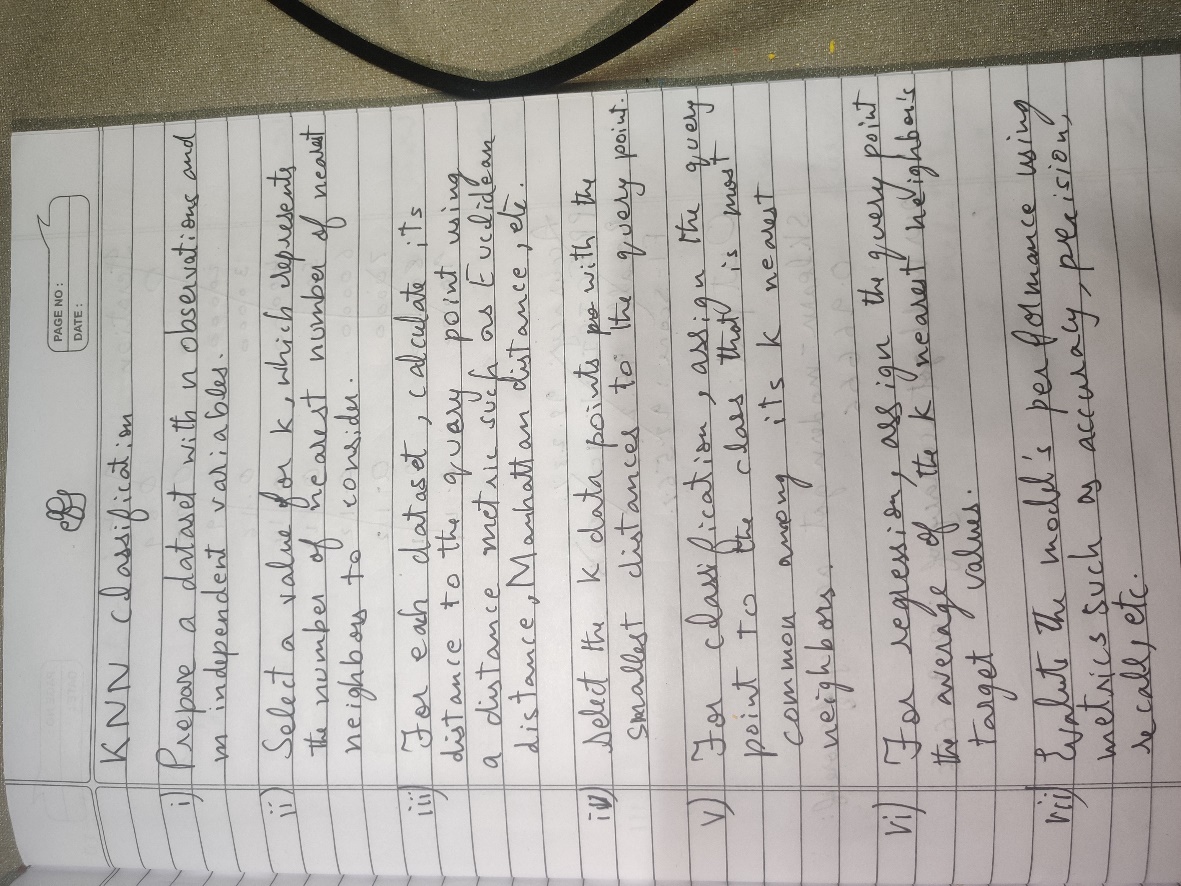
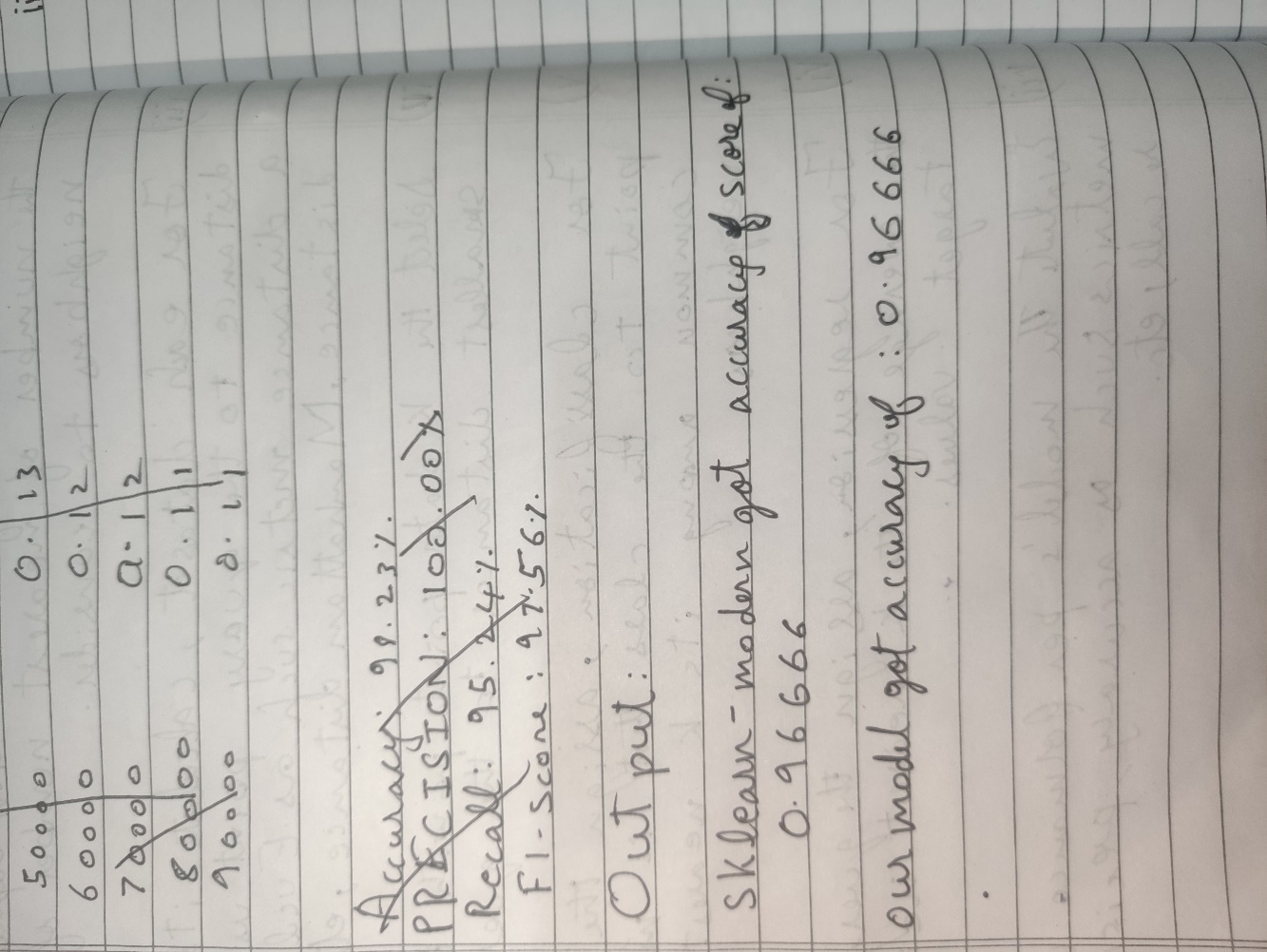
**LAB-4**

**Build KNN Classification model for a given dataset.**

**Observation:**





**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import plotly.express as px

import seaborn as sns

In [ ]:

iris = pd.read\_csv("Iris.csv") *#Load Data*

iris.drop('Id',inplace=True,axis=1) *#Drop Id column*

In [ ]:

iris.head()

In [ ]:

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

y = iris.iloc[:,-1] *#Set training labels*

In [ ]:

fig = px.pie(iris, 'Species',color\_discrete\_sequence=['#ffffd4 ','#fe9929 ','#993404 '],title='Data Distribution',template='plotly\_dark')

fig.show()

In [ ]:

fig = px.scatter(data\_frame=iris, x='SepalLengthCm',y='SepalWidthCm'

,color='Species',size='PetalLengthCm', color\_discrete\_sequence=['#ffffd4 ','#fe9929 ','#993404 '], template='plotly\_dark',)

fig.show()

In [ ]:

fig = px.scatter(data\_frame=iris, x='PetalLengthCm',y='PetalWidthCm'

,color='Species',size='SepalLengthCm', color\_discrete\_sequence=['#ffffd4 ','#fe9929 ','#993404 '], template='plotly\_dark',)

fig.show()

In [ ]:

class KNN:

"""

K-Nearest Neighbors (KNN) classification algorithm

Parameters:

-----------

n\_neighbors : int, optional (default=5)

Number of neighbors to use in the majority vote.

Methods:

--------

fit(X\_train, y\_train):

Stores the values of X\_train and y\_train.

predict(X):

Predicts the class labels for each example in X.

"""

def \_\_init\_\_(self, n\_neighbors=5):

self.n\_neighbors = n\_neighbors

def euclidean\_distance(self, x1, x2):

"""

Calculate the Euclidean distance between two data points.

Parameters:

-----------

x1 : numpy.ndarray, shape (n\_features,)

A data point in the dataset.

x2 : numpy.ndarray, shape (n\_features,)

A data point in the dataset.

Returns:

--------

distance : float

The Euclidean distance between x1 and x2.

"""

return np.linalg.norm(x1 - x2)

def fit(self, X\_train, y\_train):

"""

Stores the values of X\_train and y\_train.

Parameters:

-----------

X\_train : numpy.ndarray, shape (n\_samples, n\_features)

The training dataset.

y\_train : numpy.ndarray, shape (n\_samples,)

The target labels.

"""

self.X\_train = X\_train

self.y\_train = y\_train

def predict(self, X):

"""

Predicts the class labels for each example in X.

Parameters:

-----------

X : numpy.ndarray, shape (n\_samples, n\_features)

The test dataset.

Returns:

--------

predictions : numpy.ndarray, shape (n\_samples,)

The predicted class labels for each example in X.

"""

*# Create empty array to store the predictions*

predictions = []

*# Loop over X examples*

for x in X:

*# Get prediction using the prediction helper function*

prediction = self.\_predict(x)

*# Append the prediction to the predictions list*

predictions.append(prediction)

return np.array(predictions)

def \_predict(self, x):

"""

Predicts the class label for a single example.

Parameters:

-----------

x : numpy.ndarray, shape (n\_features,)

A data point in the test dataset.

Returns:

--------

most\_occuring\_value : int

The predicted class label for x.

"""

*# Create empty array to store distances*

distances = []

*# Loop over all training examples and compute the distance between x and all the training examples*

for x\_train in self.X\_train:

distance = self.euclidean\_distance(x, x\_train)

distances.append(distance)

distances = np.array(distances)

*# Sort by ascendingly distance and return indices of the given n neighbours*

n\_neighbors\_idxs = np.argsort(distances)[: self.n\_neighbors]

*# Get labels of n-neighbour indexes*

labels = self.y\_train[n\_neighbors\_idxs]

labels = list(labels)

*# Get the most frequent class in the array*

most\_occuring\_value = max(labels, key=labels.count)

return most\_occuring\_value

In [ ]:

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

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X.values, y.values, test\_size = 0.2, random\_state=42) *#*

In [ ]:

model = KNN(7)

model.fit(X\_train, y\_train)

In [ ]:

def compute\_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, expressed as a percentage.

"""

y\_true = y\_true.flatten()

total\_samples = len(y\_true)

correct\_predictions = np.sum(y\_true == y\_pred)

return (correct\_predictions / total\_samples)

In [ ]:

X\_test

In [ ]:

predictions = model.predict(X\_test)

accuracy = compute\_accuracy(y\_test, predictions)

print(f" our model got accuracy score of : {accuracy}")

our model got accuracy score of : 0.9666666666666667

In [ ]:

model.predict([[7.7, 2.6, 6.9, 2.3]])

In [ ]:

from sklearn.neighbors import KNeighborsClassifier

skmodel = KNeighborsClassifier(n\_neighbors=7)

skmodel.fit(X\_train, y\_train)

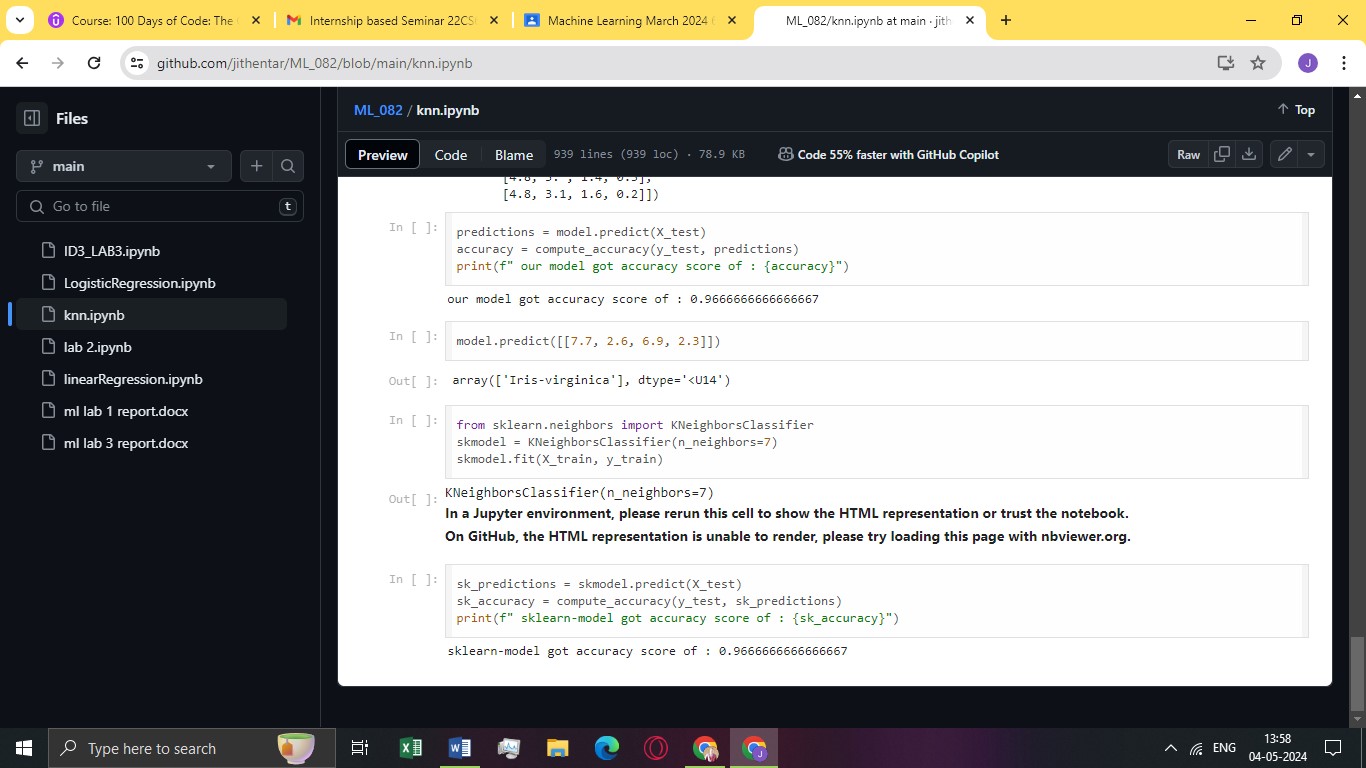
In [ ]:

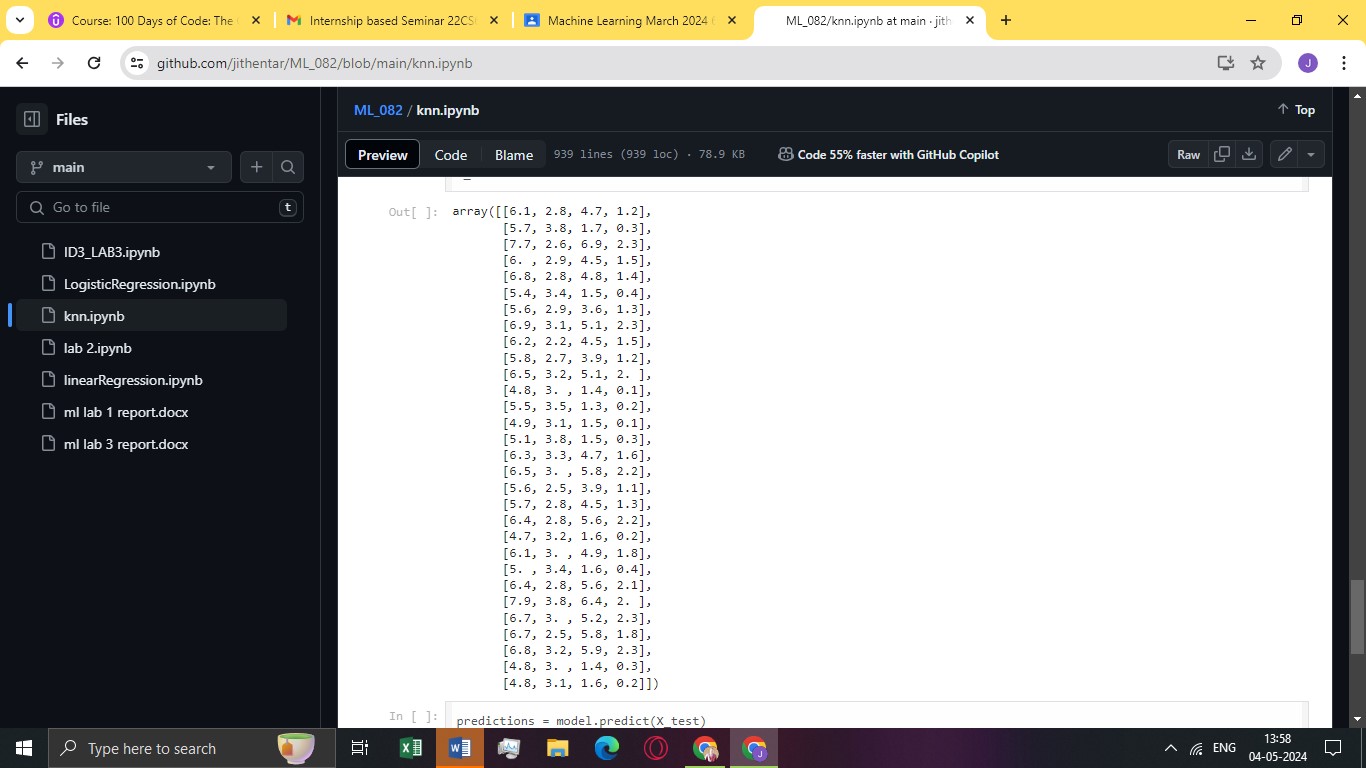
sk\_predictions = skmodel.predict(X\_test)

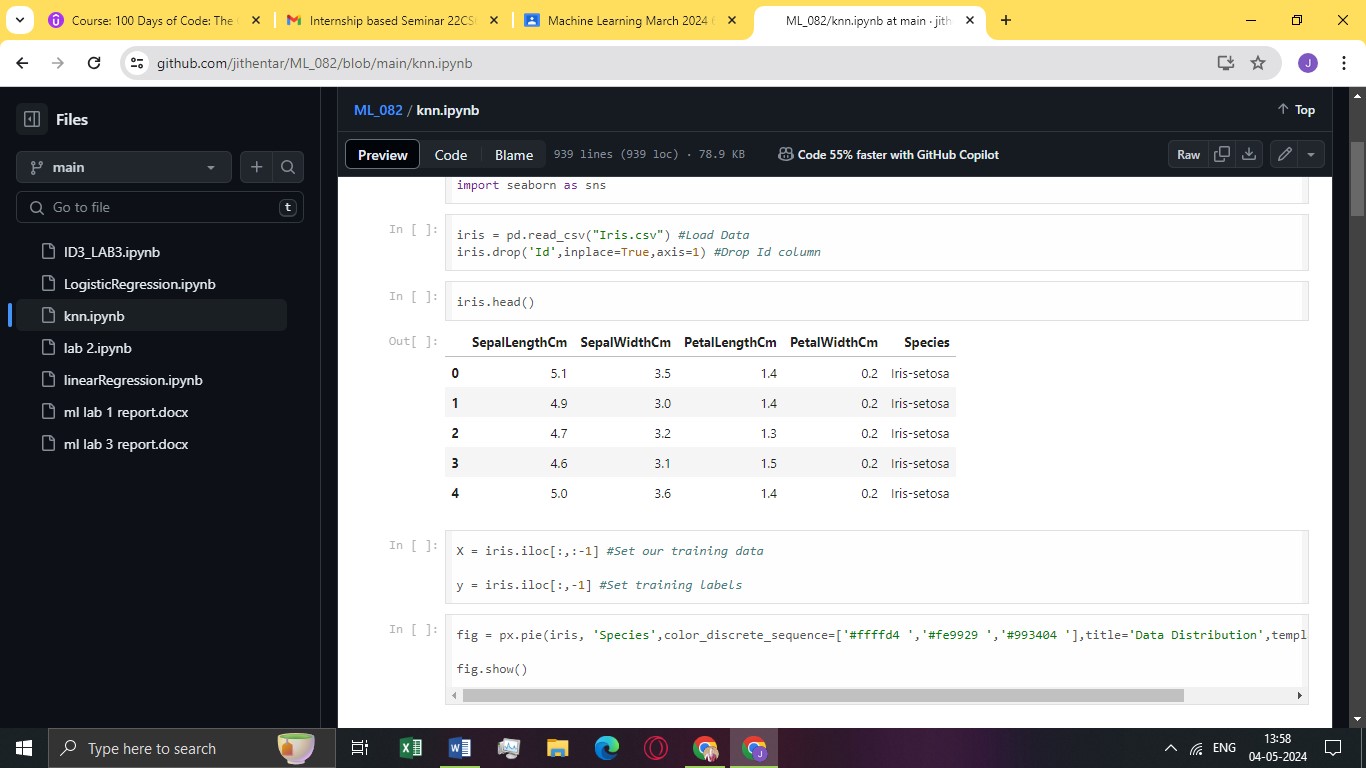
sk\_accuracy = compute\_accuracy(y\_test, sk\_predictions)

print(f" sklearn-model got accuracy score of : {sk\_accuracy}")

**outputs:**





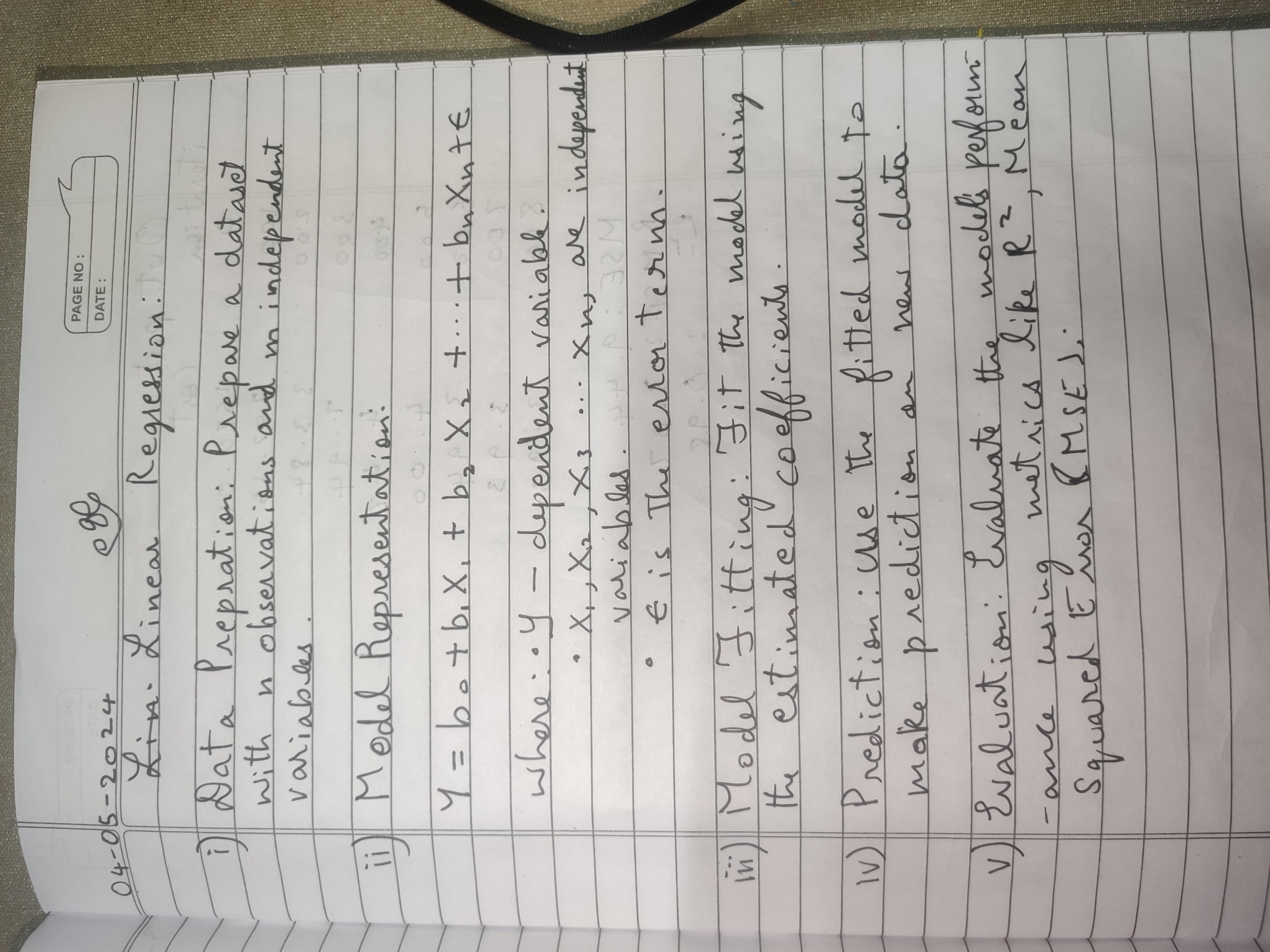


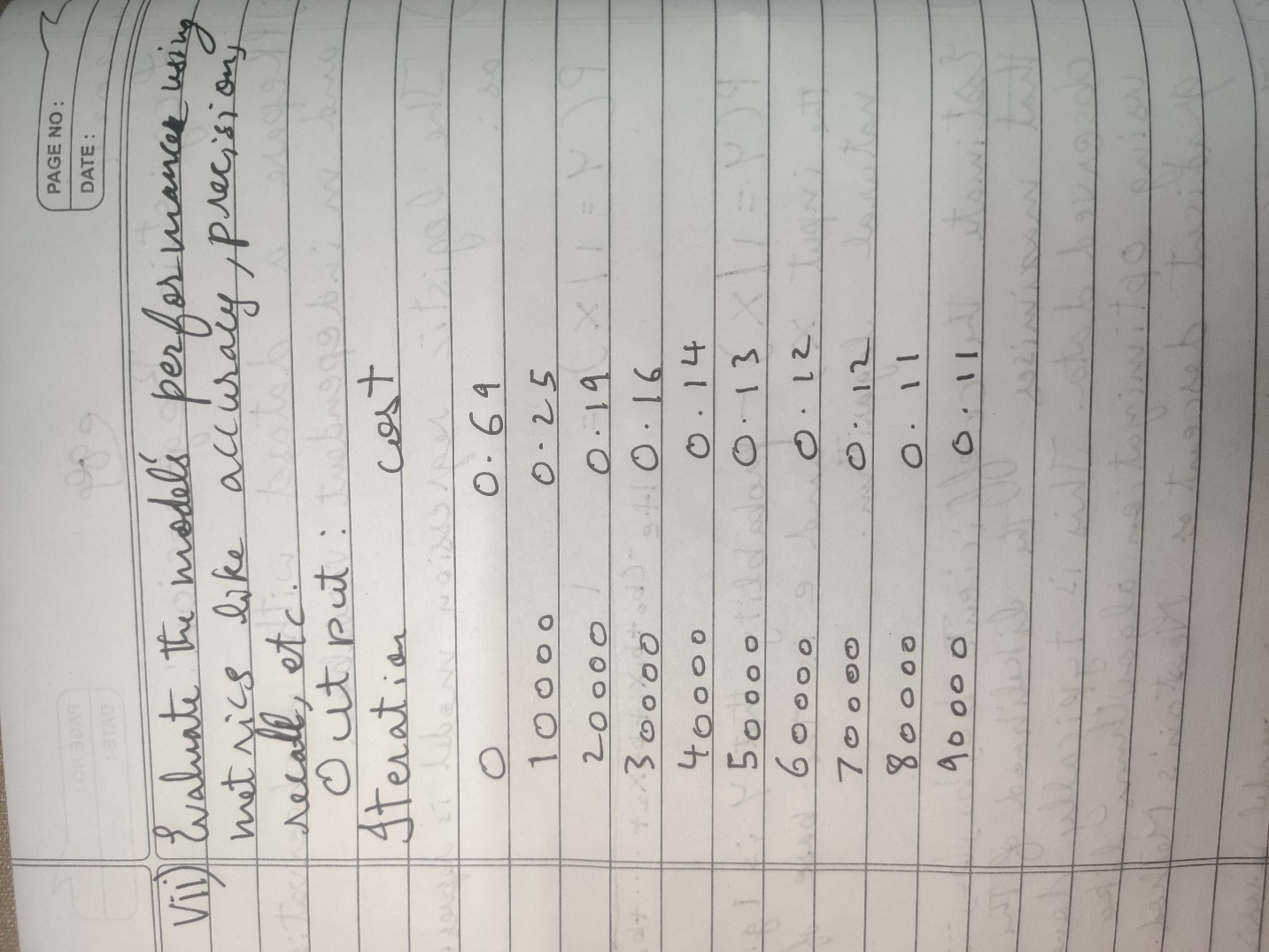
**LAB-3**

**Implement Linear and Multi-Linear Regression algorithm using appropriate dataset.**

**LINEAR REGRESSION:**

**OBSERVATION:**





**CODE:**

import math

import numpy as np

import pandas as pd

import plotly.express as px

import pickle

In [ ]:

*# Load the training and test datasets*

train\_data = pd.read\_csv('train.csv')

test\_data = pd.read\_csv('test.csv')

*# Remove rows with missing values*

train\_data = train\_data.dropna()

test\_data = test\_data.dropna()

In [ ]:

train\_data.head()

In [ ]:

px.scatter(x=train\_data['x'], y=train\_data['y'],template='seaborn')

In [ ]:

*# Set training data and target*

X\_train = train\_data['x'].values

y\_train = train\_data['y'].values

*# Set testing data and target*

X\_test = test\_data['x'].values

y\_test = test\_data['y'].values

In [ ]:

def standardize\_data(X\_train, X\_test):

"""

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)

In [ ]:

X\_train = np.expand\_dims(X\_train, axis=-1)

X\_test = np.expand\_dims(X\_test, axis=-1)

In [ ]:

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:

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).

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.

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()

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.

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

In [ ]:

lr = LinearRegression(0.01)

lr.fit(X\_train, y\_train, 10000)

In [ ]:

lr.save\_model('model.pkl')

In [ ]:

model = LinearRegression.load\_model("model.pkl")

In [ ]:

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:

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

In [ ]:

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}")

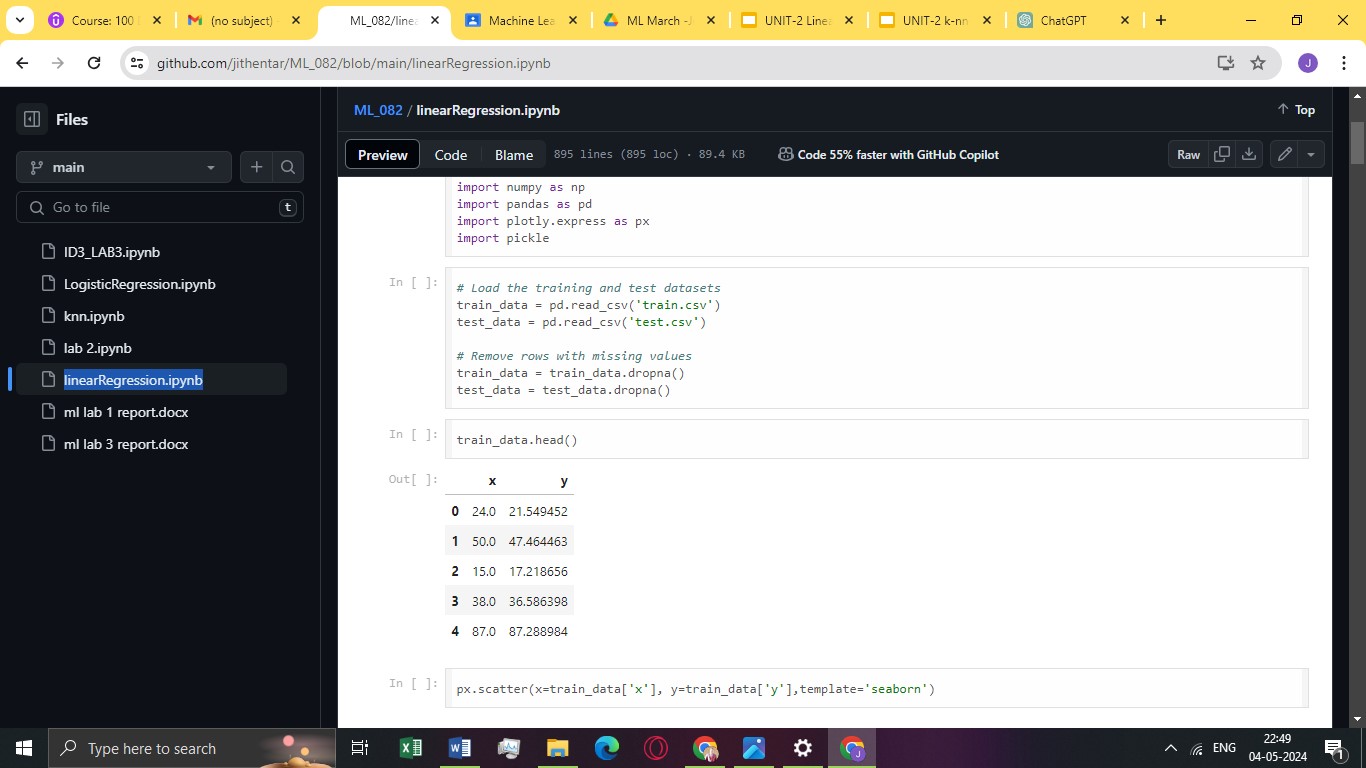
print(f"Root Mean Squared Error (RMSE): {rmse\_value}")

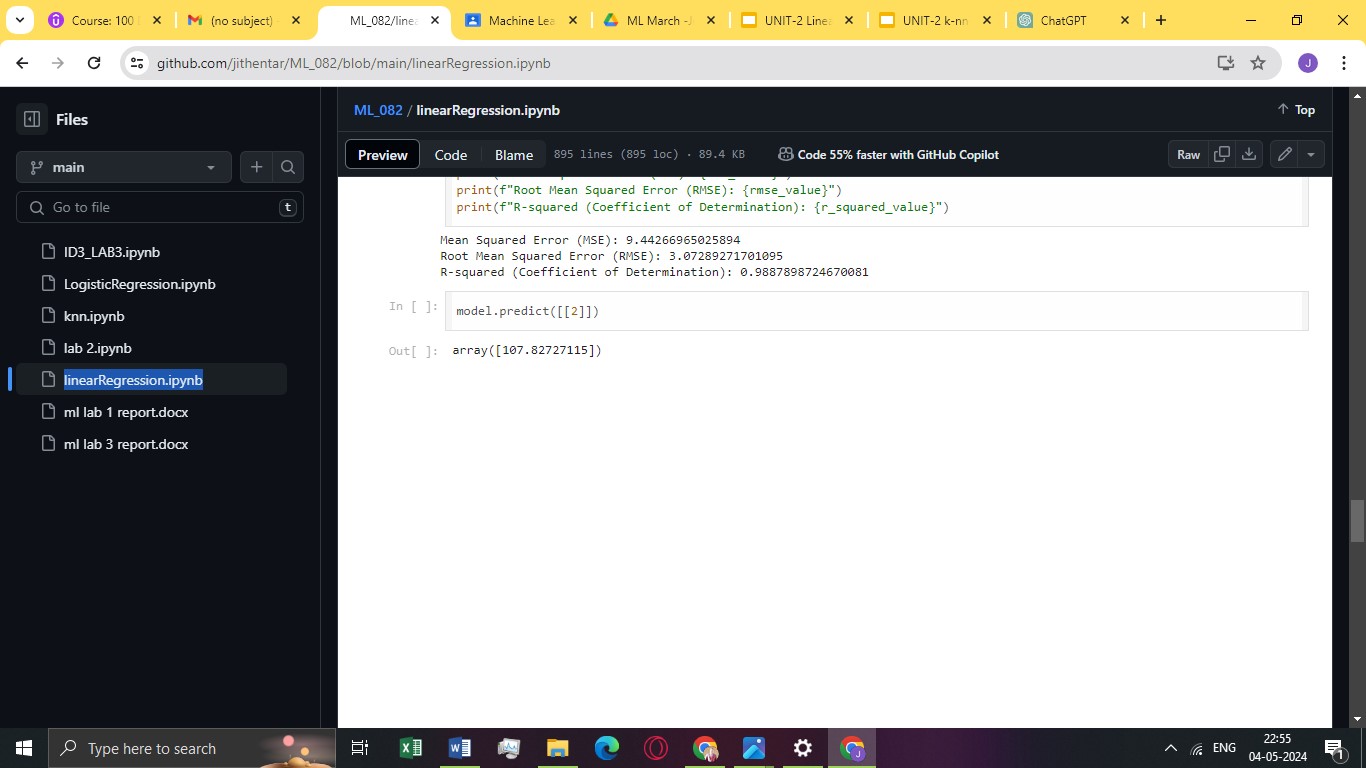
print(f"R-squared (Coefficient of Determination): {r\_squared\_value}")

In [ ]:

model.predict([[2]])

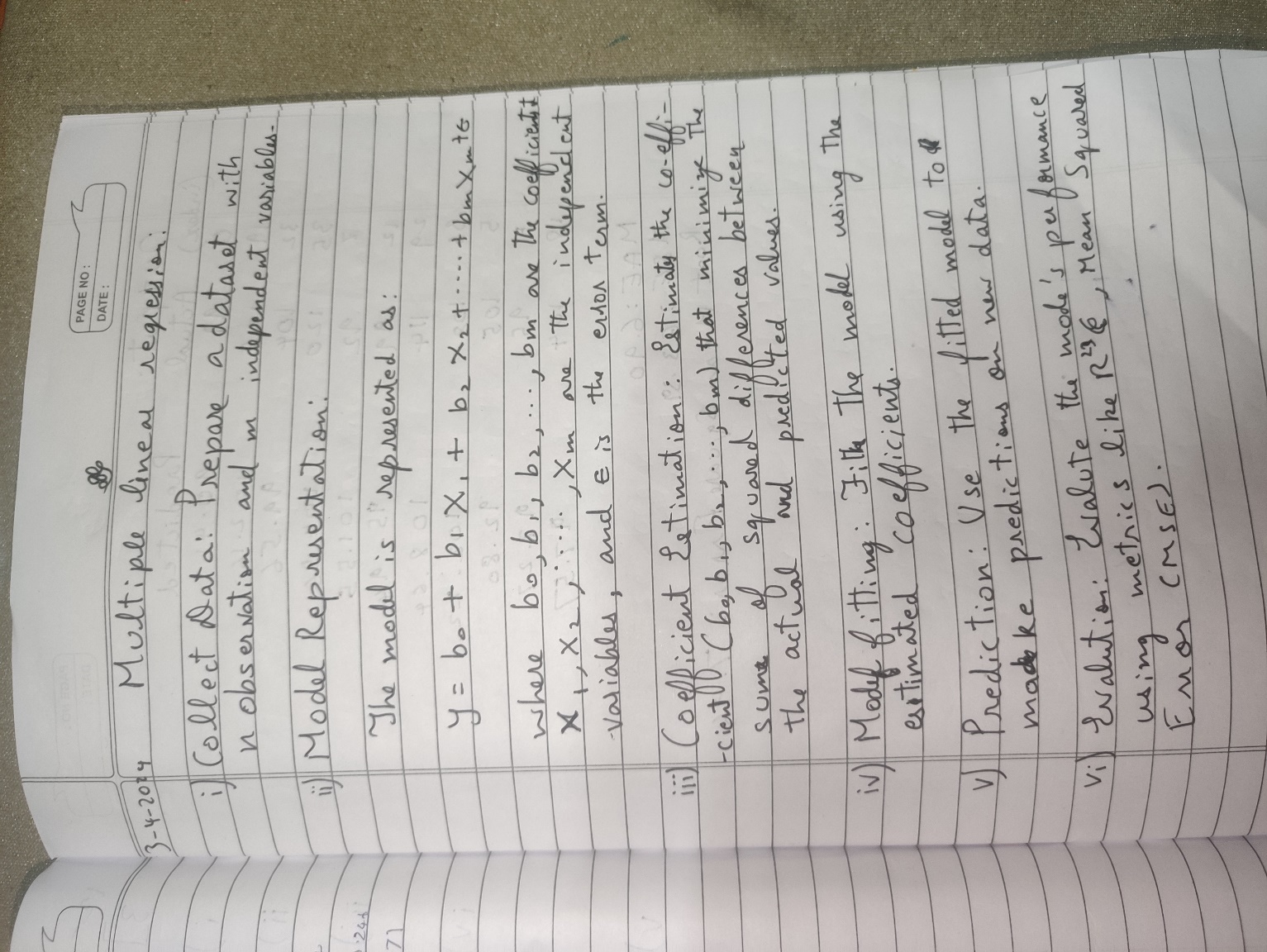
**OUTPUT:**

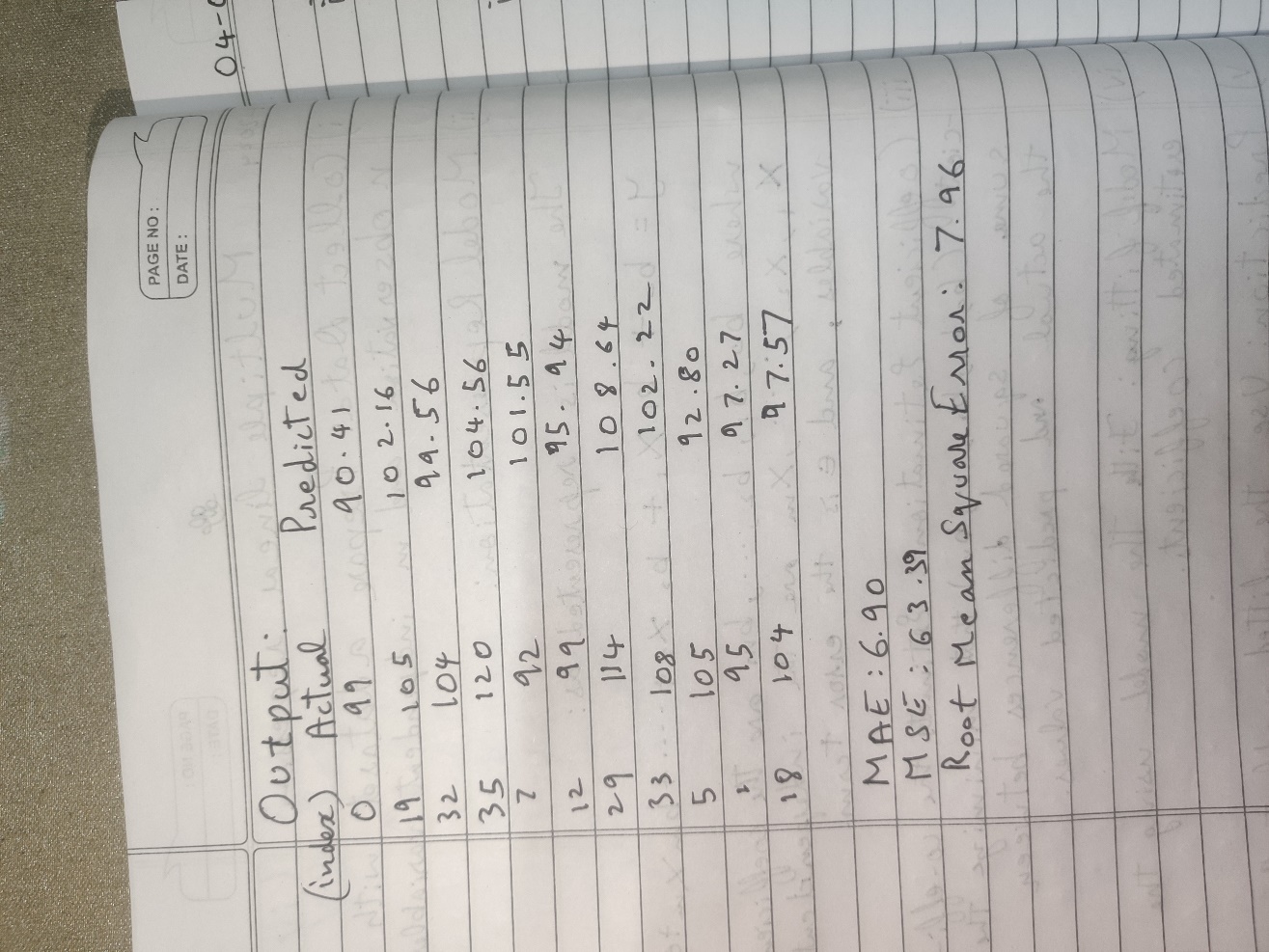




**MULTIPLE LINEAR REGRESSION:**

**OBSERVATION:**





**CODE:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

*# import warnings*

import warnings

warnings.filterwarnings("ignore")

*# We will use some methods from the sklearn module*

from sklearn import linear\_model

from sklearn.linear\_model import LinearRegression

from sklearn import metrics

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

In [3]:

*# Reading the Dataset*

df = pd.read\_csv("/kaggle/input/cardataset/data.csv")

In [4]:

df.head()

df.shape

s

In [6]:

print(df.corr())

X = df[['Weight', 'Volume']]

y = df['CO2']

In [9]:

fig, axs = plt.subplots(2, figsize = (5,5))

plt1 = sns.boxplot(df['Weight'], ax = axs[0])

plt2 = sns.boxplot(df['Volume'], ax = axs[1])

plt.tight\_layout()

sns.distplot(df['CO2']);

In [11]:

sns.pairplot(df, x\_vars=['Weight', 'Volume'], y\_vars='CO2', height=4, aspect=1, kind='scatter')

plt.show()

In [12]:

*# Create the correlation matrix and represent it as a heatmap.*

sns.heatmap(df.corr(), annot = True, cmap = 'coolwarm')

plt.show()

In [13]:

X\_train,X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 100)

In [14]:

y\_train.shape

y\_test.shape

Out[15]:

(11,)

In [16]:

reg\_model = linear\_model.LinearRegression()

In [17]:

*#Fitting the Multiple Linear Regression model*

reg\_model = LinearRegression().fit(X\_train, y\_train)

In [18]:

*#Printing the model coefficients*

print('Intercept: ',reg\_model.intercept\_)

*# pair the feature names with the coefficients*

list(zip(X, reg\_model.coef\_))

Intercept: 74.33882836589245

*#Predicting the Test and Train set result*

y\_pred= reg\_model.predict(X\_test)

x\_pred= reg\_model.predict(X\_train)

In [20]:

print("Prediction for test set: **{}**".format(y\_pred))

Prediction for test set: [ 90.41571939 102.16323413 99.56363213 104.56661845 101.54657652

95.94770019 108.64011848 102.22654214 92.80374837 97.27327129

97.57074463]

In [21]:

*#Actual value and the predicted value*

reg\_model\_diff = pd.DataFrame({'Actual value': y\_test, 'Predicted value': y\_pred})

In [22]:

mae = metrics.mean\_absolute\_error(y\_test, y\_pred)

mse = metrics.mean\_squared\_error(y\_test, y\_pred)

r2 = np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred))

print('Mean Absolute Error:', mae)

print('Mean Square Error:', mse)

print('Root Mean Square Error:', r2)

**OUTPUT:**

