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

Exercise 3: Logistic regression

* Biểu diễn dữ liệu

import matplotlib.pyplot as plt

import plotly.graph\_objects as go

import numpy as np

import pandas as pd

ex4\_data = pd.read\_csv('ex4data.txt', header= None).to\_numpy()

x1, x2, y = ex4\_data[:, 0], ex4\_data[:, 1], ex4\_data[:, 2]

colors = np.where(y == 1, 'red', 'blue')

fig = go.Figure()

# Plot data points

fig.add\_trace(go.Scatter(

x=x1, y=x2,

mode='markers',

marker=dict(color=colors, size=10),

name='Data Points'

))

fig.update\_xaxes(title= 'x1')

fig.update\_yaxes(title= 'x2')

A graph with red and blue dots

Description automatically generated

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

class Logistic\_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)

if std == 0:

return vector - mean

return (vector - mean) / std

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

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

return norm\_X

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

* Viết hàm tính cost J ở mỗi vòng lặp:
* Viết chương trình cho phép học các tham số của mô hình phân loại tuyến tính

class Logistic\_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)

if std == 0:

return vector - mean

return (vector - mean) / std

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

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

return norm\_X

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)

y\_pred = 1/(1 + np.exp(-y\_pred))

return y\_pred

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

m = len(y\_true)

epsilon = 1e-15

y\_pred = np.clip(y\_pred, epsilon, 1 - epsilon)

J = np.sum(- y\_true\*np.log(y\_pred) - (1 - y\_true)\*np.log(1 - y\_pred)) / (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 train(self, \*, epochs: int, theta: np.ndarray, input: np.ndarray,

output: np.ndarray, lr: float, plot\_graph: False, color: list, time\_delay: float) -> np.ndarray:

output = output.reshape(-1, 1)

normalized\_input = self.normalize\_input(X= input)

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

J\_array = np.array([])

for epoch in range(epochs):

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

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

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

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

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

return J\_array, theta

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

m = len(y\_true)

epsilon = 1e-15

y\_pred = np.clip(y\_pred, epsilon, 1 - epsilon)

J = np.sum(- y\_true\*np.log(y\_pred) - (1 - y\_true)\*np.log(1 - y\_pred)) / (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 train(self, \*, epochs: int, theta: np.ndarray, input: np.ndarray,

output: np.ndarray, lr: float, plot\_graph: False, color: list, time\_delay: float) -> np.ndarray:

output = output.reshape(-1, 1)

normalized\_input = self.normalize\_input(X= input)

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

J\_array = np.array([])

for epoch in range(epochs):

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

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

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

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

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

return J\_array, theta