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

Exercise 6: Regularization

* Chia dữ liệu ra thành 2 tập: training (70%) và validation (30%). Phải đảm bảo   
  việc chia dữ liệu là ngẫu nhiên và tỷ lệ positive và negative cân bằng.

import matplotlib.pyplot as plt

import plotly.graph\_objects as go

import numpy as np

import pandas as pd

ex5\_data = pd.read\_csv('ex4data.txt', header= None).to\_numpy()

x1, x2, y = ex5\_data[:, 0], ex5\_data[:, 1], ex5\_data[:, 2]

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

pos = ex5\_data[y == 1]

neg = ex5\_data[y == 0]

rng = np.random.default\_rng()

rng.shuffle(pos)

rng.shuffle(neg)

len\_train = int(0.7 \* len(y))

len\_valid = len(y) - len\_train

train = np.vstack((pos[0: int(len\_train / 2)], neg[0: len\_train - int(len\_train / 2)]))

valid = np.vstack((pos[int(len\_train / 2): ], neg[(len\_train - int(len\_train / 2)): ]))

np.random.shuffle(train)

np.random.shuffle(valid)

x\_train = train[:, :2]

y\_train = train[:, -1]

x\_valid = valid[:, :2]

y\_valid = valid[:, -1]

* Viết chương trình cho phép học các tham số của mô hình phân loại phi   
  tuyến trên có sử dụng regularization L2 và L1

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)

        std = np.where(std == 0, 1, std) # avoid divide 0

        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, lambda\_: float, theta: np.ndarray, type\_norm: None) -> np.ndarray:

        m = len(y\_true)

        epsilon = 1e-15

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

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

        theta\_update[1:] = theta[1:]

        if type\_norm == "L2":

            J = np.sum(- y\_true\*np.log(y\_pred) - (1 - y\_true)\*np.log(1 - y\_pred)) / (m) + lambda\_\*np.sum(theta\_update\*\*2) / (2 \* m)

        elif type\_norm == "L1":

            J = np.sum(- y\_true\*np.log(y\_pred) - (1 - y\_true)\*np.log(1 - y\_pred)) / (m) + lambda\_\*np.sum(np.abs(theta\_update)) / m

        return J

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

                      y\_true: np.ndarray, normalized\_input: np.ndarray, lambda\_: float, type\_norm: None) -> np.ndarray:

        m = len(y\_true)

        E = y\_pred - y\_true

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

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

        theta\_regular[1:] = theta[1:]

        if type\_norm == "L2":

            theta\_updated = theta - lr \* (dJ\_dtheta + lambda\_ \* theta\_regular) / m

        elif type\_norm == "L1":

            theta\_updated = theta - lr \* (dJ\_dtheta + lambda\_ \* np.sign(theta\_regular) ) / m

        return theta\_updated

    def train(self, \*, epochs: int, theta: np.ndarray, input: np.ndarray, lambda\_: float,

              output: np.ndarray, lr: float, type\_norm: str) -> 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, lambda\_= lambda\_, theta= theta, type\_norm= type\_norm)

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

            theta = self.update\_params(theta= theta, lr= lr, y\_pred= y\_pred, lambda\_= lambda\_, type\_norm= type\_norm,

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

        return J\_array, theta

    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, lambda\_: float, theta: np.ndarray, type\_norm: None) -> np.ndarray:

        m = len(y\_true)

        epsilon = 1e-15

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

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

        theta\_update[1:] = theta[1:]

        if type\_norm == "L2":

            J = np.sum(- y\_true\*np.log(y\_pred) - (1 - y\_true)\*np.log(1 - y\_pred)) / (m) + lambda\_\*np.sum(theta\_update\*\*2) / (2 \* m)

        elif type\_norm == "L1":

            J = np.sum(- y\_true\*np.log(y\_pred) - (1 - y\_true)\*np.log(1 - y\_pred)) / (m) + lambda\_\*np.sum(np.abs(theta\_update)) / m

        return J

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

                      y\_true: np.ndarray, normalized\_input: np.ndarray, lambda\_: float, type\_norm: None) -> np.ndarray:

        m = len(y\_true)

        E = y\_pred - y\_true

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

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

        theta\_regular[1:] = theta[1:]

        if type\_norm == "L2":

            theta\_updated = theta - lr \* (dJ\_dtheta + lambda\_ \* theta\_regular) / m

        elif type\_norm == "L1":

            theta\_updated = theta - lr \* (dJ\_dtheta + lambda\_ \* np.sign(theta\_regular) ) / m

        return theta\_updated

    def train(self, \*, epochs: int, theta: np.ndarray, input: np.ndarray, lambda\_: float,

              output: np.ndarray, lr: float, type\_norm: str) -> 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, lambda\_= lambda\_, theta= theta, type\_norm= type\_norm)

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

            theta = self.update\_params(theta= theta, lr= lr, y\_pred= y\_pred, lambda\_= lambda\_, type\_norm= type\_norm,

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

        return J\_array, theta

    def train(self, \*, epochs: int, theta: np.ndarray, input: np.ndarray, lambda\_: float,

              output: np.ndarray, lr: float, type\_norm: str) -> 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, lambda\_= lambda\_, theta= theta, type\_norm= type\_norm)

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

            theta = self.update\_params(theta= theta, lr= lr, y\_pred= y\_pred, lambda\_= lambda\_, type\_norm= type\_norm,

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

        return J\_array, theta

* Tính J ở mỗi vòng lặp cho cả hai tập, chọn điểm dừng phù hợp.

n = 0

X\_mapFeature\_train = np.zeros((len(x\_train), 27))

X\_mapFeature\_valid = np.zeros((len(x\_valid), 27))

for order in range(1, 7, 1):

    for i in range(0, order+1):

        x12\_train = x\_train[:, 0]\*\*(order - i) \* x\_train[:, 1]\*\*(i)

        X\_mapFeature\_train[:, n] = x12\_train

        x12\_valid = x\_valid[:, 0]\*\*(order - i) \* x\_valid[:, 1]\*\*(i)

        X\_mapFeature\_valid[:, n] = x12\_valid

        n = n + 1

num\_feature = X\_mapFeature\_train.shape[1]

Logistic\_Regression = Logistic\_Regression\_Multivariables(number\_of\_feature= num\_feature)

np.random.seed(1)

theta\_init = np.random.rand(num\_feature + 1, 1)

# theta\_init

J = np.zeros((1000, ))

theta\_ = np.zeros\_like(theta\_init)

J\_train, theta\_train = Logistic\_Regression.train(epochs= 50, theta= theta\_init, type\_norm= 'L2',

                                        input= X\_mapFeature\_train, output= y\_train, lr= i, lambda\_= 10)

J\_test, theta\_test = Logistic\_Regression.train(epochs= 50, theta= theta\_init, type\_norm= 'L2',

                                      input= X\_mapFeature\_valid, output= y\_valid, lr= i, lambda\_= 10)

fig = go.Figure()

fig.add\_trace(go.Scatter(x=np.arange(1000), y= J\_train,

                        mode= 'lines', name= f'train'))

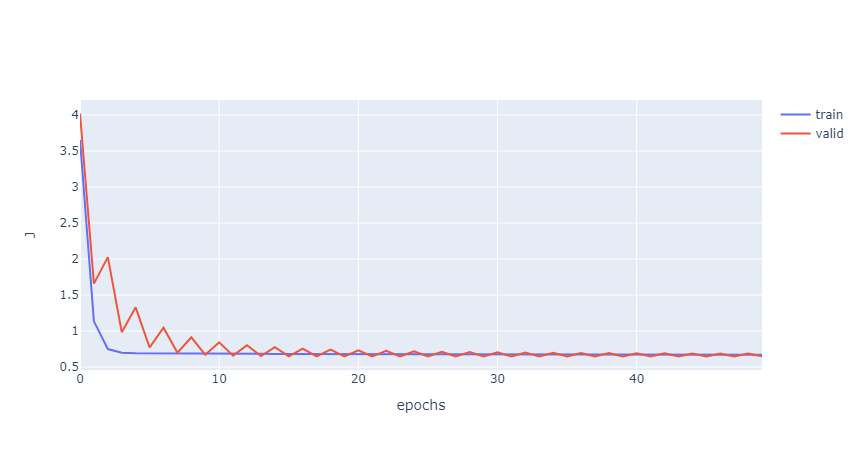
fig.add\_trace(go.Scatter(x=np.arange(1000), y=J\_test,

                        mode= 'lines', name= f'valid'))

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

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

fig.show()



* Epoch = 50