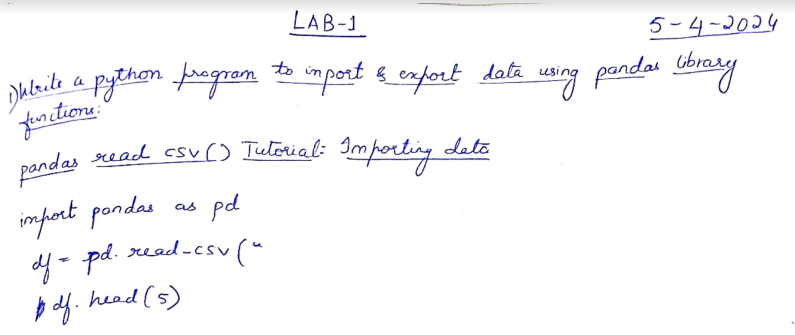
**LAB 1**

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

Importing data

Algorithm(Observation book)



Code

import pandas as pd

df=pd.read\_csv("/content/austinHousingData.csv")

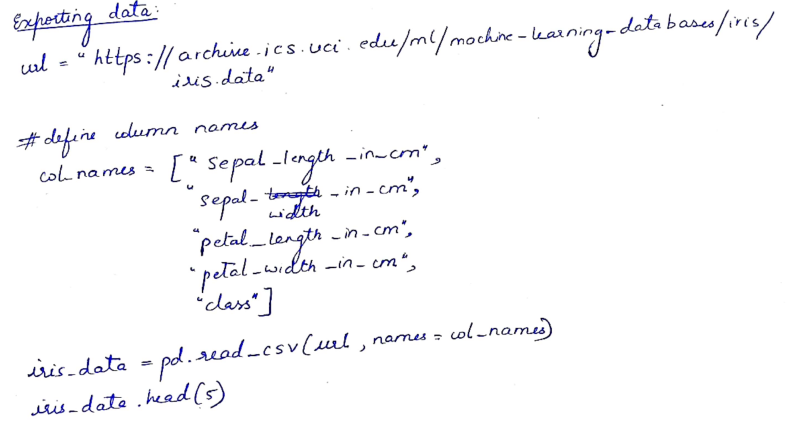
df.head(5)

Output



Exporting data

Algorithm(Observation book)



Code

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

# Define the column names

col\_names = [“sepal\_length\_in\_cm”,

            “sepal\_width\_in\_cm”,

            “petal\_length\_in\_cm”,

            “petal\_width\_in\_cm”,

            “class”]

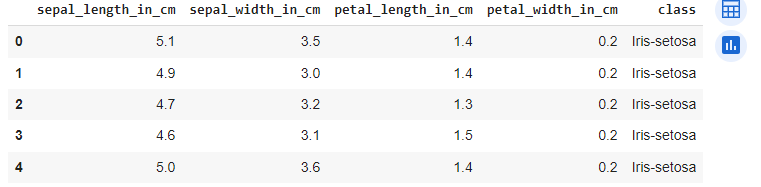
# Read data from URL

iris\_data = pd.read\_csv(url, names=col\_names)

iris\_data.head(5)

iris\_data.to\_csv(“/content/exported\_irisData.csv”)

Output:



Demonstrate various data pre-processing techniques for a given dataset.

%matplotlib inline

import numpy as np

import pandas as pd

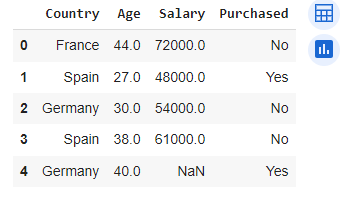
import matplotlib.pyplot as plt

import seaborn as sns

import sklearn

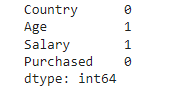
df1=pd.read\_csv(“/content/Data.csv”)

df1.head(5)



#Identifying and handling the missing values

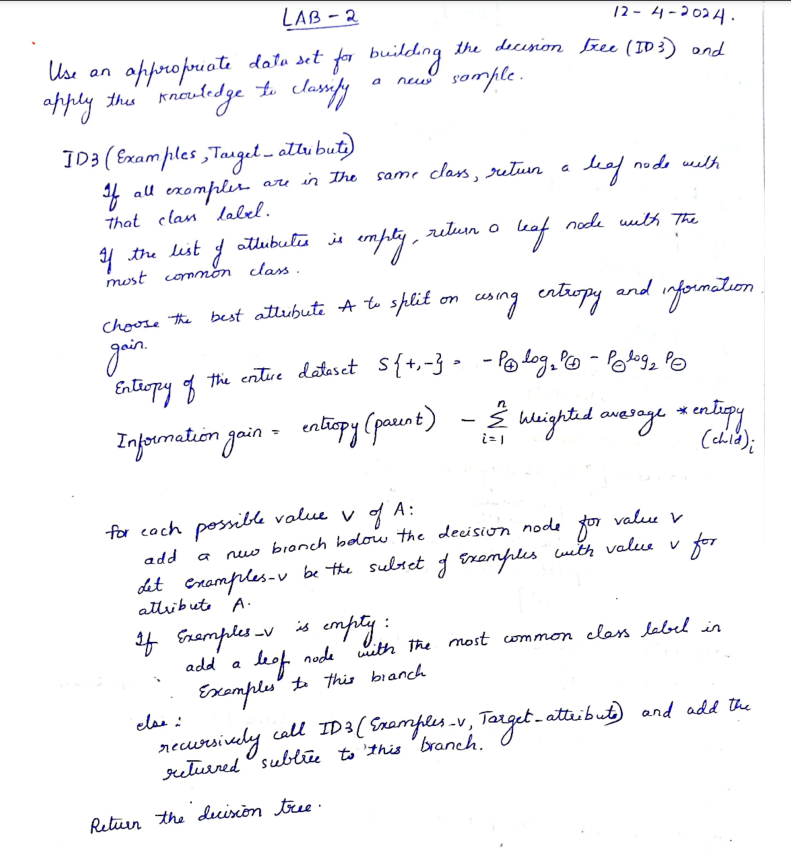
df1.isnull().sum()



**LAB 2**

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

Algorithm(Observation book)



Code

# Importing the required libraries

import pandas as pd

import numpy as np

import math

data = pd.read\_csv('/content/PlayTennis.csv')

def highlight(cell\_value):

    '''

    Highlight yes / no values in the dataframe

    '''

    color\_1 = 'background-color: pink;'

    color\_2 = 'background-color: lightgreen;'

    if cell\_value == 'no':

        return color\_1

    elif cell\_value == 'yes':

        return color\_2

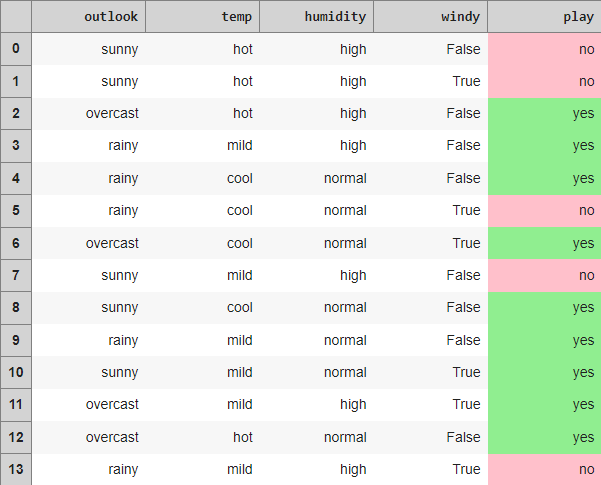
data.style.applymap(highlight)\

    .set\_properties(subset=data.columns, \*\*{'width': '100px'})\

    .set\_table\_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'), ('border', '1px solid gray'),

                                                    ('font-weight', 'bold')]},

     {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid black')]}])



def find\_entropy(data):

    """

    Returns the entropy of the class or features

    formula: - ∑ P(X)logP(X)

    """

    entropy = 0

    for i in range(data.nunique()):

        x = data.value\_counts()[i]/data.shape[0]

        entropy += (- x \* math.log(x,2))

    return round(entropy,3)

def information\_gain(data, data\_):

    """

    Returns the information gain of the features

    """

    info = 0

    for i in range(data\_.nunique()):

        df = data[data\_ == data\_.unique()[i]]

        w\_avg = df.shape[0]/data.shape[0]

        entropy = find\_entropy(df.play)

        x = w\_avg \* entropy

        info += x

    ig = find\_entropy(data.play) - info

    return round(ig, 3)

def entropy\_and\_infogain(datax, feature):

    """

    Grouping features with the same class and computing their

    entropy and information gain for splitting

    """

    for i in range(data[feature].nunique()):

        df = datax[datax[feature]==data[feature].unique()[i]]

        if df.shape[0] < 1:

            continue

        display(df[[feature, 'play']].style.applymap(highlight)\

                .set\_properties(subset=[feature, 'play'], \*\*{'width': '80px'})\

                .set\_table\_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'),

                                                                ('border', '1px solid gray'),

                                                                ('font-weight', 'bold')]},

                                   {'selector': 'td', 'props': [('border', '1px solid gray')]},

                                   {'selector': 'tr:hover', 'props': [('background-color', 'white'),

                                                                      ('border', '1.5px solid black')]}]))

        print(f'Entropy of {feature} - {data[feature].unique()[i]} = {find\_entropy(df.play)}')

    print(f'Information Gain for {feature} = {information\_gain(datax, datax[feature])}')

#Computing entropy for the entire dataset

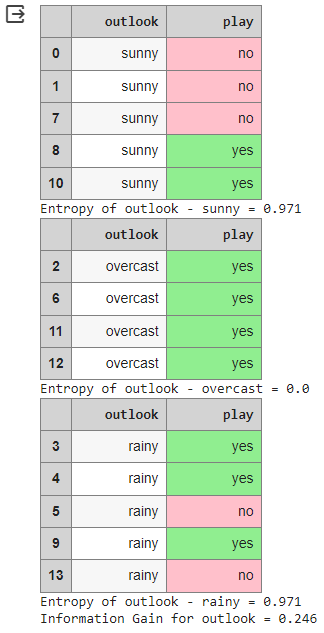
print(f'Entropy of the entire dataset: {find\_entropy(data.play)}')



#Calculate the Information Gain for each feature.

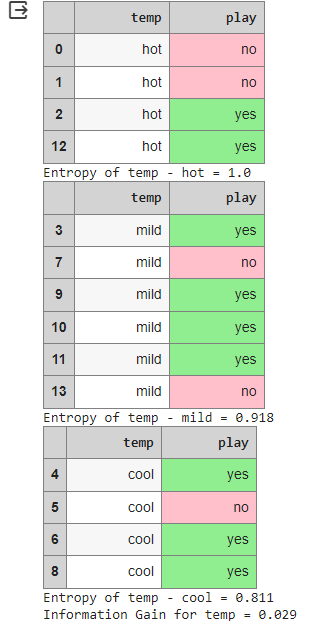
#Outlook

entropy\_and\_infogain(data, 'outlook')



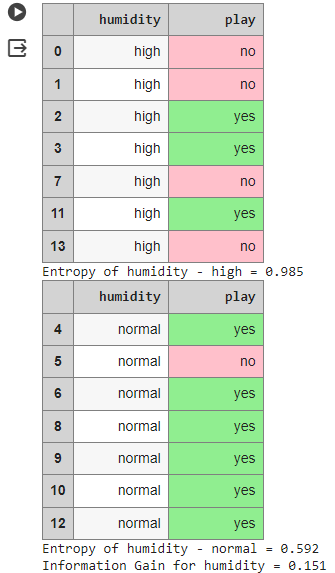
#Temp

entropy\_and\_infogain(data, 'temp')



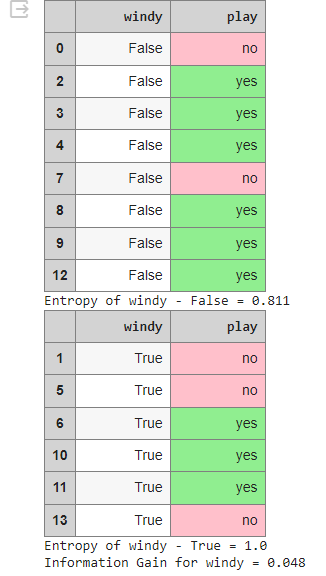
#Humidity

entropy\_and\_infogain(data, 'humidity')

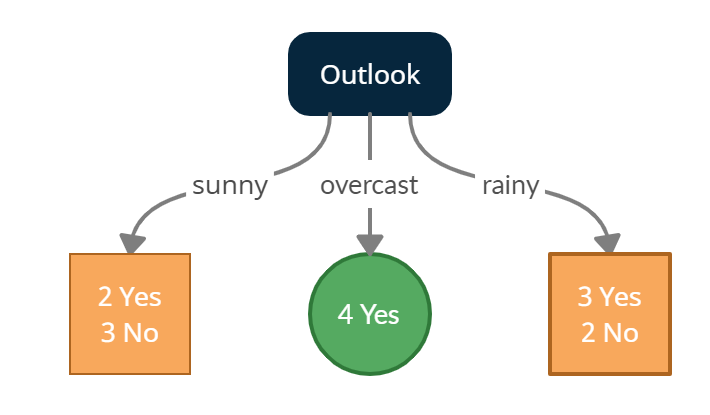


#Windy

entropy\_and\_infogain(data, 'windy')



#Make a decision tree node using the feature with the maximum Information Gain.



sunny = data[data['outlook'] == 'sunny']

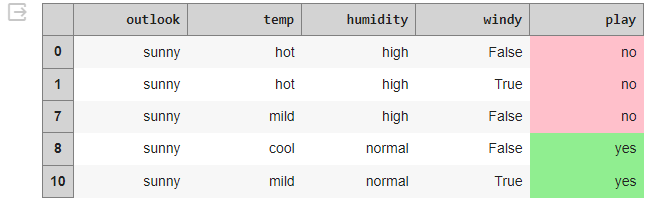
sunny.style.applymap(highlight)\

    .set\_properties(subset=data.columns, \*\*{'width': '100px'})\

    .set\_table\_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'), ('border', '1px solid gray'),

                                                    ('font-weight', 'bold')]},

     {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid black')]}])



print(f'Entropy of the Sunny dataset: {find\_entropy(sunny.play)}')



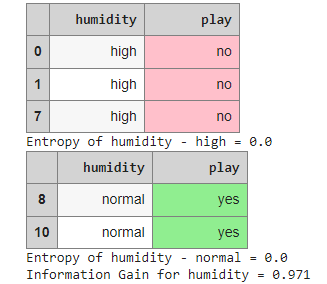
#temp

entropy\_and\_infogain(sunny, 'temp')



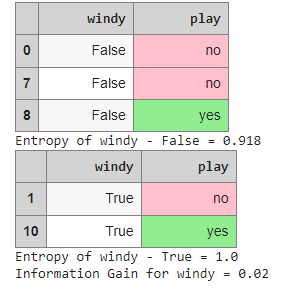
#Humidity

entropy\_and\_infogain(sunny, 'humidity')

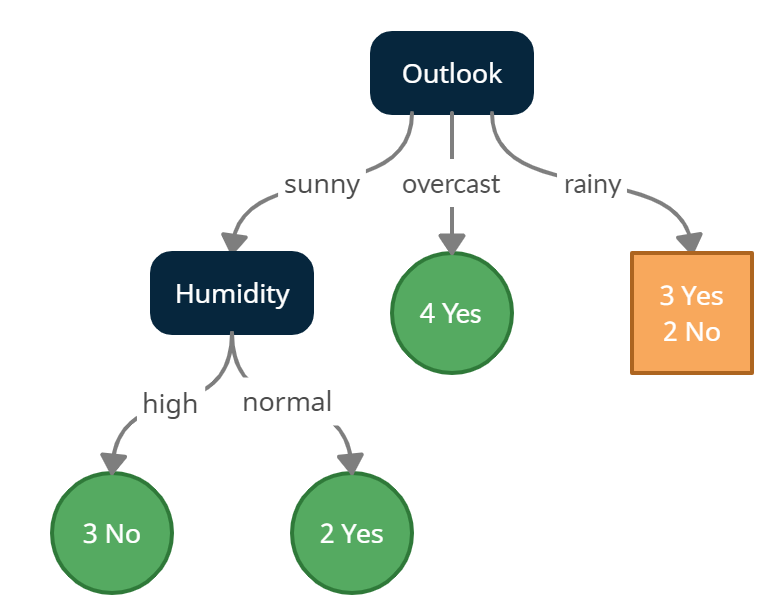


#Windy

entropy\_and\_infogain(sunny, 'windy')



**#Making a decision tree node using the feature which has the maximum Information Gain**

****

#Outlook - Rainy

rainy = data[data['outlook'] == 'rainy']

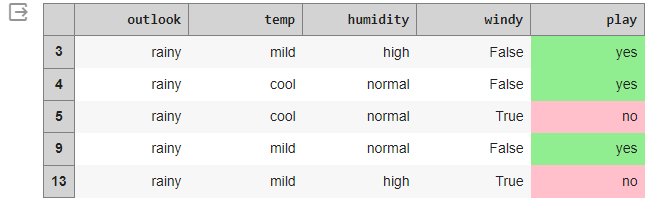
rainy.style.applymap(highlight)\

    .set\_properties(subset=data.columns, \*\*{'width': '100px'})\

    .set\_table\_styles([{'selector': 'th', 'props': [('background-color', 'lightgray'), ('border', '1px solid gray'),

                                                    ('font-weight', 'bold')]},

     {'selector': 'tr:hover', 'props': [('background-color', 'white'), ('border', '1.5px solid black')]}])



print(f'Entropy of the Rainy dataset: {find\_entropy(rainy.play)}')



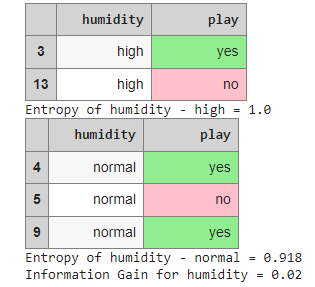
#temp

entropy\_and\_infogain(rainy, 'temp')



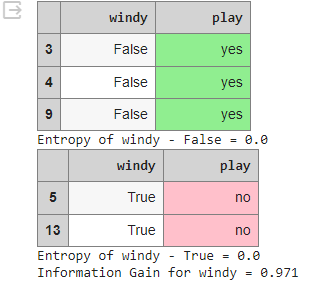
#Humidity

entropy\_and\_infogain(rainy, 'humidity')

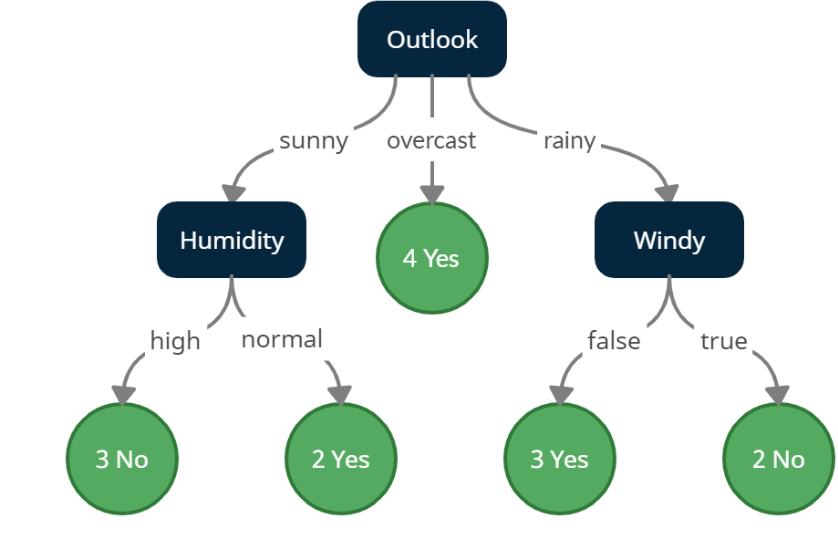


#Windy

entropy\_and\_infogain(rainy, 'windy')



**#Making a decision tree node using the feature which has the maximum Information Gain.**

****

**LAB 3**

Build KNN Classification model for a given dataset.

Algorithm(Observation book)

Code

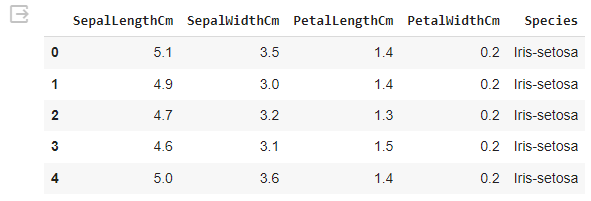
import numpy as np

import pandas as pd

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

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

iris.head(5)



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

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

class KNN:

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

        self.n\_neighbors = n\_neighbors

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

        return np.linalg.norm(x1 - x2)

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

        self.X\_train = X\_train

        self.y\_train = y\_train

    def predict(self, 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):

        # 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

def train\_test\_split(X, y, random\_state=42, test\_size=0.2):

    # 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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X.values, y.values, test\_size = 0.2, random\_state=42) #split the  data into traing and validating

model = KNN(7)

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

def compute\_accuracy(y\_true, y\_pred):

    y\_true = y\_true.flatten()

    total\_samples = len(y\_true)

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

    return (correct\_predictions / total\_samples)

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

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

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

Linear regression

Algorithm(Observation book)

Code

import math

import numpy as np

import pandas as pd

import plotly.express as px

import pickle

# Load the training and test datasets

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

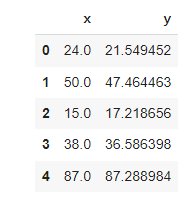
test\_data = pd.read\_csv('/content/test.csv')

# Remove rows with missing values

train\_data = train\_data.dropna()

test\_data = test\_data.dropna()

train\_data.head()



# 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

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)

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

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

class LinearRegression:

  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):

    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):

      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'])

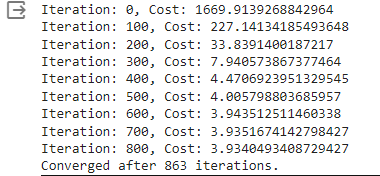
        loaded\_model.W = model\_data['W']

        loaded\_model.b = model\_data['b']

        return loaded\_model

lr = LinearRegression(0.01)

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



**LAB 4**

Implement logistic regression using appropriate dataset.