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

Exercise 2: Linear regression multiple variables

* Chuẩn hóa dữ liệu: scale dữ liệu, format kích thước dữ liệu

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

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

* 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

def add\_ones\_columns(self, \*, normalized\_input: np.ndarray) -> np.ndarray:

ones = np.ones(len(normalized\_input)).reshape(-1, 1)

x\_add = np.hstack((ones, normalized\_input))

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 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 plot\_graph(self, data):

clear\_output(wait= True)

fig = go.FigureWidget()

for i in range(len(data)):

fig.add\_trace(go.Scatter(x=data[i]['x'], y=data[i]['y'],

mode= data[i]['mode'], name= data[i]['title']))

fig.show()

time.sleep(1/2)

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

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

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

Viết hàm cập nhật giá trị theta 0 và theta 1 sử dụng phương pháp gradient descent với số lượng vòng lặp và learning rate (alpha) tùy chọn.

def predict(\*, theta: np.array, X: np.array) -> np.ndarray:

    # Compute the hypothesis (predicted values) as the dot product of X and theta

    H = np.matmul(X, theta)

    return H

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

# Read csv file ex2.csv

pd\_ex2 = pd.read\_csv('ex2.csv')

# Get collumns of file

X\_cols = pd\_ex2.columns[:-1]

Y\_col = pd\_ex2.columns[-1]

# Get vector input and output

X = pd\_ex2[X\_cols].values

Y = pd\_ex2[Y\_col].values

np.random.seed(1)

theta\_init = np.random.randn(len(X\_cols) + 1, 1)

theta\_init.shape

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.train(epoch= 1000, theta= theta\_init,

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

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()

A graph of different colored lines

Description automatically generated

* Kiểm chứng các theta mà các bạn tìm được bằng phương pháp gradient descent với phương pháp normal equation.

model = Linear\_Regression\_Multivariables(number\_of\_feature= 8)

J\_array\_ex\_2, theta\_ex\_2 = model.train(

epoch= 10000, theta= theta\_init, input= X, output= Y, lr= 0.3, plot\_graph= False)

theta\_real = compute\_true\_theta(X= X, Y= Y)

X\_with\_ones = model.add\_ones\_columns(normalized\_input= X)

theta\_difference = np.round((theta\_real - theta\_ex\_2) / theta\_real, 2)

theta\_difference

* Link kaggle to visualize answer: <https://www.kaggle.com/code/honggquan/ex2-homework>