* Name: Vo Hong Quan
* ID: 22134012
* Class: 22134

Exercise 3: Linear regression multiple orders

* Biểu diễn dữ liệu:

import numpy as np

import pandas as pd

import plotly.graph\_objects as go

import matplotlib.pyplot as plt

from IPython.display import clear\_output

import time

# Read csv file ex2.csv

pd\_ex3 = pd.read\_csv('ex3.csv')

# Get collumns of file

X\_cols = pd\_ex3.columns[:-1]

Y\_col = pd\_ex3.columns[-1]

# Get vector input and output

x\_value = pd\_ex3[X\_cols].values

y\_value = pd\_ex3[Y\_col].values

fig = go.Figure()

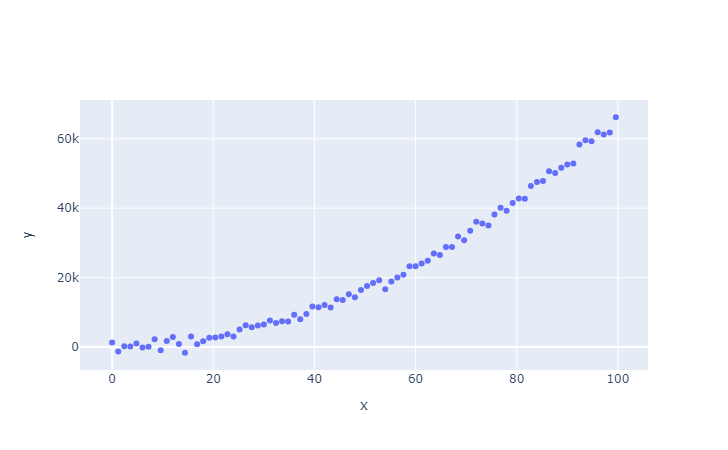
fig.add\_trace(go.Scatter(x=x\_value.reshape(-1,), y=y\_value,

                        mode= 'markers', name= f'Data'))

fig.update\_xaxes(title= 'x')

fig.update\_yaxes(title= 'y', tickangle= 0)

fig.show()



* Chọn mô hình "h" phù hợp (bậc bao nhiêu?): Bậc 2 là phù hợp

order= 2

model\_poly = Linear\_Regression\_Multivariables(number\_of\_feature= order)

* Chuyển bài toán đa bậc thành bài toán đa biến và chuẩn hóa dữ liệu: scale dữ liệu, format kích thước dữ liệu:

X\_poly = create\_polynomial\_features(X= x\_value, order= order)

Y\_poly = y\_value.reshape(-1, 1)

def create\_polynomial\_features(\*, X: np.ndarray, order: int) -> np.ndarray:

    X = X.reshape(-1,)

    poly\_matrix = np.ones((len(X), order))  # Initialize the matrix with ones

    for i in range(order):

        poly\_matrix[:, i] = X \*\* (i + 1)  # Raise each column to the respective power

    return poly\_matrix

def normalize\_vector(vector: np.ndarray) -> np.ndarray:

    mean = np.mean(vector)  # Calculate the mean

    std = np.std(vector)  # Calculate the standard deviation

    normalized\_vector = (vector - mean) / std  # Normalize the vector

    return normalized\_vector

* Viết chương trình cho phép học các tham số của mô hình hồi quy tuyến tính đa biến

class Linear\_Regression\_Multivariables:

def \_\_init\_\_(self, \*, number\_of\_feature: int) -> None:

self.number\_of\_features = number\_of\_feature

def normalize\_vector(self, vector: np.ndarray) -> np.ndarray:

mean = np.mean(vector)

std = np.std(vector)

return (vector - mean) / std

def normalize\_input\_output(self, \*, X: np.ndarray, Y: np.ndarray) -> tuple:

norm\_X = np.apply\_along\_axis(self.normalize\_vector, arr=X, axis=0).reshape(-1, self.number\_of\_features)

norm\_Y = np.apply\_along\_axis(self.normalize\_vector, arr=Y, axis=0).reshape(-1, 1)

return norm\_X, norm\_Y

return x\_add

def predict(self, \*, theta: np.ndarray, normalized\_input: np.ndarray) -> np.ndarray:

y\_pred = np.matmul(normalized\_input, theta)

return y\_pred

def compute\_loss(self, \*, y\_true: np.ndarray, y\_pred: np.ndarray) -> np.ndarray:

m = len(y\_true)

E = y\_pred - y\_true

J = np.sum((E)\*\*2) / (2\*m)

return J

def update\_params(self, \*, theta: np.ndarray, lr: float, y\_pred: np.ndarray,

y\_true: np.ndarray, normalized\_input: np.ndarray) -> np.ndarray:

m = len(y\_true)

E = y\_pred - y\_true

dJ\_dtheta = np.dot(normalized\_input.T, E) / (m)

theta\_updated = theta - lr\*dJ\_dtheta

return theta\_updated

def denormalize\_theta(self, \*, theta\_normalized: np.ndarray, input: np.ndarray, output: np.ndarray) -> np.ndarray:

theta = np.zeros\_like(theta\_normalized)

mean\_x = np.mean(input, axis= 0)

std\_x = np.std(input, axis= 0)

mean\_y = np.mean(output, axis= 0)

std\_y = np.std(output, axis= 0)

theta[1:] = std\_y\*theta\_normalized[1:]/(std\_x.reshape(-1, 1))

theta[0] = mean\_y + std\_y\*theta\_normalized[0] - np.dot(std\_y\*mean\_x/std\_x, theta\_normalized[1:])

return theta

def train(self, \*, epoch: int, theta: np.ndarray, input: np.ndarray,

output: np.ndarray, lr: float) -> np.ndarray:

"""

Trains the Linear Regression model using gradient descent.

Args:

epochs (int): Number of training iterations.

theta (np.ndarray): Initial model parameters.

X (np.ndarray): Input features.

Y (np.ndarray): Output values.

lr (float): Learning rate for parameter updates.

Returns:

tuple: Array of loss values and the trained model parameters.

"""

normalized\_input, normalized\_ouput = self.normalize\_input\_output(X= input, Y= output)

normalized\_input\_with\_ones = self.add\_ones\_columns(normalized\_input= normalized\_input)

theta\_array = theta

J\_array = np.array([])

for i in range(epoch):

y\_pred = self.predict(theta= theta,

normalized\_input= normalized\_input\_with\_ones)

J = self.compute\_loss(y\_true= normalized\_ouput, y\_pred= y\_pred)

theta = self.update\_params(theta= theta, lr= lr, y\_pred= y\_pred,

y\_true= normalized\_ouput, normalized\_input= normalized\_input\_with\_ones)

J\_array = np.append(arr= J\_array, values= J)

theta\_array = np.hstack((theta\_array, theta))

theta = self.denormalize\_theta(theta\_normalized= theta, input= input, output= output)

return J\_array, theta, theta\_array

def predict(self, \*, theta: np.ndarray, normalized\_input: np.ndarray) -> np.ndarray:

y\_pred = np.matmul(normalized\_input, theta)

return y\_pred

def compute\_loss(self, \*, y\_true: np.ndarray, y\_pred: np.ndarray) -> np.ndarray:

m = len(y\_true)

E = y\_pred - y\_true

J = np.sum((E)\*\*2) / (2\*m)

return J

def update\_params(self, \*, theta: np.ndarray, lr: float, y\_pred: np.ndarray,

y\_true: np.ndarray, normalized\_input: np.ndarray) -> np.ndarray:

m = len(y\_true)

E = y\_pred - y\_true

dJ\_dtheta = np.dot(normalized\_input.T, E) / (m)

theta\_updated = theta - lr\*dJ\_dtheta

return theta\_updated

def denormalize\_theta(self, \*, theta\_normalized: np.ndarray, input: np.ndarray, output: np.ndarray) -> np.ndarray:

theta = np.zeros\_like(theta\_normalized)

mean\_x = np.mean(input, axis= 0)

std\_x = np.std(input, axis= 0)

mean\_y = np.mean(output, axis= 0)

std\_y = np.std(output, axis= 0)

theta[1:] = std\_y\*theta\_normalized[1:]/(std\_x.reshape(-1, 1))

theta[0] = mean\_y + std\_y\*theta\_normalized[0] - np.dot(std\_y\*mean\_x/std\_x, theta\_normalized[1:])

return theta

def train(self, \*, epoch: int, theta: np.ndarray, input: np.ndarray,

output: np.ndarray, lr: float) -> np.ndarray:

"""

Trains the Linear Regression model using gradient descent.

Args:

epochs (int): Number of training iterations.

theta (np.ndarray): Initial model parameters.

X (np.ndarray): Input features.

Y (np.ndarray): Output values.

lr (float): Learning rate for parameter updates.

Returns:

tuple: Array of loss values and the trained model parameters.

"""

normalized\_input, normalized\_ouput = self.normalize\_input\_output(X= input, Y= output)

normalized\_input\_with\_ones = self.add\_ones\_columns(normalized\_input= normalized\_input)

theta\_array = theta

J\_array = np.array([])

for i in range(epoch):

y\_pred = self.predict(theta= theta,

normalized\_input= normalized\_input\_with\_ones)

J = self.compute\_loss(y\_true= normalized\_ouput, y\_pred= y\_pred)

theta = self.update\_params(theta= theta, lr= lr, y\_pred= y\_pred,

y\_true= normalized\_ouput, normalized\_input= normalized\_input\_with\_ones)

J\_array = np.append(arr= J\_array, values= J)

theta\_array = np.hstack((theta\_array, theta))

theta = self.denormalize\_theta(theta\_normalized= theta, input= input, output= output)

return J\_array, theta, theta\_array

    def train(self, \*, epoch: int, theta: np.ndarray, input: np.ndarray,

              output: np.ndarray, lr: float, plot\_graph: False) -> np.ndarray:

        normalized\_input, normalized\_ouput = self.normalize\_input\_output(X= input, Y= output)

        normalized\_input\_with\_ones = self.add\_ones\_columns(normalized\_input= normalized\_input)

        J\_array = np.array([])

        for i in range(epoch):

            y\_pred = self.predict(theta= theta, normalized\_input= normalized\_input\_with\_ones)

            J = self.compute\_loss(y\_true= normalized\_ouput, y\_pred= y\_pred)

            theta = self.update\_params(theta= theta, lr= lr, y\_pred= y\_pred,

                                       y\_true= normalized\_ouput, normalized\_input= normalized\_input\_with\_ones)

            J\_array = np.append(arr= J\_array, values= J)

            if plot\_graph == True:

                x\_with\_ones = self.add\_ones\_columns(normalized\_input= input)

                theta\_praph = self.denormalize\_theta(theta\_normalized= theta, input= input, output= output)

                data\_1 = dict(x= input[:, 0].reshape(-1,), y= output.reshape(-1,), mode= 'markers', title= 'Data')

                data\_2 = dict(x= input[:, 0].reshape(-1,), y= function(X= x\_with\_ones, theta= theta\_praph, add\_ones= False) , mode= 'lines', title= 'Predicted')

                data = [data\_1, data\_2]

                self.plot\_graph(data= data)

        theta = self.denormalize\_theta(theta\_normalized= theta, input= input, output= output)

        return J\_array, theta

* Tính J ở mỗi vòng lặp, và vẽ biểu đồ J ở các giá trị learning rate khác nhau sau khi chạy hết các vòng lặp.

learning\_rate\_ = [0.01, 0.001, 0.003, 0.3, 0.04, 0.1]

J = np.zeros((1000, ))

for i in learning\_rate\_:

    J\_arr, theta\_arr = model\_poly.train(epoch= 1000, theta= theta\_init,

                                           input= X\_poly, output= Y\_poly, lr= i, plot\_graph= True)

    J = np.vstack([J, J\_arr])

fig = go.Figure()

for i in range(len(J[1:, :])):

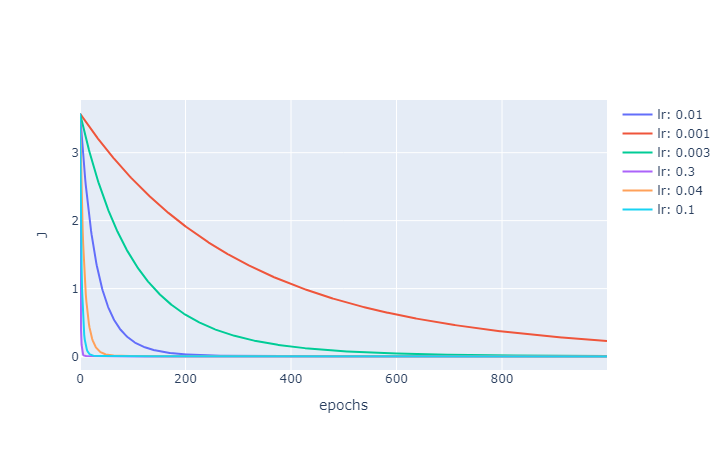
    fig.add\_trace(go.Scatter(x=np.arange(1000), y=J[(1+i), :],

                            mode= 'lines', name= f'lr: {learning\_rate\_[i]}'))

fig.update\_xaxes(title= 'epochs')

fig.update\_yaxes(title= 'J', tickangle= 0)

fig.show()



* Biển diễn đường cong học được và dữ liệu trên cùng 1 hình ảnh:

J\_arr, theta\_arr = model\_poly.train(epoch= 1000, theta= theta\_init,

                                           input= X\_poly, output= Y\_poly, lr= 0.3, plot\_graph= True )

A graph with a red line

Description automatically generated

* Link kaggle to visualize answer: <https://www.kaggle.com/code/honggquan/ex3-homework>