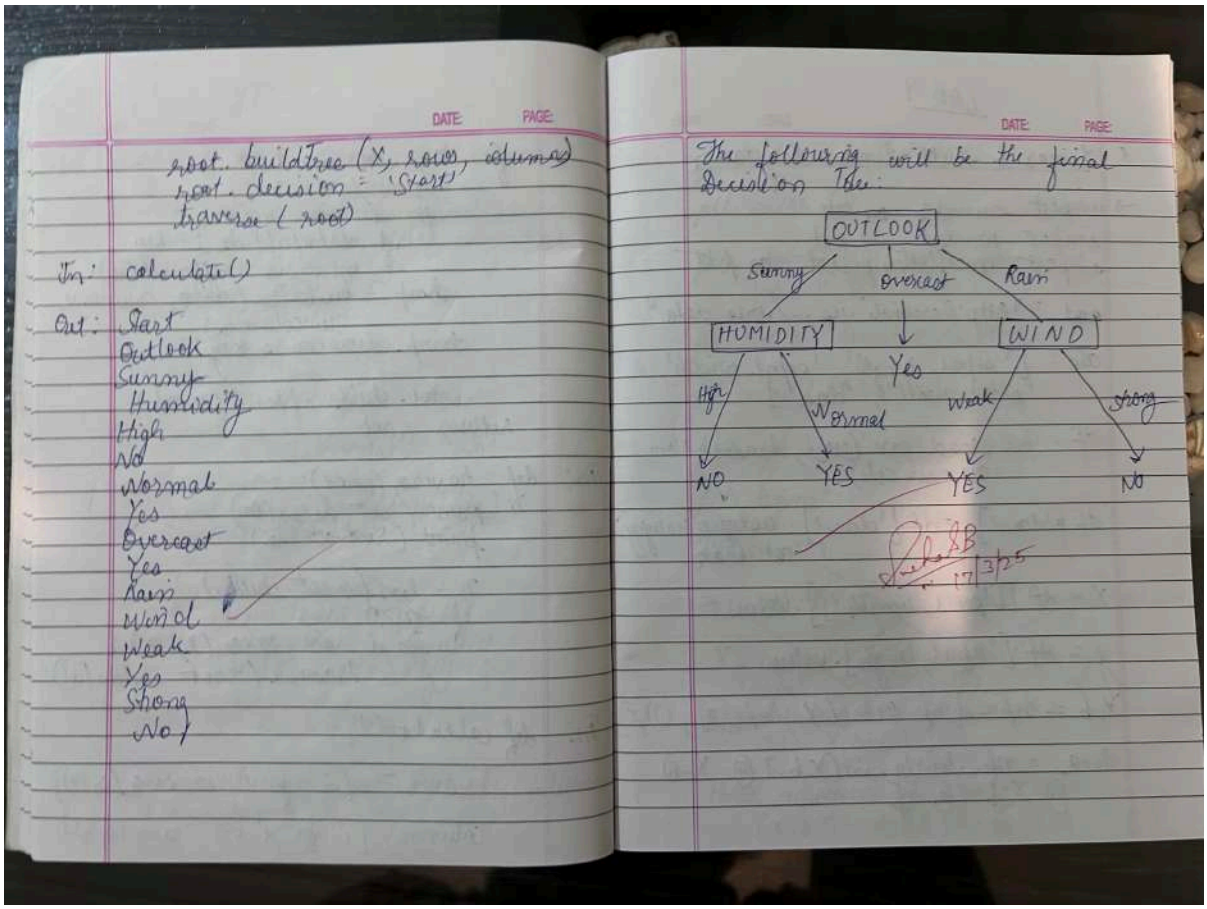




```
if x!=0 and y!=0:
    entropy = -1*(x*math.log2(x)
        + y*math.log2(y))

if x==1:
    ans = 1
if y==1:
    ans = 0
return entropy, ans

In: 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] + 1
        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.child = []

    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, 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)]
```



▼ LAB 4: Linear Regression, Multiple Regression, Logical Regression

Code

```
import numpy as np
import pandas as pd

# Sample data
data = {
    "Temp": [30, 28, 25, 27, 32],
    "Humidity": [70, 65, 80, 85, 75],
    "Windspeed": [5, 10, 12, 8, 7],
    "Rain": [1, 0, 1, 0, 1]
}
```

```
df = pd.DataFrame(data)

# Features and target
X = df[["Temp", "Humidity", "Windspeed"]].values
y = df["Rain"].values

# Feature scaling (Standardization)
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

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

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Cost function
def compute_cost(X, y, weights):
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    cost = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
    return cost

# Gradient Descent
def gradient_descent(X, y, weights, alpha, iterations):
    m = len(y)
    cost_history = []
    for i in range(iterations):
        h = sigmoid(np.dot(X, weights))
        gradient = np.dot(X.T, (h - y)) / m
        weights -= alpha * gradient
        cost_history.append(compute_cost(X, y, weights))
    return weights, cost_history

# Initialize weights, learning rate, and iterations
```

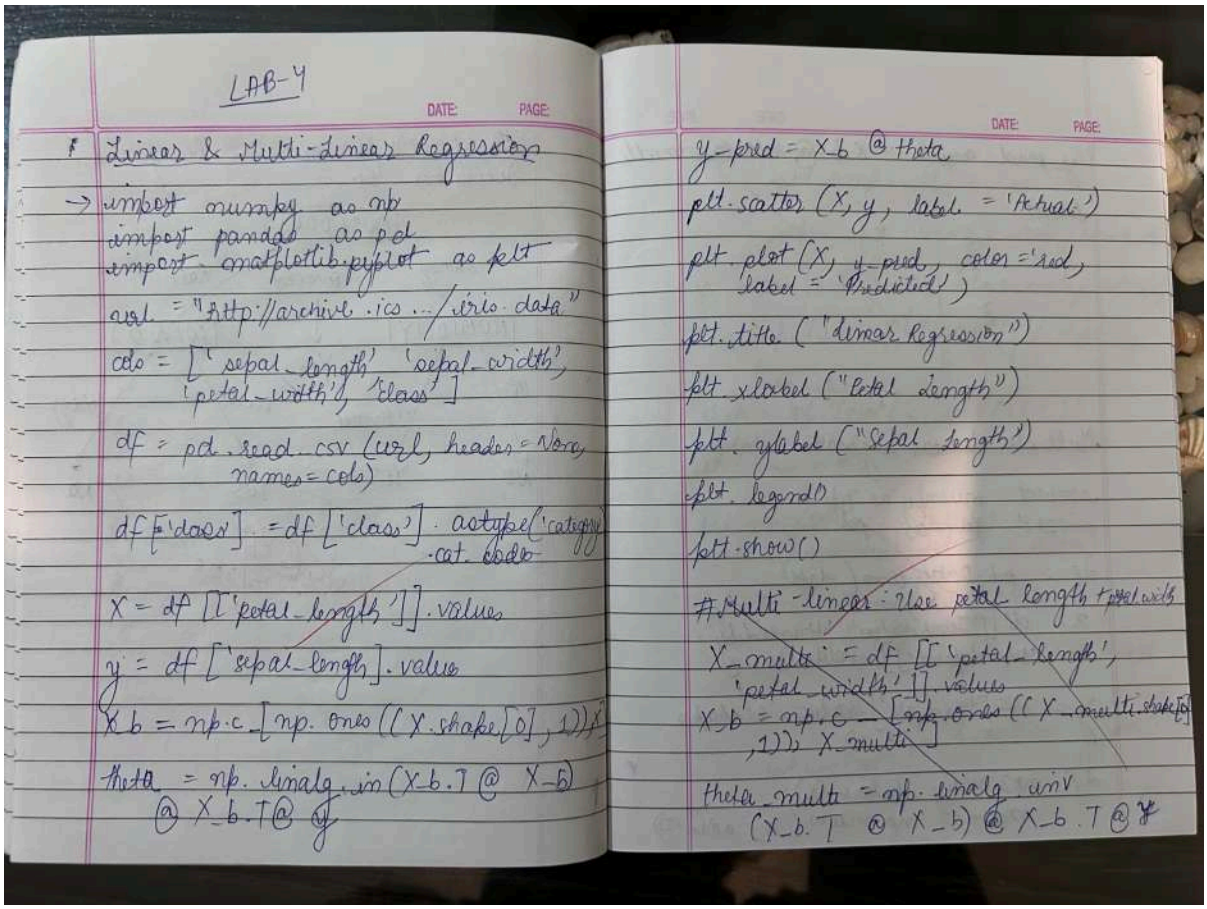
```
weights = np.zeros(X.shape[1])
alpha = 0.1
iterations = 1000

# Train model
weights, cost_history = gradient_descent(X, y, weights, alpha, iterations)
print("Final parameters:", weights)

# Prediction function
def predict(X, weights):
    return sigmoid(np.dot(X, weights)) >= 0.5

# Make predictions
predictions = predict(X, weights).astype(int)
print("Predicted labels:", predictions)

# Calculate accuracy
accuracy = np.mean(predictions == y)
print("Accuracy:", accuracy * 100, "%")



import numpy as np
import pandas as pd

# Example data
data = {
    "Area": [2104, 1600, 2400, 1416, 3000],
    "Bedrooms": [3, 3, 3, 2, 4],
    "Price": [399900, 329900, 369000, 232000, 539900]
}
df = pd.DataFrame(data)

X = df[["Area", "Bedrooms"]].values
y = df["Price"].values

# Feature scaling
```

```
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
X = np.c_[np.ones(X.shape[0]), X]  # Add bias

theta = np.zeros(X.shape[1])
alpha = 0.01
iterations = 1000

def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1/(2*m)) * np.sum((predictions - y) ** 2)
    return cost

def gradient_descent(X, y, theta, alpha, iterations):
    m = len(y)
    cost_history = []
    for i in range(iterations):
        predictions = X.dot(theta)
        gradient = (1/m) * X.T.dot(predictions - y)
        theta -= alpha * gradient
        cost_history.append(compute_cost(X, y, theta))
    return theta, cost_history

theta, cost_history = gradient_descent(X, y, theta, alpha, iterations)
print("Final parameters:", theta)



import numpy as np
import pandas as pd

# Sample data
data = {
    "Temp": [30, 28, 25, 27, 32],
    "Humidity": [70, 65, 80, 85, 75],
    "Windspeed": [5, 10, 12, 8, 7],
```

```
    "Rain": [1, 0, 1, 0, 1]
}

df = pd.DataFrame(data)

# Features and target
X = df[["Temp", "Humidity", "Windspeed"]].values
y = df["Rain"].values

# Feature scaling (Standardization)
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

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

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Cost function
def compute_cost(X, y, weights):
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    cost = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
    return cost

# Gradient Descent
def gradient_descent(X, y, weights, alpha, iterations):
    m = len(y)
    cost_history = []
    for i in range(iterations):
        h = sigmoid(np.dot(X, weights))
        gradient = np.dot(X.T, (h - y)) / m
        weights -= alpha * gradient
        cost_history.append(compute_cost(X, y, weights))
    return weights, cost_history
```

```
# Initialize weights, learning rate, and iterations
weights = np.zeros(X.shape[1])
alpha = 0.1
iterations = 1000

# Train model
weights, cost_history = gradient_descent(X, y, weights, alpha, iterations)
print("Final parameters:", weights)

# Prediction function
def predict(X, weights):
    return sigmoid(np.dot(X, weights)) >= 0.5

# Make predictions
predictions = predict(X, weights).astype(int)
print("Predicted labels:", predictions)

# Calculate accuracy
accuracy = np.mean(predictions == y)
print("Accuracy:", accuracy * 100, "%")
```

Record Book

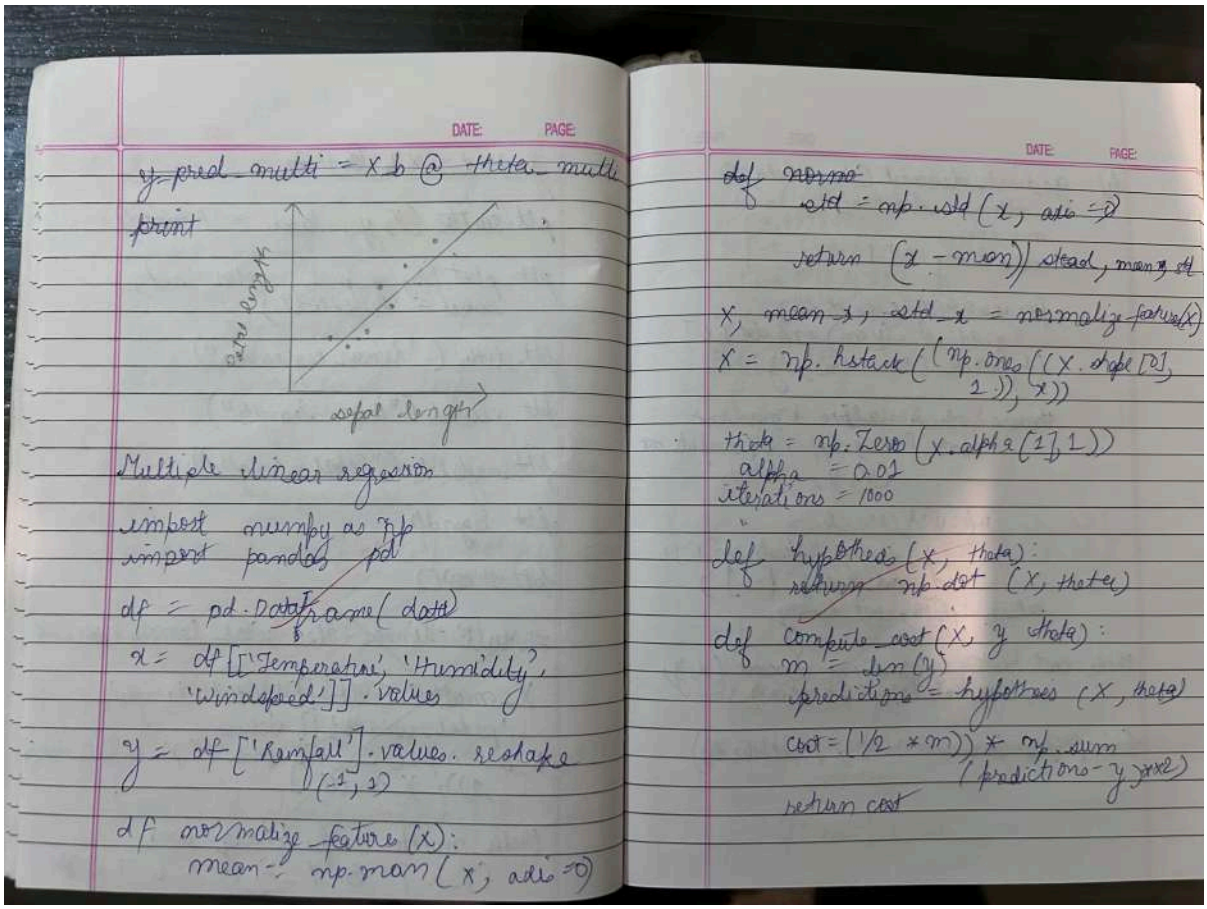


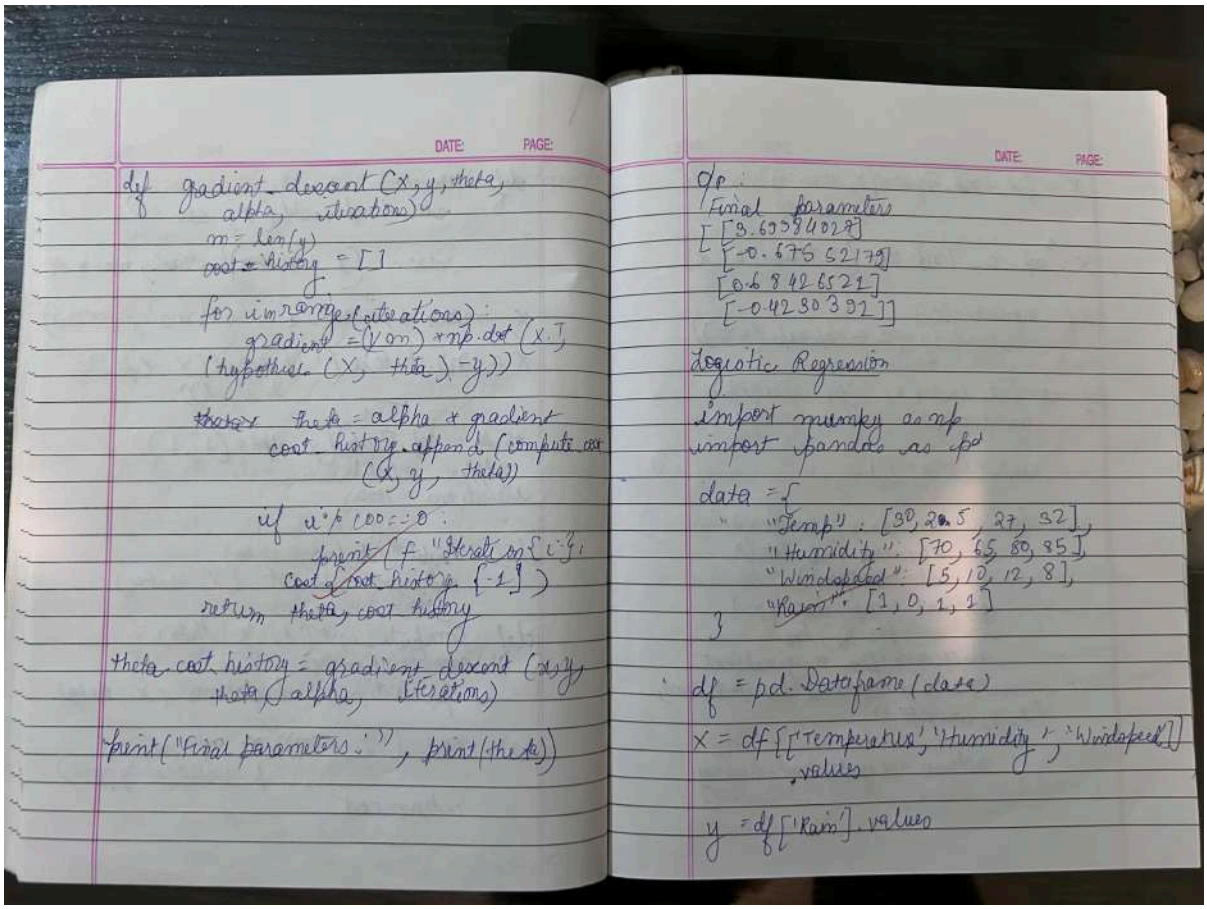
LAB-4

DATE: PAGE:

1 Linear & Multi-Linear Regression

```
-> import numpy as np
   import pandas as pd
   import matplotlib.pyplot as plt
   url = "http://archive.ics.../iris.data"
   cols = ['sepal_length' 'sepal_width',
           'petal_width', 'class']
   df = pd.read_csv(url, header=None,
                    names=cols)
   df['class'] = df['class'].astype('category')
                 .cat.codes
   X = df[['petal_length']].values
   y = df['sepal_length'].values
   X_b = np.c_[np.ones((X.shape[0], 1)), X]
   theta = np.linalg.inv(X_b.T @ X_b)
           @ X_b.T @ y
   y_pred = X_b @ theta
   plt.scatter(X, y, label='Actual')
   plt.plot(X, y_pred, color='red',
            label='Predicted')
   plt.title("Linear Regression")
   plt.xlabel("Petal length")
   plt.ylabel("Sepal length")
   plt.legend()
   plt.show()
   #Multi-linear: Use petal length + petal width
   X_multi = df[['petal_length',
                 'petal_width']].values
   X_b = np.c_[np.ones((X_multi.shape[0]
               ,1)), X_multi]
   theta_multi = np.linalg.inv
                 (X_b.T @ X_b) @ X_b.T @ y
```


DATE: PAGE:
y_pred_multi = X_b @ theta_multi
print
sepal length
sepal length
Multiple linear regression
import numpy as np
import pandas as pd
df = pd.DataFrame(data)
x = df[['Temperature', 'Humidity', 'Windspeed']].values
y = df['Rainfall'].values.reshape(-1, 1)
def normalize_feature(X):
mean = np.mean(X, axis=0)
DATE: PAGE:
def normalize
std = np.std(X, axis=0)
return (X - mean)/std, mean, std
X, mean_x, std_x = normalize_feature(X)
X = np.hstack((np.ones((X.shape[0], 1)), X))
theta = np.zeros(X.shape[1], 1))
alpha = 0.01
iterations = 1000
def hypothesis(X, theta):
return np.dot(X, theta)
def compute_cost(X, y, theta):
m = len(y)
predictions = hypothesis(X, theta)
cost = (1/2 * m)) * np.sum
(predictions - y)**2)
return cost

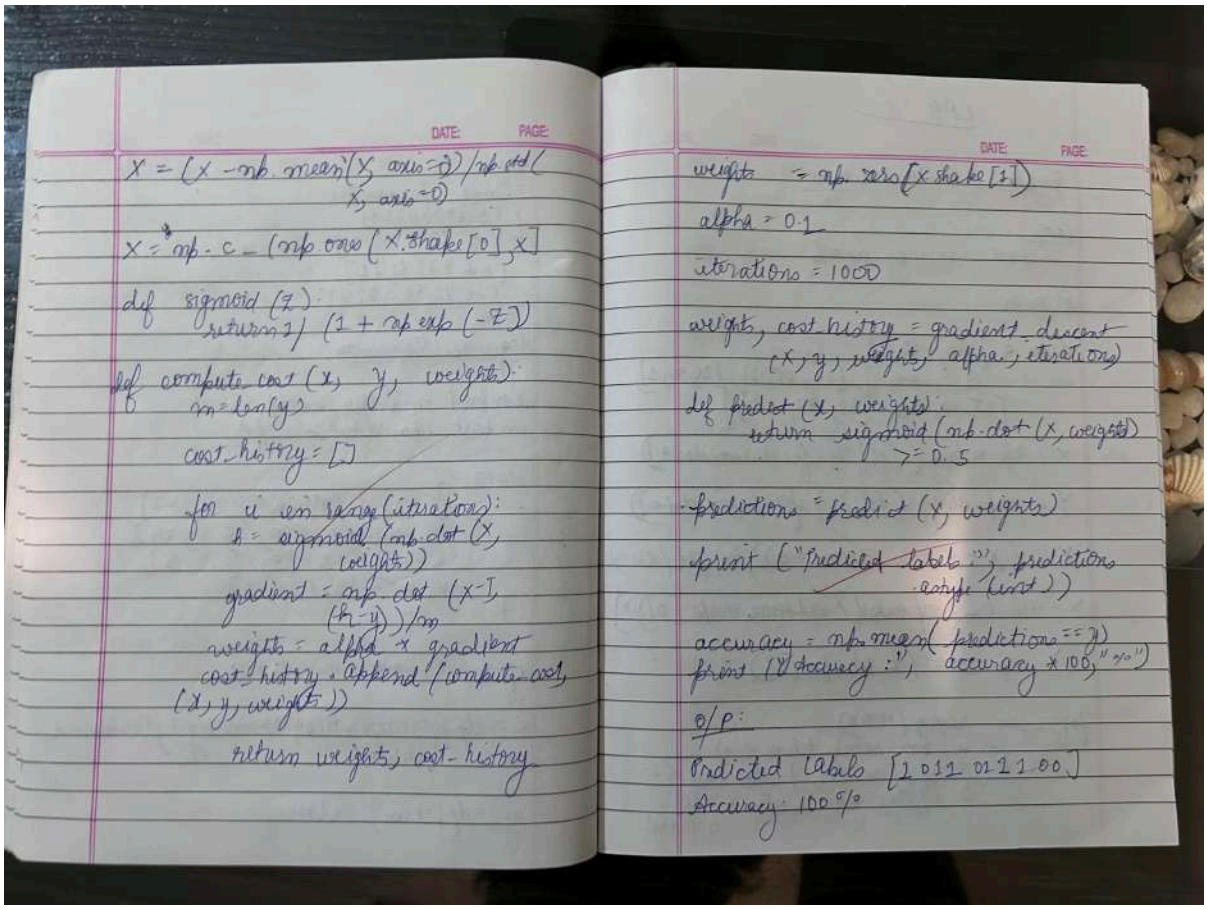


```
def gradient_descent(X, y, theta, alpha, iterations):
    m = len(y)
    cost_history = []
    for i in range(iterations):
        gradient = (1/m) * np.dot(X.T, (hypothesis(X, theta) - y))
        theta = theta - alpha * gradient
        cost_history.append(compute_cost(X, y, theta))
        if i % 100 == 0:
            print(f"Iteration {i}: Cost {cost_history[-1]}")
    return theta, cost_history

theta, cost_history = gradient_descent(X, y, theta, alpha, iterations)

print("Final parameters:", print(theta))
```

```
Op:
Final parameters
[[3.69594028]
 [-0.67552179]
 [0.68426521]
 [-0.42303921]]
```

Logistic Regression

```
import numpy as np
import pandas as pd

data = {
    "Temp": [30, 28.5, 27, 32],
    "Humidity": [70, 65, 80, 85],
    "Windspeed": [5, 10, 12, 8],
    "Rain": [1, 0, 1, 1]
}

df = pd.DataFrame(data)

X = df[['Temperature', 'Humidity', 'Windspeed']].values

y = df['Rain'].values
```



▼ LAB 5: SVM & KNN

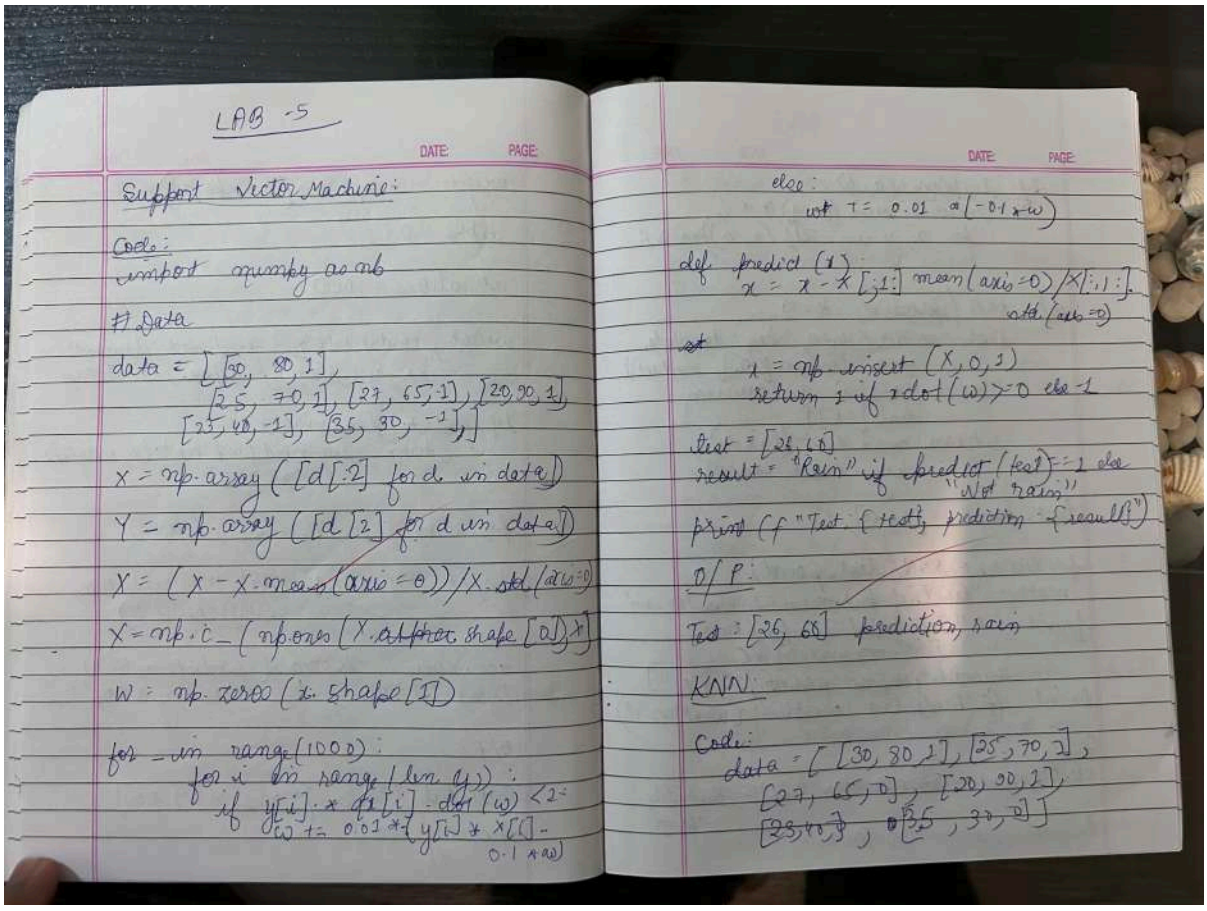
Code

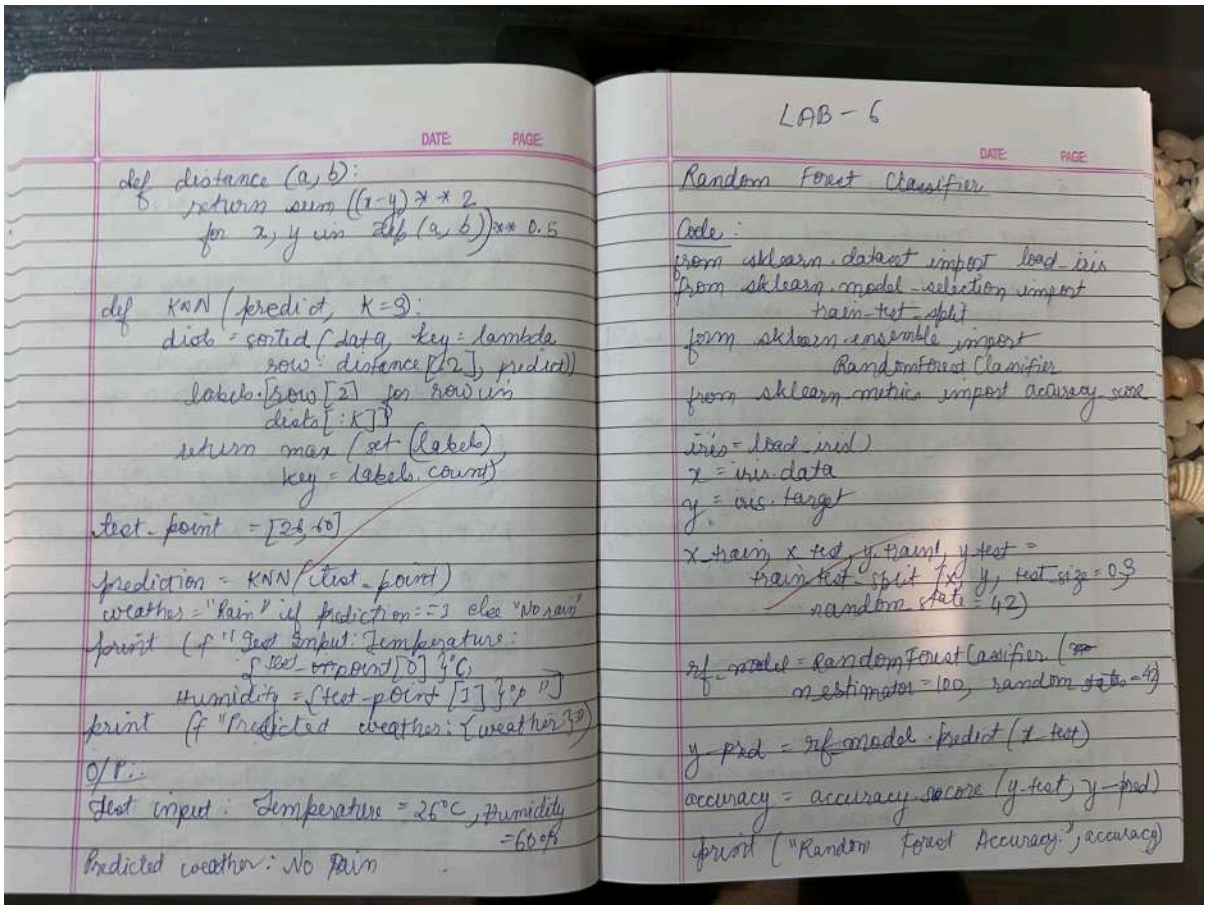
```
import numpy as np

# Data: [Temp, Humidity, Label]
data = [
    [30, 80, 1], [25, 70, 1], [27, 65, 1], [20, 90, 1],
    [25, 40, 0], [35, 30, 0]
]

def distance(a, b):
    return sum((x - y) ** 2 for x, y in zip(a, b)) ** 0.5

def KNN_predict(test_point, k=3):
    # Compute distances
```

```
    dists = sorted(data, key=lambda row: distance(row[:2], test_point))
    # Get labels of k nearest neighbors
    labels = [row[2] for row in dists[:k]]
    # Majority vote
    return 1 if labels.count(1) > labels.count(0) else 0

test_point = [26, 60]
prediction = KNN_predict(test_point)
weather = "Rain" if prediction == 1 else "No Rain"
print(f"Test input: Temperature = {test_point[0]}°C, Humidity = {test_point[1]}
print(f"Predicted weather: {weather}")

import numpy as np

# Data: [Temp, Humidity, Label]
data = [
    [30, 80, 1], [25, 70, 1], [27, 65, 1], [20, 90, 1],
    [25, 40, 0], [35, 30, 0]
]

def distance(a, b):
    return sum((x - y) ** 2 for x, y in zip(a, b)) ** 0.5

def KNN_predict(test_point, k=3):
    # Compute distances
    dists = sorted(data, key=lambda row: distance(row[:2], test_point))
    # Get labels of k nearest neighbors
    labels = [row[2] for row in dists[:k]]
    # Majority vote
    return 1 if labels.count(1) > labels.count(0) else 0

test_point = [26, 60]
prediction = KNN_predict(test_point)
weather = "Rain" if prediction == 1 else "No Rain"
print(f"Test input: Temperature = {test_point[0]}°C, Humidity = {test_point[1]}
```

```
print(f"Predicted weather: {weather}")
```

Record Book





▼ LAB 6: Random Forest, AdaBoost, K-Means Clustering

Code

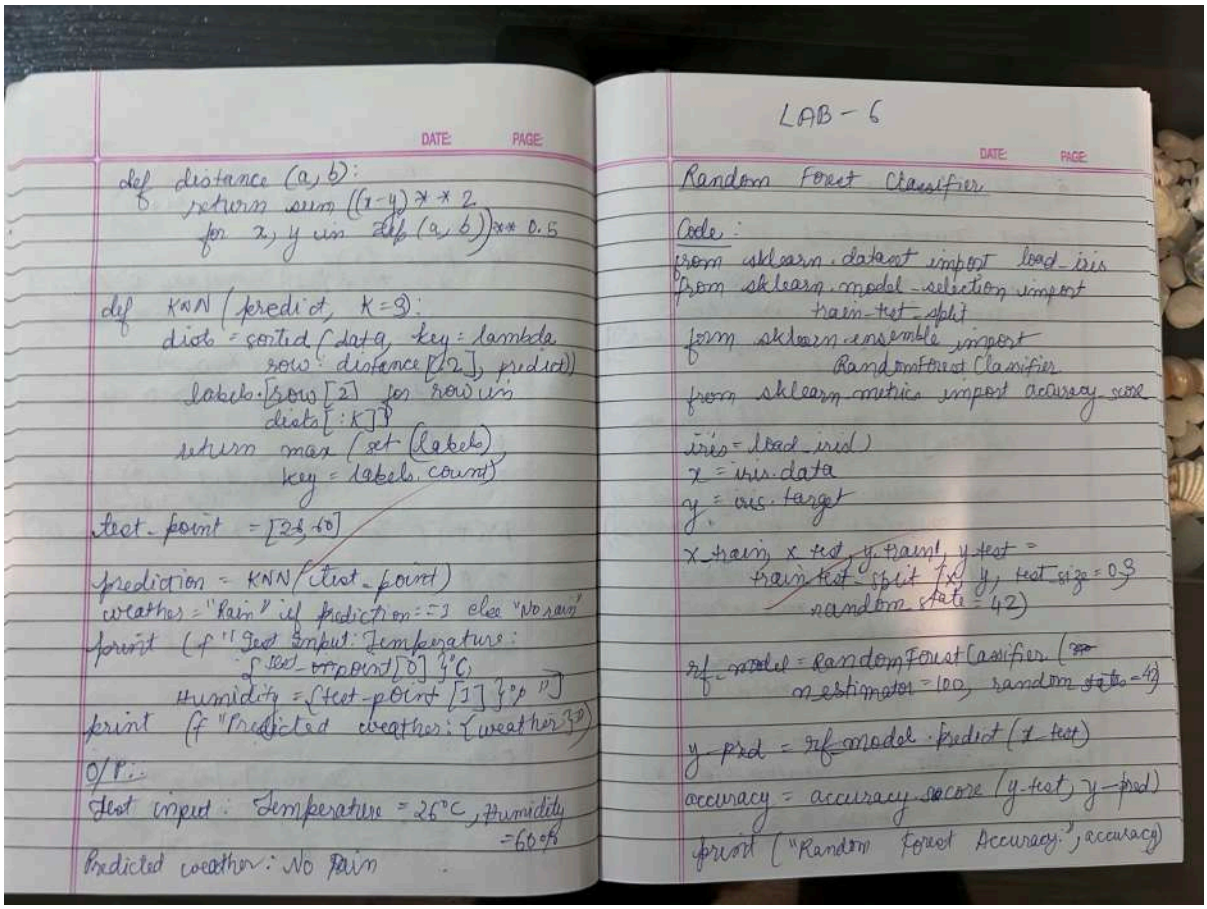
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

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

# Split data
```

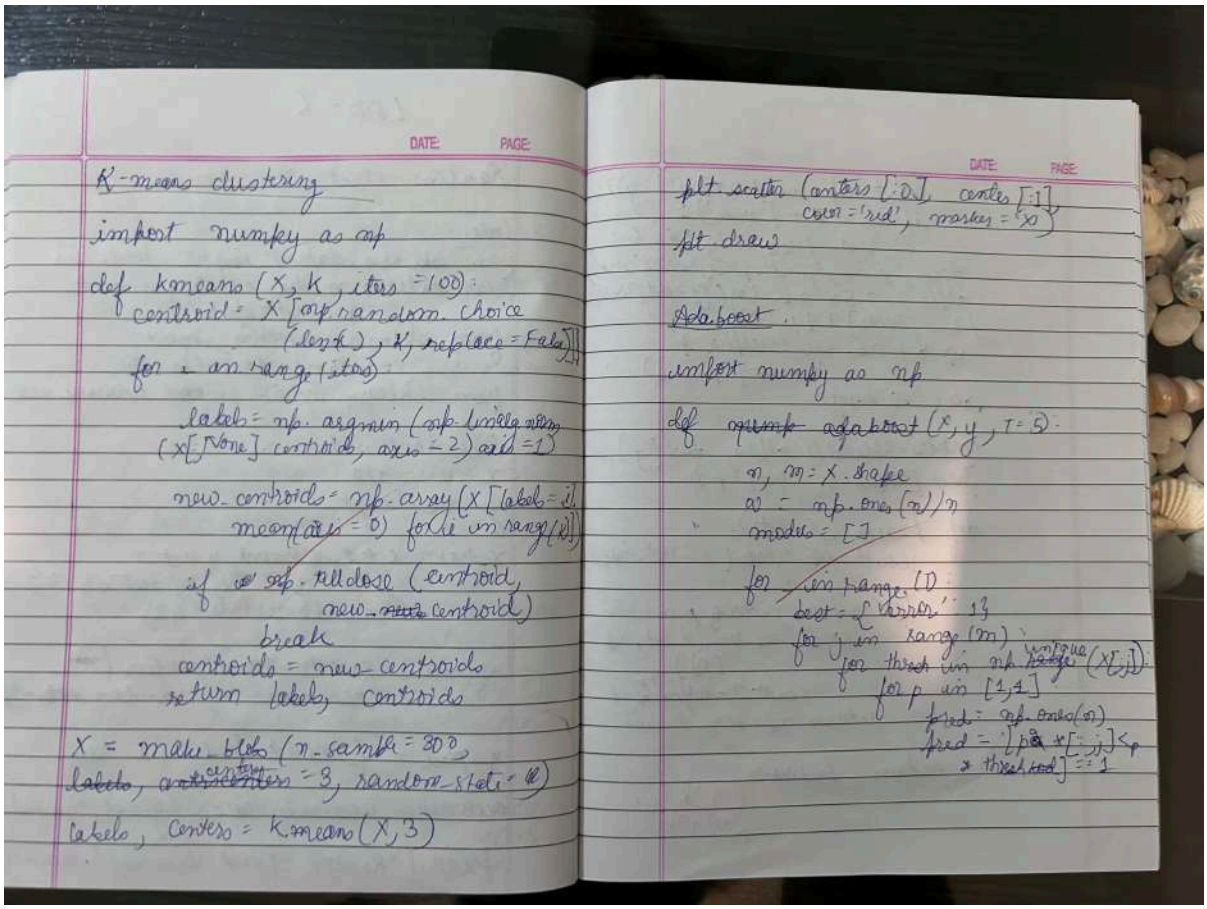
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Train Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict
y_pred = rf_model.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Random Forest Accuracy:", accuracy)

import numpy as np

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

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

def adaboost(X, y, n_clf=5):
    n_samples, n_features = X.shape
    w = np.full(n_samples, (1 / n_samples))
    models = []
```

```
    alphas = []

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

        # Find best stump
        for feature_i in range(n_features):
            feature_values = np.unique(X[:, feature_i])
            for threshold in feature_values:
                for polarity in [1, -1]:
                    predictions = np.ones(n_samples)
                    if polarity == 1:
                        predictions[X[:, feature_i] < threshold] = -1
                    else:
                        predictions[X[:, feature_i] > threshold] = -1
                    error = np.sum(w[y != predictions])
                    if error < min_error:
                        min_error = error
                        clf.polarity = polarity
                        clf.threshold = threshold
                        clf.feature_index = feature_i

        # Compute alpha
        EPS = 1e-10
        alpha = 0.5 * np.log((1 - min_error + EPS) / (min_error + EPS))
        predictions = clf.predict(X)
        w *= np.exp(-alpha * y * predictions)
        w /= np.sum(w)

        models.append(clf)
        alphas.append(alpha)

    return models, alphas

def predict(X, models, alphas):
```

```
    clf_preds = [alpha * clf.predict(X) for clf, alpha in zip(models, alphas)]
    y_pred = np.sign(np.sum(clf_preds, axis=0))
    return y_pred

# Example usage:
# X = np.array([[...], ...])  # shape (n_samples, n_features)
# y = np.array([...])         # shape (n_samples,), labels must be -1 or 1

# models, alphas = adaboost(X, y, n_clf=5)
# y_pred = predict(X, models, alphas)
```

Record Book





DATE: PAGE:

K-means clustering

import numpy as np

def kmeans(X, K, iters=100):
centroid = X[np.random.choice(len(X), K, replace=False)]
for i in range(iters):
labels = np.argmin(np.linalg.norm(X[:None] centroid, axis=2), axis=1)
new_centroids = np.array(X[labels==i].mean(axis=0) for i in range(K))
if np.allclose(centroid, new_centroid)
break
centroids = new_centroids
return labels, centroids

X = make_blobs(n_sample=300, labels, centers=3, random_state=42)
labels, centers = K_means(X, 3)

DATE: PAGE:

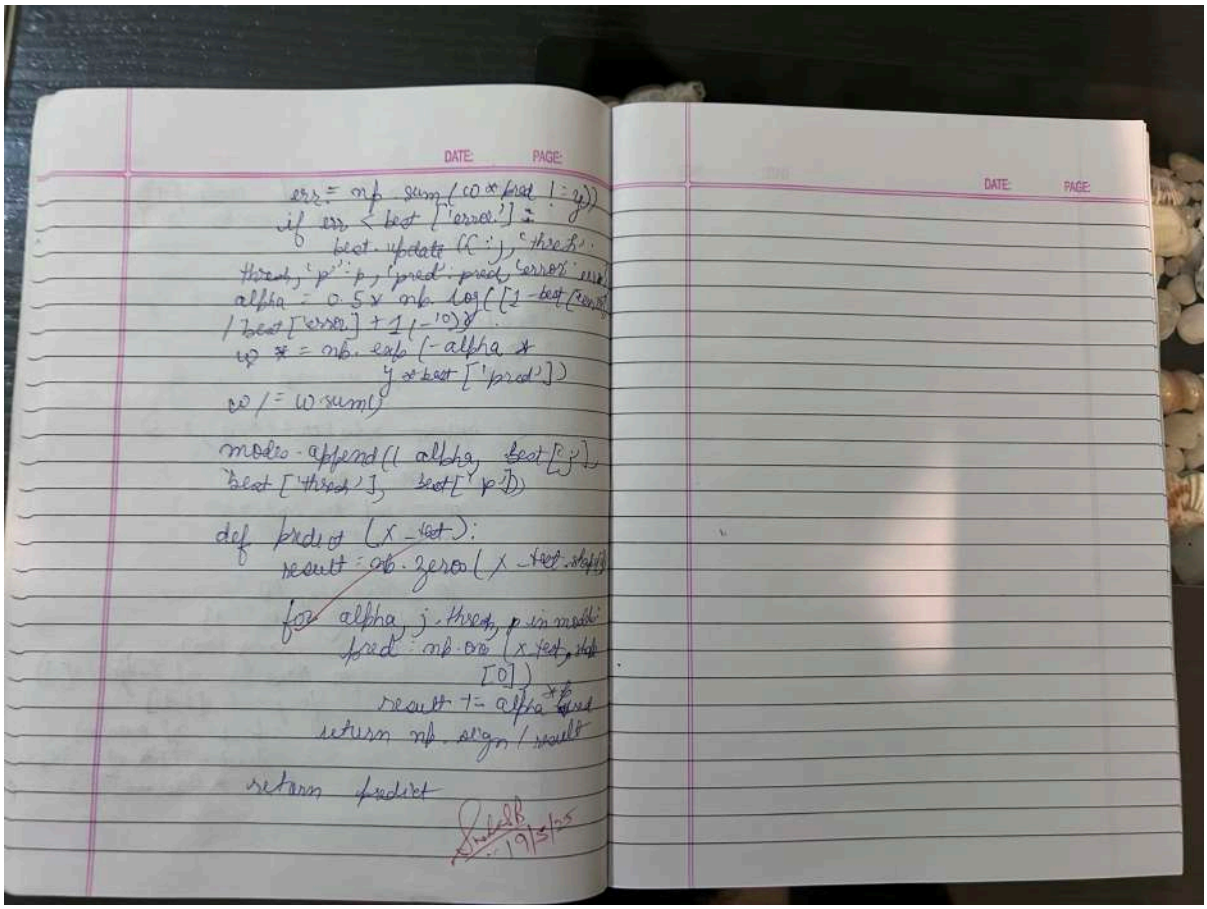
plt.scatter(centers[:,0], centers[:1], color='red', marker='x')
plt.show()

Ada.boost

import numpy as np

def adaboost(X, y, T=5):
n, m = X.shape
w = np.ones(n)/n
models = []

for in range(T):
best = {'error': 1}
for j in range(m):
for thresh in unique(X[:,j]):
for p in [1,-1]:
pred = np.ones(n)
pred = [p * X[:,j] < p * thresh] = -1


DATE:
PAGE:
err = np.sum(w * (pred != y))
if err < best['error']:
best.update({'i': j, 'thresh':
'p': p, 'pred': pred, 'error': err})
alpha = 0.5 * np.log((1 - best['error']
/ best['error']) + 1e-10)
w *= np.exp(-alpha *
y * best['pred'])
w /= w.sum()
models.append((alpha, best['j'],
best['thresh'], best['p']))
def predict(X_test):
result = np.zeros(X_test.shape
for alpha, j, thresh, p in models:
pred = np.where(X_test,
[0])
result += alpha *
return np.sign(result)
return predict
19/5/25
DATE:
PAGE: