VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Machine Learning (23CS6PCMAL)

Submitted by

Pranav Y (1BM22CS204)

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 **Pranav Y (1BM22CS000),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Machine Learning (23CS6PCMAL) work prescribed for the said degree.

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

Index

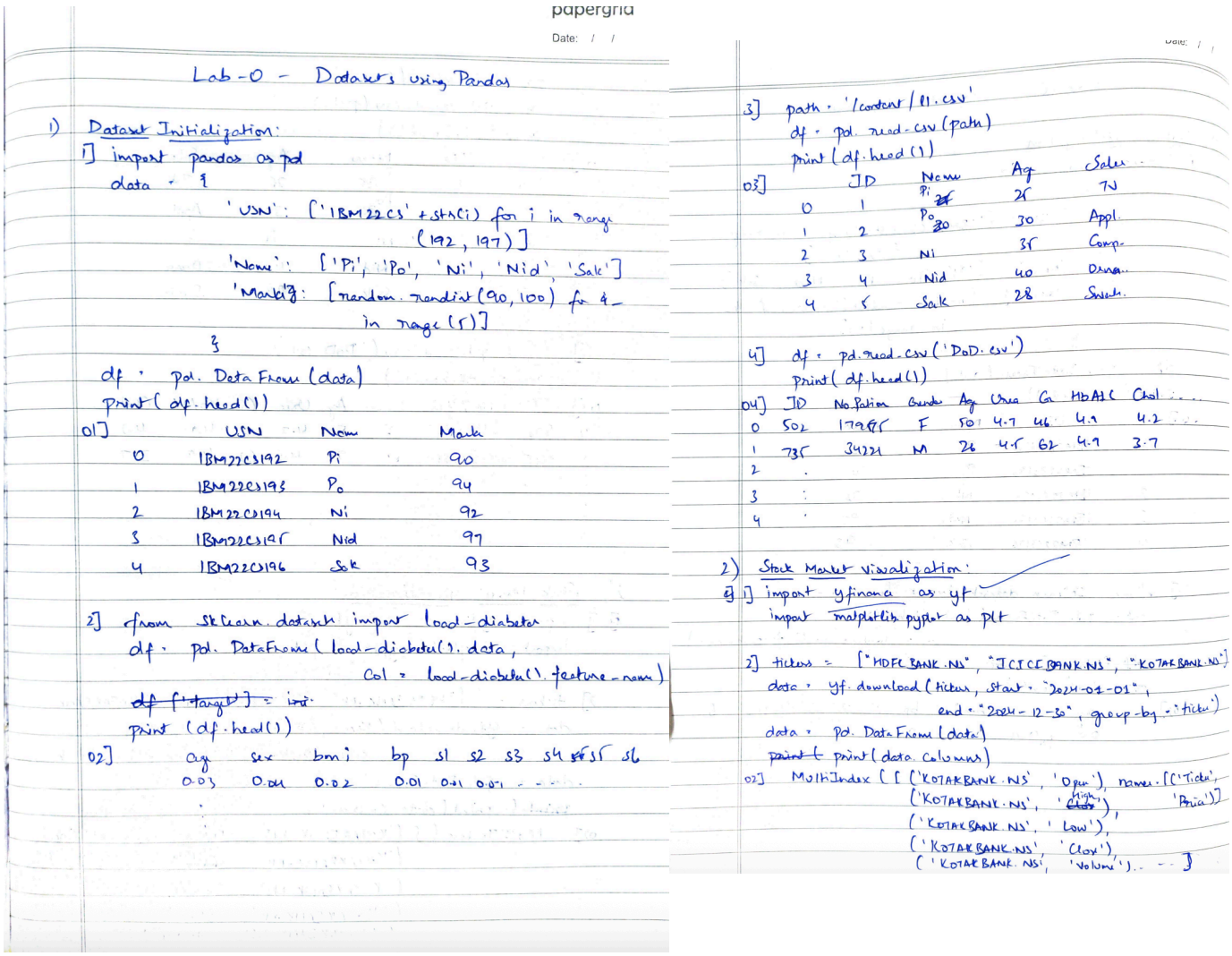
Github Link:

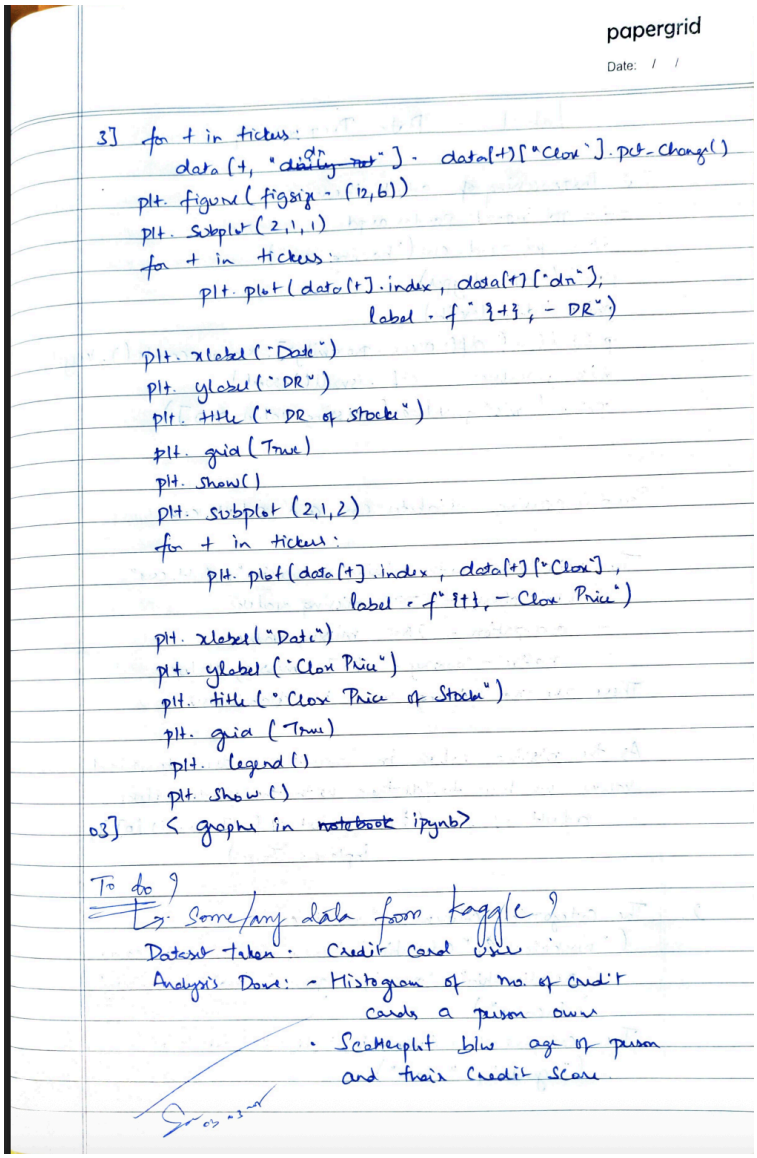
https://github.com/PranavY204/ML

Program 1

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

Screenshot





papergrid

Date: / /

```
3] for t in tickers:
       data[t, 'daily ret'] = data[t]['Close'].pct_change()
   plt.figure(figsize = (12,6))
   plt.subplot(2,1,1)
   for t in tickers:
       plt.plot(data[t].index, data[t]['dr'],
                label = f"{t} - DR")
   plt.xlabel("Date")
   plt.ylabel("DR")
   plt.title("DR of Stocks")
   plt.grid(True)
   plt.show()
   plt.subplot(2,1,2)
   for t in tickers:
       plt.plot(data[t].index, data[t]['Close'],
                label = f"{t} - Close Price")
   plt.xlabel("Date")
   plt.ylabel("Close Price")
   plt.title("Close Price of Stocks")
   plt.grid(True)
   plt.legend()
   plt.show()
03] < graphs in notebook ipynb>
```

To do ?

1. Some/any data from kaggle ?

Dataset taken: Credit card use

Analysis Done: - Histogram of no. of credit cards a person owns

- Scatterplot blw age of person and their Credit score

Code

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

# Dataset Initializations
df1 = pd.DataFrame({
   "USN": ["1BM22CS" + str(i) for i in range (200, 211)] # 10 records
   "Name": [i for i in "abcdefghij"]
   "Marks": [random.randint(0, 100) for i in range(10)]
})
print(df1.head())

from sklearn.datasets import load_diabetes
df2 = pd.DataFram(load_diabetes())
print(df2.head())
```

```
# Step 2: Downloading Stock Market Data
# Define the ticker symbols for Indian companies
# Example: Reliance Industries (RELIANCE.NS), TCS (TCS.NS), Infosys (INFY.NS)
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')
data = pd.DataFrame(data)
print("\n", type(data))
# Display the first 5 rows of the dataset
print("First 5 rows of the dataset:")
print(data.head())

print(f"Shape of dataset:\n{data.shape}")
print(f"Column names:\n{data.columns}")
print(f"Summary of Reliance Stats:\n{data['RELIANCE.NS'].describe()}")

reliance_data = data['RELIANCE.NS']
reliance_data["daily_returns"] = reliance_data["Close"].pct_change()
print("Daily Returns:")
print(reliance_data["daily_returns"])

plt.figure(figsize=(12, 6))
plt.subplot(2,1,1)
reliance_data["Close"].plot(title="Reliance - Closing Price")
plt.subplot(2, 1, 2)
reliance_data["daily_returns"].plot(title='Reliance - Daily Returns', color="orange")
plt.tight_layout()
plt.show()

tickers = ["HDFCBANK.NS", "ICICIBANK.NS", "KOTAKBANK.NS"]
data = yf.download(tickers, start="2024-01-01", end="2024-12-30", group_by='ticker')
data = pd.DataFrame(data)
print("\n", type(data))
print(data.describe())

print(data.columns)

for t in tickers:
    # Calculate daily returns first
    data[t, "daily_returns"] = data[t]["Close"].pct_change()
    # Then, print them
    print(f"Daily Returns for {t}:")
    print(data[t]["daily_returns"])
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
for t in tickers:
    plt.plot(data[t].index, data[t]["daily_returns"], label=f"{t} - Daily Returns")
plt.xlabel("Date")
plt.ylabel("Daily Returns")
plt.title("Daily Returns of Selected Stocks")
plt.grid(True)
plt.legend()
plt.show()
plt.subplot(2, 1, 2)
```

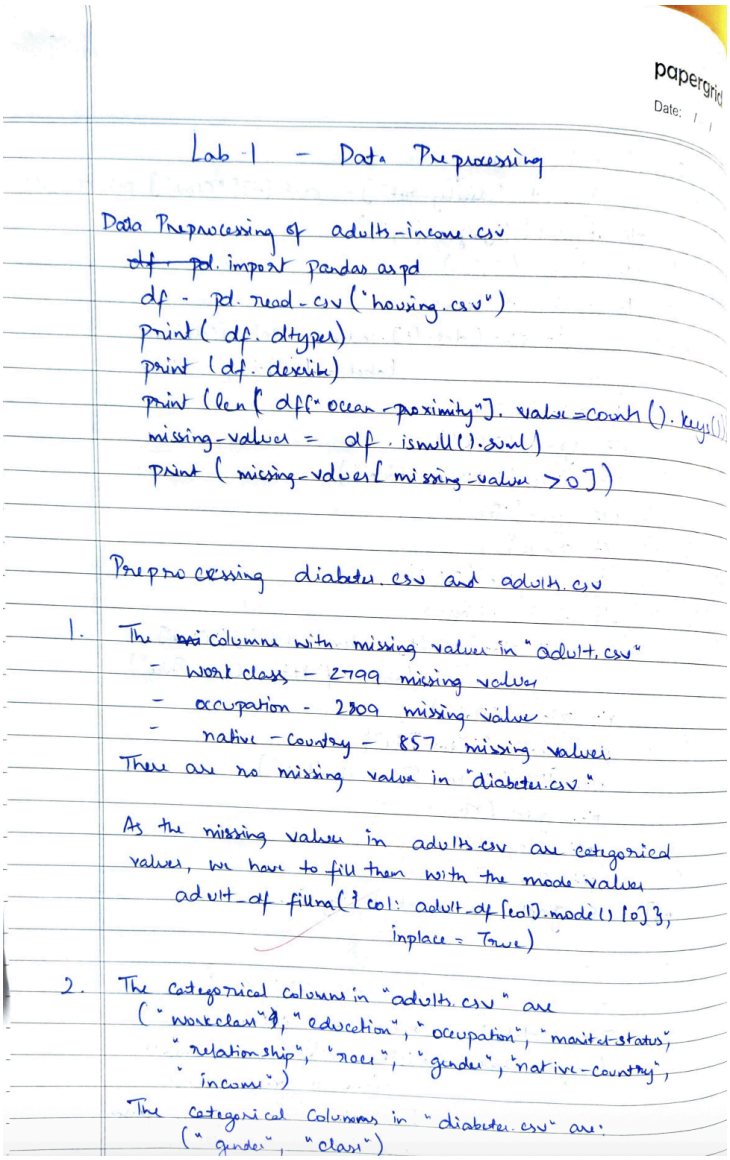
```
for t in tickers:
    plt.plot(data[t].index, data[t]["Close"], label=f"{t} - Close Price")
# plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel("Date")
plt.ylabel("Close Price")
plt.title("Close Price of Selected Stocks")
plt.grid(True)
plt.legend()
plt.show()



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

import pandas as pd
df = pd.read_csv("/content/drive/My Drive/users_data.csv")
print(df.head())

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
df["num_credit_cards"].plot(kind="hist", bins=10, title="Distribution of Number of Credit Cards")
plt.xlabel("Number of Credit Cards")
plt.ylabel("Frequency")
plt.show()

import numpy as np

plt.scatter(df["current_age"], df["credit_score"], marker="o")
plt.title("Age vs. Credit Score")
plt.xlabel("Age")
plt.ylabel("Credit Score")
plt.grid(True)
plt.show()

x = df["current_age"]
y = df["credit_score"]

b, a = np.polyfit(x, y, deg=1)

# Create sequence of 100 numbers from 0 to 100
xseq = np.linspace(0, 100, 100)

# Plot regression line
plt.plot(xseq, a + b * xseq, color="k", lw=2.5)
```

(You should repeat as above, for all the remaining programs from 2 to 11)

Program 2

Demonstrate various data pre-processing techniques for a given dataset

Screenshot



Lab-1 - Data Preprocessing

Data Preprocessing of adult-income.csv

```
df - pd. import pandas as pd
df - pd.read_csv('housing.csv')
print(df.dtypes)
print(df.describe)
print(len(df["ocean_proximity"].value_counts().keys())
missing_values = df.isnull().sum()
print(missing_values[missing_values > 0])
```

Preprocessing diabetes.csv and adult.csv

1. The ~~no~~ columns with missing values in "adult.csv"
   - work class - 2799 missing values
   - occupation - 2809 missing values
   - native-country - 857 missing values

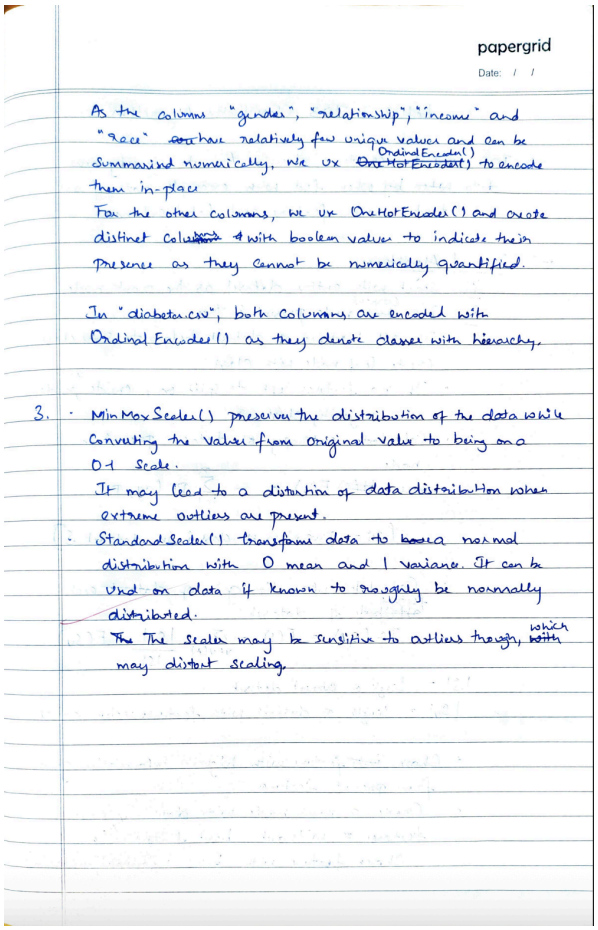
   There are no missing value in "diabetes.csv".

   As the missing values in adults.csv are categorical values, we have to fill them with the mode value

```
adult_df.fillna({col: adult_df[col].mode()[0]},
                inplace = True)
```

2. The categorical columns in "adults.csv" are ("workclass", "education", "occupation", "marital-status", "relationship", "race", "gender", "native-country", "income")

   The categorical columns in "diabetes.csv" are: ("gender", "class")



As the columns "gender", "relationship", "income" and "race" ~~are~~ have relatively few unique values and can be summarized numerically, we use ~~One Hot Encoder~~ OrdinalEncoder() to encode them in-place

For the other columns, we use OneHotEncoder() and create distinct columns with boolean values to indicate their presence as they cannot be numerically quantified.

In "diabetes.csv", both columns are encoded with OrdinalEncoder() as they denote classes with hierarchy.

3. · MinMaxScaler() preserves the distribution of the data while converting the values from original value to being on a 0-1 scale.
   It may lead to a distortion of data distribution when extreme outliers are present.
   · StandardScaler() transforms data to ~~be~~ a normal distribution with 0 mean and 1 variance. It can be used on data if known to roughly be normally distributed.
   ~~The~~ The scaler may be sensitive to outliers though, ~~with~~ which may distort scaling.

Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from scipy import stats
```

```
def createdata():
  data = {
      'Age': np.random.randint(18, 70, size=20),
      'Salary': np.random.randint(30000, 120000, size=20),
      'Purchased': np.random.choice([0, 1], size=20),
      'Gender': np.random.choice(['Male', 'Female'], size=20),
      'City': np.random.choice(['New York', 'San Francisco', 'Los Angeles'], size=20)
  }

  df = pd.DataFrame(data)
  return df

df = createdata()
df.head(10)

df.shape

# Introduce some missing values for demonstration
df.loc[5, 'Age'] = np.nan
df.loc[10, 'Salary'] = np.nan
df.head(10)

# Basic information about the dataset
print(df.info())

# Summary statistics
print(df.describe())

#Code to Find Missing Values
# Check for missing values in each column
missing_values = df.isnull().sum()

# Display columns with missing values
print(missing_values[missing_values > 0])

#Set the values to some value (zero, the mean, the median, etc.).
# Step 1: Create an instance of SimpleImputer with the median strategy for Age and mean stratergy for Salary
imputer1 = SimpleImputer(strategy="median")
imputer2 = SimpleImputer(strategy="mean")

df_copy=df

# Step 2: Fit the imputer on the "Age" and "Salary"column
# Note: SimpleImputer expects a 2D array, so we reshape the column
imputer1.fit(df_copy[["Age"]])
imputer2.fit(df_copy[["Salary"]])

# Step 3: Transform (fill) the missing values in the "Age" and "Salary"c column
df_copy["Age"] = imputer1.transform(df[["Age"]])
df_copy["Salary"] = imputer2.transform(df[["Salary"]])

# Verify that there are no missing values left
```

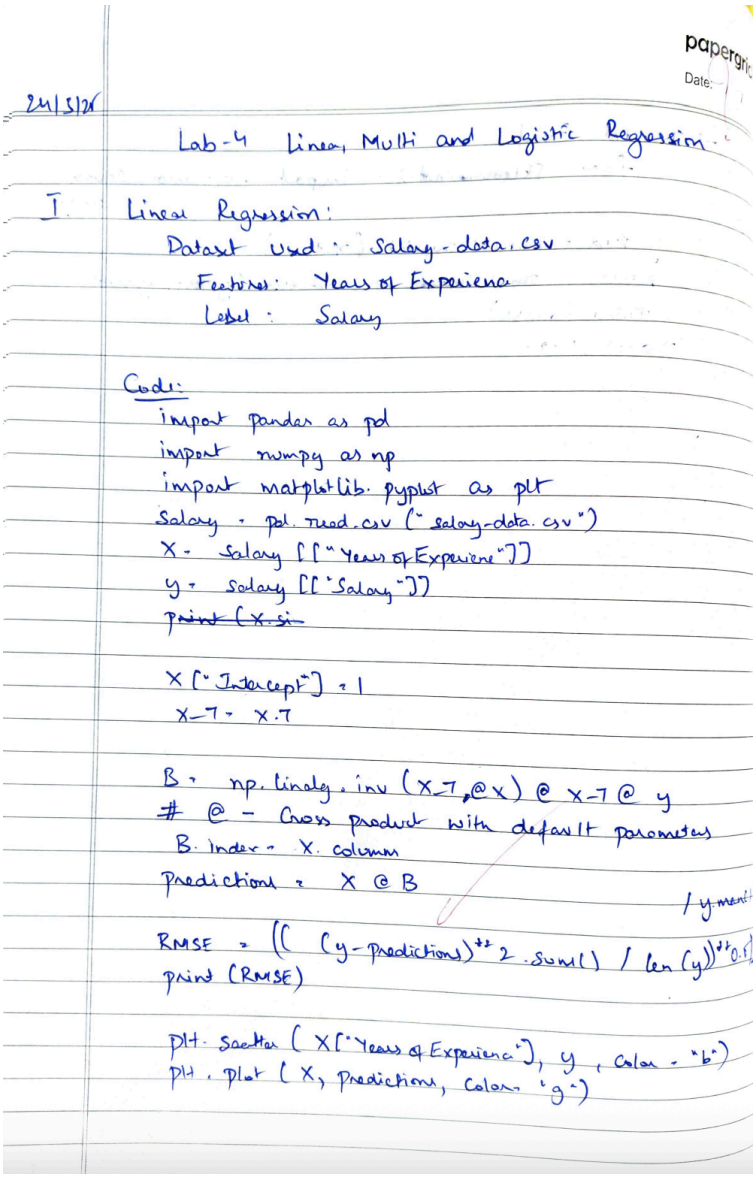
```
print(df_copy["Age"].isnull().sum())
print(df_copy["Salary"].isnull().sum())

#Handling Categorical Attributes
#Using Ordinal Encoding for gender COlumn and One-Hot Encoding for City Column

# Initialize OrdinalEncoder
ordinal_encoder = OrdinalEncoder(categories=[["Male", "Female"]])
# Fit and transform the data
df_copy["Gender_Encoded"] = ordinal_encoder.fit_transform(df_copy[["Gender"]])

# Initialize OneHotEncoder
onehot_encoder = OneHotEncoder()

# Fit and transform the "City" column
encoded_data = onehot_encoder.fit_transform(df[["City"]])

# Convert the sparse matrix to a dense array
encoded_array = encoded_data.toarray()

# Convert to DataFrame for better visualization
encoded_df = pd.DataFrame(encoded_array, columns=onehot_encoder.get_feature_names_out(["City"]))
df_encoded = pd.concat([df_copy, encoded_df], axis=1)

df_encoded.drop("Gender", axis=1, inplace=True)
df_encoded.drop("City", axis=1, inplace=True)

print(df_encoded. head())

#Data Transformation
# Min-Max Scaler/Normalization (range 0-1)
#Pros: Keeps all data between 0 and 1; ideal for distance-based models.
#Cons: Can distort data distribution, especially with extreme outliers.
normalizer = MinMaxScaler()
df_encoded[['Salary']] = normalizer.fit_transform(df_encoded[['Salary']])
df_encoded.head()

# Standardization (mean=0, variance=1)
#Pros: Works well for normally distributed data; suitable for many models.
#Cons: Sensitive to outliers.
scaler = StandardScaler()
df_encoded[['Age']] = scaler.fit_transform(df_encoded[['Age']])
df_encoded.head()

#Removing Outliers
# Outlier Detection and Treatment using IQR
#Pros: Simple and effective for mild outliers.
#Cons: May overly reduce variation if there are many extreme outliers.
df_encoded_copy1=df_encoded
df_encoded_copy2=df_encoded
df_encoded_copy3=df_encoded
```

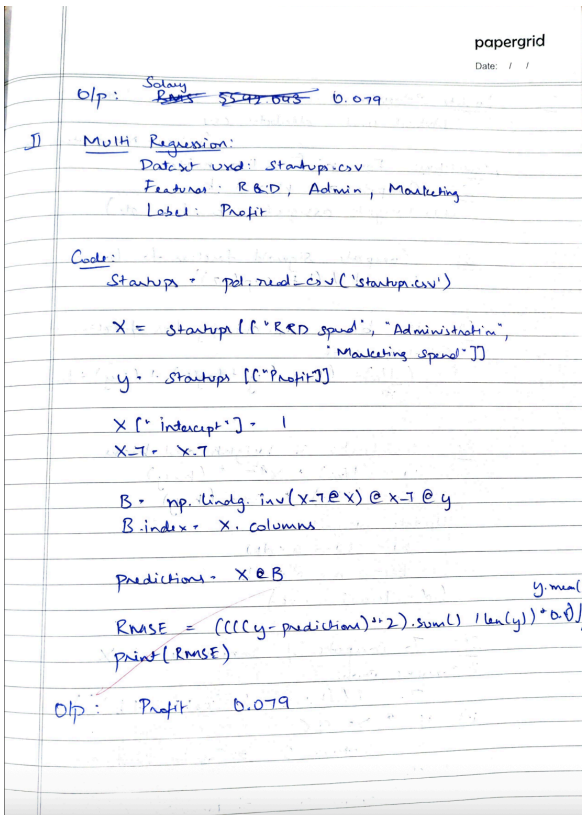
```
Q1 = df_encoded_copy1['Salary'].quantile(0.25)
Q3 = df_encoded_copy1['Salary'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
df_encoded_copy1['Salary'] = np.where(df_encoded_copy1['Salary'] > upper_bound, upper_bound,
                    np.where(df_encoded_copy1['Salary'] < lower_bound, lower_bound, df_encoded_copy1['Salary']))

print(df_encoded_copy1.head())

#Removing Outliers
# Z-score method
#Pros: Good for normally distributed data.
#Cons: Not suitable for non-normal data; may miss outliers in skewed distributions.

df_encoded_copy2['Salary_zscore'] = stats.zscore(df_encoded_copy2['Salary'])
df_encoded_copy2['Salary'] = np.where(df_encoded_copy2['Salary_zscore'].abs() > 3, np.nan,
df_encoded_copy2['Salary'])  # Replace outliers with NaN
print(df_encoded_copy2.head())

#Removing Outliers
# Median replacement for outliers
#Pros: Keeps distribution shape intact, useful when capping isn't feasible.
#Cons: May distort data if outliers represent real phenomena.
df_encoded_copy3['Salary_zscore'] = stats.zscore(df_encoded_copy3['Salary'])
median_salary = df_encoded_copy3['Salary'].median()
df_encoded_copy3['Salary'] = np.where(df_encoded_copy3['Salary_zscore'].abs() > 3, median_salary,
df_encoded_copy3['Salary'])
print(df_encoded_copy3.head())
```

Program 3

Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

Screenshot





Code

```
import pandas as pd
import numpy as np

salary = pd.read_csv('Salary_Data.csv')
print(salary.head())

X = salary[["YearsExperience"]]
y = salary[["Salary"]]
print(X.size, y.size)
```

```
X["intercept"] = 1

X.head()

X_T = X.T

X_T

B = np.linalg.inv(X_T @ X) @ X_T @ y

B.index = X.columns

predictions = X @ B
print(predictions)

SSR = ((((y - predictions) ** 2).sum() / len(y))**0.5) / y.mean()
print(SSR)

import matplotlib.pyplot as plt

plt.scatter(X['YearsExperience'], y, color="b")
plt.plot(X, predictions, color="g")
```

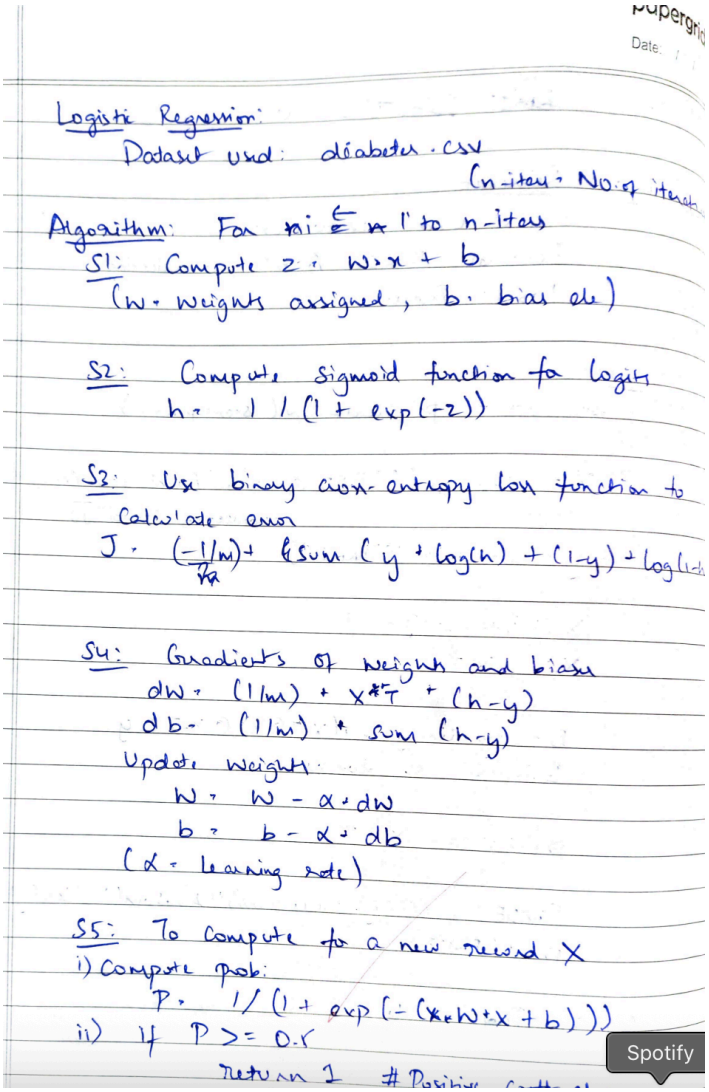
Multiple Regression:

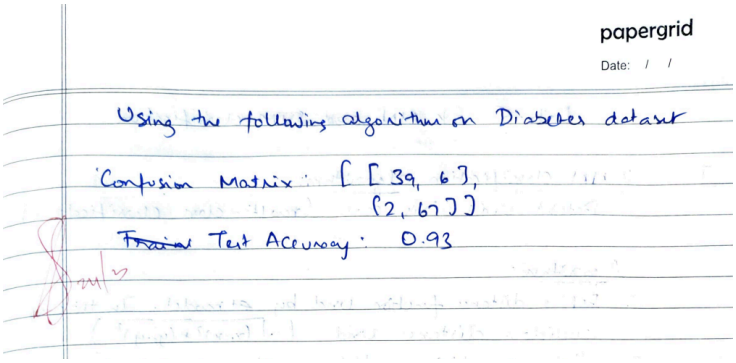
```
startups = pd.read_csv('50_Startups.csv')
print(startups.head())

X = startups[["R&D Spend", "Administration", "Marketing Spend"]]
y = startups[["Profit"]]
print(X.shape, y.size)

X["intercept"] = 1

X_T = X.T

B = np.linalg.inv(X_T @ X) @ X_T @ y
B.index = X.columns
print(B)

predictions = X @ B
print(predictions)

# plt.scatter(X['R&D Spend'], y, color="b")
# plt.scatter(X['Administration'], y, color="y")
# plt.scatter(X['Marketing Spend'], y, color="r")
# plt.plot(X, predictions, color="g")
RMSE= ((((y - predictions) ** 2).sum() / len(y))**0.5) / y.mean()
print(RMSE)
```

Program 4

Build Logistic Regression Model for a given dataset

Screenshot





Code

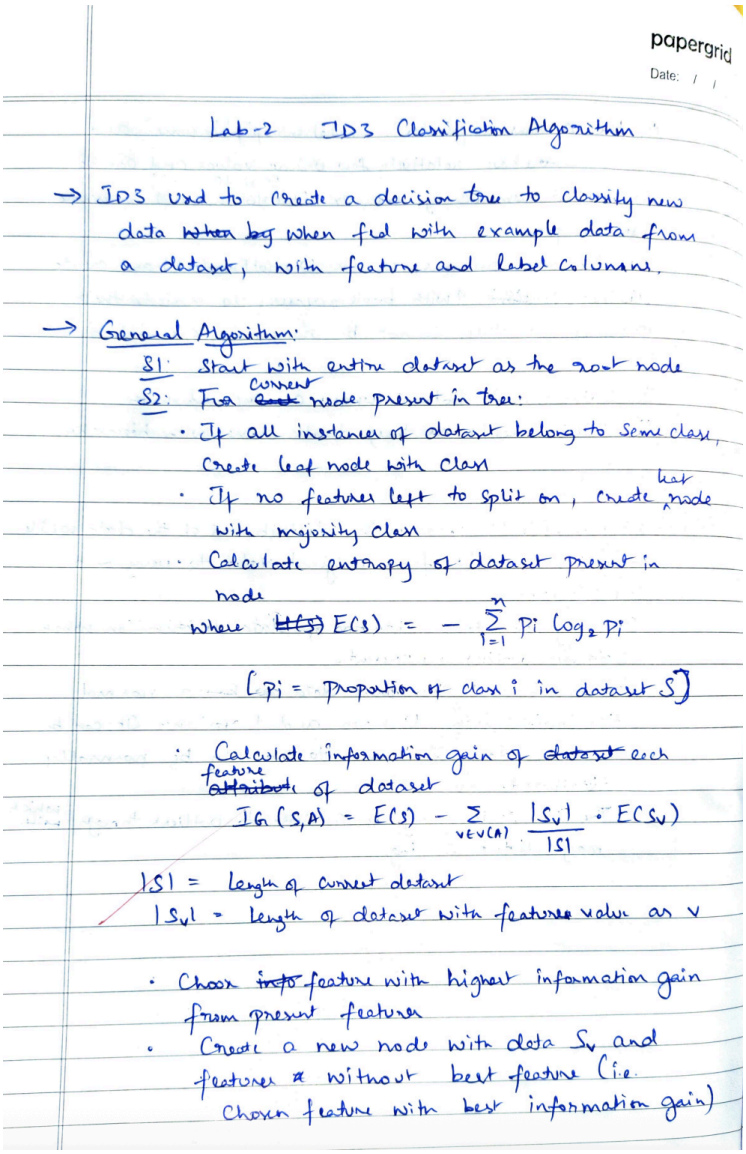
```
class LogisticRegression:
    def __init__(self, learning_rate=0.001, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
        self.losses = []

    #Sigmoid method
    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def compute_loss(self, y_true, y_pred):
        # binary cross entropy
        epsilon = 1e-9
        y1 = y_true * np.log(y_pred + epsilon)
        y2 = (1-y_true) * np.log(1 - y_pred + epsilon)
        return -np.mean(y1 + y2)

    def feed_forward(self,X):
        z = np.dot(X, self.weights) + self.bias
        A = self._sigmoid(z)
        return A

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

        # init parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        # gradient descent
        for _ in range(self.n_iters):
            A = self.feed_forward(X)
            self.losses.append(self.compute_loss(y,A))
            dz = A - y # derivative of sigmoid and bce X.T*(A-y)
            # compute gradients
            dw = (1 / n_samples) * np.dot(X.T, dz)
            db = (1 / n_samples) * np.sum(dz)
            # update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        threshold = .5
        y_hat = np.dot(X, self.weights) + self.bias
        y_predicted = self._sigmoid(y_hat)
        y_predicted_cls = [1 if i > threshold else 0 for i in y_predicted]

        return np.array(y_predicted_cls)
```

```
from sklearn.model_selection import train_test_split
from sklearn import datasets

dataset = datasets.load_breast_cancer()
X, y = dataset.data, dataset.target

X, y = dataset.data, dataset.target
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=1234
)

def confusion_matrix(y_actual, y_predicted):
    tp = 0
    tn = 0
    fp = 0
    fn = 0
    epsilon = 1e-9
    for i in range(len(y_actual)):
        if y_actual[i] > 0:
            if y_actual[i] == y_predicted[i]:
                tp = tp + 1
            else:
                fn = fn + 1
        if y_actual[i] < 1:
            if y_actual[i] == y_predicted[i]:
                tn = tn + 1
            else:
                fp = fp + 1

    cm = [[tn, fp], [fn, tp]]
    accuracy = (tp+tn)/(tp+tn+fp+fn+epsilon)
    sens = tp/(tp+fn+epsilon)
    prec = tp/(tp+fp+epsilon)
    f_score = (2*prec*sens)/(prec+sens+epsilon)
    return cm,accuracy,sens,prec,f_score

regressor = LogisticRegression(learning_rate=0.0001, n_iters=1000)
regressor.fit(X_train, y_train)
predictions = regressor.predict(X_test)
cm ,accuracy,sens,precision,f_score  = confusion_matrix(np.asarray(y_test), np.asarray(predictions))
print("Test accuracy: {0:.3f}".format(accuracy))
print("Confusion Matrix:",cm)
```

Program 5

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

Screenshot



Lab-2 ID3 Classification Algorithm

→ ID3 used to create a decision tree to classify new data when fed with example data from a dataset, with feature and label columns.

→ General Algorithm:

S1: Start with entire dataset as the root node

S2: For current node present in tree:

- If all instances of dataset belong to same class, Create leaf node with class
- If no features left to split on, create leaf node with majority class
- Calculate entropy of dataset present in node

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

[p_i = Proportion of class i in dataset S]

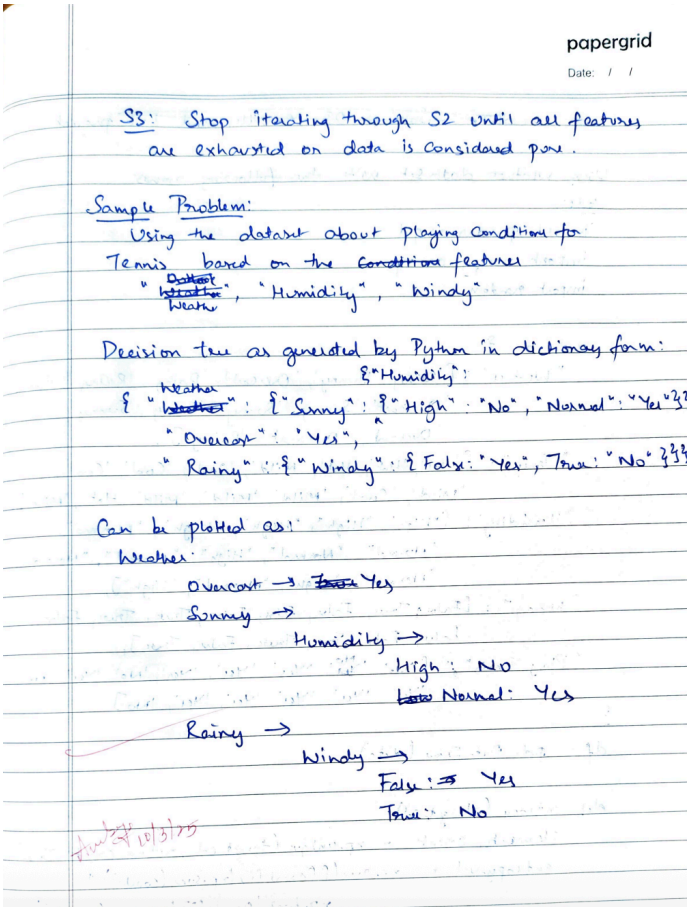
- Calculate information gain of each feature of dataset

$IG(S,A) = E(S) - \sum_{v \in V(A)} \frac{|S_v|}{|S|} \cdot E(S_v)$

|S| = Length of current dataset

$|S_v|$ = Length of dataset with feature value as v

- Choose feature with highest information gain from present features
- Create a new node with data S_v and features = without best feature (i.e. Chosen feature with best information gain)



S3: Stop iterating through S2 until all features are exhausted or data is considered pure.

Sample Problem:

Using the dataset about playing conditions for Tennis based on the features "Weather", "Humidity", "Windy"

Decision tree as generated by Python in dictionary form:

{"Humidity":
{"Weather": {"Sunny": {"High": "No", "Normal": "Yes"},
"Overcast": "Yes",
"Rainy": {"Windy": {False: "Yes", True: "No"}}}}}

Can be plotted as:

Weather:
Overcast → Yes
Sunny →
Humidity →
High: No
Normal: Yes
Rainy →
Windy →
False: Yes
True: No

Code

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt
import math

df = pd.read_csv('DoD.csv')
df.head()

print(df.dtypes)
```

```
def calculate_entropy(data, target_column):
    total_rows = len(data)
    target_values = data[target_column].unique()

    entropy = 0
    for value in target_values:
        # Calculate the proportion of instances with the current value
        value_count = len(data[data[target_column] == value])
        proportion = value_count / total_rows
        entropy -= proportion * math.log2(proportion)

    return entropy

print(f"Total Entropy of dataset: {calculate_entropy(df, 'Play Tennis')}")

def calculate_information_gain(data, feature, target_column):

    # Calculate weighted average entropy for the feature
    unique_values = data[feature].unique()
    weighted_entropy = 0
    entropy_outcome = calculate_entropy(data, target_column)

    for value in unique_values:
        subset = data[data[feature] == value]
        proportion = len(subset) / len(data)
        weighted_entropy += proportion * calculate_entropy(subset, target_column)

    # Calculate information gain
    information_gain = entropy_outcome - weighted_entropy

    return information_gain

for col in df.columns[:-1]:
    print(f"Information gain for {col}: {calculate_information_gain(df, col, 'Play Tennis')}")

def id3(data, target_column, features):
    if len(data[target_column].unique()) == 1:
        return data[target_column].iloc[0]


    if len(features) == 0:
        return data[target_column].mode().iloc[0]

    best_feature = max(features, key=lambda x: calculate_information_gain(data, x, target_column))

    tree = {best_feature: {}}

    features = [f for f in features if f != best_feature]

    for value in data[best_feature].unique():
        subset = data[data[best_feature] == value]
```

```
        tree[best_feature][value] = id3(subset, target_column, features)

    return tree

tree = id3(df, 'Play Tennis', df.columns[:-1])
print(tree)

import pandas as pd
import numpy as np
import random

# Define the dataset
data = {
    'Weather': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Rainy', 'Sunny',
'Overcast', 'Overcast', 'Rainy'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal', 'Normal',
'High', 'Normal', 'High'],
    'Windy': [False, True, False, False, False, True, True, False, False, False, True, True, False, True],
    'Play Tennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
}

df = pd.DataFrame(data)

def entropy(target_col):
    elements, counts = np.unique(target_col, return_counts=True)
    entropy_val = -np.sum([(counts[i] / np.sum(counts)) * np.log2(counts[i] / np.sum(counts)) for i in
range(len(elements))])
    return entropy_val

def information_gain(data, split_attribute_name, target_name):
    total_entropy = entropy(data[target_name])
    vals, counts= np.unique(data[split_attribute_name], return_counts=True)
    weighted_entropy = np.sum([(counts[i] / np.sum(counts)) *
entropy(data.where(data[split_attribute_name]==vals[i]).dropna()[target_name]) for i in range(len(vals))])
    information_gain_val = total_entropy - weighted_entropy
    return information_gain_val

def id3_algorithm(data, original_data, features, target_attribute_name, parent_node_class):
    # Base cases
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
    elif len(data) == 0:
        return
np.unique(original_data[target_attribute_name])[np.argmax(np.unique(original_data[target_attribute_name],
return_counts=True)[1])]
    elif len(features) == 0:
        return parent_node_class
    else:
        parent_node_class =
np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],
return_counts=True)[1])]
```

```
        item_values = [information_gain(data, feature, target_attribute_name) for feature in features]
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]
        tree = {best_feature: {}}
        features = [i for i in features if i != best_feature]
        for value in np.unique(data[best_feature]):
            value = value
            sub_data = data.where(data[best_feature] == value).dropna()
            subtree = id3_algorithm(sub_data, data, features, target_attribute_name, parent_node_class)
            tree[best_feature][value] = subtree
        return tree


def fit(df, target_attribute_name, features):
    return id3_algorithm(df, df, features, target_attribute_name, None)


tree = fit(df, 'Play Tennis', ['Weather', 'Temperature', 'Humidity', 'Windy'])
# accuracy = get_accuracy(test_data, tree)
print("Decision Tree:")
print(tree)
# print("Accuracy:", accuracy)

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]
```

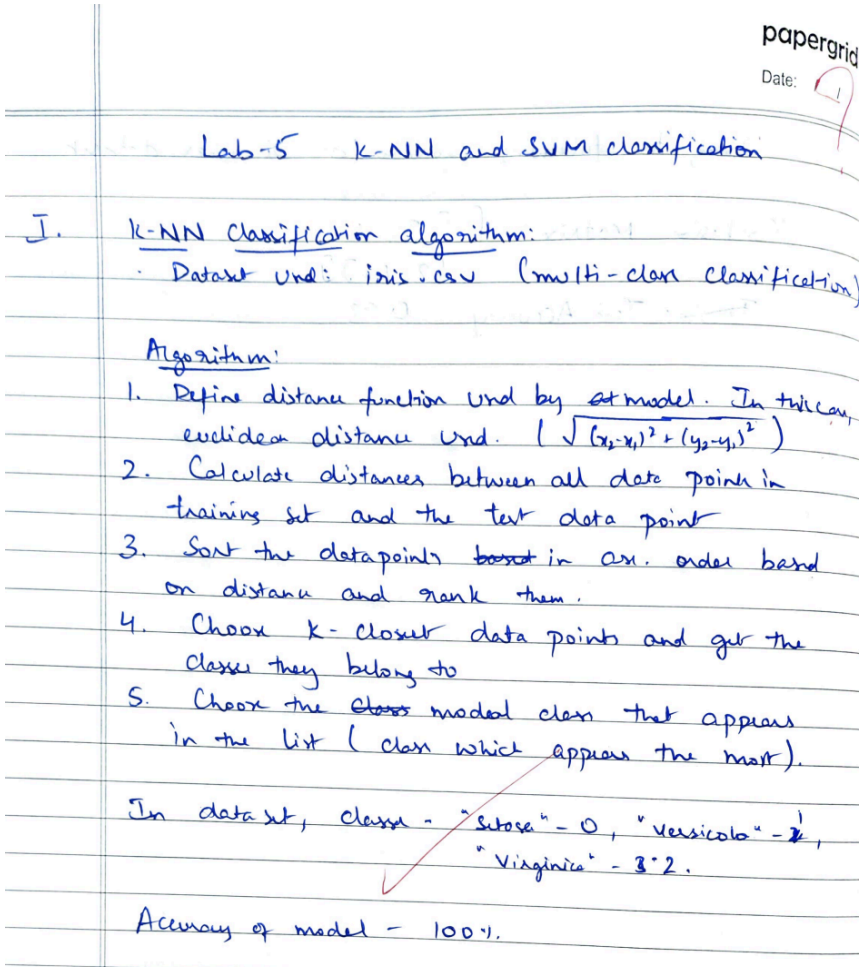
```
    if len(features) == 0:  # No more features to split on
        return data.iloc[:, -1].mode()[0]

    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

Screenshot



Code

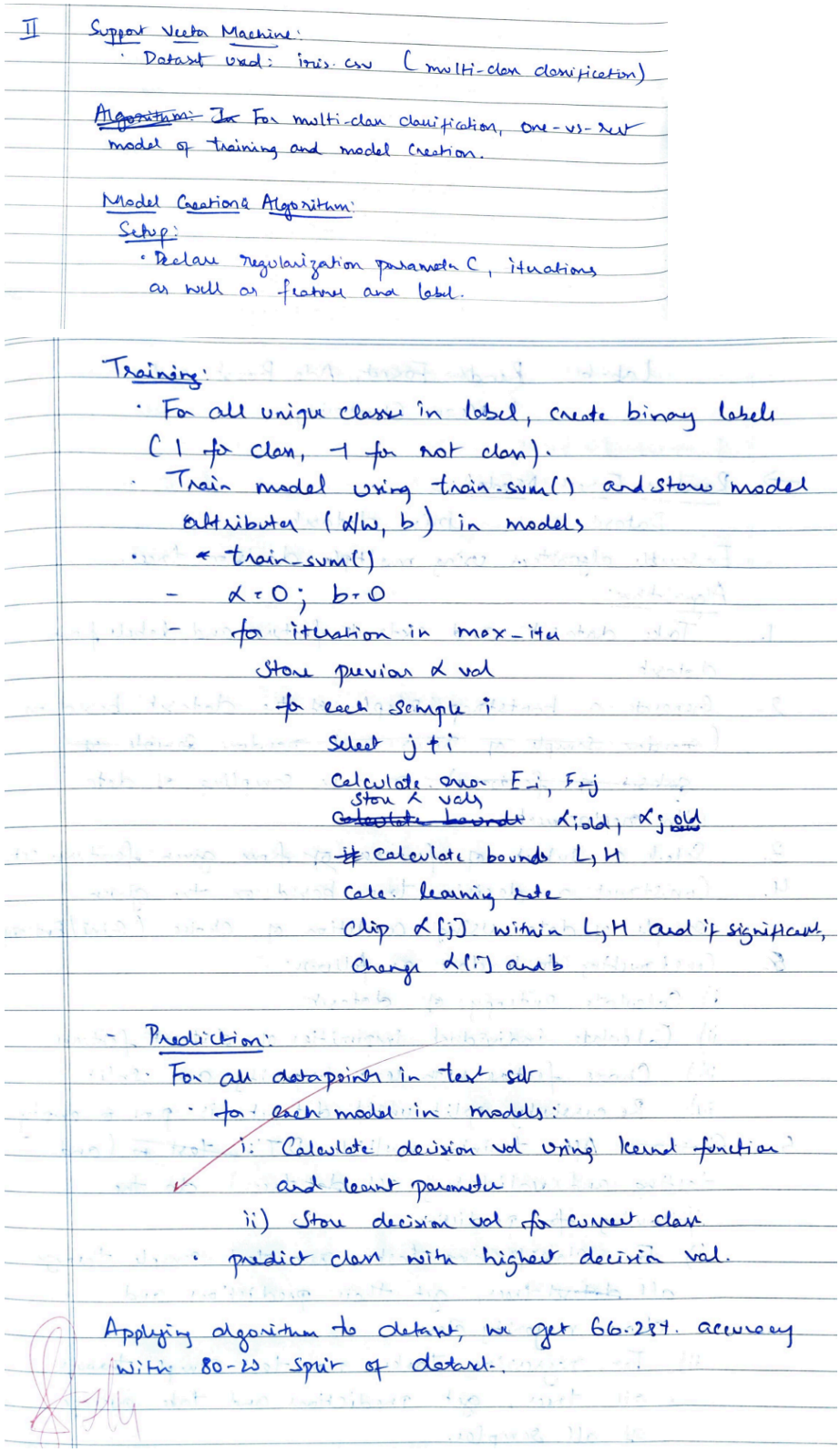
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

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

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

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

```
    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        # Compute distances between x and all examples in the training set
        distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]
        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Extract the labels of the k nearest neighbor training samples
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        # Return the most common class label
        most_common = np.bincount(k_nearest_labels).argmax()
        return most_common

df = pd.read_csv('iris.csv')
df['variety'].replace({'Setosa': 0, 'Versicolor': 1, 'Virginica': 2}, inplace=True)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, :-1].values, df.iloc[:, -1].values, test_size=0.2,
random_state=42)

model = KNN()
model.fit(X_train, y_train)

predicted_classes = model.predict(X_test)
actual_classes = y_test
print(predicted_classes, actual_classes)
correct = len([i for i in range(len(predicted_classes)) if predicted_classes[i] == actual_classes[i]])
accuracy = correct / len(predicted_classes)
print("Accuracy:", accuracy*100)
```

Program 7

Build Support vector machine model for a given dataset

Screenshot



II Support Vector Machine:

- Dataset used: iris.csv (multi-class classification)

Algorithm: For multi-class classification, one-vs-rest model of training and model creation.

Model Creation Algorithm:

Setup:

- Declare regularization parameter C, iterations as well as feature and label.

Training:

- For all unique classes in label, create binary labels (1 for class, -1 for not class).
- Train model using train_svm() and store model attributes (α/w, b) in models
- * train_svm()
  - $\alpha = 0$; $b = 0$
  - for iteration in max_iter
    - Store previous α val
    - for each sample i
      - Select j ≠ i
      - Calculate error E_i, E_j
      - Store α vals
      - ~~Calculate bounds~~ $\alpha_{i\,old}$, $\alpha_{j\,old}$
      - # Calculate bounds L, H
      - Calc. learning rate
      - Clip $\alpha[j]$ within L, H and if significant, Change $\alpha[i]$ and b

- Prediction:
  - For all datapoints in test set
    - for each model in models:
      - i. Calculate decision val using kernel function and learnt parameters
      - ii) Store decision val for current class
    - predict class with highest decision val.

Applying algorithm to dataset, we get 66.28% accuracy with 80-20 split of dataset.

Code

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

# Linear kernel function
def linear_kernel(x1, x2):
    return np.dot(x1, x2)

# Support Vector Machine (SVM) class
class SVM:
    def __init__(self, C=1, max_iter=1000):
        self.C = C  # Regularization parameter
        self.max_iter = max_iter
        self.alpha = None
        self.b = None
        self.kernel = linear_kernel
        self.X_train = None
        self.y_train = None

    # Fit the SVM model (train the model)
    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.X_train = X
        self.y_train = y

        # One-vs-rest strategy: Train a separate SVM for each class
        self.models = []
        self.classes = np.unique(y)

        for class_label in self.classes:
            y_binary = np.where(y == class_label, 1, -1)  # Class vs all other classes
            model = self._train_svm(X, y_binary)
            self.models.append(model)

    # Train a binary SVM for one class vs all others
    def _train_svm(self, X, y):
        n_samples, n_features = X.shape
        alpha = np.zeros(n_samples)
        b = 0

        for _ in range(self.max_iter):
            alpha_prev = np.copy(alpha)
            for i in range(n_samples):
                j = np.random.choice([x for x in range(n_samples) if x != i])

                E_i = self._compute_error(X, y, i, alpha, b)
                E_j = self._compute_error(X, y, j, alpha, b)

                alpha_i_old = alpha[i]
```

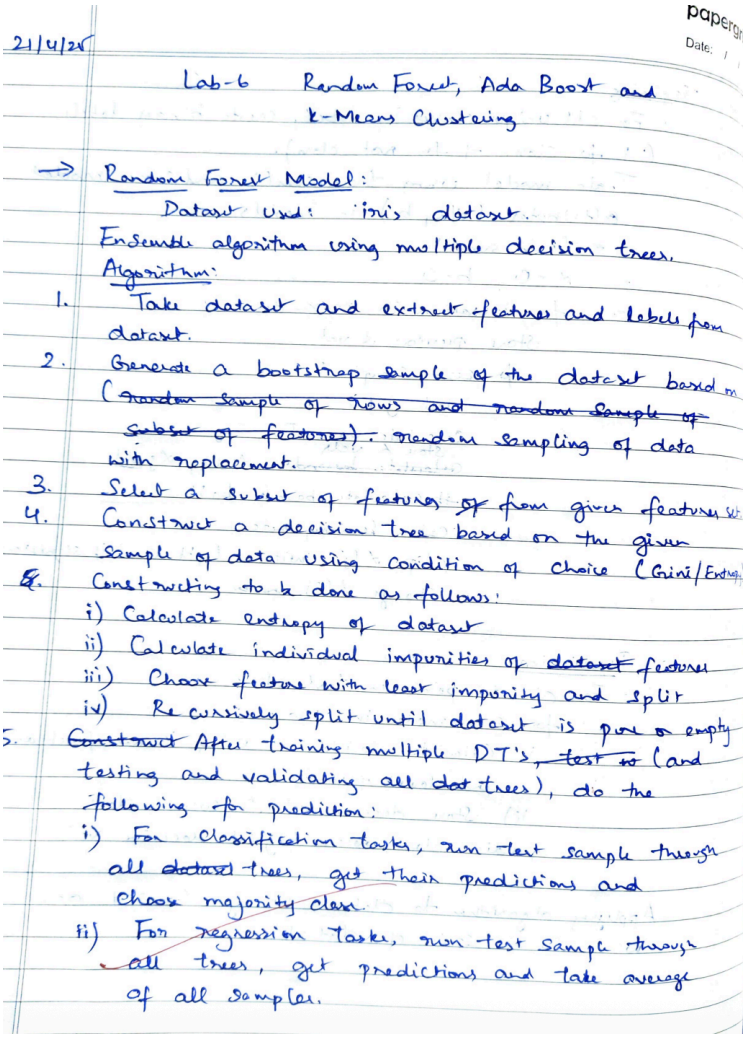
```
            alpha_j_old = alpha[j]

            L, H = self._compute_bounds(y, i, j, alpha)
            if L == H:
                continue

            eta = 2 * self.kernel(X[i], X[j]) - self.kernel(X[i], X[i]) - self.kernel(X[j], X[j])
            if eta >= 0:
                continue

            alpha[j] -= (y[j] * (E_i - E_j)) / eta
            alpha[j] = min(H, max(L, alpha[j]))

            if abs(alpha[j] - alpha_j_old) < 1e-5:
                continue

            alpha[i] += y[i] * y[j] * (alpha_j_old - alpha[j])

            b1 = b - E_i - y[i] * (alpha[i] - alpha_i_old) * self.kernel(X[i], X[i]) - y[j] * (alpha[j] - alpha_j_old) *
self.kernel(X[i], X[j])
            b2 = b - E_j - y[i] * (alpha[i] - alpha_i_old) * self.kernel(X[i], X[j]) - y[j] * (alpha[j] - alpha_j_old) *
self.kernel(X[j], X[j])

            if 0 < alpha[i] < self.C:
                b = b1
            elif 0 < alpha[j] < self.C:
                b = b2
            else:
                b = (b1 + b2) / 2

        if np.linalg.norm(alpha - alpha_prev) < 1e-5:
            break

    return (alpha, b)

# Compute the error for a given sample
def _compute_error(self, X, y, i, alpha, b):
    return np.dot(alpha * y, [self.kernel(X[i], X[j]) for j in range(len(X))]) + b - y[i]

# Compute the bounds L and H for the optimization problem
def _compute_bounds(self, y, i, j, alpha):
    if y[i] != y[j]:
        L = max(0, alpha[j] - alpha[i])
        H = min(self.C, self.C + alpha[j] - alpha[i])
    else:
        L = max(0, alpha[i] + alpha[j] - self.C)
        H = min(self.C, alpha[i] + alpha[j])
    return L, H

# Predict the class labels for a given set of samples
def predict(self, X):
    predictions = []
```

```
        for i in range(len(X)):
            class_scores = []
            for model in self.models:
                alpha, b = model
                decision_value = 0
                for j in range(len(self.X_train)):
                    if alpha[j] > 0:
                        decision_value += alpha[j] * self.y_train[j] * self.kernel(X[i], self.X_train[j])
                decision_value += b
                class_scores.append(decision_value)

            # Choose the class with the highest decision value
            predictions.append(self.classes[np.argmax(class_scores)])

        return np.array(predictions)

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

# Preprocess the data: Standardize it
scaler = StandardScaler()
X = scaler.fit_transform(X)

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

# Initialize the SVM model
svm = SVM(C=1, max_iter=1000)

# Fit the SVM model
svm.fit(X_train, y_train)

# Make predictions on the test data
predictions = svm.predict(X_test)

# Print the results
print("Predictions:", predictions)
print("Actual Labels:", y_test)

# Calculate accuracy
accuracy = np.sum(predictions == y_test) / len(y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Program 8

Implement Random forest ensemble method on a given dataset

Screenshot



21/4/25

Lab-6 Random Forest, Ada Boost and K-Means Clustering

→ Random Forest Model:

Dataset used: iris dataset.

Ensemble algorithm using multiple decision trees.

Algorithm:

1. Take dataset and extract features and labels from dataset.
2. Generate a bootstrap sample of the dataset based on (~~Random sample of rows and random sample of subset of features~~). Random sampling of data with replacement.
3. Select a subset of features ~~of~~ from given features set
4. Construct a decision tree based on the given sample of data using condition of choice (Gini/Entro
5. Constructing to be done as follows:
   i) Calculate entropy of dataset
   ii) Calculate individual impurities of ~~dataset~~ features
   iii) Choose feature with least impurity and split
   iv) Recursively split until dataset is pure or empty
6. ~~Construct~~ After training multiple DT's, ~~test~~ (and testing and validating all dot trees), do the following ~~for~~ prediction:
   i) For classification tasks, run test sample through all ~~dataset~~ trees, get their predictions and choose majority class.
   ii) For regression tasks, run test sample through all trees, get predictions and take average of all samples.

Code

```
import numpy as np
import sklearn
from sklearn import datasets, model_selection, metrics
import matplotlib.pyplot as plt
from collections import Counter

"""# Decision Tree

"""

class TreeNode():
    def __init__(self, data, feature_idx, feature_val, prediction_probs, information_gain) -> None:
        self.data = data
```

```
        self.feature_idx = feature_idx
        self.feature_val = feature_val
        self.prediction_probs = prediction_probs
        self.information_gain = information_gain
        self.feature_importance = self.data.shape[0] * self.information_gain
        self.left = None
        self.right = None

    def node_def(self) -> str:

        if (self.left or self.right):
            return f"NODE | Information Gain = {self.information_gain} | Split IF X[{self.feature_idx}] <
{self.feature_val} THEN left O/W right"
        else:
            unique_values, value_counts = np.unique(self.data[:,-1], return_counts=True)
            output = ", ".join([f"{value}->{count}" for value, count in zip(unique_values, value_counts)])
            return f"LEAF | Label Counts = {output} | Pred Probs = {self.prediction_probs}"




class DecisionTree():
    """
    Decision Tree Classifier
    Training: Use "train" function with train set features and labels
    Predicting: Use "predict" function with test set features
    """

    def __init__(self, max_depth=4, min_samples_leaf=1,
                 min_information_gain=0.0, numb_of_features_splitting=None,
                 amount_of_say=None) -> None:
        """
        Setting the class with hyperparameters
        max_depth: (int) -> max depth of the tree
        min_samples_leaf: (int) -> min # of samples required to be in a leaf to make the splitting possible
        min_information_gain: (float) -> min information gain required to make the splitting possible
        num_of_features_splitting: (str) ->  when splitting if sqrt then sqrt(# of features) features considered,
                                             if log then log(# of features) features considered
                                             else all features are considered
        amount_of_say: (float) -> used for Adaboost algorithm
        """
        self.max_depth = max_depth
        self.min_samples_leaf = min_samples_leaf
        self.min_information_gain = min_information_gain
        self.numb_of_features_splitting = numb_of_features_splitting
        self.amount_of_say = amount_of_say

    def _entropy(self, class_probabilities: list) -> float:
        return sum([-p * np.log2(p) for p in class_probabilities if p>0])

    def _class_probabilities(self, labels: list) -> list:
        total_count = len(labels)
```

```
    return [label_count / total_count for label_count in Counter(labels).values()]

def _data_entropy(self, labels: list) -> float:
    return self._entropy(self._class_probabilities(labels))

def _partition_entropy(self, subsets: list) -> float:
    """subsets = list of label lists (EX: [[1,0,0], [1,1,1])"""
    total_count = sum([len(subset) for subset in subsets])
    return sum([self._data_entropy(subset) * (len(subset) / total_count) for subset in subsets])

def _split(self, data: np.array, feature_idx: int, feature_val: float) -> tuple:

    mask_below_threshold = data[:, feature_idx] < feature_val
    group1 = data[mask_below_threshold]
    group2 = data[~mask_below_threshold]

    return group1, group2

def _select_features_to_use(self, data: np.array) -> list:
    """
    Randomly selects the features to use while splitting w.r.t. hyperparameter numb_of_features_splitting
    """
    feature_idx = list(range(data.shape[1]-1))

    if self.numb_of_features_splitting == "sqrt":
        feature_idx_to_use = np.random.choice(feature_idx, size=int(np.sqrt(len(feature_idx))))
    elif self.numb_of_features_splitting == "log":
        feature_idx_to_use = np.random.choice(feature_idx, size=int(np.log2(len(feature_idx))))
    else:
        feature_idx_to_use = feature_idx

    return feature_idx_to_use

def _find_best_split(self, data: np.array) -> tuple:
    """
    Finds the best split (with the lowest entropy) given data
    Returns 2 splitted groups and split information
    """
    min_part_entropy = 1e9
    feature_idx_to_use =  self._select_features_to_use(data)

    for idx in feature_idx_to_use:
        feature_vals = np.percentile(data[:, idx], q=np.arange(25, 100, 25))
        for feature_val in feature_vals:
            g1, g2, = self._split(data, idx, feature_val)
            part_entropy = self._partition_entropy([g1[:, -1], g2[:, -1]])
            if part_entropy < min_part_entropy:
                min_part_entropy = part_entropy
                min_entropy_feature_idx = idx
                min_entropy_feature_val = feature_val
                g1_min, g2_min = g1, g2
```

```
    return g1_min, g2_min, min_entropy_feature_idx, min_entropy_feature_val, min_part_entropy

def _find_label_probs(self, data: np.array) -> np.array:

    labels_as_integers = data[:,-1].astype(int)
    # Calculate the total number of labels
    total_labels = len(labels_as_integers)
    # Calculate the ratios (probabilities) for each label
    label_probabilities = np.zeros(len(self.labels_in_train), dtype=float)

    # Populate the label_probabilities array based on the specific labels
    for i, label in enumerate(self.labels_in_train):
        label_index = np.where(labels_as_integers == i)[0]
        if len(label_index) > 0:
            label_probabilities[i] = len(label_index) / total_labels

    return label_probabilities

def _create_tree(self, data: np.array, current_depth: int) -> TreeNode:
    """
    Recursive, depth first tree creation algorithm
    """

    # Check if the max depth has been reached (stopping criteria)
    if current_depth > self.max_depth:
        return None

    # Find best split
    split_1_data, split_2_data, split_feature_idx, split_feature_val, split_entropy = self._find_best_split(data)

    # Find label probs for the node
    label_probabilities = self._find_label_probs(data)

    # Calculate information gain
    node_entropy = self._entropy(label_probabilities)
    information_gain = node_entropy - split_entropy

    # Create node
    node = TreeNode(data, split_feature_idx, split_feature_val, label_probabilities, information_gain)

    # Check if the min_samples_leaf has been satisfied (stopping criteria)
    if self.min_samples_leaf > split_1_data.shape[0] or self.min_samples_leaf > split_2_data.shape[0]:
        return node
    # Check if the min_information_gain has been satisfied (stopping criteria)
    elif information_gain < self.min_information_gain:
        return node

    current_depth += 1
    node.left = self._create_tree(split_1_data, current_depth)
    node.right = self._create_tree(split_2_data, current_depth)

    return node
```

```
    def _predict_one_sample(self, X: np.array) -> np.array:
        """Returns prediction for 1 dim array"""
        node = self.tree

        # Finds the leaf which X belongs
        while node:
            pred_probs = node.prediction_probs
            if X[node.feature_idx] < node.feature_val:
                node = node.left
            else:
                node = node.right

        return pred_probs

    def train(self, X_train: np.array, Y_train: np.array) -> None:
        """
        Trains the model with given X and Y datasets
        """

        # Concat features and labels
        self.labels_in_train = np.unique(Y_train)
        train_data = np.concatenate((X_train, np.reshape(Y_train, (-1, 1))), axis=1)

        # Start creating the tree
        self.tree = self._create_tree(data=train_data, current_depth=0)

        # Calculate feature importance
        self.feature_importances = dict.fromkeys(range(X_train.shape[1]), 0)
        self._calculate_feature_importance(self.tree)
        # Normalize the feature importance values
        self.feature_importances = {k: v / total for total in (sum(self.feature_importances.values()),) for k, v in
self.feature_importances.items()}

    def predict_proba(self, X_set: np.array) -> np.array:
        """Returns the predicted probs for a given data set"""

        pred_probs = np.apply_along_axis(self._predict_one_sample, 1, X_set)

        return pred_probs

    def predict(self, X_set: np.array) -> np.array:
        """Returns the predicted labels for a given data set"""

        pred_probs = self.predict_proba(X_set)
        preds = np.argmax(pred_probs, axis=1)

        return preds

    def _print_recursive(self, node: TreeNode, level=0) -> None:
        if node != None:
            self._print_recursive(node.left, level + 1)
```

```
            print('    ' * 4 * level + '-> ' + node.node_def())
            self._print_recursive(node.right, level + 1)

    def print_tree(self) -> None:
        self._print_recursive(node=self.tree)

    def _calculate_feature_importance(self, node):
        """Calculates the feature importance by visiting each node in the tree recursively"""
        if node != None:
            self.feature_importances[node.feature_idx] += node.feature_importance
            self._calculate_feature_importance(node.left)
            self._calculate_feature_importance(node.right)

"""# Random Forest Classifier"""

class RandomForestClassifier():
    """
    Random Forest Classifier
    Training: Use "train" function with train set features and labels
    Predicting: Use "predict" function with test set features
    """

    def __init__(self, n_base_learner=10, max_depth=5, min_samples_leaf=1, min_information_gain=0.0, \
                 numb_of_features_splitting=None, bootstrap_sample_size=None) -> None:
        self.n_base_learner = n_base_learner
        self.max_depth = max_depth
        self.min_samples_leaf = min_samples_leaf
        self.min_information_gain = min_information_gain
        self.numb_of_features_splitting = numb_of_features_splitting
        self.bootstrap_sample_size = bootstrap_sample_size

    def _create_bootstrap_samples(self, X, Y) -> tuple:
        """
        Creates bootstrap samples for each base learner
        """
        bootstrap_samples_X = []
        bootstrap_samples_Y = []

        for i in range(self.n_base_learner):

            if not self.bootstrap_sample_size:
                self.bootstrap_sample_size = X.shape[0]

            sampled_idx = np.random.choice(X.shape[0], size=self.bootstrap_sample_size, replace=True)
            bootstrap_samples_X.append(X[sampled_idx])
            bootstrap_samples_Y.append(Y[sampled_idx])

        return bootstrap_samples_X, bootstrap_samples_Y

    def train(self, X_train: np.array, Y_train: np.array) -> None:
        """Trains the model with given X and Y datasets"""
        bootstrap_samples_X, bootstrap_samples_Y = self._create_bootstrap_samples(X_train, Y_train)
```

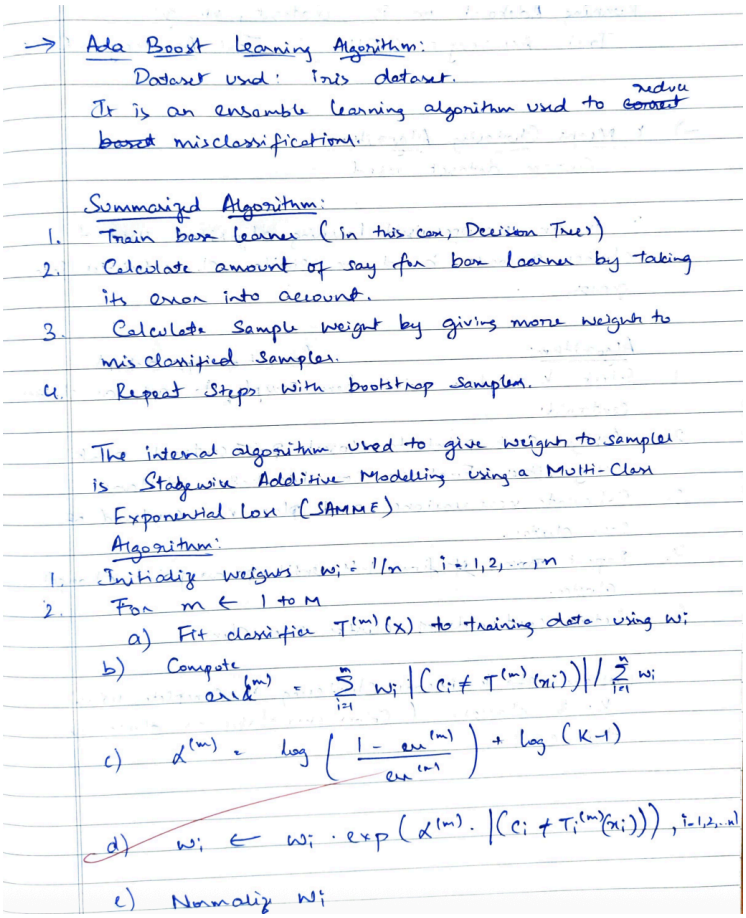
```
        self.base_learner_list = []
        for base_learner_idx in range(self.n_base_learner):
            base_learner = DecisionTree(max_depth=self.max_depth, min_samples_leaf=self.min_samples_leaf, \
                                min_information_gain=self.min_information_gain,
                                numb_of_features_splitting=self.numb_of_features_splitting)

            base_learner.train(bootstrap_samples_X[base_learner_idx], bootstrap_samples_Y[base_learner_idx])
            self.base_learner_list.append(base_learner)

        # Calculate feature importance
        self.feature_importances = self._calculate_rf_feature_importance(self.base_learner_list)

    def _predict_proba_w_base_learners(self,  X_set: np.array) -> list:
        """
        Creates list of predictions for all base learners
        """
        pred_prob_list = []
        for base_learner in self.base_learner_list:
            pred_prob_list.append(base_learner.predict_proba(X_set))

        return pred_prob_list

    def predict_proba(self, X_set: np.array) -> list:
        """Returns the predicted probs for a given data set"""

        pred_probs = []
        base_learners_pred_probs = self._predict_proba_w_base_learners(X_set)

        # Average the predicted probabilities of base learners
        for obs in range(X_set.shape[0]):
            base_learner_probs_for_obs = [a[obs] for a in base_learners_pred_probs]
            # Calculate the average for each index
            obs_average_pred_probs = np.mean(base_learner_probs_for_obs, axis=0)
            pred_probs.append(obs_average_pred_probs)

        return pred_probs

    def predict(self, X_set: np.array) -> np.array:
        """Returns the predicted labels for a given data set"""

        pred_probs = self.predict_proba(X_set)
        preds = np.argmax(pred_probs, axis=1)

        return preds

    def _calculate_rf_feature_importance(self, base_learners):
        """Calcalates the average feature importance of the base learners"""
        feature_importance_dict_list = []
        for base_learner in base_learners:
            feature_importance_dict_list.append(base_learner.feature_importances)
```

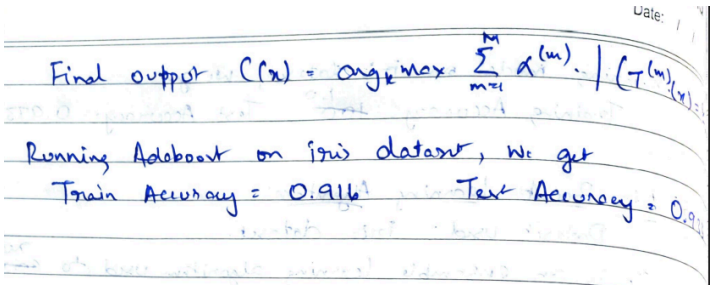
```
feature_importance_list = [list(x.values()) for x in feature_importance_dict_list]
average_feature_importance = np.mean(feature_importance_list, axis=0)

return average_feature_importance
```

Program 9

Implement Boosting ensemble method on a given dataset

Screenshot





Code

```
class AdaBoostClassifier():
    """
    AdaBoost Classifier Model
    Training: Use "train" function with train set features and labels
    Predicting: Use "predict" function with test set features
    The algorithm used in this class is SAMME algorithm with boosting with resampling
    """

    def __init__(self, n_base_learner=10) -> None:
        """
        Initialize the object with the hyperparameters
        n_base_learner: # of base learners in the model (base learners are DecisionTree with max_depth=1)
        """
        self.n_base_learner = n_base_learner

    def _calculate_amount_of_say(self, base_learner: DecisionTree, X: np.array, y: np.array) -> float:
        """calculates the amount of say (see SAMME)"""
        K = self.label_count
```

```
    preds = base_learner.predict(X)
    err = 1 - np.sum(preds==y) / preds.shape[0]
    amount_of_say = np.log((1-err)/err) + np.log(K-1)
    return amount_of_say

def _fit_base_learner(self, X_bootstrapped: np.array, y_bootstrapped: np.array) -> DecisionTree:
    """Trains a Decision Tree model with depth 1 and returns the model"""
    base_learner = DecisionTree(max_depth=1)
    base_learner.train(X_bootstrapped, y_bootstrapped)
    base_learner.amount_of_say = self._calculate_amount_of_say(base_learner, self.X_train, self.y_train)

    return base_learner

def _update_dataset(self, sample_weights: np.array) -> tuple:
    """Creates bootstrapped samples w.r.t. sample weights"""
    n_samples = self.X_train.shape[0]
    bootstrap_indices = np.random.choice(n_samples, size=n_samples, replace=True, p=sample_weights)
    X_bootstrapped = self.X_train[bootstrap_indices]
    y_bootstrapped = self.y_train[bootstrap_indices]

    return X_bootstrapped, y_bootstrapped

def _calculate_sample_weights(self, base_learner: DecisionTree) -> np.array:
    """Calculates sample weights (see SAMME)"""
    preds = base_learner.predict(self.X_train)
    matches = (preds == self.y_train)
    not_matches = (~matches).astype(int)
    sample_weights = 1/self.X_train.shape[0] * np.exp(base_learner.amount_of_say*not_matches)
    # Normalize weights
    sample_weights = sample_weights / np.sum(sample_weights)

    return sample_weights

def train(self, X_train: np.array, y_train: np.array) -> None:
    """
    trains base learners with given feature and label dataset
    """
    self.X_train = X_train
    self.y_train = y_train
    X_bootstrapped = X_train
    y_bootstrapped = y_train
    self.label_count = len(np.unique(y_train))

    self.base_learner_list = []
    for i in range(self.n_base_learner):
        base_learner = self._fit_base_learner(X_bootstrapped, y_bootstrapped)
        self.base_learner_list.append(base_learner)
        sample_weights = self._calculate_sample_weights(base_learner)
        X_bootstrapped, y_bootstrapped = self._update_dataset(sample_weights)

def _predict_scores_w_base_learners(self,  X: np.array) -> list:
    """
```

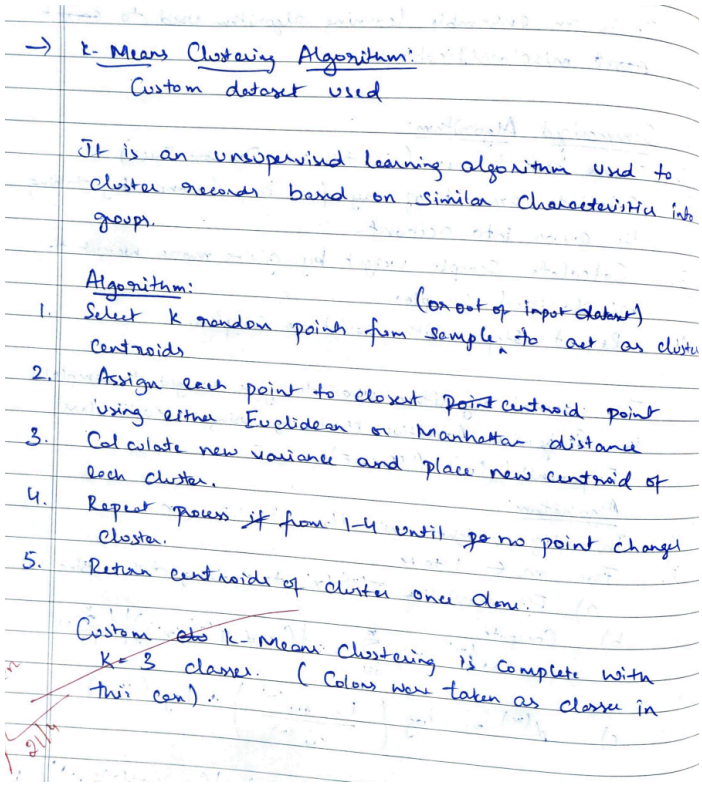
```
        Creates list of predictions for all base learners
        """
        pred_scores = np.zeros(shape=(self.n_base_learner, X.shape[0], self.label_count))
        for idx, base_learner in enumerate(self.base_learner_list):
            pred_probs = base_learner.predict_proba(X)
            pred_scores[idx] = pred_probs*base_learner.amount_of_say

        return pred_scores

    def predict_proba(self, X: np.array) -> np.array:
        """Returns the predicted probs for a given data set"""

        pred_probs = []
        base_learners_pred_scores = self._predict_scores_w_base_learners(X)

        # Take the avg scores and turn them to probabilities
        avg_base_learners_pred_scores = np.mean(base_learners_pred_scores, axis=0)
        column_sums = np.sum(avg_base_learners_pred_scores, axis=1)
        pred_probs = avg_base_learners_pred_scores / column_sums[:, np.newaxis]

        return pred_probs

    def predict(self, X: np.array) -> np.array:
        """Returns the predicted labels for a given data set"""

        pred_probs = self.predict_proba(X)
        preds = np.argmax(pred_probs, axis=1)

        return preds

iris = datasets.load_iris()

X = np.array(iris.data)
Y = np.array(iris.target)
iris_feature_names = iris.feature_names

X_train, X_test, Y_train, Y_test = model_selection.train_test_split(X, Y, test_size=0.25, random_state=0)
print("Train Shape:", X_train.shape)
print("Train Shape:", X_test.shape)
# Building random forest model
rf_model = RandomForestClassifier(n_base_learner=50, numb_of_features_splitting="sqrt")
rf_model.train(X_train, Y_train)
# Performans increases when compared to the basic DecisionTree
train_preds = rf_model.predict(X_set=X_train)
print("TRAIN PERFORMANCE")
print("Train size", len(Y_train))
print("True preds", sum(train_preds == Y_train))
print("Accuracy", sum(train_preds == Y_train) / len(Y_train))

test_preds = rf_model.predict(X_set=X_test)
print("TEST PERFORMANCE")
print("Test size", len(Y_test))
```

```
print("True preds", sum(test_preds == Y_test))
print("Accuracy", sum(test_preds == Y_test) / len(Y_test))

iris = datasets.load_iris()

X = np.array(iris.data)
Y = np.array(iris.target)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, Y, test_size=0.2, random_state=0)

model = AdaBoostClassifier(n_base_learner=50)
model.train(X_train, y_train)

train_accuracy = sum(model.predict(X=X_train) == y_train) / len(y_train)
test_accuracy = sum(model.predict(X=X_test) == y_test) / len(y_test)
print("Our Model Performance")
print("Train Accuracy: ", train_accuracy)
print("Test Accuracy: ", test_accuracy)
iris = datasets.load_iris()

X = np.array(iris.data)
Y = np.array(iris.target)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, Y, test_size=0.2, random_state=0)

model = sklearn.ensemble.AdaBoostClassifier(n_estimators=50)
model.fit(X_train, y_train)

train_accuracy = sum(model.predict(X=X_train) == y_train) / len(y_train)
test_accuracy = sum(model.predict(X=X_test) == y_test) / len(y_test)
print("Sklearn Model Performance")
print("Train Accuracy: ", train_accuracy)
print("Test Accuracy: ", test_accuracy)
```

Program 10

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

Screenshot



Code

```
class K_Means:

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

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

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

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

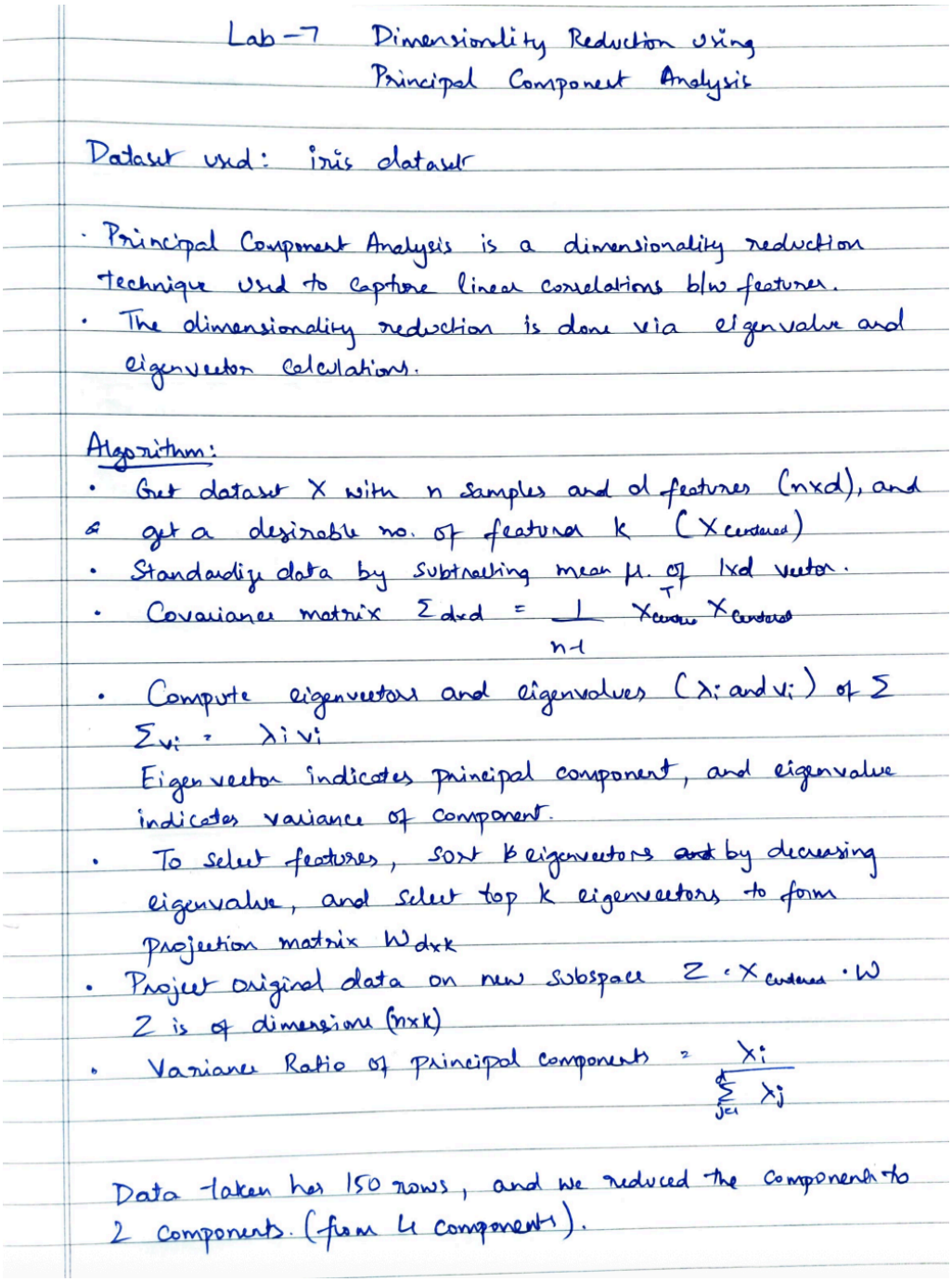
```
        for i in range(self.max_iterations):
            self.classes = {}
            for j in range(self.k):
                self.classes[j] = []

            for point in data:
                distances = []
                for index in self.centroids:
                    distances.append(self.euclidean_distance(point,self.centroids[index]))
                cluster_index = distances.index(min(distances))
                self.classes[cluster_index].append(point)

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



            isOptimal = True

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

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

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

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

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


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

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

for cluster_index in k_means.classes:
```

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

Program 11

Implement Dimensionality reduction using Principal Component Analysis (PCA) method

Screenshot



Lab – 7 Dimensionality Reduction using Principal Component Analysis

Dataset used: iris dataset

- Principal Component Analysis is a dimensionality reduction technique used to capture linear correlations b/w features.
- The dimensionality reduction is done via eigenvalue and eigenvector calculations.

Algorithm:

- Get dataset X with n samples and d features (nxd), and get a desirable no. of features k ($X_{centered}$)
- Standardize data by subtracting mean μ. of 1xd vector.
- Covariance matrix $\Sigma_{dxd} = \frac{1}{n-1} X_{centered}^{T} X_{centered}$
- Compute eigenvectors and eigenvalues (λ_i and v_i) of Σ
  $\Sigma v_i = \lambda_i v_i$
  Eigenvector indicates principal component, and eigenvalue indicates variance of component.
- To select features, sort k eigenvectors ~~and~~ by decreasing eigenvalue, and select top k eigenvectors to form projection matrix W_{dxk}
- Project original data on new subspace $Z = X_{centered} \cdot W$
  Z is of dimensions (nxk)
- Variance Ratio of principal components = $\frac{\lambda_i}{\sum_{j=1}^{} \lambda_j}$

Data taken has 150 rows, and we reduced the components to 2 components. (from 4 components).

Code

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris['data']
```

```
y = iris['target']

n_samples, n_features = X.shape

print('Number of samples:', n_samples)
print('Number of features:', n_features)

# Commented out IPython magic to ensure Python compatibility.
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline

fig, ax = plt.subplots(nrows=n_features, ncols=n_features, figsize= (8, 8))
fig.tight_layout()

names = iris.feature_names

for i, j in zip(*np.triu_indices_from(ax, k=1)):
    ax[j, i].scatter(X[:, j], X[:, i], c = y)
    ax[j, i].set_xlabel(names[j])
    ax[j, i].set_ylabel(names[i])
    ax[i, j].set_axis_off()

for i in range(n_features):
    ax[i, i].hist(X[:, i], color = 'lightblue')
    ax[i, i].set_ylabel('Count')
    ax[i, i].set_xlabel(names[i])

def mean(x): # np.mean(X, axis = 0)
    return sum(x)/len(x)

def std(x): # np.std(X, axis = 0)
    return (sum((i - mean(x))**2 for i in x)/len(x))**0.5

def Standardize_data(X):
    return (X - mean(X))/std(X)

X_std = Standardize_data(X)

def covariance(x):
    return (x.T @ x)/(x.shape[0]-1)

cov_mat = covariance(X_std) # np.cov(X_std.T)

from numpy.linalg import eig

# Eigendecomposition of covariance matrix
eig_vals, eig_vecs = eig(cov_mat)

# Adjusting the eigenvectors (loadings) that are largest in absolute value to be positive
max_abs_idx = np.argmax(np.abs(eig_vecs), axis=0)
signs = np.sign(eig_vecs[max_abs_idx, range(eig_vecs.shape[0])])
```

```
eig_vecs = eig_vecs*signs[np.newaxis,:]
eig_vecs = eig_vecs.T

print('Eigenvalues \n', eig_vals)
print('Eigenvectors \n', eig_vecs)

# We first make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[i,:]) for i in range(len(eig_vals))]

# Then, we sort the tuples from the highest to the lowest based on eigenvalues magnitude
eig_pairs.sort(key=lambda x: x[0], reverse=True)

# For further usage
eig_vals_sorted = np.array([x[0] for x in eig_pairs])
eig_vecs_sorted = np.array([x[1] for x in eig_pairs])

print(eig_pairs)

# Select top k eigenvectors
k = 2
W = eig_vecs_sorted[:k, :] # Projection matrix

print(W.shape)

eig_vals_total = sum(eig_vals)
explained_variance = [(i / eig_vals_total)*100 for i in eig_vals_sorted]
explained_variance = np.round(explained_variance, 2)
cum_explained_variance = np.cumsum(explained_variance)

print('Explained variance: {}'.format(explained_variance))
print('Cumulative explained variance: {}'.format(cum_explained_variance))

plt.plot(np.arange(1,n_features+1), cum_explained_variance, '-o')
plt.xticks(np.arange(1,n_features+1))
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance');
plt.show()

X_proj = X_std.dot(W.T)

print(X_proj.shape)

plt.scatter(X_proj[:, 0], X_proj[:, 1], c = y)
plt.xlabel('PC1'); plt.xticks([])
plt.ylabel('PC2'); plt.yticks([])
plt.title('2 components, captures {} of total variation'.format(cum_explained_variance[1]))
plt.show()

class PCA:

    def __init__(self, n_components):
        self.n_components = n_components
```

```
    def fit(self, X):
        # Standardize data
        X = X.copy()
        self.mean = np.mean(X, axis = 0)
        self.scale = np.std(X, axis = 0)
        X_std = (X - self.mean) / self.scale

        # Eigendecomposition of covariance matrix
        cov_mat = np.cov(X_std.T)
        eig_vals, eig_vecs = np.linalg.eig(cov_mat)

        # Adjusting the eigenvectors that are largest in absolute value to be positive
        max_abs_idx = np.argmax(np.abs(eig_vecs), axis=0)
        signs = np.sign(eig_vecs[max_abs_idx, range(eig_vecs.shape[0])])
        eig_vecs = eig_vecs*signs[np.newaxis,:]
        eig_vecs = eig_vecs.T

        eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[i,:]) for i in range(len(eig_vals))]
        eig_pairs.sort(key=lambda x: x[0], reverse=True)
        eig_vals_sorted = np.array([x[0] for x in eig_pairs])
        eig_vecs_sorted = np.array([x[1] for x in eig_pairs])

        self.components = eig_vecs_sorted[:self.n_components,:]

        # Explained variance ratio
        self.explained_variance_ratio = [i/np.sum(eig_vals) for i in eig_vals_sorted[:self.n_components]]

        self.cum_explained_variance = np.cumsum(self.explained_variance_ratio)

        return self

    def transform(self, X):
        X = X.copy()
        X_std = (X - self.mean) / self.scale
        X_proj = X_std.dot(self.components.T)

        return X_proj
# ----------------------------------------------------------
my_pca = PCA(n_components = 2).fit(X)

print('Components:\n', my_pca.components)
print('Explained variance ratio from scratch:\n', my_pca.explained_variance_ratio)
print('Cumulative explained variance from scratch:\n', my_pca.cum_explained_variance)

X_proj = my_pca.transform(X)
print('Transformed data shape from scratch:', X_proj.shape)
```