VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB

REPORTON

MACHINE LEARNING

Submitted by

Jagadeesh A Latti(1BM21CS079)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B. M. S. College of Engineering,

Bull Temple Road, Bangalore

560019(March 2024 to June 2024)

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**" is carried out by **Jagadeesh A Latti (1BM21CS079)** 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 Visveswaraya Technological University, Belgaum during the year 2023-2024. The lab report has been approved as it satisfies the academic requirements in respect of **Machine Learning Lab - (22CS3PCMAL)** work prescribed for thesaid degree.

Dr. K. Panimozhi
Assistant Professor
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Prof.& Head, Dept. of CSE
BMSCE, Bengaluru

Index

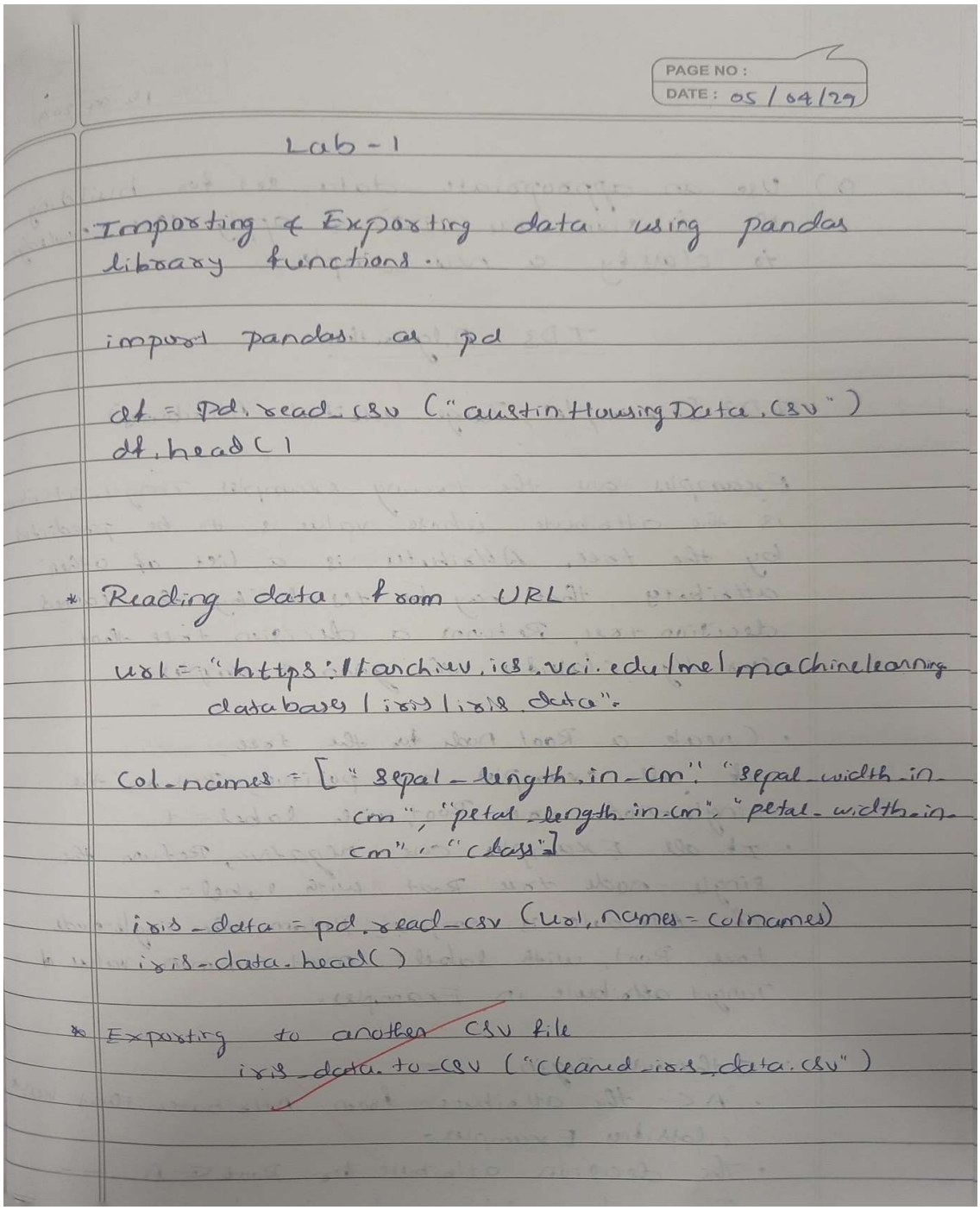
Course outcomes:

CO1	Apply machine learning techniques in computing systems
CO2	Evaluate the model using metrics
CO3	Design a model using machine learning to solve a problem
CO4	Conduct experiments to solve real-world problems using appropriate machine learning techniques

Lab1

Date: 05/04/2024

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



PAGE NO :
DATE : 05 / 04 / 24

Lab - 1

Importing & Exporting data using pandas library functions.

```
import pandas as pd
df = pd.read_csv("austinHousingData.csv")
df.head()
```

* Reading data from URL

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

col_names = ["sepal_length_in_cm", "sepal_width_in_cm", "petal_length_in_cm", "petal_width_in_cm", "class"]

iris_data = pd.read_csv(url, names = colnames)
iris_data.head()
```

* Exporting to another CSV file

```
iris_data.to_csv("cleaned_iris_data.csv")
```

OUTPUT :

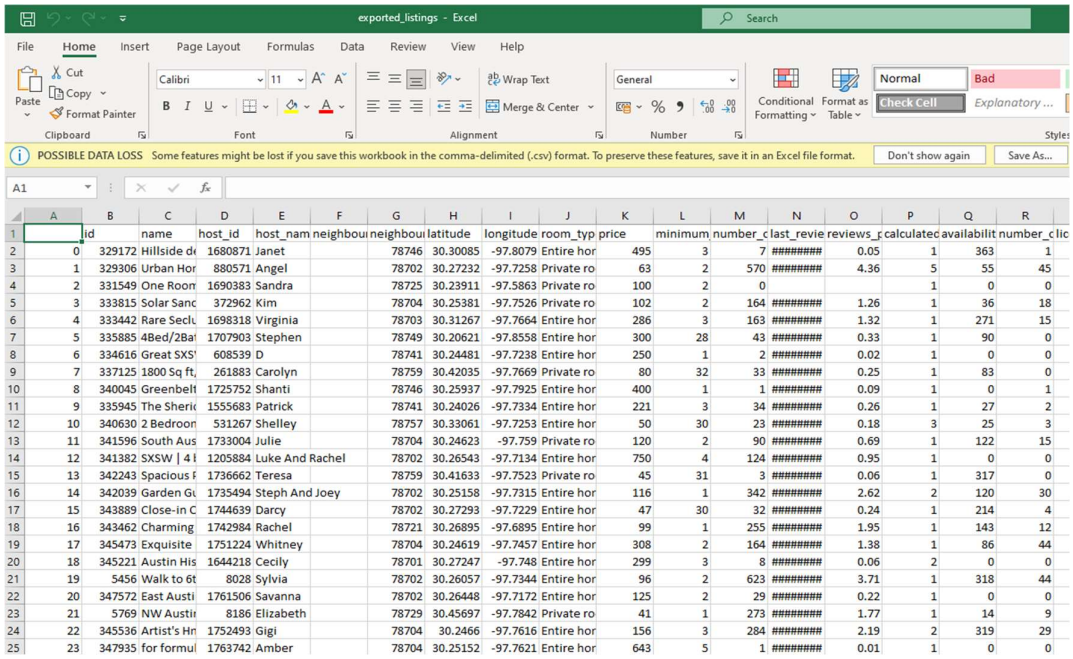
Dataset -

	zpid	city	streetAddress	zipcode	description	latitude	longitude	propertyTaxRate	garageSpaces	hasAssociation	...	numOfMiddleSchools
0	111373431	pflugerville	14424 Lake Victor Dr	78660	14424 Lake Victor Dr, Pflugerville, TX 78660 i...	30.430632	-97.663078	1.98	2	True	...	1
1	120900430	pflugerville	1104 Strickling Dr	78660	Absolutely GORGEOUS 4 Bedroom home with 2 full...	30.432673	-97.661697	1.98	2	True	...	1
2	2084491383	pflugerville	1408 Fort Dessau Rd	78660	Under construction - estimated completion in A...	30.409748	-97.639771	1.98	0	True	...	1
3	120901374	pflugerville	1025 Strickling Dr	78660	Absolutely darling one story home in charming ...	30.432112	-97.661659	1.98	2	True	...	1
4	60134862	pflugerville	15005 Donna Jane Loop	78660	Brimming with appeal & warm livability! Sleek ...	30.437368	-97.656860	1.98	0	True	...	1

After reading dataset from URL –

	sepal_length_in_cm	sepal_width_in_cm	petal_length_in_cm	petal_width_in_cm	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

CSV file after exporting –

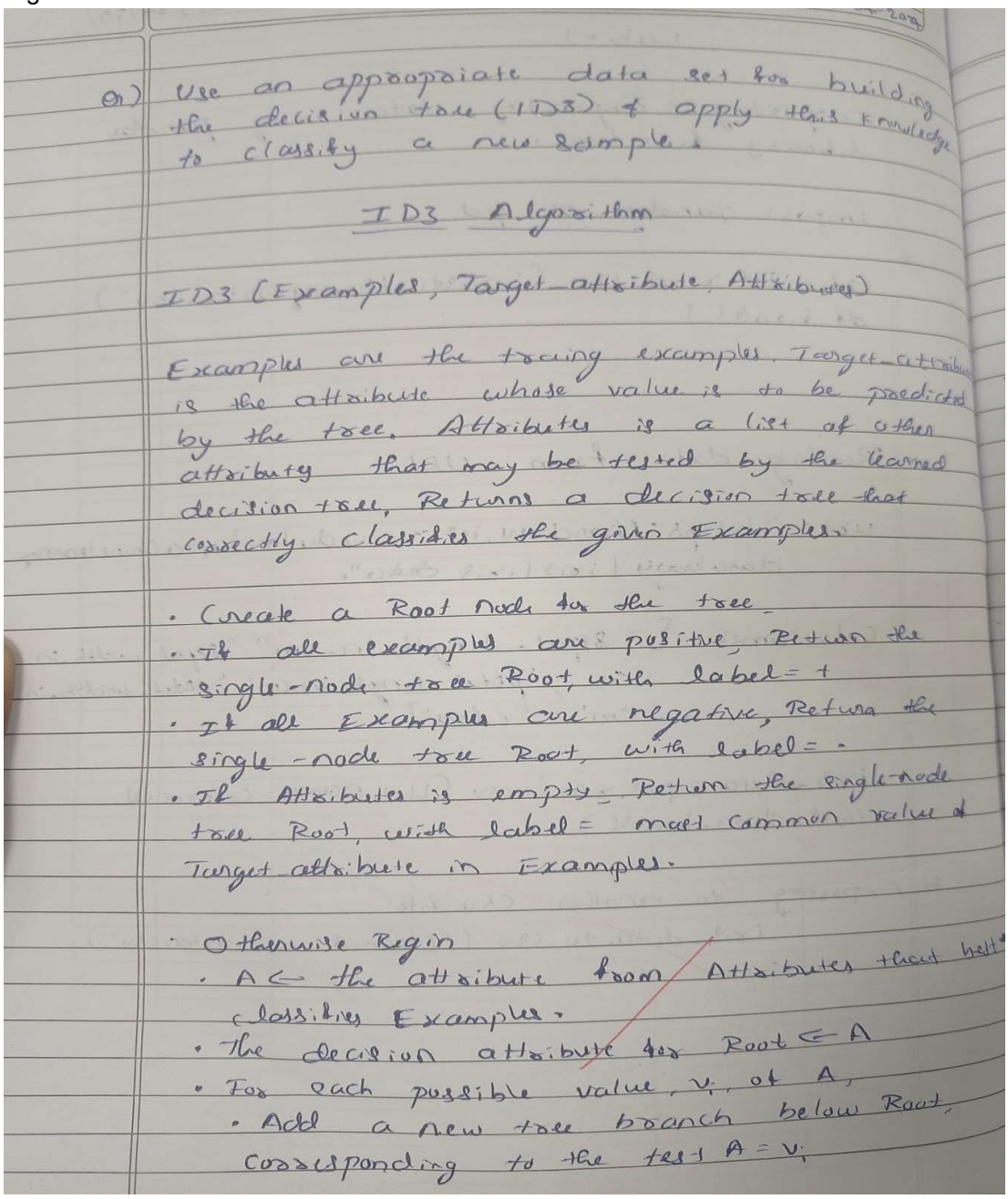


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1		id	name	host_id	host_nam	neighbour	neighbour	latitude	longitude	room_typ	price	minimum	number_c	last_revie	reviews_p	calculated	availabilit	number_c	lic
2	0	329172	Hillside de	1680871	Janet		78746	30.30085	-97.8079	Entire hor	495	3	7	########	0.05	1	363	1	
3	1	329306	Urban Hor	880571	Angel		78702	30.27232	-97.7258	Private ro	63	2	570	########	4.36	5	55	45	
4	2	331549	One Room	1690383	Sandra		78725	30.23911	-97.5863	Private ro	100	2	0			1	0	0	
5	3	333815	Solar Sanc	372962	Kim		78704	30.25381	-97.7526	Private ro	102	2	164	########	1.26	1	36	18	
6	4	333442	Rare Seclu	1698318	Virginia		78703	30.31267	-97.7664	Entire hor	286	3	163	########	1.32	1	271	15	
7	5	335885	4Bed/2Bat	1707903	Stephen		78749	30.20621	-97.8558	Entire hor	300	28	43	########	0.33	1	90	0	
8	6	334616	Great SXS'	608539	D		78741	30.24481	-97.7238	Entire hor	250	1	2	########	0.02	1	0	0	
9	7	337125	1800 Sq ft,	261883	Carolyn		78759	30.42035	-97.7669	Private ro	80	32	33	########	0.25	1	83	0	
10	8	340045	Greenbelt	1725752	Shanti		78746	30.25937	-97.7925	Entire hor	400	1	1	########	0.09	1	0	1	
11	9	335945	The Sheric	1555683	Patrick		78741	30.24026	-97.7334	Entire hor	221	3	34	########	0.26	1	27	2	
12	10	340630	2 Bedroon	531267	Shelley		78757	30.33061	-97.7253	Entire hor	50	30	23	########	0.18	3	25	3	
13	11	341596	South Aus	1733004	Julie		78704	30.24623	-97.759	Private ro	120	2	90	########	0.69	1	122	15	
14	12	341382	SXSW | 4 E	1205884	Luke And	Rachel	78702	30.26543	-97.7134	Entire hor	750	4	124	########	0.95	1	0	0	
15	13	342243	Spacious F	1736662	Teresa		78759	30.41633	-97.7523	Private ro	45	31	3	########	0.06	1	317	0	
16	14	342039	Garden Gu	1735494	Steph And	Joey	78702	30.25158	-97.7315	Entire hor	116	1	342	########	2.62	2	120	30	
17	15	343889	Close-in C	1744639	Darcy		78702	30.27293	-97.7229	Entire hor	47	30	32	########	0.24	1	214	4	
18	16	343462	Charming	1742984	Rachel		78721	30.26895	-97.6895	Entire hor	99	1	255	########	1.95	1	143	12	
19	17	345473	Exquisite	1751224	Whitney		78704	30.24619	-97.7457	Entire hor	308	2	164	########	1.38	1	86	44	
20	18	345221	Austin His	1644218	Cecily		78701	30.27247	-97.748	Entire hor	299	3	8	########	0.06	2	0	0	
21	19	5456	Walk to 6t	8028	Sylvia		78702	30.26057	-97.7344	Entire hor	96	2	623	########	3.71	1	318	44	
22	20	347572	East Austi	1761506	Savanna		78702	30.26448	-97.7172	Entire hor	125	2	29	########	0.22	1	0	0	
23	21	5769	NW Austir	8186	Elizabeth		78729	30.45697	-97.7842	Private ro	41	1	273	########	1.77	1	14	9	
24	22	345536	Artist's Hr	1752493	Gigi		78704	30.2466	-97.7616	Entire hor	156	3	284	########	2.19	2	319	29	
25	23	347935	for formul	1763742	Amber		78704	30.25152	-97.7621	Entire hor	643	5	1	########	0.01	1	0	0	

Lab – 2

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

Algorithm:



Q1) Use an appropriate data set for building the decision tree (ID3) & apply this knowledge to classify a new sample.

ID3 Algorithm

ID3 (Examples, Target_attribute, Attributes)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a Root node for the tree
- If all examples are positive, Return the single-node tree Root, with label = +
- If all Examples are negative, Return the single-node tree Root, with label = -
- If Attributes is empty, Return the single-node tree Root, with label = most common value of Target_attribute in Examples.

- Otherwise Begin
  - A ← the attribute from Attributes that best classifies Examples.
  - The decision attribute for Root ← A
  - For each possible value, v_i, of A,
    - Add a new tree branch below Root, corresponding to the test $A = v_i$

PAGE NO :
DATE :

- Let Examples v_i be the subset of Examples that have value v_i for A.
- If Examples v_i is empty
  - Then below this new branch add a leaf node with label = most common value of Target-attribute in Examples.
  - Else below this new branch add the subtree ID3 (Examples, Target_attribute, Attributes - {A}).
  - End
  - Return Root

Output:

-> df

	Pregnancies	Glucose	...	Age	Outcome
0	6	148	...	50	1
1	1	85	...	31	0
:	:	:		:	:
767	1	93	...	23	0

-> Entropy of dataset : 0.9331393166407831

-> Pregnancies - Entropy: 3.482, Information Gain: 0.862
Glucose - Entropy: 6.751, Information Gain: 0.304
BP - Entropy: 4.792, Information Gain: 0.059
:
Diabetes Pedigree Function - Entropy: 8.829, Infor Gain: 0.051
Age - Entropy: 5.029, Information Gain: 0.141

Code:

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

df = pd.read_csv('/content/diabetes.csv')
df

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

entropy_outcome = calculate_entropy(df, 'Outcome')
print(f"Entropy of the dataset: {entropy_outcome}")

def calculate_entropy(data, target_column): # for each categorical variable
    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) if proportion != 0 else 0

    return entropy

def calculate_information_gain(data, feature, target_column):

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

    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
```

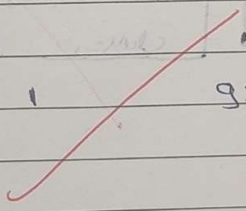
```
    information_gain = entropy_outcome - weighted_entropy

    return information_gain

for column in df.columns[:-1]:
    entropy = calculate_entropy(df, column)
    information_gain = calculate_information_gain(df, column, 'Outcome')
    print(f"{column} - Entropy: {entropy:.3f}, Information Gain: {information_gain:.3f}")

# Feature selection for the first step in making decision tree
selected_feature = 'DiabetesPedigreeFunction'

# Create a decision tree
clf = DecisionTreeClassifier(criterion='entropy', max_depth=1)
X = df[[selected_feature]]
y = df['Outcome']
clf.fit(X, y)

plt.figure(figsize=(8, 6))
plot_tree(clf, feature_names=[selected_feature], class_names=['0', '1'], filled=True, rounded=True)
plt.show()

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

id3(df, 'Outcome', ['Pregnancies',  'Glucose',  'BloodPressure',    'SkinThickness',    'Insulin',  'BMI',
'DiabetesPedigreeFunction', 'Age'] )
```

Output:

Output:

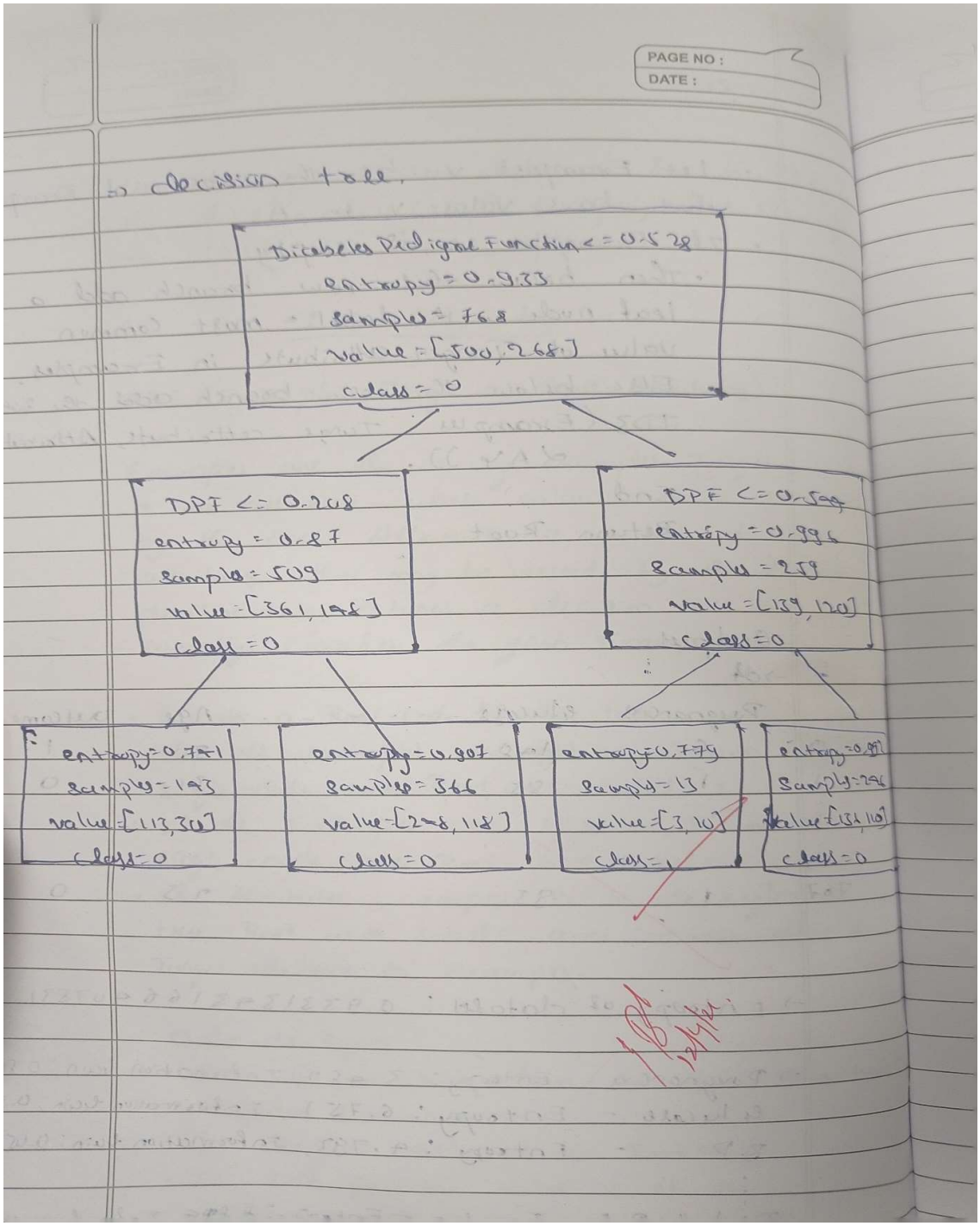
-> df

	Pregnancies	Glucose	...	Age	Outcome
0	6	148	...	50	1
1	1	85	...	31	0
⋮	⋮	⋮	⋮	⋮	⋮
767	1	93	...	23	0

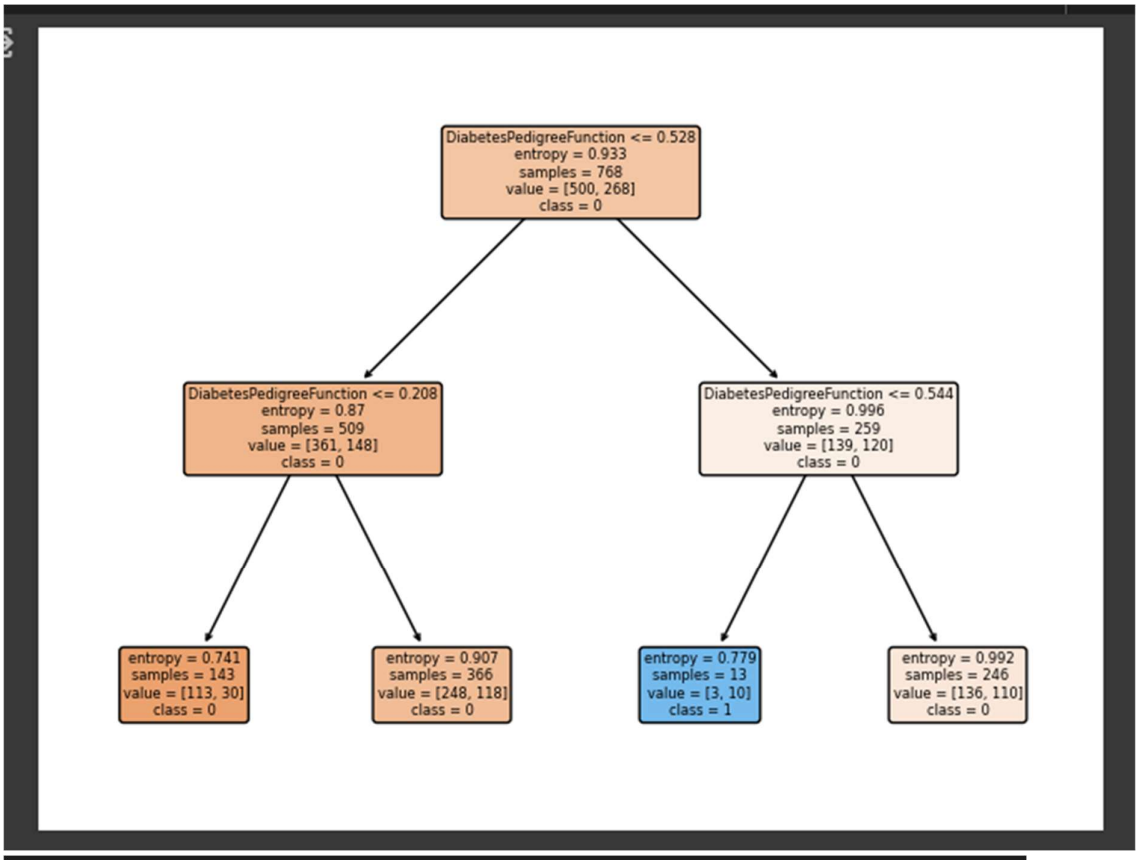


-> Entropy of dataset : 0.9331393166407831

-> Pregnancies - Entropy: 3.482, Information Gain: 0.862
Glucose - Entropy: 6.751, Information Gain: 0.314
BP - Entropy: 9.792, Information Gain: 0.059
⋮
Diabetes Pedigree Function - Entropy: 8.829, Info Gain: 0.051
Age - Entropy: 5.029, Information Gain: 0.141



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

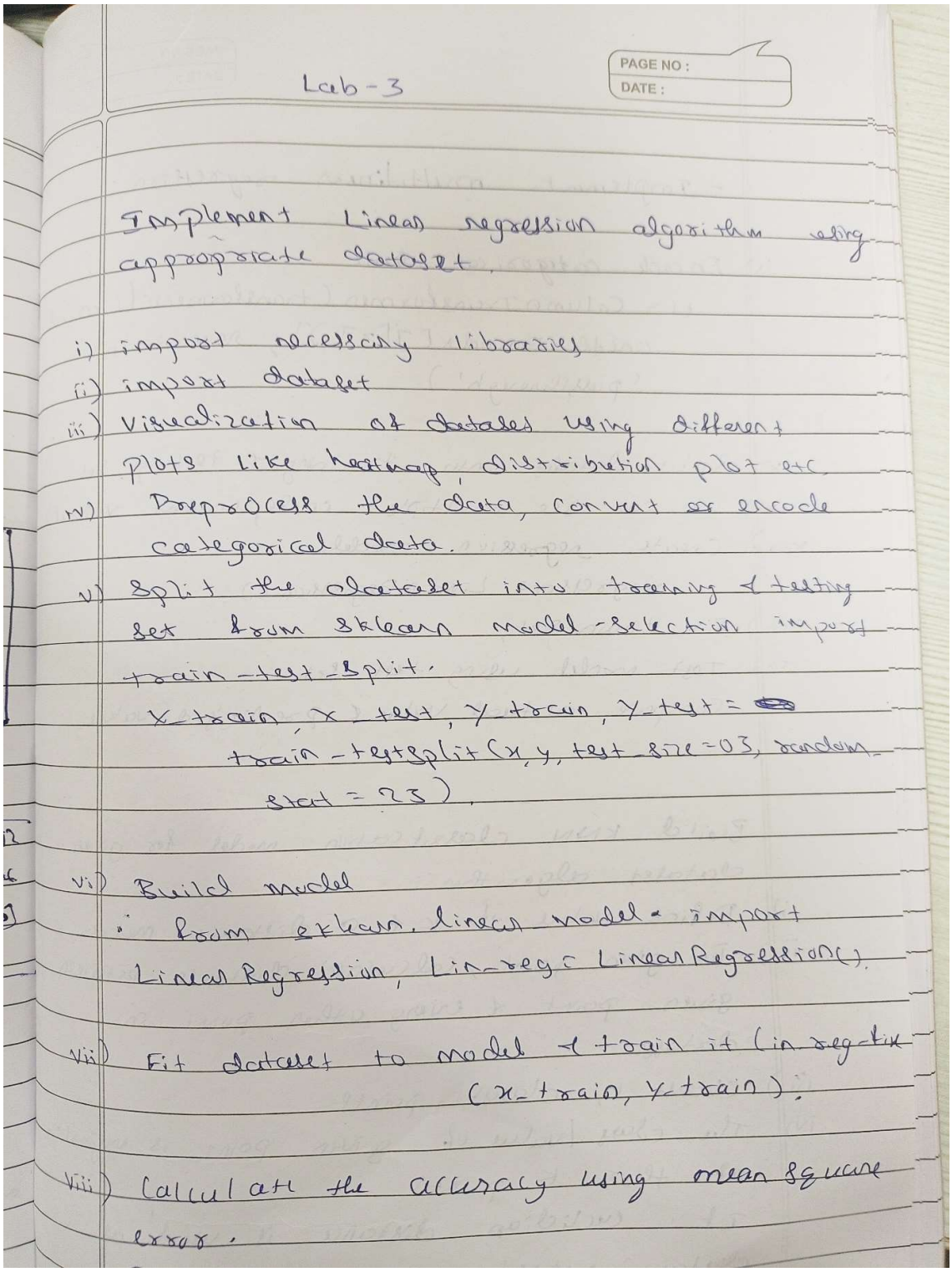



```
Pregnancies - Entropy: 3.482, Information Gain: 0.062
Glucose - Entropy: 6.751, Information Gain: 0.304
BloodPressure - Entropy: 4.792, Information Gain: 0.059
SkinThickness - Entropy: 4.586, Information Gain: 0.082
Insulin - Entropy: 4.682, Information Gain: 0.277
BMI - Entropy: 7.594, Information Gain: 0.344
DiabetesPedigreeFunction - Entropy: 8.829, Information Gain: 0.651
Age - Entropy: 5.029, Information Gain: 0.141
```

LAB-3

a)Implement Linear Regression algorithm using appropriate dataset

a)



Lab-3

PAGE NO :
DATE :

Implement Linear regression algorithm using appropriate dataset.

i) import necessary libraries
ii) import dataset
iii) Visualization of datasets using different plots like heatmap, distribution plot etc.
iv) Preprocess the data, convert or encode categorical data.
v) Split the dataset into training & testing set from sklearn model_selection import train_test_split.
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state = 23).
vi) Build model
from sklearn.linear_model import LinearRegression, Lin_reg = LinearRegression().
vii) Fit dataset to model Lin_reg.fit(x_train, y_train).
viii) Calculate the accuracy using mean square error.

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

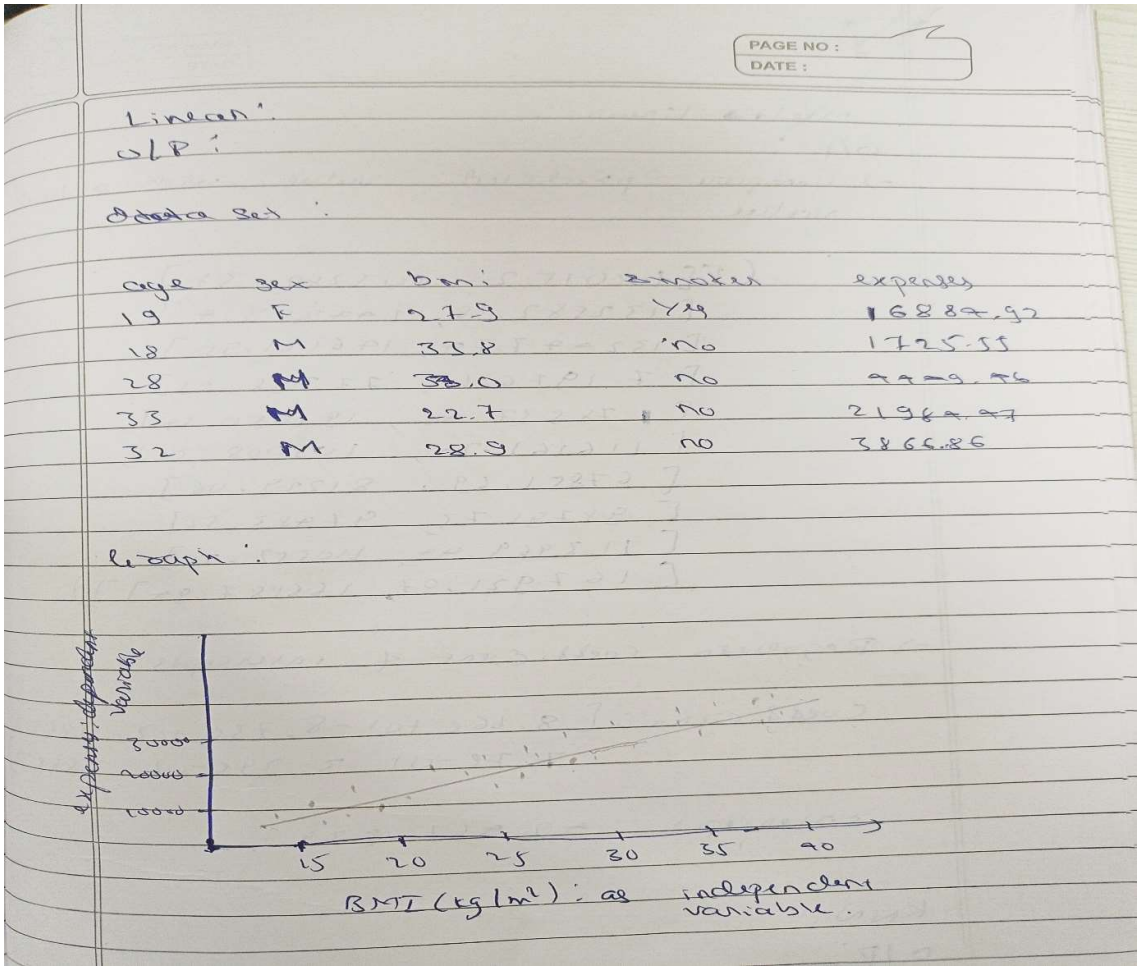
```
[ ]  def estimate_coef(x, y):
       # number of observations/points
       n = np.size(x)

       # mean of x and y vector
       m_x = np.mean(x)
       m_y = np.mean(y)

       # calculating cross-deviation and deviation about x
       SS_xy = np.sum(y*x) - n*m_y*m_x
       SS_xx = np.sum(x*x) - n*m_x*m_x

       # calculating regression coefficients
       b_1 = SS_xy / SS_xx
       b_0 = m_y - b_1*m_x

       return (b_0, b_1)
```

```
def plot_regression_line(x, y, b):
  # plotting the actual points as scatter plot
  plt.scatter(x, y, color = "m",
      marker = "o", s = 30)

  # predicted response vector
  y_pred = b[0] + b[1]*x

  # plotting the regression line
  plt.plot(x, y_pred, color = "g")

  # putting labels
  plt.xlabel('x')
  plt.ylabel('y')
```

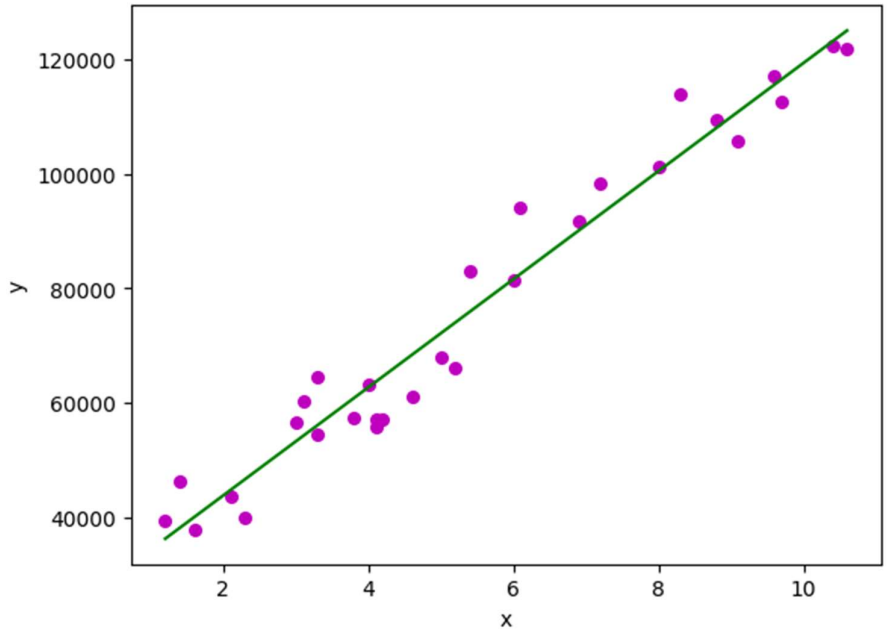
```
[ ]  import pandas as pd
     def main():
       # observations / data
       df=pd.read_csv("Salary_dataset.csv")
       x = np.array(df['YearsExperience'])
       y = np.array(df['Salary'])

       # estimating coefficients
       b = estimate_coef(x, y)
       plot_regression_line(x,y,b)
       print("Estimated coefficients:\nb_0 = {}\nb_1 = {}".format(b[0],b[1]))
       x=int(input("Enter X value:"))
       y=b[0]+b[1]*x
       print(f"The predicted salary is:{y}")
```

```
[ ]  main()
```

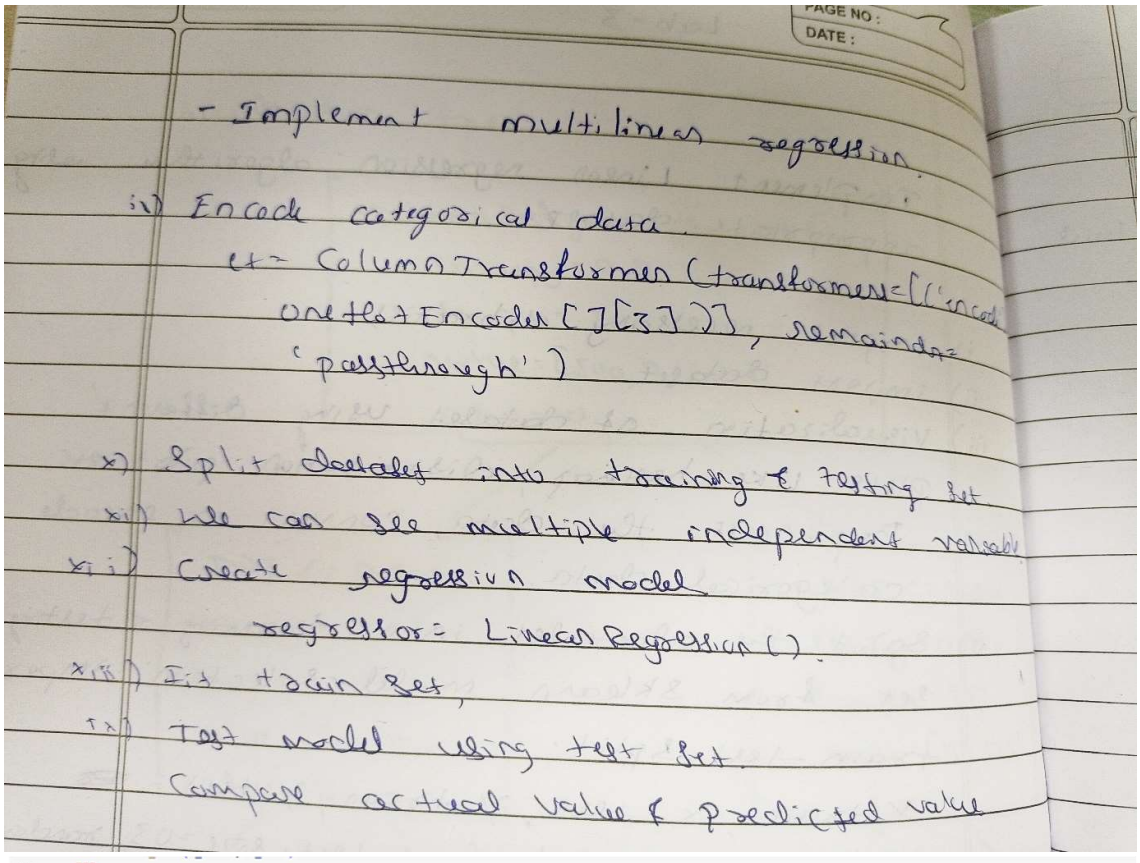
OUTPUT:



```
Estimated coefficients:
b_0 = 24848.203966523113
b_1 = 9449.962321455092
Enter X value45
The predicted salary is:450096.5084320023
```



b)Implement Multi-Linear Regression algorithm using appropriate dataset




```
[6]
    """
    Standardizes the input data using mean and standard deviation.

    Parameters:
        X_train (numpy.ndarray): Training data.
        X_test (numpy.ndarray): Testing data.

    Returns:
        Tuple of standardized training and testing data.
    """
    # Calculate the mean and standard deviation using the training data
    mean = np.mean(X_train, axis=0)
    std = np.std(X_train, axis=0)

    # Standardize the data
    X_train = (X_train - mean) / std
    X_test = (X_test - mean) / std

    return X_train, X_test

X_train, X_test = standardize_data(X_train, X_test)
```

```
[7] X_train = np.expand_dims(X_train, axis=-1)
    X_test = np.expand_dims(X_test, axis=-1)
```

```
[8] class LinearRegression:
    """
    Linear Regression Model with Gradient Descent

    Linear regression is a supervised machine learning algorithm used for modeling the relationship
    between a dependent variable (target) and one or more independent variables (features) by fitting
    a linear equation to the observed data.

    This class implements a linear regression model using gradient descent optimization for training.
    It provides methods for model initialization, training, prediction, and model persistence.

    Parameters:
        learning_rate (float): The learning rate used in gradient descent.
        convergence_tol (float, optional): The tolerance for convergence (stopping criterion). Defaults to 1e-6.

    Attributes:
```

[8]
```
    Attributes:
        W (numpy.ndarray): Coefficients (weights) for the linear regression model.
        b (float): Intercept (bias) for the linear regression model.

    Methods:
        initialize_parameters(n_features): Initialize model parameters.
        forward(X): Compute the forward pass of the linear regression model.
        compute_cost(predictions): Compute the mean squared error cost.
        backward(predictions): Compute gradients for model parameters.
        fit(X, y, iterations, plot_cost=True): Fit the linear regression model to training data.
        predict(X): Predict target values for new input data.
        save_model(filename=None): Save the trained model to a file using pickle.
        load_model(filename): Load a trained model from a file using pickle.

    Examples:
        >>> from linear_regression import LinearRegression
        >>> model = LinearRegression(learning_rate=0.01)
        >>> model.fit(X_train, y_train, iterations=1000)
        >>> predictions = model.predict(X_test)
    """

    def __init__(self, learning_rate, convergence_tol=1e-6):
        self.learning_rate = learning_rate
        self.convergence_tol = convergence_tol
        self.W = None
        self.b = None

    def initialize_parameters(self, n_features):
        """
        Initialize model parameters.

        Parameters:
            n_features (int): The number of features in the input data.
        """
        self.W = np.random.randn(n_features) * 0.01
        self.b = 0
    def forward(self, X):
        """
        Compute the forward pass of the linear regression model.

        Parameters:
            X (numpy.ndarray): Input data of shape (m, n_features).
```

[8]
```
        Returns:
            numpy.ndarray: Predictions of shape (m,).
        """
        return np.dot(X, self.W) + self.b

    def compute_cost(self, predictions):
        """
        Compute the mean squared error cost.

        Parameters:
            predictions (numpy.ndarray): Predictions of shape (m,).

        Returns:
            float: Mean squared error cost.
        """
        m = len(predictions)
        cost = np.sum(np.square(predictions - self.y)) / (2 * m)
        return cost

    def backward(self, predictions):
        """
        Compute gradients for model parameters.

        Parameters:
            predictions (numpy.ndarray): Predictions of shape (m,).

        Updates:
            numpy.ndarray: Gradient of W.
            float: Gradient of b.
        """
        m = len(predictions)
        self.dW = np.dot(predictions - self.y, self.X) / m
        self.db = np.sum(predictions - self.y) / m
    def fit(self, X, y, iterations, plot_cost=True):
        """
        Fit the linear regression model to the training data.

        Parameters:
            X (numpy.ndarray): Training input data of shape (m, n_features).
            y (numpy.ndarray): Training labels of shape (m,).
            iterations (int): The number of iterations for gradient descent.
            plot_cost (bool, optional): Whether to plot the cost during training. Defaults to True.
```

[8]

```
        Raises:
            AssertionError: If input data and labels are not NumPy arrays or have mismatched shapes.

        Plots:
            Plotly line chart showing cost vs. iteration (if plot_cost is True).
        """
        assert isinstance(X, np.ndarray), "X must be a NumPy array"
        assert isinstance(y, np.ndarray), "y must be a NumPy array"
        assert X.shape[0] == y.shape[0], "X and y must have the same number of samples"
        assert iterations > 0, "Iterations must be greater than 0"

        self.X = X
        self.y = y
        self.initialize_parameters(X.shape[1])
        costs = []

        for i in range(iterations):
            predictions = self.forward(X)
            cost = self.compute_cost(predictions)
            self.backward(predictions)
            self.W -= self.learning_rate * self.dW
            self.b -= self.learning_rate * self.db
            costs.append(cost)

            if i % 100 == 0:
                print(f'Iteration: {i}, Cost: {cost}')

            if i > 0 and abs(costs[-1] - costs[-2]) < self.convergence_tol:
                print(f'Converged after {i} iterations.')
                break

        if plot_cost:
            fig = px.line(y=costs, title="Cost vs Iteration", template="plotly_dark")
            fig.update_layout(
                title_font_color="#41BEE9",
                xaxis=dict(color="#41BEE9", title="Iterations"),
                yaxis=dict(color="#41BEE9", title="Cost")
            )

            fig.show()
```

```
            fig.show()
    def predict(self, X):
        """
        Predict target values for new input data.

        Parameters:
            X (numpy.ndarray): Input data of shape (m, n_features).

        Returns:
            numpy.ndarray: Predicted target values of shape (m,).
        """
        return self.forward(X)


    def save_model(self, filename=None):
        """
        Save the trained model to a file using pickle.

        Parameters:
            filename (str): The name of the file to save the model to.
        """
        model_data = {
            'learning_rate': self.learning_rate,
            'convergence_tol': self.convergence_tol,
            'W': self.W,
            'b': self.b
        }

        with open(filename, 'wb') as file:
            pickle.dump(model_data, file)

    @classmethod
    def load_model(cls, filename):
        """
        Load a trained model from a file using pickle.

        Parameters:
            filename (str): The name of the file to load the model from.
```

```
8]          with open(filename, 'wb') as file:
                pickle.dump(model_data, file)

        @classmethod
        def load_model(cls, filename):
            """
            Load a trained model from a file using pickle.

            Parameters:
                filename (str): The name of the file to load the model from.

            Returns:
                LinearRegression: An instance of the LinearRegression class with loaded parameters.
            """
            with open(filename, 'rb') as file:
                model_data = pickle.load(file)

            # Create a new instance of the class and initialize it with the loaded parameters
            loaded_model = cls(model_data['learning_rate'], model_data['convergence_tol'])
            loaded_model.W = model_data['W']
            loaded_model.b = model_data['b']

            return loaded_model
```

```
9]  lr = LinearRegression(0.01)
    lr.fit(X_train, y_train, 10000)
```

```
Iteration: 0, Cost: 1670.0184887161677
Iteration: 100, Cost: 227.15535101517312
Iteration: 200, Cost: 33.84101696145528
Iteration: 300, Cost: 7.9408253395546575
Iteration: 400, Cost: 4.4707260872934835
Iteration: 500, Cost: 4.005803317750673
Iteration: 600, Cost: 3.943513116253261
Iteration: 700, Cost: 3.9351674953098015
Iteration: 800, Cost: 3.9340493517293096
Converged after 863 iterations.
```

DATE:

Multiple linear :

O/P :

=) Compare predicted value with actual value

a ([[[103015.2, 103282.38],
[132582.28, 144259.4],
[132447.79, 146121.95],
[71976.1, 77798.83],
[78537.48, 191050.39],
[116761.24, 105008.31],
[67851.69, 81229.06],
[98791.73, 97483.56],
[113969.44, 110352.25],
[167921.07, 166187.94]]])

=) Regression Coefficient & intercepts.

Coefficient : [8.66e+01 -8.73e+02 7.86+02
7.7e-01 3.29e-02 3.66(-02)]

intercept : 42967.529 .

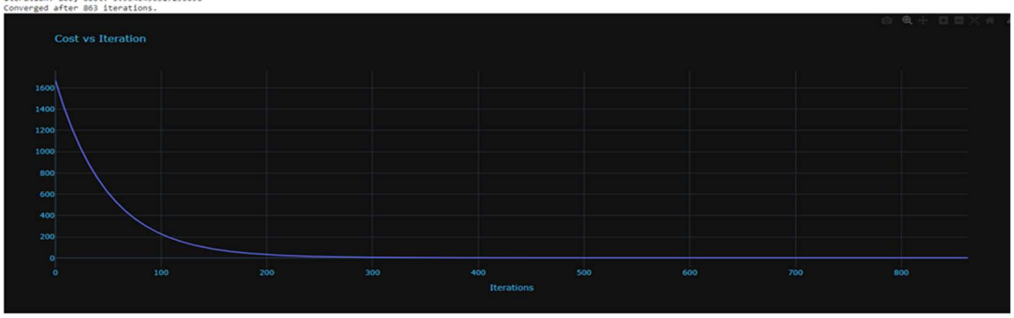
KNN :

O/P :

Accuracy with K=5 93.60000

Accuracy with K=1 90.4

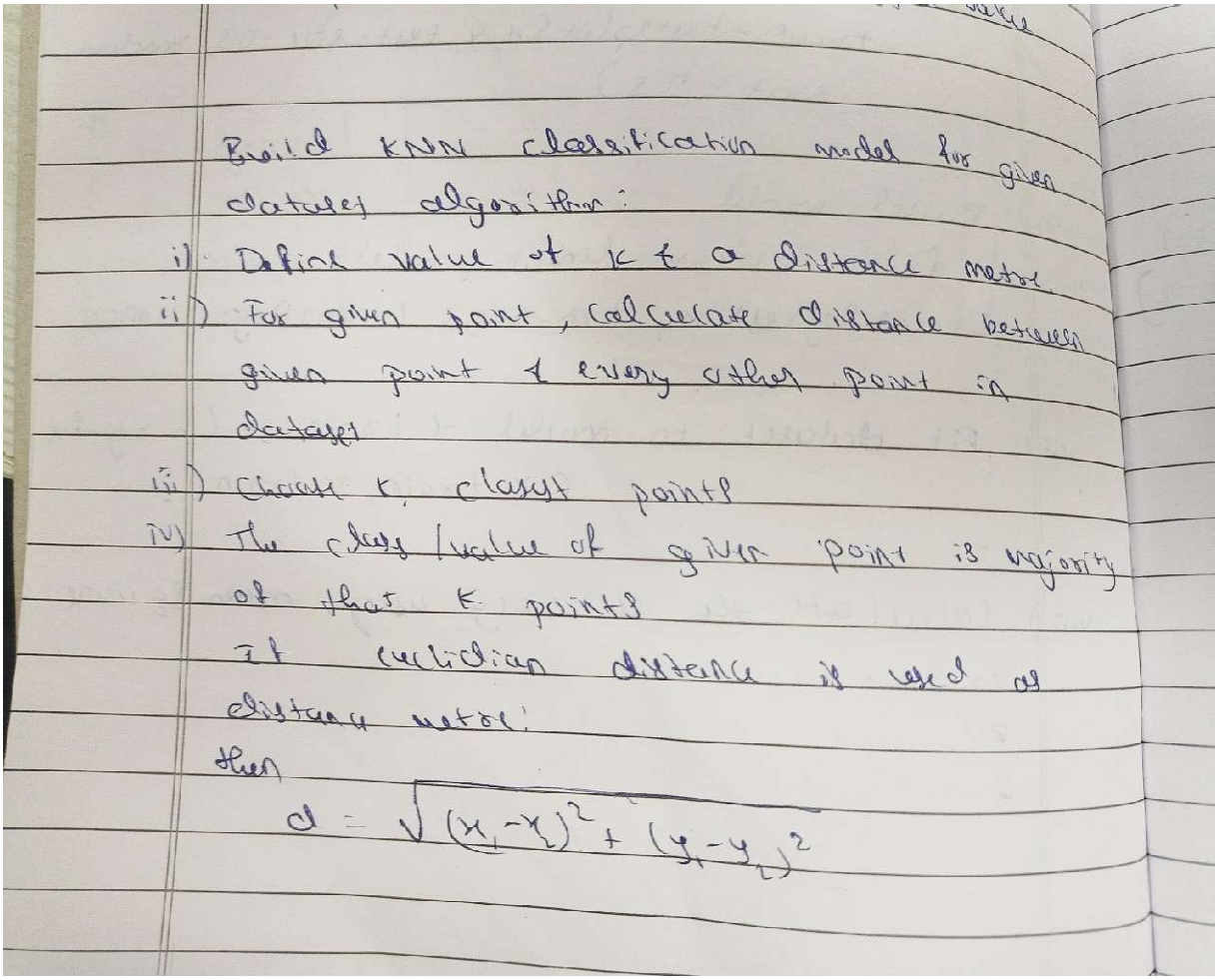
Iteration: 800, Cost: 3.9340493517293096
Converged after 863 iterations.



[10] lr.save_model('model.pkl')

[11] model = LinearRegression.load_model("model.pkl")

c)Build KNN Classification model for a given dataset.



Build KNN classification model for given dataset algorithm:

i) Define value of K & a distance metric.

ii) For given point, calculate distance between given point & every other point in dataset

iii) Choose K closest points

iv) The class value of given point is majority of that K points

If euclidian distance is used as distance metric then

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
[12] class RegressionMetrics:
        @staticmethod
        def mean_squared_error(y_true, y_pred):
            """
            Calculate the Mean Squared Error (MSE).

            Args:
                y_true (numpy.ndarray): The true target values.
                y_pred (numpy.ndarray): The predicted target values.

            Returns:
                float: The Mean Squared Error.
            """
            assert len(y_true) == len(y_pred), "Input arrays must have the same length."
            mse = np.mean((y_true - y_pred) ** 2)
            return mse

        @staticmethod
        def root_mean_squared_error(y_true, y_pred):
            """
            Calculate the Root Mean Squared Error (RMSE).

            Args:
                y_true (numpy.ndarray): The true target values.
                y_pred (numpy.ndarray): The predicted target values.

            Returns:
                float: The Root Mean Squared Error.
            """
            assert len(y_true) == len(y_pred), "Input arrays must have the same length."
            mse = RegressionMetrics.mean_squared_error(y_true, y_pred)
            rmse = np.sqrt(mse)
            return rmse

        @staticmethod
        def r_squared(y_true, y_pred):
            """
            Calculate the R-squared (R^2) coefficient of determination.

            Args:
```

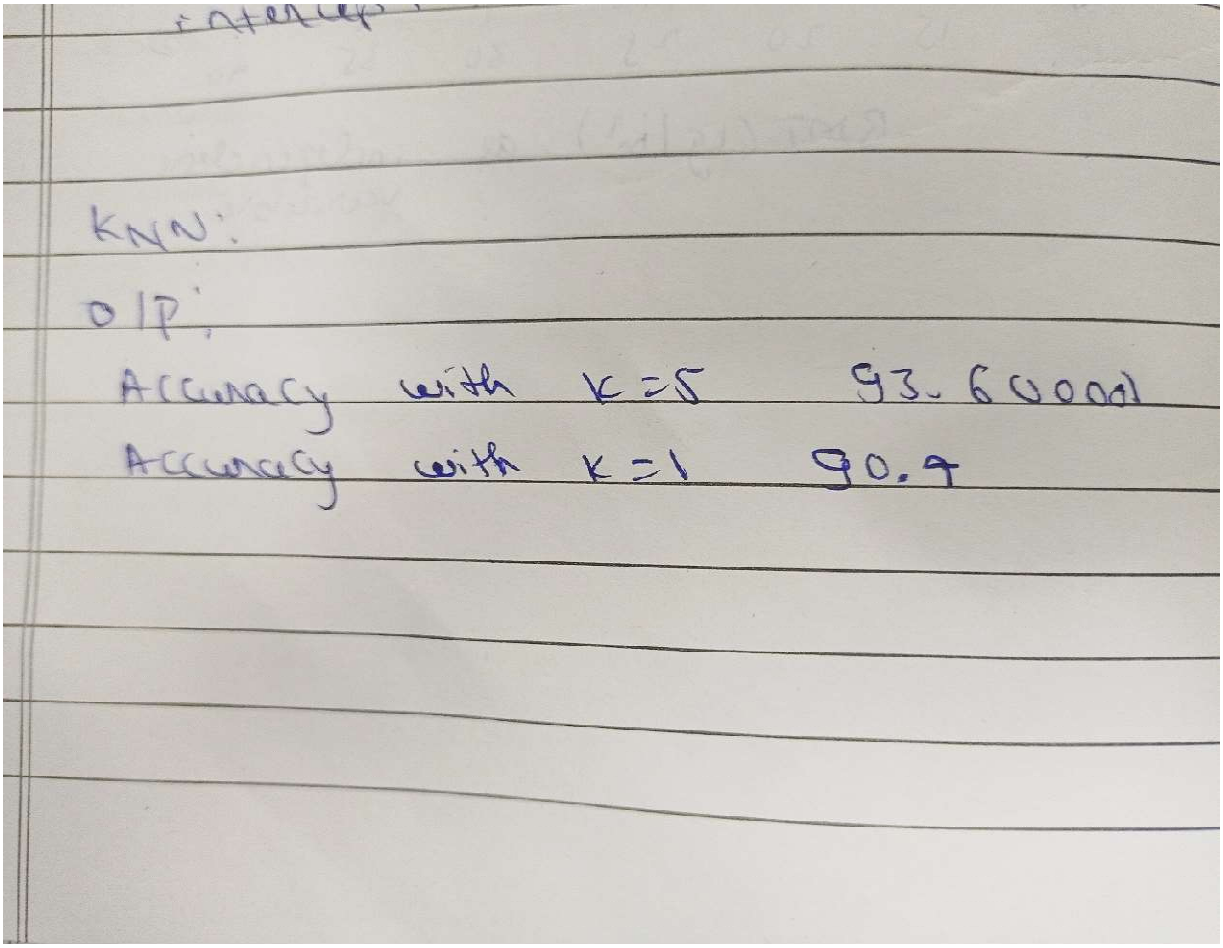
```
[12]    @staticmethod
        def r_squared(y_true, y_pred):
            """
            Calculate the R-squared (R^2) coefficient of determination.

            Args:
                y_true (numpy.ndarray): The true target values.
                y_pred (numpy.ndarray): The predicted target values.

            Returns:
                float: The R-squared (R^2) value.
            """
            assert len(y_true) == len(y_pred), "Input arrays must have the same length."
            mean_y = np.mean(y_true)
            ss_total = np.sum((y_true - mean_y) ** 2)
            ss_residual = np.sum((y_true - y_pred) ** 2)
            r2 = 1 - (ss_residual / ss_total)
            return r2
```

```
[13] y_pred = model.predict(X_test)
     mse_value = RegressionMetrics.mean_squared_error(y_test, y_pred)
     rmse_value = RegressionMetrics.root_mean_squared_error(y_test, y_pred)
     r_squared_value = RegressionMetrics.r_squared(y_test, y_pred)

     print(f"Mean Squared Error (MSE): {mse_value}")
     print(f"Root Mean Squared Error (RMSE): {rmse_value}")
     print(f"R-squared (Coefficient of Determination): {r_squared_value}")
```

```
Mean Squared Error (MSE): 9.44266965025894
Root Mean Squared Error (RMSE): 3.07289271701095
R-squared (Coefficient of Determination): 0.9887898724670081
```

```
model.predict([[2]])
```

```
array([107.82727115])
```


KNN:
O/P:
Accuracy with K=5 93.600001
Accuracy with K=1 90.9

Lab 4

1. Build Logistic Regression Model for a given dataset

```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import pprint
import pickle
```

```
[4] df = pd.read_csv('breast-cancer.csv')
```

```
[5] df.head()
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	radius_worst	texture_worst	perimeter_worst	are
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38	17.33	184.60	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	

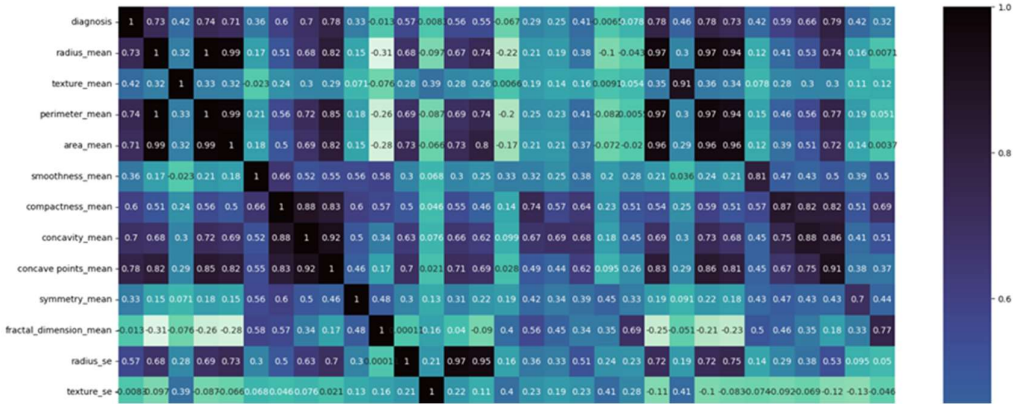
5 rows × 32 columns

```
[6] df.drop('id', axis=1, inplace=True) #drop redundant columns
```

```
[7] df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0
```

```
[8] corr = df.corr()
```

```
[9] plt.figure(figsize=(20,20))
    sns.heatmap(corr, cmap='mako_r',annot=True)
    plt.show()
```




```
[12] # Get the absolute value of the correlation
     cor_target = abs(corr["diagnosis"])

     # Select highly correlated features (thresold = 0.2)
     relevant_features = cor_target[cor_target>0.2]

     # Collect the names of the features
     names = [index for index, value in relevant_features.items()]

     # Drop the target variable from the results
     names.remove('diagnosis')

     # Display the results
     pprint.pprint(names)
```

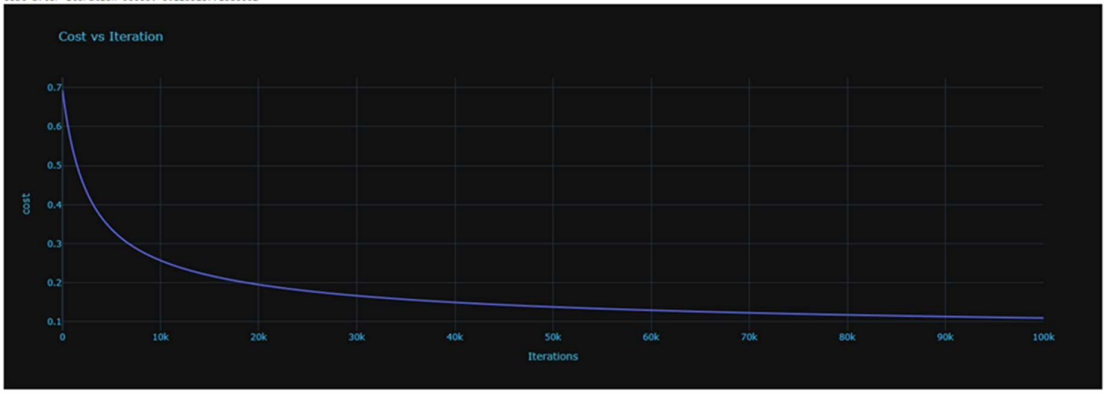
```
['radius_mean',
 'texture_mean',
 'perimeter_mean',
 'area_mean',
 'smoothness_mean',
 'compactness_mean',
 'concavity_mean',
 'concave points_mean',
 'symmetry_mean',
 'radius_se',
 'perimeter_se',
 'area_se',
 'compactness_se',
 'concavity_se',
 'concave points_se',
 'radius_worst',
 'texture_worst',
 'perimeter_worst',
 'area_worst',
 'smoothness_worst',
 'compactness_worst',
 'concavity_worst',
 'concave points_worst',
 'symmetry_worst',
 'fractal_dimension_worst']
```

```
[13] X = df[names].values
     y = df['diagnosis'].values
```

```
[14] def train_test_split(X, y, random_state=42, test_size=0.2):
         """
         Splits the data into training and testing sets.

         Parameters:
             X (numpy.ndarray): Features array of shape (n_samples, n_features).
             y (numpy.ndarray): Target array of shape (n_samples,).
             random_state (int): Seed for the random number generator. Default is 42.
             test_size (float): Proportion of samples to include in the test set. Default is 0.2.

         Returns:
             Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
         """
         # Get number of samples
         n_samples = X.shape[0]

         # Set the seed for the random number generator
         np.random.seed(random_state)

         # Shuffle the indices
         shuffled_indices = np.random.permutation(np.arange(n_samples))

         # Determine the size of the test set
         test_size = int(n_samples * test_size)

         # Split the indices into test and train
         test_indices = shuffled_indices[:test_size]
         train_indices = shuffled_indices[test_size:]

         # Split the features and target arrays into test and train
         X_train, X_test = X[train_indices], X[test_indices]
         y_train, y_test = y[train_indices], y[test_indices]

         return X_train, X_test, y_train, y_test
```

```
[ ] lg = LogisticRegression()
    lg.fit(X_train, y_train,100000)
```

```
Cost after iteration 0: 0.6931471605599454
Cost after iteration 10000: 0.2570778370558246
Cost after iteration 20000: 0.19529178673689726
Cost after iteration 30000: 0.16685820756163852
Cost after iteration 40000: 0.14978939548676498
Cost after iteration 50000: 0.1381876134031554
Cost after iteration 60000: 0.1296814121248933
Cost after iteration 70000: 0.1231144039988139
Cost after iteration 80000: 0.11785163708790082
Cost after iteration 90000: 0.113513771386002
```



```python
    @staticmethod
    def recall(y_true, y_pred):
        """
        Computes the recall (sensitivity) of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The recall of the model, which measures the proportion of true positive predictions
        out of all actual positive instances in the dataset.
        """
        true_positives = np.sum((y_true == 1) & (y_pred == 1))
        false_negatives = np.sum((y_true == 1) & (y_pred == 0))
        return true_positives / (true_positives + false_negatives)

    @staticmethod
    def f1_score(y_true, y_pred):
        """
        Computes the F1-score of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The F1-score of the model, which is the harmonic mean of precision and recall.
        """
        precision_value = ClassificationMetrics.precision(y_true, y_pred)
        recall_value = ClassificationMetrics.recall(y_true, y_pred)
        return 2 * (precision_value * recall_value) / (precision_value + recall_value)
```

```python
model = LogisticRegression.load_model("model.pkl")
```


```python
model = LogisticRegression.load_model("model.pkl")
```


```python
y_pred = model.predict(X_test)
accuracy = ClassificationMetrics.accuracy(y_test, y_pred)
precision = ClassificationMetrics.precision(y_test, y_pred)
recall = ClassificationMetrics.recall(y_test, y_pred)
f1_score = ClassificationMetrics.f1_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2%}")
print(f"Precision: {precision:.2%}")
print(f"Recall: {recall:.2%}")
print(f"F1-Score: {f1_score:.2%}")
```

```
Accuracy: 98.23%
Precision: 100.00%
Recall: 95.24%
F1-Score: 97.56%
```

PAGE NO :
DATE :

Lab-7

Build Logistic regression model for a given dataset.

Algorithm:

1) Import required libraries
2) Load, visualize & explore the dataset
3) Clean the dataset.
4) Deal with the outliers
5) Define dependent & independent variable & then split the data into a training set & testing set.
6) a Logistic regression modulating

o/p:

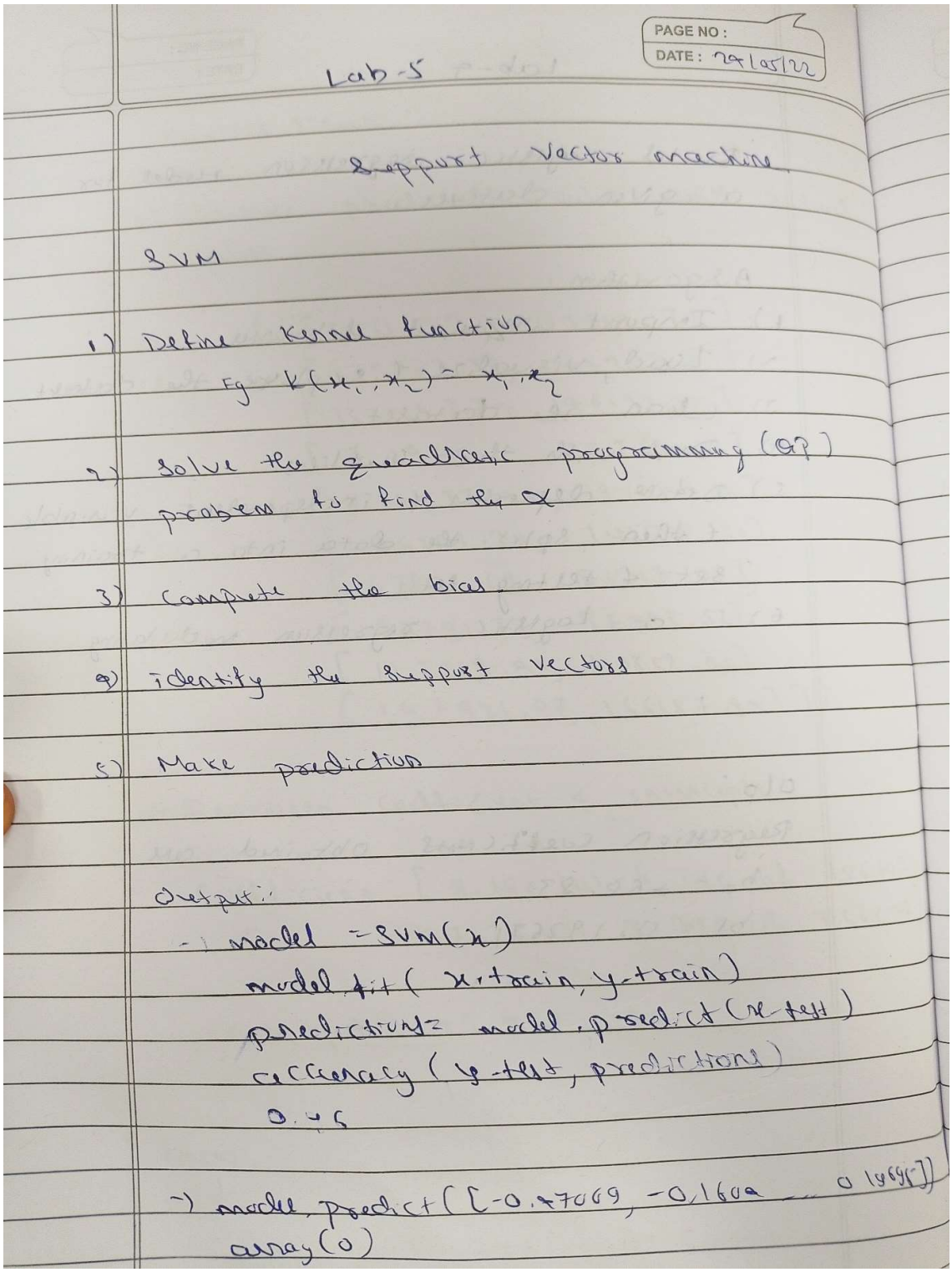
Regression coefficients obtained are

$b_0 = -68.83$

$b_1 = 0.192671$

LAB 5:

a)Build support vector machine for the dataset:



Lab-5

PAGE NO :
DATE : 29/05/22

Support Vector machine

SVM

1) Define Kernel function
   Eg $K(x_1, x_2) = x_1 \cdot x_2$

2) Solve the quadratic programming (QP) problem to find the α

3) Compute the bias

4) identify the support vectors

5) Make prediction

Output:

- model = SVM(x)
  model.fit(x_train, y_train)
  prediction = model.predict(x_test)
  accuracy(y_test, prediction)
  0.95

- model.predict([[-0.47069, -0.1602 ... 0.1966]])
  array(0)

SVM:

Code:

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

```
Mounted at /content/drive
```

```
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
```

```
df = pd.read_csv('/content/drive/MyDrive/breast-cancer.csv')
df.head()
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	rac
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	

5 rows × 32 columns

```
df.drop('id', axis=1, inplace=True) #drop redundant columns
```

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
radius_mean	569.0	14.127292	3.524049	6.981000	11.700000	13.370000	15.780000	28.11000
texture_mean	569.0	19.289649	4.301036	9.710000	16.170000	18.840000	21.800000	39.28000
perimeter_mean	569.0	91.969033	24.298981	43.790000	75.170000	86.240000	104.100000	188.50000
area_mean	569.0	654.889104	351.914129	143.500000	420.300000	551.100000	782.700000	2501.00000
smoothness_mean	569.0	0.096360	0.014064	0.052630	0.086370	0.095870	0.105300	0.16340
compactness_mean	569.0	0.104341	0.052813	0.019380	0.064920	0.092630	0.130400	0.34540
concavity_mean	569.0	0.088799	0.079720	0.000000	0.029560	0.061540	0.130700	0.42680
concave points_mean	569.0	0.048919	0.038803	0.000000	0.020310	0.033500	0.074000	0.20120
symmetry_mean	569.0	0.181162	0.027414	0.106000	0.161900	0.179200	0.195700	0.30400
fractal_dimension_mean	569.0	0.062798	0.007060	0.049960	0.057700	0.061540	0.066120	0.09744
radius_se	569.0	0.405172	0.277313	0.111500	0.232400	0.324200	0.478900	2.87300
texture_se	569.0	1.216853	0.551648	0.360200	0.833900	1.108000	1.474000	4.88500
perimeter_se	569.0	2.866059	2.021855	0.757000	1.606000	2.287000	3.357000	21.98000
area_se	569.0	40.337079	45.491006	6.802000	17.850000	24.530000	45.190000	542.20000
smoothness_se	569.0	0.007041	0.003003	0.001713	0.005169	0.006380	0.008146	0.03113

```
df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0
```

```
corr = df.corr()
```

```
# Get the absolute value of the correlation
cor_target = abs(corr["diagnosis"])

# Select highly correlated features (thresold = 0.2)
relevant_features = cor_target[cor_target>0.2]

# Collect the names of the features
names = [index for index, value in relevant_features.items()]

# Drop the target variable from the results
names.remove('diagnosis')

# Display the results
print(names)
```

```
['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave points_mean', 'symmetry_
```

```
X = df[names].values
y = df['diagnosis']
```

```
[12] def scale(X):
         """
         Standardizes the data in the array X.

         Parameters:
             X (numpy.ndarray): Features array of shape (n_samples, n_features).

         Returns:
             numpy.ndarray: The standardized features array.
         """
         # Calculate the mean and standard deviation of each feature
         mean = np.mean(X, axis=0)
         std = np.std(X, axis=0)

         # Standardize the data
         X = (X - mean) / std
         return X
```

```
[13] X = scale(X)
```

```
[14] def train_test_split(X, y, random_state=41, test_size=0.2):
         """
         Splits the data into training and testing sets.

         Parameters:
             X (numpy.ndarray): Features array of shape (n_samples, n_features).
             y (numpy.ndarray): Target array of shape (n_samples,).
             random_state (int): Seed for the random number generator. Default is 42.
             test_size (float): Proportion of samples to include in the test set. Default is 0.2.

         Returns:
             Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
         """
         # Get number of samples
         n_samples = X.shape[0]

         # Set the seed for the random number generator
         np.random.seed(random_state)

         # Shuffle the indices
         shuffled_indices = np.random.permutation(np.arange(n_samples))

         # Determine the size of the test set
         test_size = int(n_samples * test_size)

         # Split the indices into test and train
         test_indices = shuffled_indices[:test_size]
         train_indices = shuffled_indices[test_size:]

         # Split the features and target arrays into test and train
         X_train, X_test = X[train_indices], X[test_indices]
         y_train, y_test = y[train_indices], y[test_indices]

         return X_train, X_test, y_train, y_test
```

```
[15] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42) #split the  data into traing and validatin
```

```
[18] class SVM:
        """
        A Support Vector Machine (SVM) implementation using gradient descent.

        Parameters:
        -----------
        iterations : int, default=1000
            The number of iterations for gradient descent.
        lr : float, default=0.01
            The learning rate for gradient descent.
        lambdaa : float, default=0.01
            The regularization parameter.

        Attributes:
        -----------
        lambdaa : float
            The regularization parameter.
        iterations : int
            The number of iterations for gradient descent.
        lr : float
            The learning rate for gradient descent.
        w : numpy array
            The weights.
        b : float
            The bias.

        Methods:
        --------
        initialize_parameters(X)
            Initializes the weights and bias.
        gradient_descent(X, y)
            Updates the weights and bias using gradient descent.
        update_parameters(dw, db)
            Updates the weights and bias.
        fit(X, y)
            Fits the SVM to the data.
        predict(X)
            Predicts the labels for the given data.

        """

        def __init__(self, iterations=1000, lr=0.01, lambdaa=0.01):
            """
            Initializes the SVM model.

            Parameters:
            -----------
            iterations : int, default=1000
                The number of iterations for gradient descent.
            lr : float, default=0.01
                The learning rate for gradient descent.
            lambdaa : float, default=0.01
                The regularization parameter.
            """
            self.lambdaa = lambdaa
            self.iterations = iterations
            self.lr = lr
            self.w = None
            self.b = None
        def initialize_parameters(self, X):
            """
            Initializes the weights and bias.

            Parameters:
            -----------
```

[18]

```
        X : numpy array
            The input data.
        """
        m, n = X.shape
        self.w = np.zeros(n)
        self.b = 0

    def gradient_descent(self, X, y):
        """
        Updates the weights and bias using gradient descent.

        Parameters:
        -----------
        X : numpy array
            The input data.
        y : numpy array
            The target values.
        """
        y_ = np.where(y <= 0, -1, 1)
        for i, x in enumerate(X):
            if y_[i] * (np.dot(x, self.w) - self.b) >= 1:
                dw = 2 * self.lambdaa * self.w
                db = 0
            else:
                dw = 2 * self.lambdaa * self.w - np.dot(x, y_[i])
                db = y_[i]
            self.update_parameters(dw, db)

    def update_parameters(self, dw, db):
        """
        Updates the weights and bias.

        Parameters:
```

[18]

```
          dw : numpy array
              The change in weights.
          db : float
              The change in bias.
          """
          self.w = self.w - self.lr * dw
          self.b = self.b - self.lr * db
      def fit(self, X, y):
          """
          Fits the SVM to the data.

          Parameters:
          -----------
          X : numpy array
              The input data.
          y : numpy array
              The target values.
          """
          self.initialize_parameters(X)
          for i in range(self.iterations):
              self.gradient_descent(X, y)


      def predict(self, X):
          """
          Predicts the class labels for the test data.

          Parameters
          ----------
          X : array-like, shape (n_samples, n_features)
              The input data.

          Returns
```

```
[18]
        Returns
        -------
        y_pred : array-like, shape (n_samples,)
            The predicted class labels.

        """
        # get the outputs
        output = np.dot(X, self.w) - self.b
        # get the signs of the labels depending on if it's greater/less than zero
        label_signs = np.sign(output)
        #set predictions to 0 if they are less than or equal to -1 else set them to 1
        predictions = np.where(label_signs <= -1, 0, 1)
        return predictions
```

```
[19] def accuracy(y_true, y_pred):
        """
        Computes the accuracy of a classification model.

        Parameters:
        ----------
            y_true (numpy array): A numpy array of true labels for each data point.
            y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        ----------
            float: The accuracy of the model
        """
        total_samples = len(y_true)
        correct_predictions = np.sum(y_true == y_pred)
        return (correct_predictions / total_samples)
```

```
[20] model = SVM()
     model.fit(X_train,y_train)
     predictions = model.predict(X_test)

     accuracy(y_test, predictions)
```

```
0.9823008849557522
```

```
[28] model.predict([-0.47069438, -0.16048584, -0.44810956, -0.49199876,  0.23411429,
            0.02765051, -0.10984741, -0.27623152,  0.41394897, -0.03274296,
           -0.18269561, -0.22105292, -0.35591235, -0.16192949, -0.23133322,
           -0.26903951, -0.16890536, -0.33393537, -0.35629925,  0.4485028 ,
           -0.10474068, -0.02441212, -0.19956318,  0.18320441,  0.19695794])
```

```
array(0)
```

b)K-MEANS Algorithm:

Code:

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

```
Mounted at /content/drive
```

```
[2]  import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import plotly.express as px
     import seaborn as sns
     import plotly.graph_objects as go
```

```
[3]  iris = pd.read_csv("/content/drive/MyDrive/Iris.csv") #Load Data
     iris.drop('Id',inplace=True,axis=1) #Drop Id column
```

```
[4]  X = iris.iloc[:,:-1] #Set our training data

     y = iris.iloc[:,-1] #We'll use this just for visualization as clustering doesn't require labels
```

```
[5]  class Kmeans:
         """
         K-Means clustering algorithm implementation.

         Parameters:
             K (int): Number of clusters

         Attributes:
             K (int): Number of clusters
             centroids (numpy.ndarray): Array containing the centroids of each cluster
```

```
[5]      Methods:
             __init__(self, K): Initializes the Kmeans instance with the specified number of clusters.
             initialize_centroids(self, X): Initializes the centroids for each cluster by selecting K random points from the dataset.
             assign_points_centroids(self, X): Assigns each point in the dataset to the nearest centroid.
             compute_mean(self, X, points): Computes the mean of the points assigned to each centroid.
             fit(self, X, iterations=10): Clusters the dataset using the K-Means algorithm.
         """

         def __init__(self, K):
             assert K > 0, "K should be a positive integer."
             self.K = K

         def initialize_centroids(self, X):
             assert X.shape[0] >= self.K, "Number of data points should be greater than or equal to K."

             randomized_X = np.random.permutation(X.shape[0])
             centroid_idx = randomized_X[:self.K] # get the indices for the centroids
             self.centroids = X[centroid_idx] # assign the centroids to the selected points

         def assign_points_centroids(self, X):
             """
             Assign each point in the dataset to the nearest centroid.

             Parameters:
             X (numpy.ndarray): dataset to cluster

             Returns:
             numpy.ndarray: array containing the index of the centroid for each point
             """
             X = np.expand_dims(X, axis=1) # expand dimensions to match shape of centroids
             distance = np.linalg.norm((X - self.centroids), axis=-1) # calculate Euclidean distance between each point and each centroid
             points = np.argmin(distance, axis=1) # assign each point to the closest centroid
             assert len(points) == X.shape[0], "Number of assigned points should equal the number of data points."
```

```
[5]         points = np.argmin(distance, axis=1) # assign each point to the closest centroid
            assert len(points) == X.shape[0], "Number of assigned points should equal the number of data points."
            return points

        def compute_mean(self, X, points):
            """
            Compute the mean of the points assigned to each centroid.

            Parameters:
            X (numpy.ndarray): dataset to cluster
            points (numpy.ndarray): array containing the index of the centroid for each point

            Returns:
            numpy.ndarray: array containing the new centroids for each cluster
            """
            centroids = np.zeros((self.K, X.shape[1])) # initialize array to store centroids
            for i in range(self.K):
                centroid_mean = X[points == i].mean(axis=0) # calculate mean of the points assigned to the current centroid
                centroids[i] = centroid_mean # assign the new centroid to the mean of its points
            return centroids

        def fit(self, X, iterations=10):
            """
            Cluster the dataset using the K-Means algorithm.

            Parameters:
            X (numpy.ndarray): dataset to cluster
            iterations (int): number of iterations to perform (default=10)

            Returns:
            numpy.ndarray: array containing the final centroids for each cluster
            numpy.ndarray: array containing the index of the centroid for each point
            """
            self.initialize_centroids(X) # initialize the centroids
            for i in range(iterations):
```

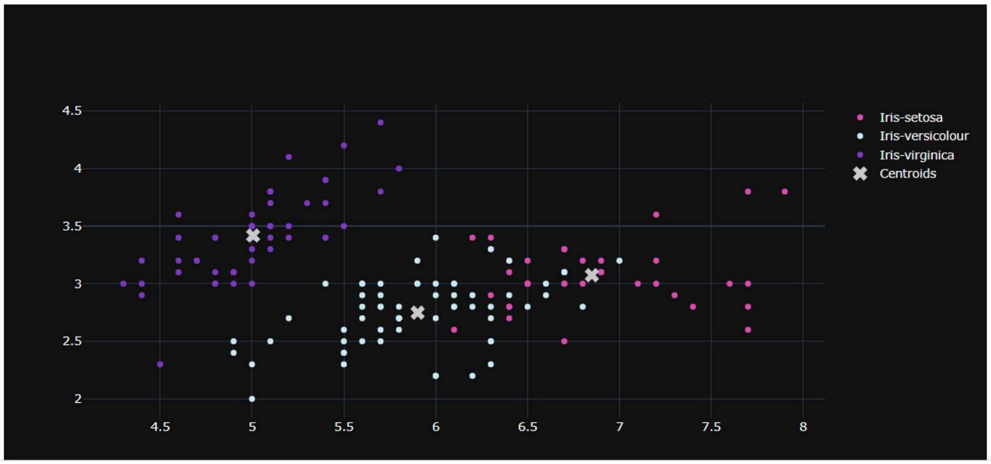
```
[5]         """
            self.initialize_centroids(X) # initialize the centroids
            for i in range(iterations):
                points = self.assign_points_centroids(X) # assign each point to the nearest centroid
                self.centroids = self.compute_mean(X, points) # compute the new centroids based on the mean of their points

                # Assertions for debugging and validation
                assert len(self.centroids) == self.K, "Number of centroids should equal K."
                assert X.shape[1] == self.centroids.shape[1], "Dimensionality of centroids should match input data."
                assert max(points) < self.K, "Cluster index should be less than K."
                assert min(points) >= 0, "Cluster index should be non-negative."

            return self.centroids, points
```

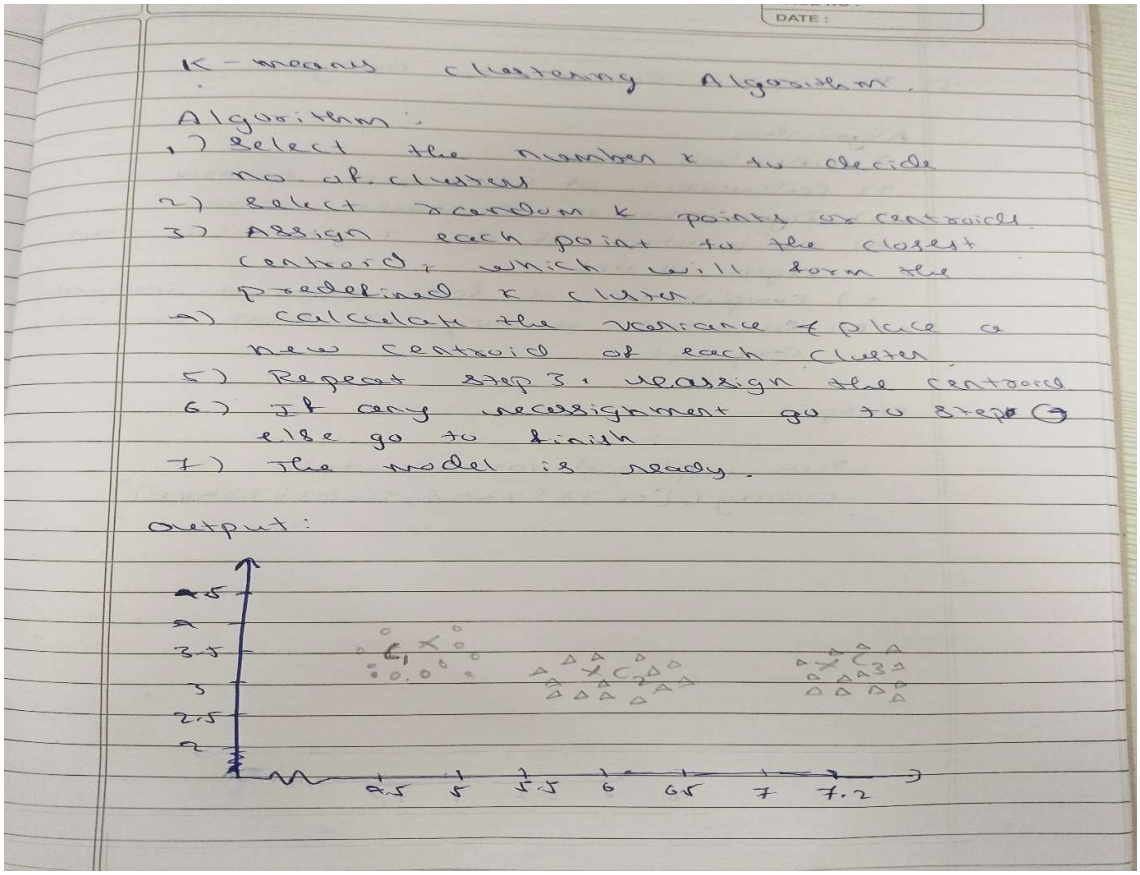
```
[6] X = X.values
```

```
[7] kmeans = Kmeans(3)

    centroids, points = kmeans.fit(X, 1000)
```

```
[8] fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=X[points == 0, 0], y=X[points == 0, 1],
        mode='markers',marker_color='#DB4CB2',name='Iris-setosa'
    ))

    fig.add_trace(go.Scatter(
        x=X[points == 1, 0], y=X[points == 1, 1],
        mode='markers',marker_color='#c9e9f6',name='Iris-versicolour'
    ))
```

```
[8] ))

    fig.add_trace(go.Scatter(
        x=X[points == 2, 0], y=X[points == 2, 1],
        mode='markers',marker_color='#7D3AC1',name='Iris-virginica'
    ))

    fig.add_trace(go.Scatter(
        x=centroids[:, 0], y=centroids[:,1],
        mode='markers',marker_color='#CAC9CD',marker_symbol=4,marker_size=13,name='Centroids'
    ))
    fig.update_layout(template='plotly_dark',width=1000, height=500,)
```





c)Principal Component Analysis:

Code:

+ Code + Text

RAM Disk

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

```
Mounted at /content/drive
```

```
[2] import seaborn as sns
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    import plotly.express as px
    import plotly.graph_objects as go
    from plotly.subplots import make_subplots
```

```
[3] df = pd.read_csv('/content/drive/MyDrive/breast-cancer.csv')
    df.head()
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	radius_worst
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54

5 rows × 32 columns

```
[4] df.drop('id', axis=1, inplace=True) #drop redundant columns
```

```
[5] df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0
```

```
[6] corr = df.corr()
```

```
[8] # Get the absolute value of the correlation
    cor_target = abs(corr["diagnosis"])

    # Select highly correlated features (thresold = 0.2)
    relevant_features = cor_target[cor_target>0.2]

    # Collect the names of the features
    names = [index for index, value in relevant_features.items()]

    # Drop the target variable from the results
    names.remove('diagnosis')

    # Display the results
    print(names)
```

```
['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave p
```

```
[9] X = df[names].values
```

```
[9]  X = df[names].values
```

```
[11] class PCA:
         """
         Principal Component Analysis (PCA) class for dimensionality reduction.
         """

         def __init__(self, n_components):
             """
             Constructor method that initializes the PCA object with the number of components to retain.

             Args:
             - n_components (int): Number of principal components to retain.
             """
             self.n_components = n_components
         def fit(self, X):
             """
             Fits the PCA model to the input data and computes the principal components.

             Args:
             - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).
             """
             # Compute the mean of the input data along each feature dimension.
             mean = np.mean(X, axis=0)

             # Subtract the mean from the input data to center it around zero.
             X = X - mean

             # Compute the covariance matrix of the centered input data.
             cov = np.cov(X.T)
```

```
[11]         # Compute the covariance matrix of the centered input data.
             cov = np.cov(X.T)

             # Compute the eigenvectors and eigenvalues of the covariance matrix.
             eigenvalues, eigenvectors = np.linalg.eigh(cov)
             # Reverse the order of the eigenvalues and eigenvectors.
             eigenvalues = eigenvalues[::-1]
             eigenvectors = eigenvectors[:,::-1]

             # Keep only the first n_components eigenvectors as the principal components.
             self.components = eigenvectors[:,:self.n_components]

             # Compute the explained variance ratio for each principal component.
             # Compute the total variance of the input data
             total_variance = np.sum(np.var(X, axis=0))

             # Compute the variance explained by each principal component
             self.explained_variances = eigenvalues[:self.n_components]

             # Compute the explained variance ratio for each principal component
             self.explained_variance_ratio_ = self.explained_variances / total_variance
         def transform(self, X):
             """
             Transforms the input data by projecting it onto the principal components.

             Args:
             - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).

             Returns:
             - transformed_data (numpy.ndarray): Transformed data matrix with shape (n_samples, n_components).
             """
             # Center the input data around zero using the mean computed during the fit step.
             X = X - np.mean(X, axis=0)
```

```
[11]
        # Project the centered input data onto the principal components.
        transformed_data = np.dot(X, self.components)

        return transformed_data

    def fit_transform(self, X):
        """
        Fits the PCA model to the input data and computes the principal components then
        transforms the input data by projecting it onto the principal components.

        Args:
        - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).
        """
        self.fit(X)
        transformed_data = self.transform(X)
        return transformed_data
```

```
[12] pca = PCA(2)
```

```
[13] pca.fit(X)
```

```
[14] pca.explained_variance_ratio_
```

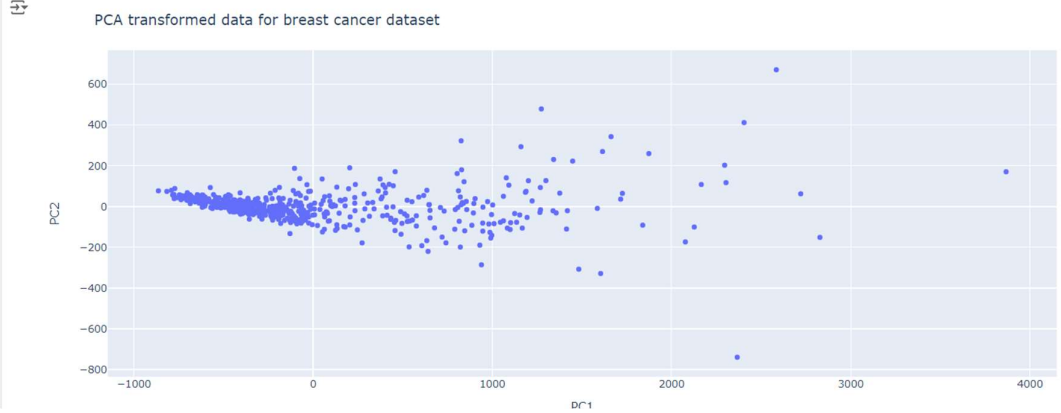
```
array([0.98377428, 0.01620498])
```

```
[15] X_transformed = pca.transform(X)
```

```
[16] X_transformed[:,1].shape
```

```
(569,)
```

```
fig = px.scatter(x=X_transformed[:,0], y=X_transformed[:,1])
fig.update_layout(
    title="PCA transformed data for breast cancer dataset",
    xaxis_title="PC1",
    yaxis_title="PC2"
)
fig.show()
```



3) Principal Component Analysis.

Algorithm:

1) Calculate mean
2) Calculation of Covariance matrix
3) Eigen values of Covariance matrix
4) Computation of the Eigen vector - unit eigen vectors
5) Computation of first principal component
6) Geometric meaning of first principal components.

Output:

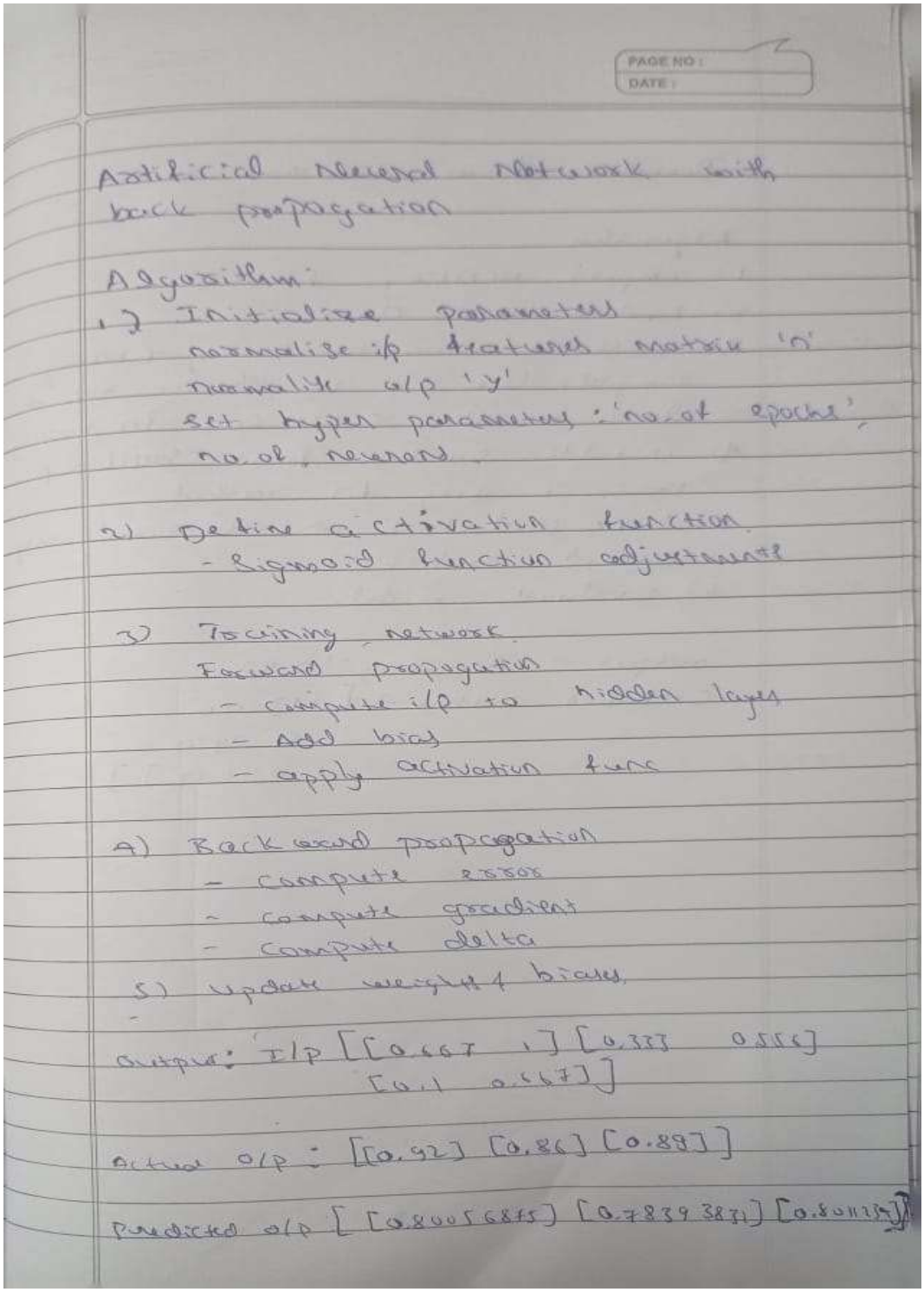
pca.explained_variance_ratio_
array([0.9837428, 0.01620968])

Lab – 6

Date : 31/05/2024

1. Build Artificial Neural Network model with back propagation on a given dataset.

Observation Screenshot :



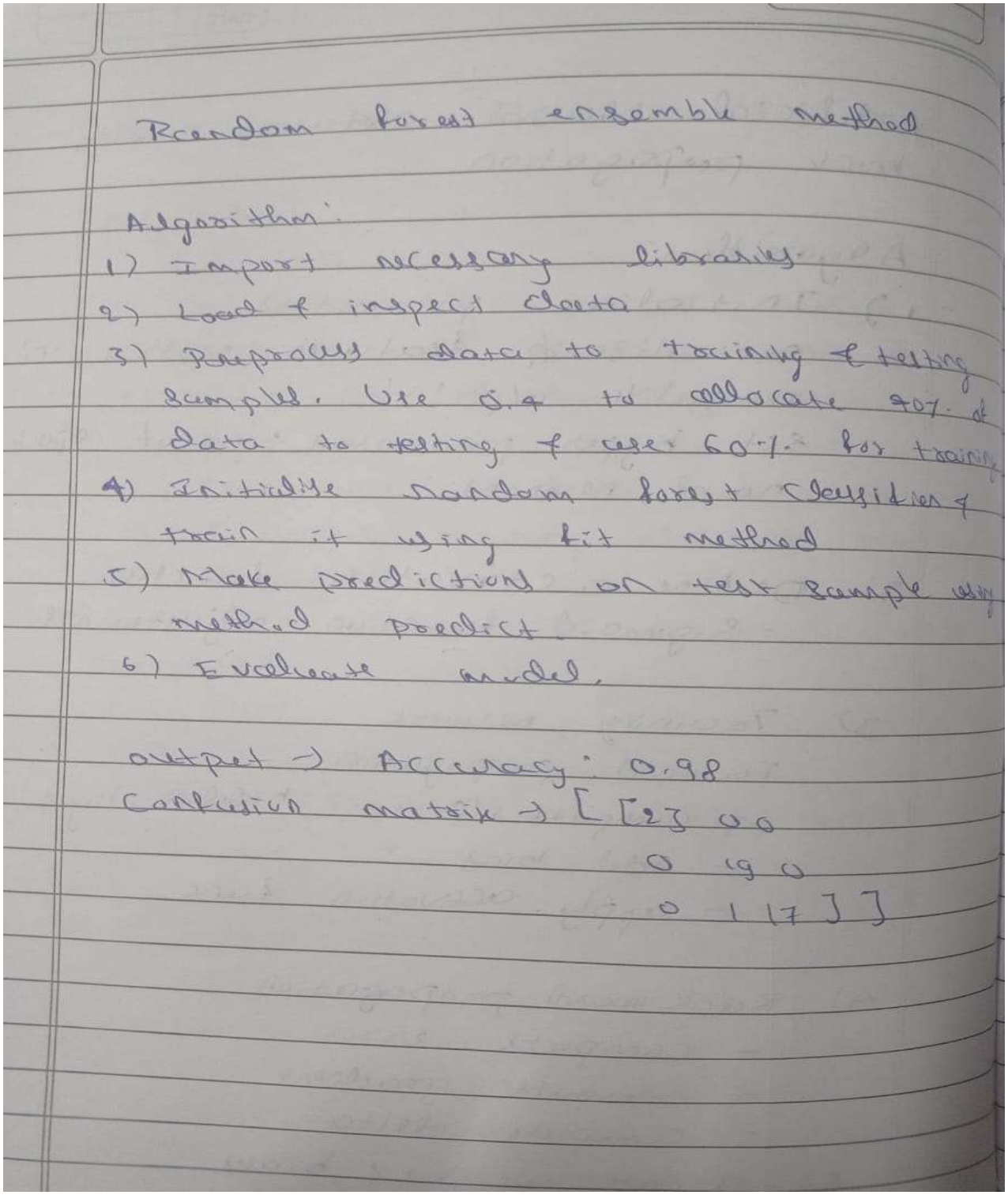
Code and Output :

```python
import numpy as np
x = np.array(([2,9],[1,5],[3,6]),dtype = float)
y = np.array(([92],[86],[89]),dtype = float)
x = x/np.amax(x,axis=0)
y = y/100


#Varialble Initialization
epoch = 5000
lr = 0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1


# weight and bias Initialization
wh = np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh = np.random.uniform(size=(1,hiddenlayer_neurons))
wout = np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout = np.random.uniform(size=(1,output_neurons))


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

# Derivative of Sigmoid
def der_sigmoid(x):
  return x*(1-x)


# Draws a random range of numbers uniformly of dim x*y

for i in range(epoch):

  # forward propagation
  hinp1 = np.dot(x,wh)
  hinp = hinp1 + bh
  hlayer_act = sigmoid(hinp)
  outinp1 = np.dot(hlayer_act,wout)
  outinp = outinp1 + bout
  output = sigmoid(outinp)

  # Backpropagation
  EO = y - output
  outgrad = der_sigmoid(output)
  d_output = EO*outgrad
  EH = d_output.dot(wout.T)

  # how much hidden layer weights contributed to error
  hiddengrad = der_sigmoid(hlayer_act)
  d_hiddenlayer = EH*hiddengrad

  #dotproduct of nextlayererror and current layer op
  wout += hlayer_act.T.dot(d_output)*lr
  wh += x.T.dot(d_hiddenlayer)*lr

  print("Input: \n" + str(x))
  print("Actual output: \n" + str(y))
  print("Predicted Output: \n",output)
```

```
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.80056875]
 [0.79393831]
 [0.80112347]]
```

2. *Implement Random forest ensemble method on a given dataset.*

Observation Screenshot :



Code and Output :

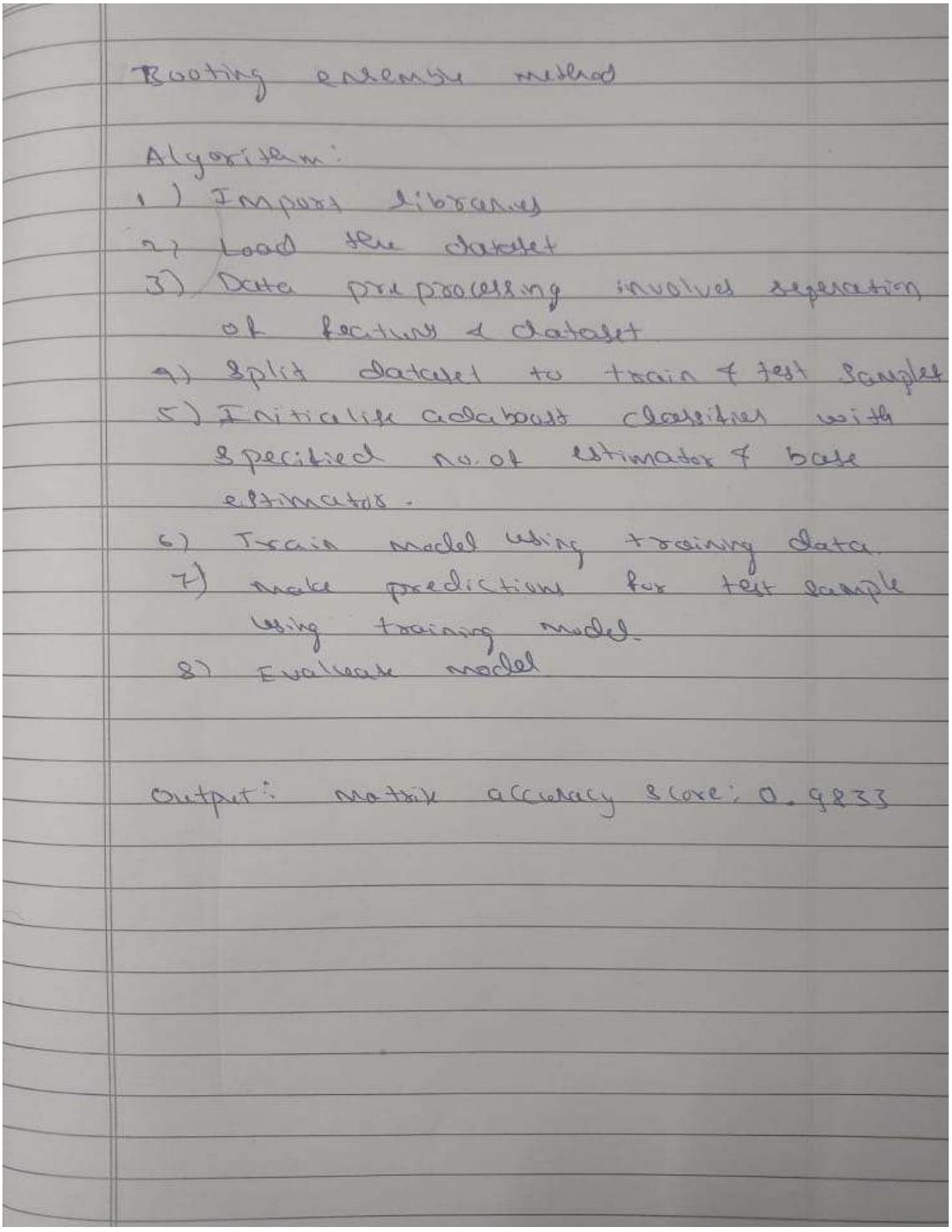
```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn import datasets

# Load the data
iris_data = datasets.load_iris()

X = pd.DataFrame(iris_data.data, columns=['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'])
y = pd.DataFrame(iris_data.target, columns=['Targets'])


# Check the info of the modified data
# print(iris_data.info())

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
rf_classifier.fit(X_train, y_train)

# Predict on the test data
y_pred = rf_classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Print classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Print confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

✓ 0.7s

```
Accuracy: 0.98
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        23
           1       0.95      1.00      0.97        19
           2       1.00      0.94      0.97        18

    accuracy                           0.98        60
   macro avg       0.98      0.98      0.98        60
weighted avg       0.98      0.98      0.98        60

Confusion Matrix:
[[23  0  0]
 [ 0 19  0]
 [ 0  1 17]]
```

3. *Implement Boosting ensemble method on a given dataset*

Observation Screenshot :



Code and Output :

```
[10] from sklearn.linear_model import LogisticRegression
     from sklearn.ensemble import AdaBoostClassifier
     from sklearn import metrics
     from sklearn import datasets
```

```
[11] import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn.model_selection import train_test_split
```

```
[12] # Load the iris dataset
     iris = datasets.load_iris()
     X = pd.DataFrame(iris.data, columns=['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'])
     y = pd.DataFrame(iris.target, columns=['Targets'])
```

```
[13] X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.4,random_state=42)
```

```
[14] mylogregmodel = LogisticRegression()
```

```
[15] adabc = AdaBoostClassifier(n_estimators = 150, estimator = mylogregmodel, learning_rate = 1)
```

```
     model = adabc.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py:1143: DataConversionWarning: A
  y = column_or_1d(y, warn=True)
```

```
[17] y_pred = model.predict(X_test)
```

```
[18] metrics.accuracy_score(y_test, y_pred)
```

```
0.9833333333333333
```