VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Machine Learning (23CS6PCMAL)

Submitted by

Niket Dugar (1BM22CS180)

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 **Niket Dugar (1BM22CS180),** 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 a Machine Learning (23CS6PCMAL) work prescribed for the said degree.

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

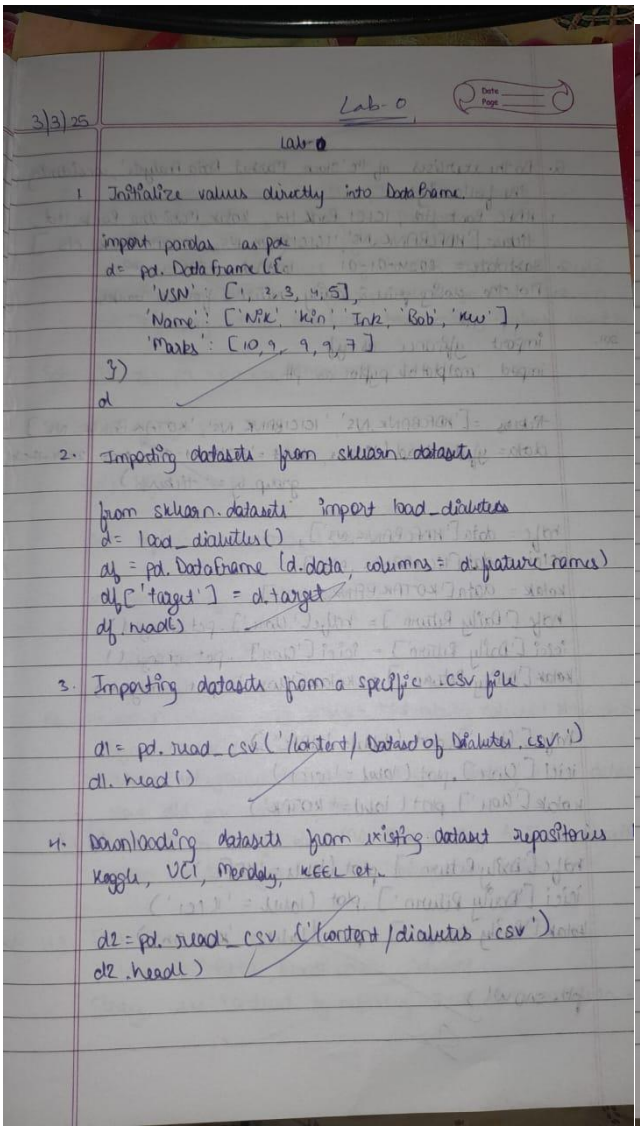
Index

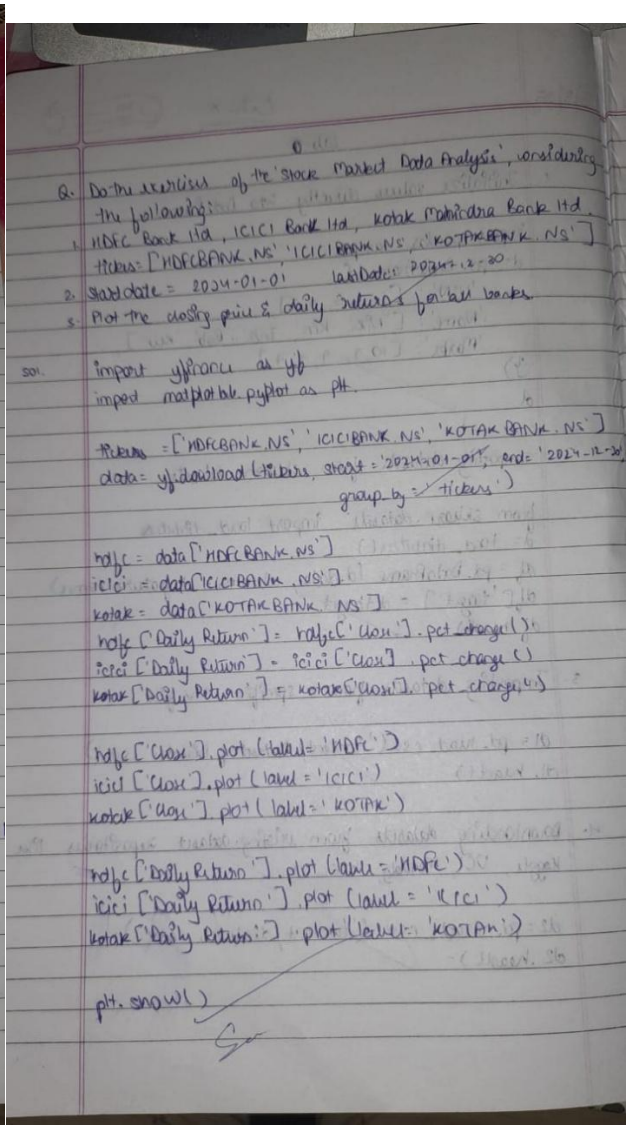
Github Link: *https://github.com/Niketjr/ML*

Program 1

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

Screenshot:





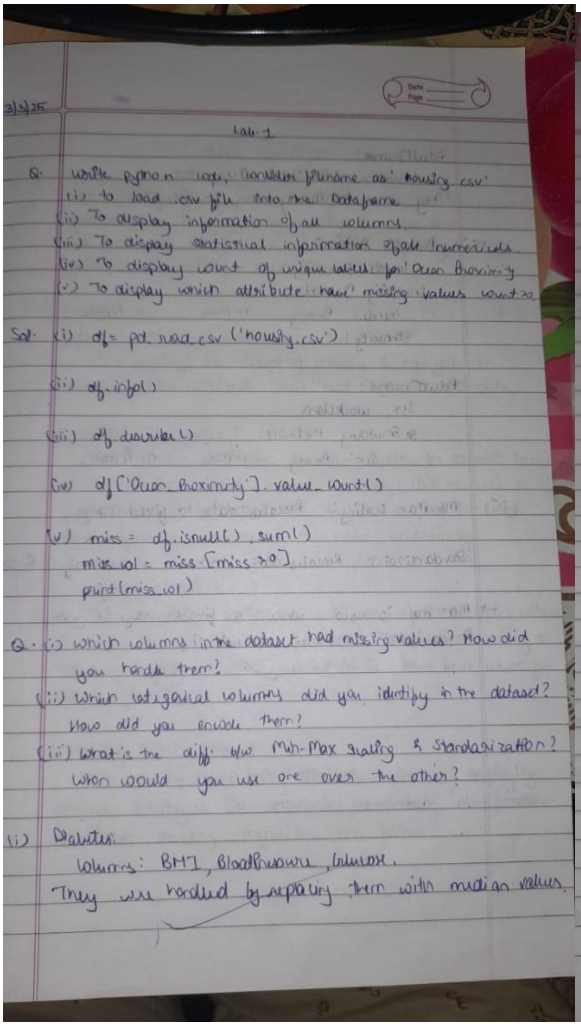
Code:

```
import pandas as pd
try:
    df = pd.read_csv('data.csv')
    print("Original Data:\n", df.head())
except FileNotFoundError:
    print("File not found. Please ensure 'data.csv' exists.")
    exit()
df = df.dropna()
if 'Quantity' in df.columns and 'Price' in df.columns:
    df['Total'] = df['Quantity'] * df['Price']

if 'Category' in df.columns:
    df = df[df['Category'] == 'Electronics']

df = df.sort_values(by='Total', ascending=False)

df.to_csv('cleaned_data.csv', index=False)
df.to_excel('cleaned_data.xlsx', index=False)

print("Cleaned and filtered data exported successfully.")
```

Program 2

Demonstrate various data pre-processing techniques for a given dataset

Screenshots:



Lab-1

Q: Write Python code, consider filename as 'housing.csv'
(i) to load .csv file into the Dataframe
(ii) To display information of all columns
(iii) To display statistical information of all numericals
(iv) To display count of unique values for 'Ocean_Proximity'
(v) To display which attribute have missing values count

Sol: (i) df = pd.read_csv('housing.csv')

(ii) df.info()

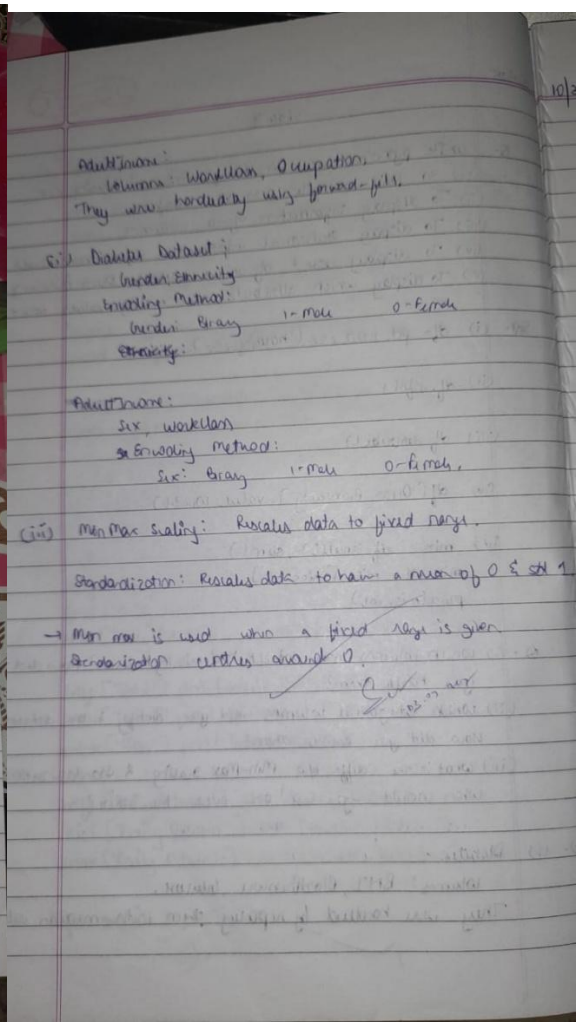
(iii) df.describe()

(iv) df['Ocean_Proximity'].value_counts()

(v) miss = df.isnull().sum()
miss_val = miss[miss>0]
print(miss_val)

Q. (i) Which columns in the dataset had missing values? How did you handle them?
(ii) Which categorical columns did you identify in the dataset? How did you encode them?
(iii) What is the diff b/w Min-Max scaling & Standardization? When would you use one over the other?

(i) Diabetes:
Columns: BMI, BloodPressure, Glucose.
They were handled by replacing them with median values.



Adult Income:
Columns: Workclass, Occupation.
They were handled by using forward-fill.

(ii) Diabetes Dataset:
Gender, Ethnicity
Encoding Method:
Gender: Binary 1-Male 0-Female
~~Ethnicity~~:

Adult Income:
Six, Workclass
Encoding Method:
Six: Binary 1-Male 0-Female.

(iii) MinMax Scaling: Rescales data to fixed range.

Standardization: Rescales data to have a mean of 0 & SD 1.

→ Min max is used when a fixed range is given.
Standardization centres around 0.

Code:

```
from sklearn.datasets import load_iris

import pandas as pd

iris = load_iris()

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

df['target'] = iris.target

print(df.head())
```

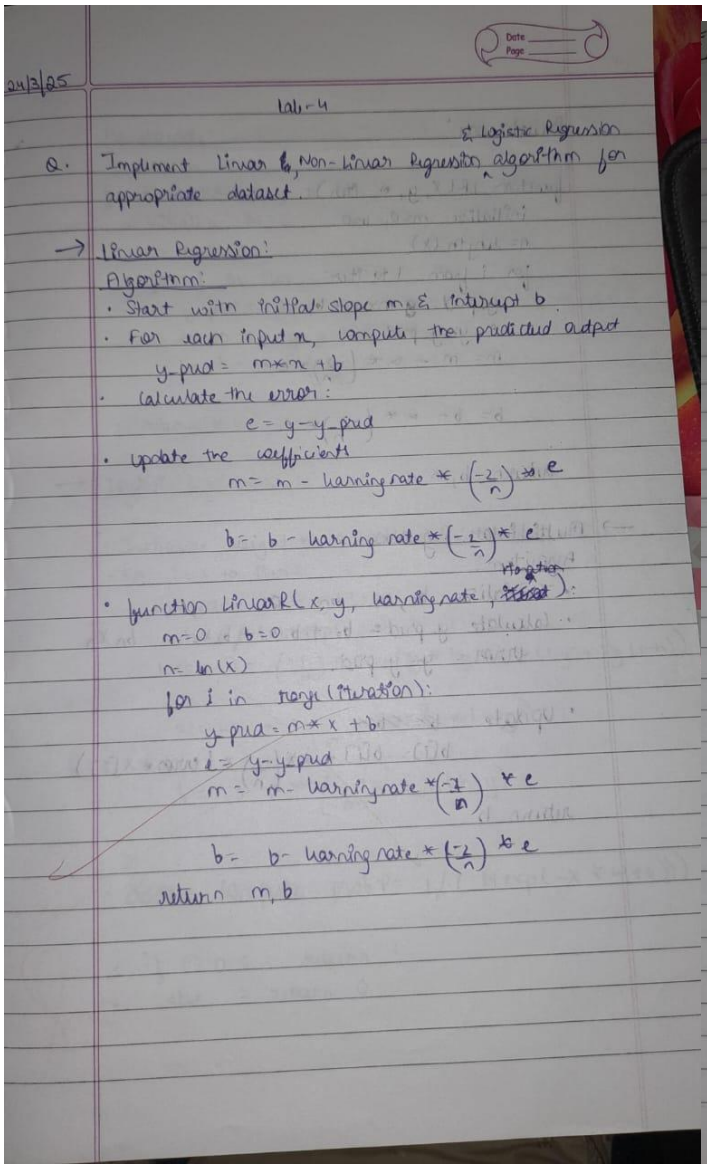
```
import kagglehub

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')

print(df.head())

print(df['Company Name'])

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

df = pd.DataFrame(data)

print(df)

from sklearn.datasets import load_diabetes

diabetes = load_diabetes()

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

print(df.head())

print(df.columns)

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

print(df.head())


import yfinance as yf

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')
```

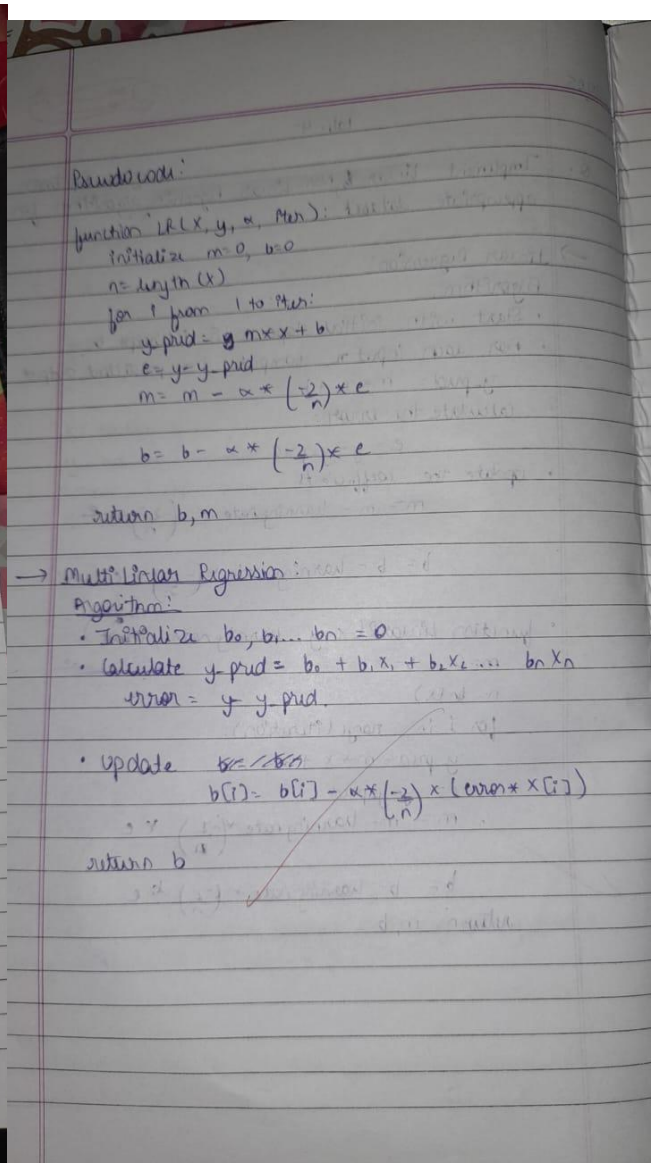
```
print("First 5 rows of the dataset:")
print(data.head())
print("\nShape of the dataset:")
print(data.shape)

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 Price")
plt.subplot(2, 1, 2)
reliance_data['Daily Return'].plot(title="Reliance Industries - Daily Returns", color='orange')
plt.tight_layout()
plt.show()
```

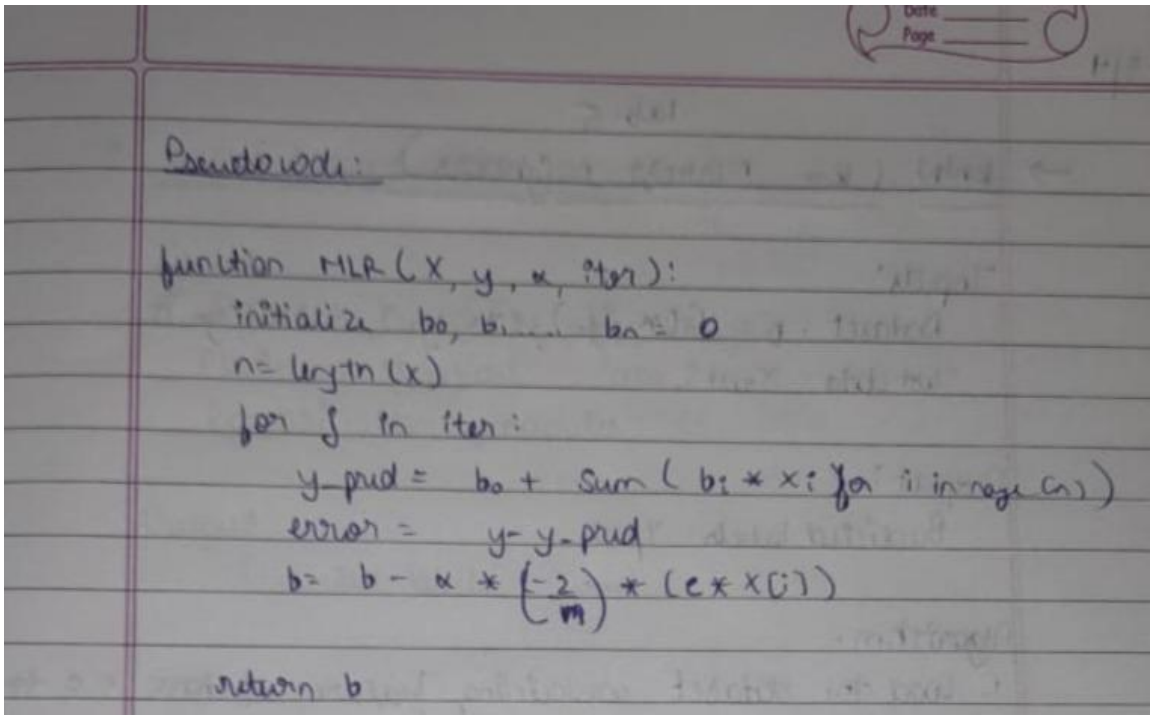
Program-3

Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

Screenshots:







Code:

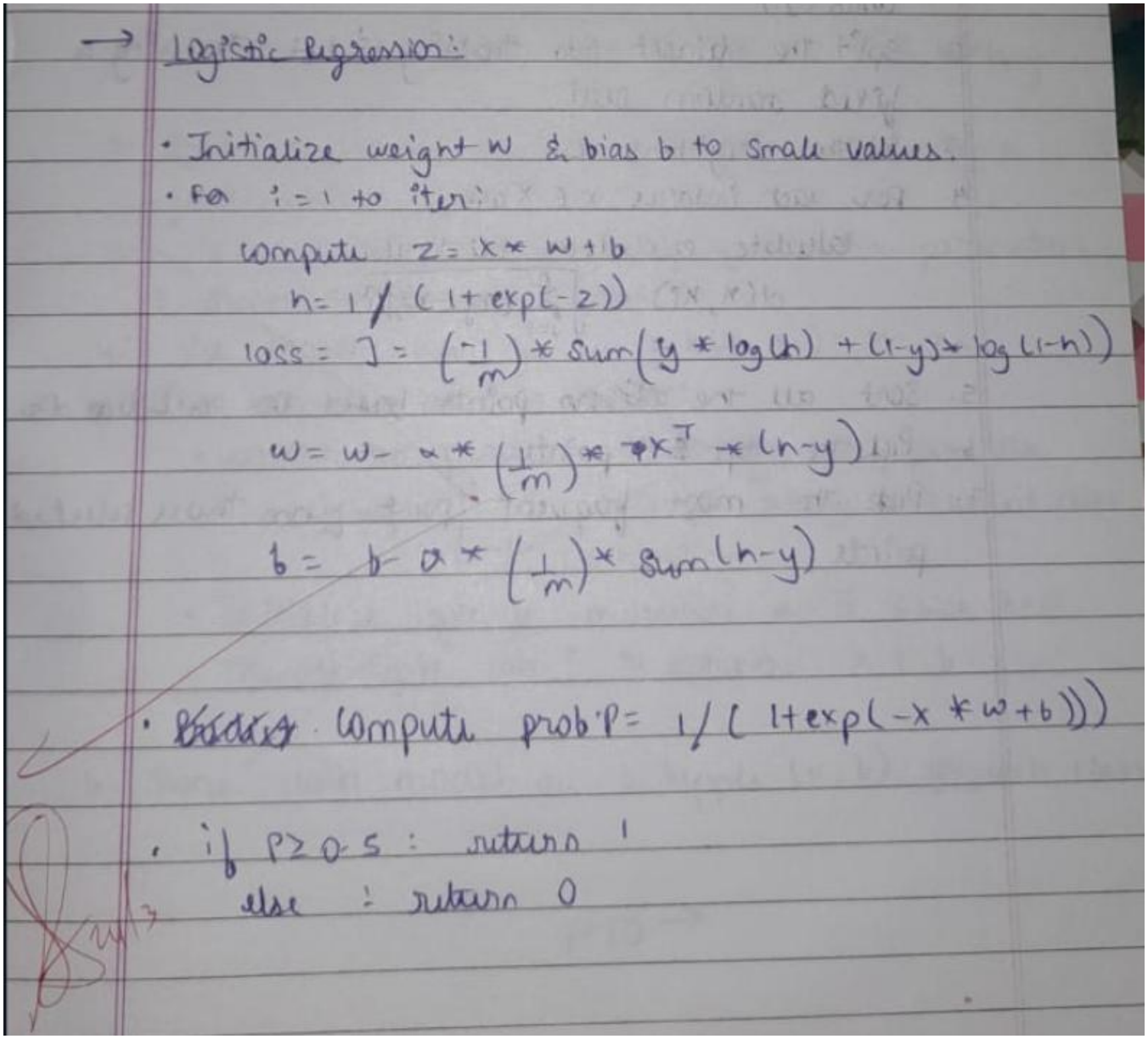
```
import pandas as pd

from sklearn.linear_model import LinearRegression

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_diabetes

import matplotlib.pyplot as plt


# Linear Regression with one feature

diabetes = load_diabetes()

X = diabetes.data[:, [2]]  # BMI feature

y = diabetes.target


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

model = LinearRegression()
```

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

plt.scatter(X_test, y_test, color='blue')
plt.plot(X_test, y_pred, color='red')
plt.title("Linear Regression - BMI vs Target")
plt.xlabel("BMI")
plt.ylabel("Target")
plt.show()

# Multiple Linear Regression with all features
X = diabetes.data
y = diabetes.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
multi_model = LinearRegression()
multi_model.fit(X_train, y_train)
y_pred_multi = multi_model.predict(X_test)

print("Multiple Linear Regression - R² Score:", multi_model.score(X_test, y_test))
```

Program-4

Build Logistic Regression Model for a given dataset

Screenshots:



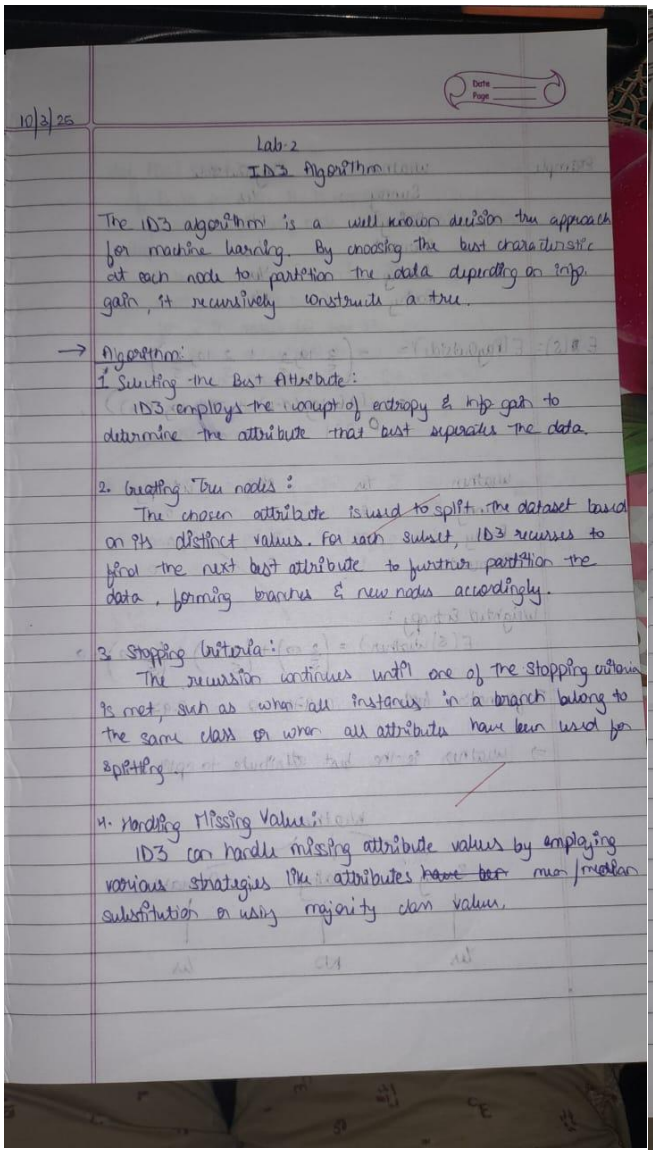
Code:

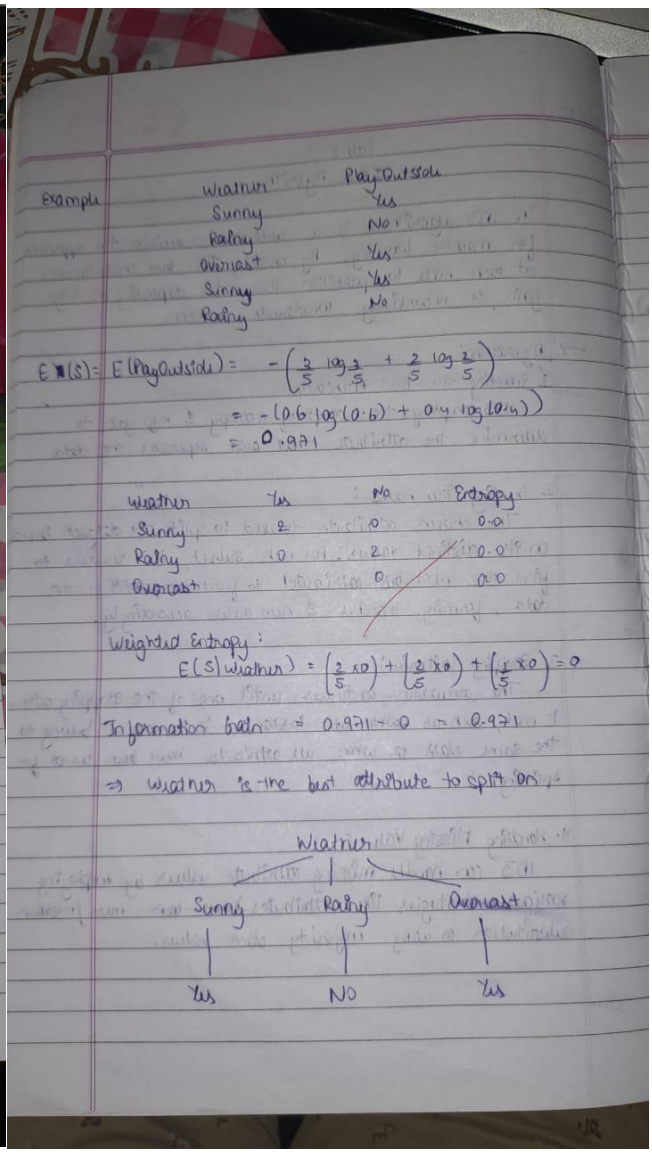
```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = load_iris()
X = iris.data
y = (iris.target == 0).astype(int)  # Binary: Setosa vs not

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

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

Program-5

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

Screenshots:





```
def ID3(D,A):
    if D is pure or A is empty:
        return a leaf node with majority class in D
    else:
        A_best = argmax(Gain(D,A))
        root = Node(A_best)
        for v in values(A_best):
            D_v = subset(D, A_best, v)
            child = ID3(D_v, A - {A_best})
            root.add_child(v, child)
        return root
```

Gain(D,A) = Entropy(D) − $\sum \frac{|S_v|}{|S|}$ Entropy(D_v)

Entropy = $p(C_i) \times \log_2(p(C_i))$

10/3/25

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")
```

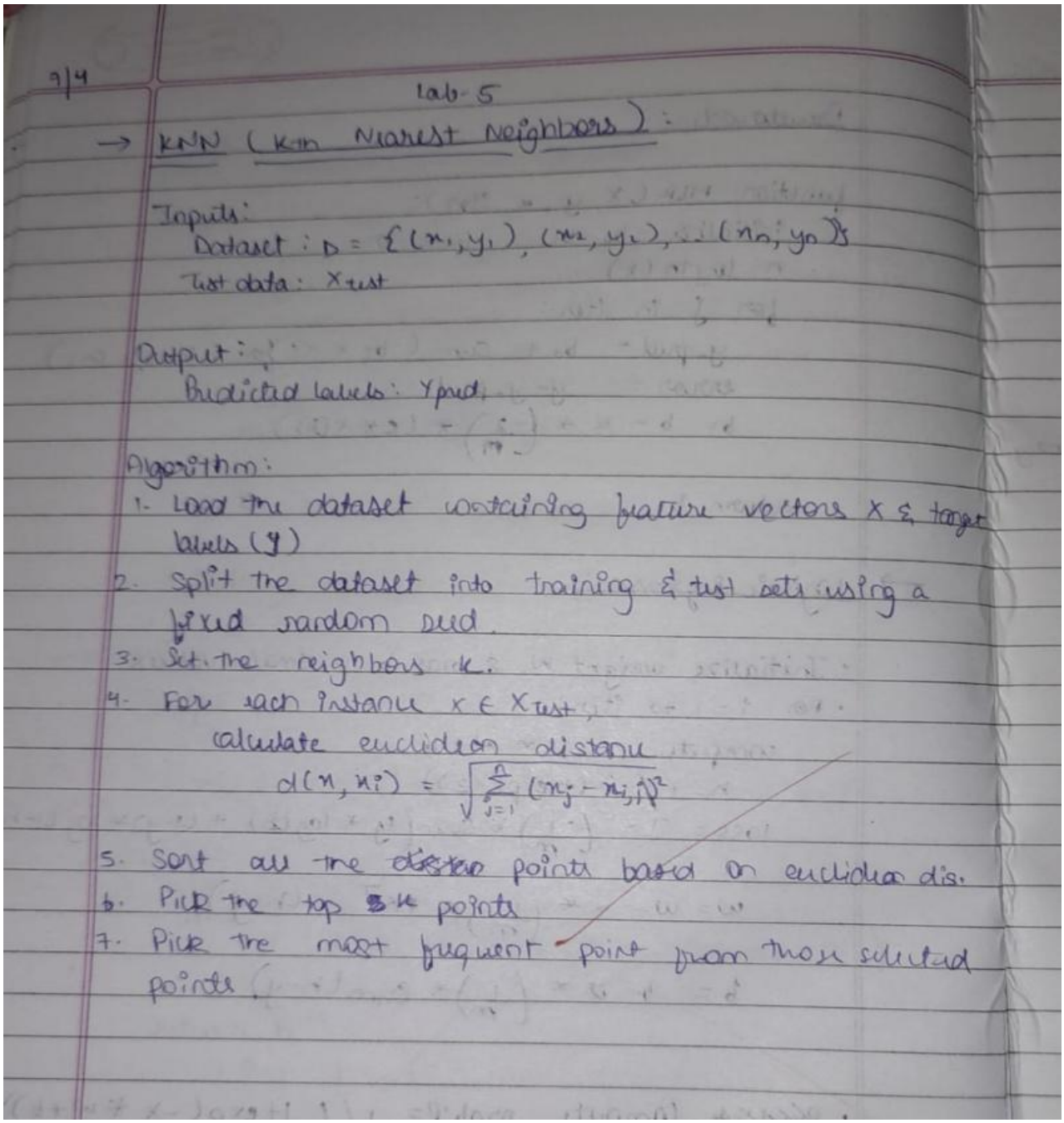
```
    if isinstance(tree, dict):  # Tree node
        for feature, branches in tree.items():
            feature_name = f"{parent_name}_{feature}"
            dot.node(feature_name, feature)
            dot.edge(parent_name, feature_name, label=parent_value)

            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.edge(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-6

Build KNN Classification model for a given dataset

Screenshots:



Code:

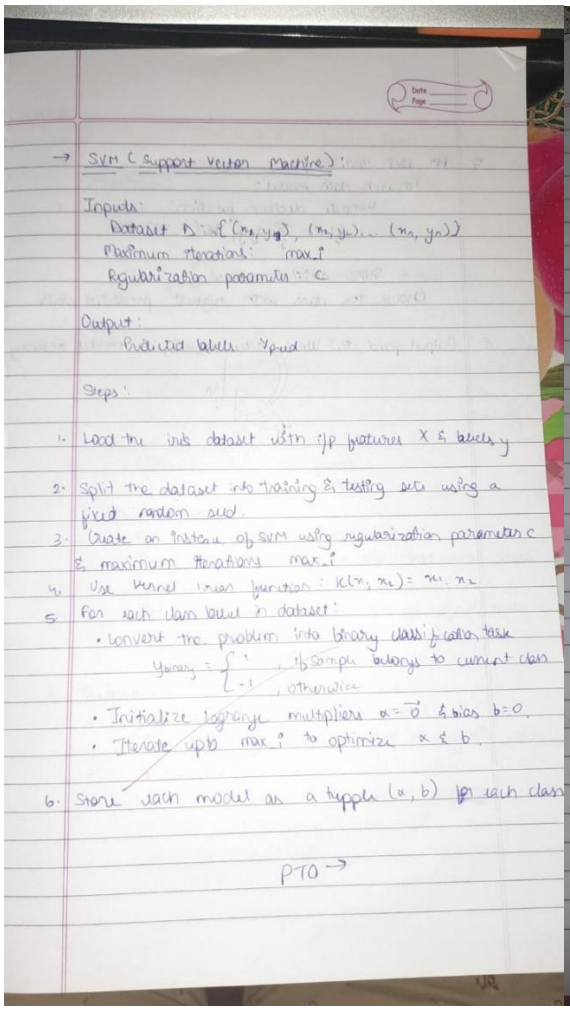
```
import numpy as np

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]
```

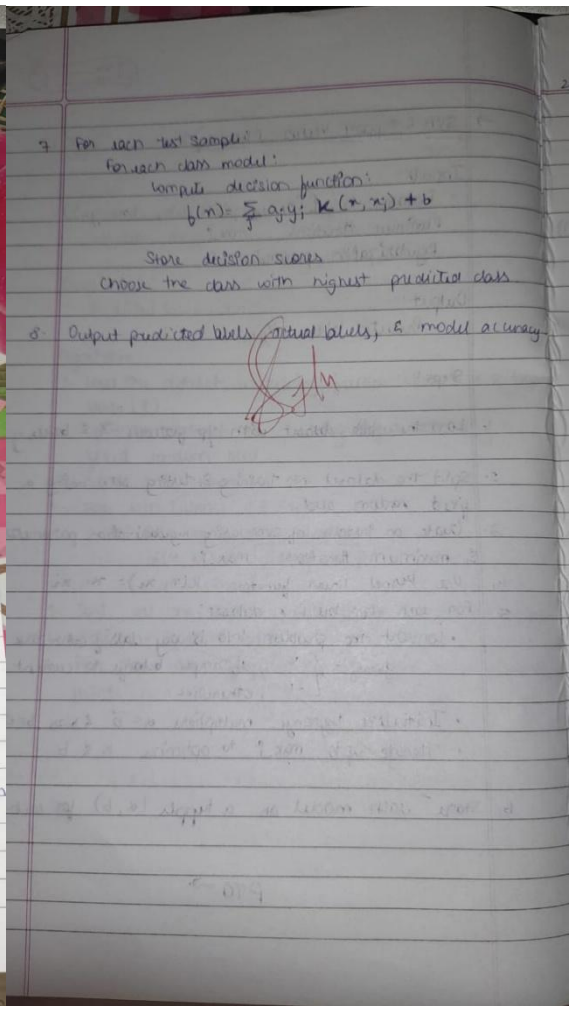
```
        # Get the labels of those neighbors
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # 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

Screenshots:





Code:

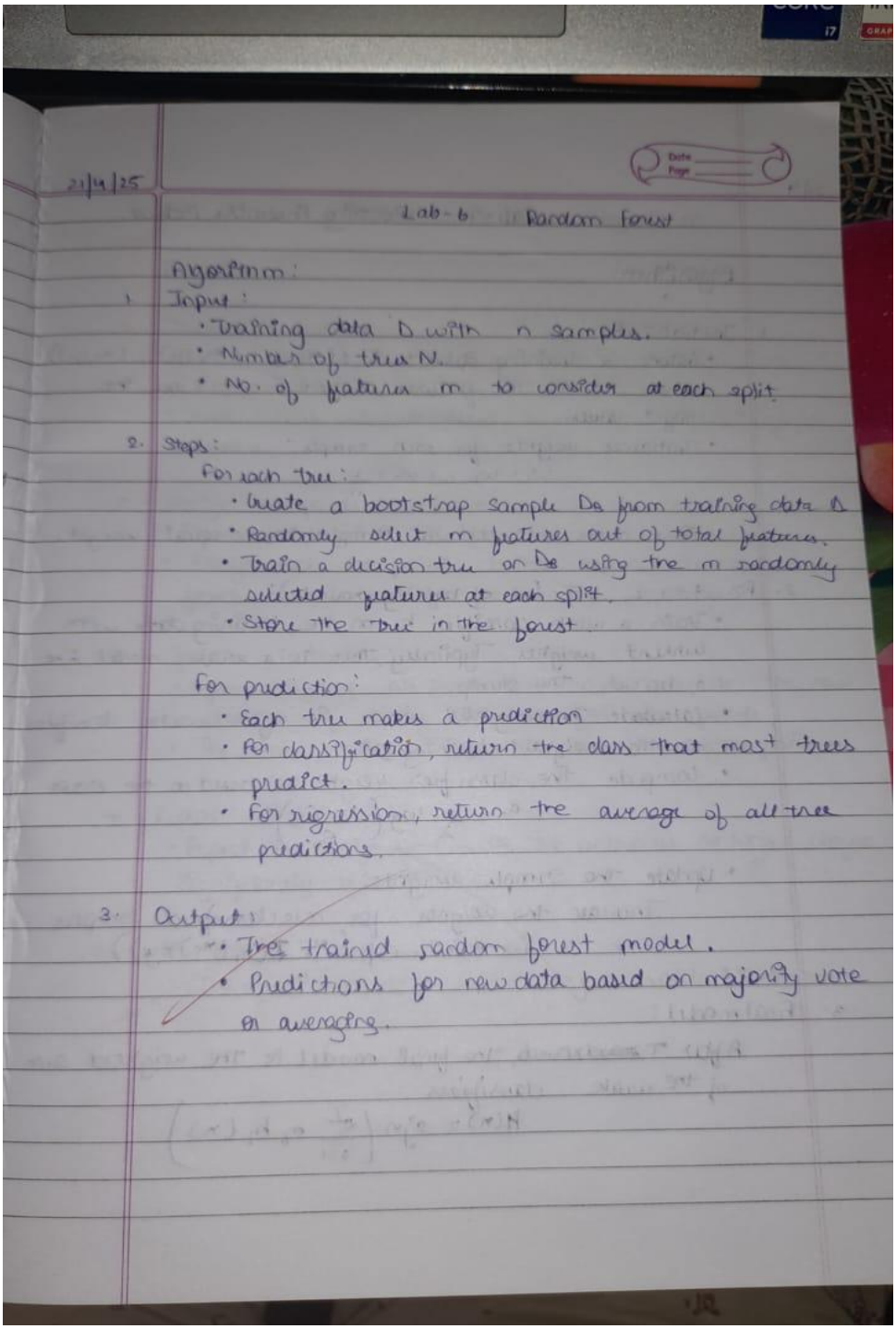
```
import pandas as pd

import seaborn as sns

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split
```

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import matplotlib.pyplot as plt

iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.Series(iris.target)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.25, random_state=1)
model = SVC(kernel='rbf', C=1.0, gamma='scale')
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

Program-8

Implement Random forest ensemble method on a given dataset

Screenshots:



Code:

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


# Load iris dataset manually
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
data = np.column_stack((X, y))
columns = iris.feature_names + ['target']
df = pd.DataFrame(data, columns=columns)
# Split dataset
def train_test_split(data, test_size=0.2):
    data = data.sample(frac=1).reset_index(drop=True)
    test_count = int(test_size * len(data))
    return data.iloc[test_count:], data.iloc[:test_count]
train_data, test_data = train_test_split(df)
# Gini index calculation
def gini_index(groups, classes):
    n_instances = sum(len(group) for group in groups)
    gini = 0.0
```

```
    for group in groups:
        size = len(group)
        if size == 0: continue
        score = 0.0
        group_labels = group[:, -1]
        for class_val in classes:
            p = np.sum(group_labels == class_val) / size
            score += p * p
        gini += (1 - score) * (size / n_instances)
    return gini
# Split datast
def test_split(index, value, dataset):
    left = dataset[dataset[:, index] < value]
    right = dataset[dataset[:, index] >= value]
    return left, right
# Choosebest split
def get_split(dataset):
    class_values = list(set(dataset[:, -1]))
    b_index, b_value, b_score, b_groups = None, None, float('inf'), None
    for index in range(dataset.shape[1] - 1):
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
```

```
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index': b_index, 'value': b_value, 'groups': b_groups}
# Create terminal node
def to_terminal(group):
    outcomes = group[:, -1]
    return Counter(outcomes).most_common(1)[0][0]
# Recursive split
def split(node, max_depth, min_size, depth):
    left, right = node['groups']
    del node['groups']
    if len(left) == 0 or len(right) == 0:
        node['left'] = node['right'] = to_terminal(np.vstack((left, right)))
        return
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left)
        split(node['left'], max_depth, min_size, depth + 1)
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
```

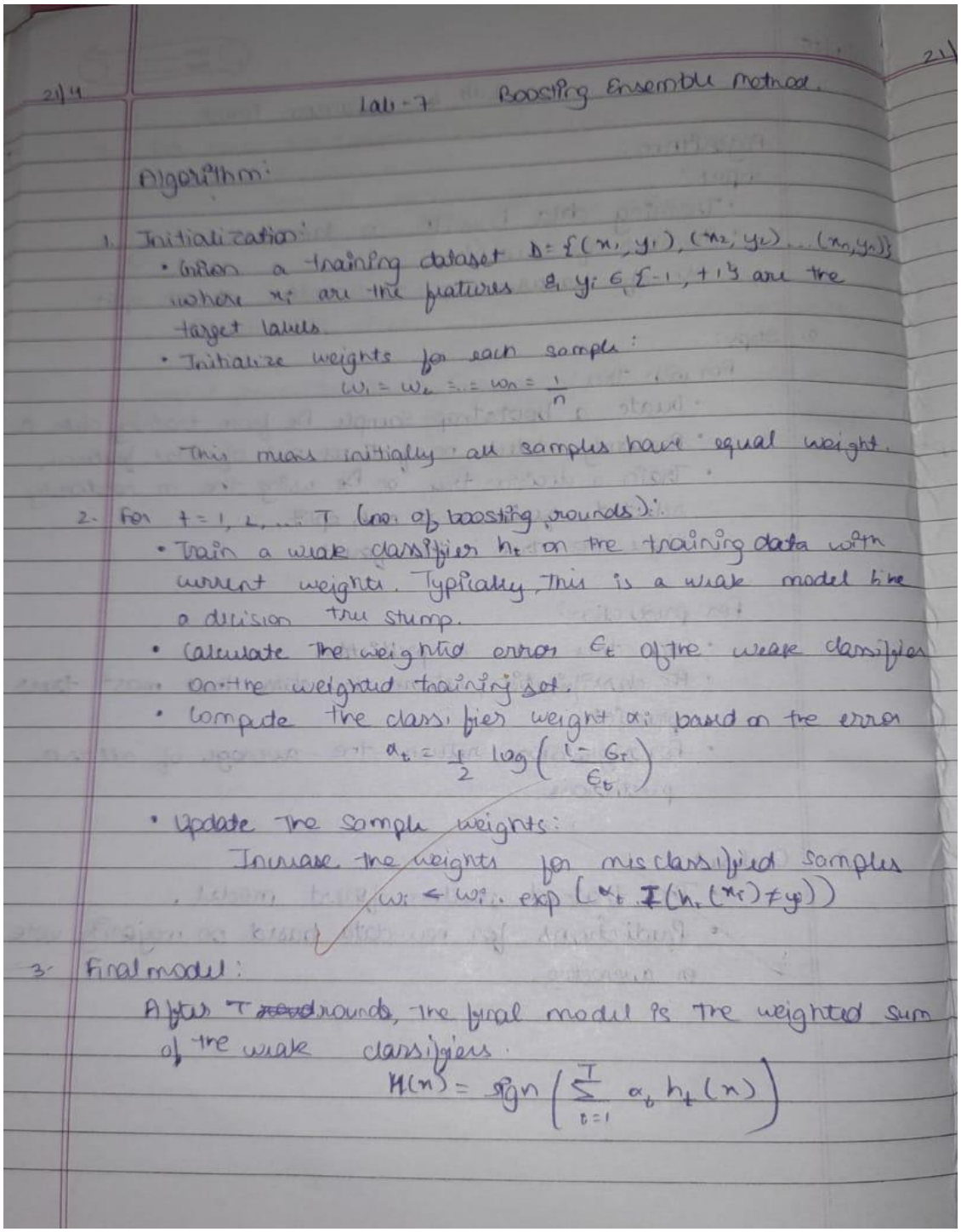
```
        node['right'] = get_split(right)
        split(node['right'], max_depth, min_size, depth + 1)
# Build tree
def build_tree(train, max_depth, min_size):
    root = get_split(train)
    split(root, max_depth, min_size, 1)
    return root
# Make prediction
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
# Build a random forest
def subsample(dataset, ratio):
    n_sample = round(len(dataset) * ratio)
    return dataset.sample(n=n_sample, replace=True).value
def random_forest(train, test, max_depth, min_size, sample_size, n_trees):
```

```
    trees = []
    for _ in range(n_trees):
        sample = subsample(train, sample_size)
        tree = build_tree(sample, max_depth, min_size)
        trees.append(tree)
    predictions = [bagging_predict(trees, row) for row in test.values]
    return predictions
# Bagging predict
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree in trees]
    return max(set(predictions), key=predictions.count)
# Evaluate accuracy
def accuracy_metric(actual, predicted):
    correct = sum(1 for i in range(len(actual)) if actual[i] == predicted[i])
    return correct / float(len(actual)) * 100.0
# Run forest
n_trees = 5
max_depth = 5
min_size = 1
sample_size = 0.8
predictions = random_forest(train_data, test_data, max_depth, min_size, sample_size, n_trees)
actual = test_data['target'].values
acc = accuracy_metric(actual, predictions)
print("Random Forest Accuracy (manual):", round(acc, 2), "%")
```

Program-9

Implement Boosting ensemble method on a given dataset

Screenshots:



Lab - 7 Boosting Ensemble Method.

Algorithm:

1. Initialization:
   - Given a training dataset $D = \{(x_1, y_1), (x_2, y_2) \ldots (x_n, y_n)\}$ where x_i are the features & $y_i \in \{-1, +1\}$ are the target labels.
   - Initialize weights for each sample:
     $$w_1 = w_2 = \ldots = w_n = \frac{1}{n}$$
     This means initially all samples have equal weight.
2. For $t = 1, 2, \ldots T$ (no. of boosting rounds):
   - Train a weak classifier h_t on the training data with current weights. Typically, this is a weak model like a decision tree stump.
   - Calculate the weighted error ϵ_t of the weak classifier on the weighted training set.
   - Compute the classifier weight α_t based on the error
     $$\alpha_t = \frac{1}{2} \log\left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$$
   - Update the sample weights:
     Increase the weights for misclassified samples
     $$w_i \leftarrow w_i \cdot \exp(\alpha_t \cdot \mathbb{I}(h_t(x_i) \neq y_i))$$
3. Final model:
   After T ~~rounds~~ rounds, the final model is the weighted sum of the weak classifiers.
   $$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load and prepare binary classification dataset
iris = load_iris()
X = iris.data
y = (iris.target == 0).astype(int)  # 1 for setosa, 0 for others
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Decision stump class
class DecisionStump:
    def __init__(self):
        self.feature_index = None
        self.threshold = None
        self.polarity = 1
        self.alpha = None

    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] = 0
```

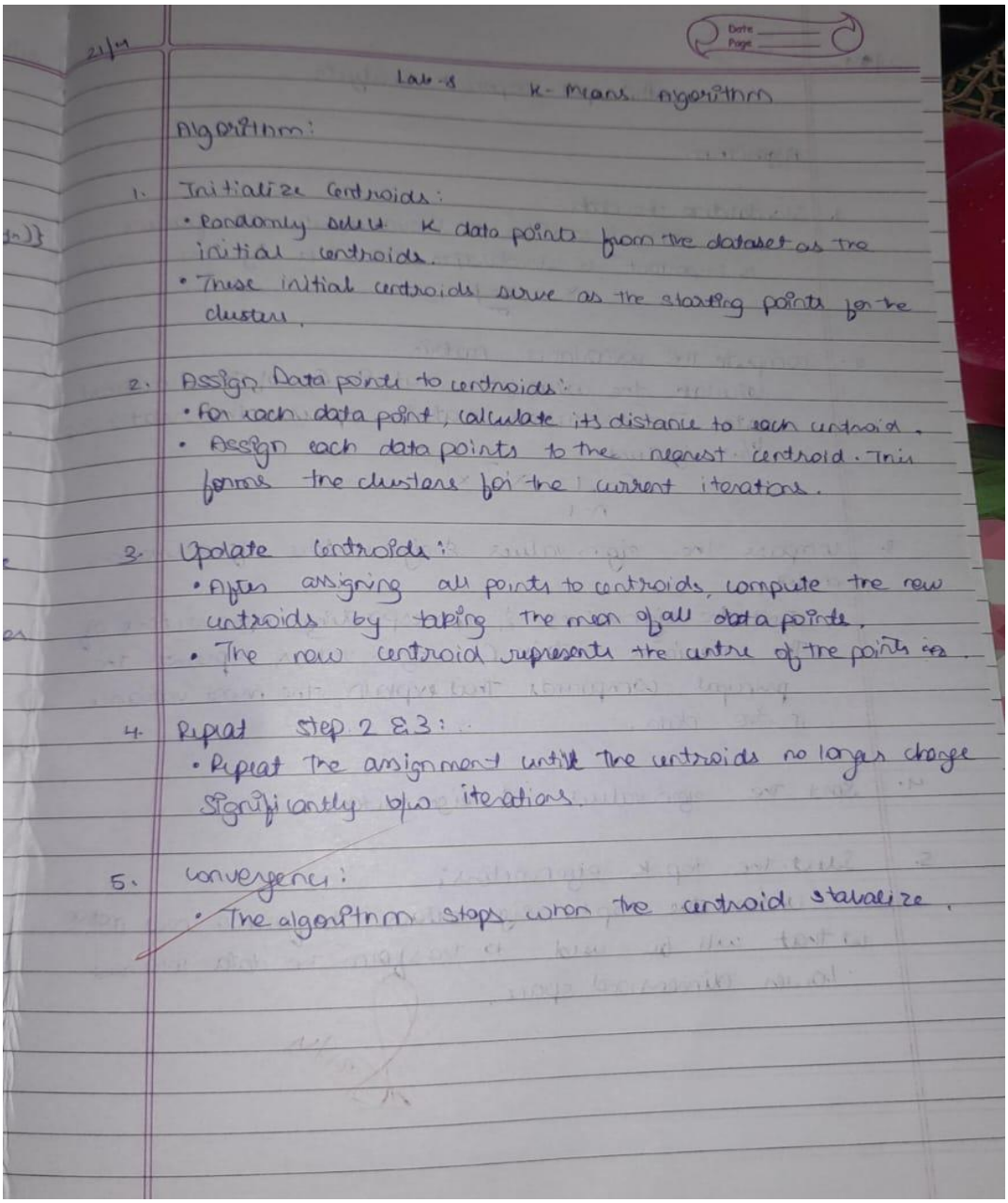
```
        else:
            predictions[X[:, self.feature_index] > self.threshold] = 0
        return predictions

# AdaBoost training
def adaboost(X, y, n_clf=10):
    n_samples, n_features = X.shape
    weights = np.ones(n_samples) / n_samples
    classifiers = []

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

        for feature in range(n_features):
            feature_values = np.unique(X[:, feature])
            for threshold in feature_values:
                for polarity in [1, -1]:
                    pred = np.ones(n_samples)
                    if polarity == 1:
                        pred[X[:, feature] < threshold] = 0
                    else:
                        pred[X[:, feature] > threshold] = 0
                    error = np.sum(weights[pred != y])
```

```
                if error < min_error:
                    clf.polarity = polarity
                    clf.threshold = threshold
                    clf.feature_index = feature
                    min_error = error

        EPS = 1e-10
        clf.alpha = 0.5 * np.log((1.0 - min_error) / (min_error + EPS))
        predictions = clf.predict(X)
        weights *= np.exp(-clf.alpha * y * (2 * predictions - 1))
        weights /= np.sum(weights)
        classifiers.append(clf)
    return classifiers
# Prediction function
def predict(X, classifiers):
    clf_preds = [clf.alpha * (2 * clf.predict(X) - 1) for clf in classifiers]
    y_pred = np.sign(np.sum(clf_preds, axis=0))
    return ((y_pred + 1) // 2).astype(int)
# Train and test
classifiers = adaboost(X_train, y_train, n_clf=10)
y_pred = predict(X_test, classifiers)
accuracy = np.mean(y_pred == y_test)
print("AdaBoost Accuracy (manual):", round(accuracy * 100, 2), "%")
```

Program-10

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

Screenshots:



Lab-8 K-Means algorithm

Algorithm:

1. Initialize centroids:
   - Randomly select K data points from the dataset as the initial centroids.
   - These initial centroids serve as the starting points for the clusters.
2. Assign Data points to centroids:
   - For each data point, calculate its distance to each centroid.
   - Assign each data points to the nearest centroid. This forms the clusters for the current iterations.
3. Update centroids:
   - After assigning all points to centroids, compute the new centroids by taking the mean of all data points.
   - The new centroid represents the centre of the points in.
4. Repeat step 2 & 3:
   - Repeat the assignment until the centroids no longer change significantly b/w iterations.
5. Convergence:
   - The algorithm stops when the centroid stabalize.

Code:

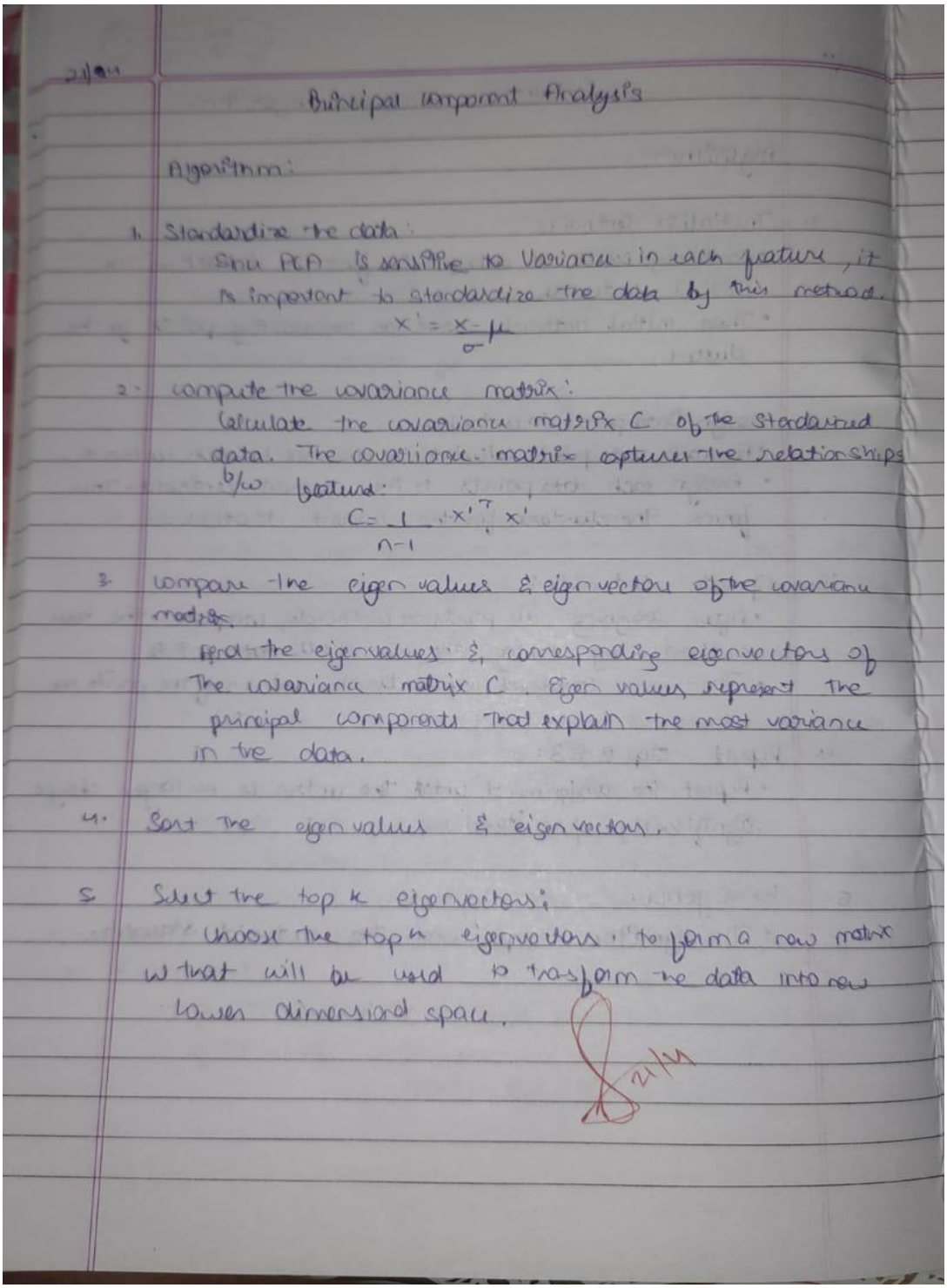
```
import pandas as pd

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:

    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

Screenshots:



Principal component Analysis

Algorithm:

1. Standardize the data:
   Since PCA is sensitive to variance in each feature, it is important to standardize the data by this method.
   $$x' = \frac{x - \mu}{\sigma}$$
2. Compute the covariance matrix:
   Calculate the covariance matrix C of the standardized data. The covariance matrix captures the relationships b/w features:
   $$C = \frac{1}{n-1} \cdot x'^{T} \cdot x'$$
3. Compare the eigen values & eigen vectors of the covariance matrix.
   Find the eigenvalues & corresponding eigenvectors of the covariance matrix C. Eigen values represent the principal components that explain the most variance in the data.
4. Sort the eigen values & eigen vectors.
5. Select the top k eigenvectors:
   Choose the top k eigenvectors to form a new matrix W that will be used to transform the data into new lower dimensional space.

Code:

```
import pandas as pd

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

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()
```