Practica 3

Regresión Logistica

**CÓDIGO FUENTE DE LA PARTE A:**

def our\_test\_A():

    #read data

    X , Y = readData("ex2data1.txt")

    #initial values

    b\_init = 1

    w\_init = np.array([0.0, 0.0])

    iterations = 10000

    alpha = 0.01

    #TRAINING

    w , b, history = lr.gradient\_descent(X, Y, w\_init, b\_init,lr.compute\_cost\_reg, lr.compute\_gradient\_reg, alpha , iterations)

    #Predict Values

    correct\_ones = 0

    for i in range(Y.shape[0]):

        if(Y[i] == lr.predict(X[i], w,b)):

            correct\_ones +=1

    print(f"\n The accuracy is {(correct\_ones/Y.shape[0])\*100} % \n")

    #Show values and function

    utils.plot\_data(X, Y, "y\_=1", "y\_=0", 'green', 'blue')

    utils.plot\_decision\_boundary(w, b, X, Y)

    plt.legend()

    plt.show()

**CÓDIGO FUENTE DE LA PARTE B:**

def our\_test\_B():

    #read data

    X , Y = readData("ex2data2.txt")

    #initial values

    b\_init = 1

    iterations = 1000000

    alpha = 0.01

    #TRAINING

    #X[:, 0] -> : todas las filas de la columna 0

    X\_stack = utils.map\_feature(X[:,0],X[:,1])

    w\_init = np.zeros(X\_stack.shape[1])

    w , b, history = lr.gradient\_descent(X\_stack, Y, w\_init, b\_init,lr.compute\_cost\_reg, lr.compute\_gradient\_reg, alpha , iterations)

    correct\_ones = 0

    for i in range(Y.shape[0]):

        if(Y[i] == lr.predict(X\_stack[i], w,b)):

            correct\_ones +=1

    print(f"\n The accuracy is {(correct\_ones/Y.shape[0])\*100} % \n")

    #Show values and function

    utils.plot\_data(X, Y, "y\_=1", "y\_=0", 'green', 'blue')

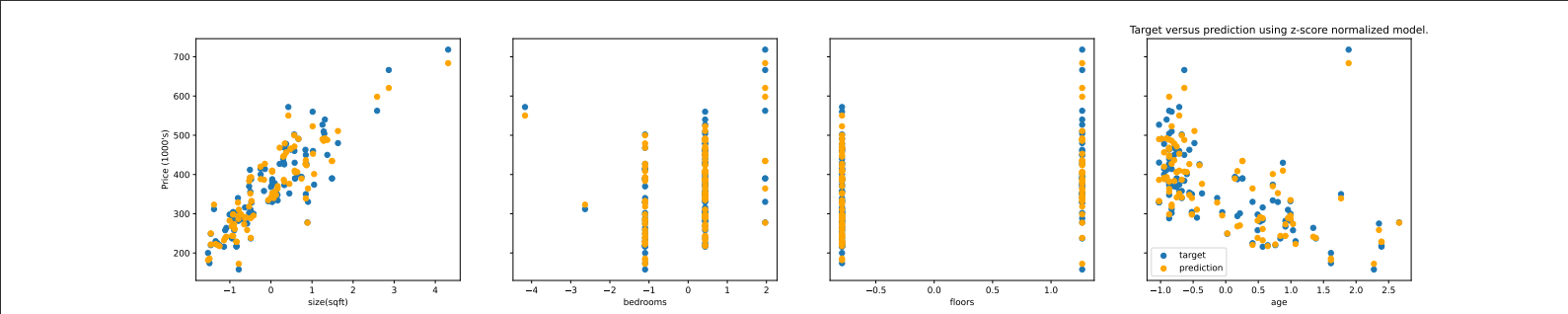
    utils.plot\_decision\_boundary(w, b, X\_stack, Y)

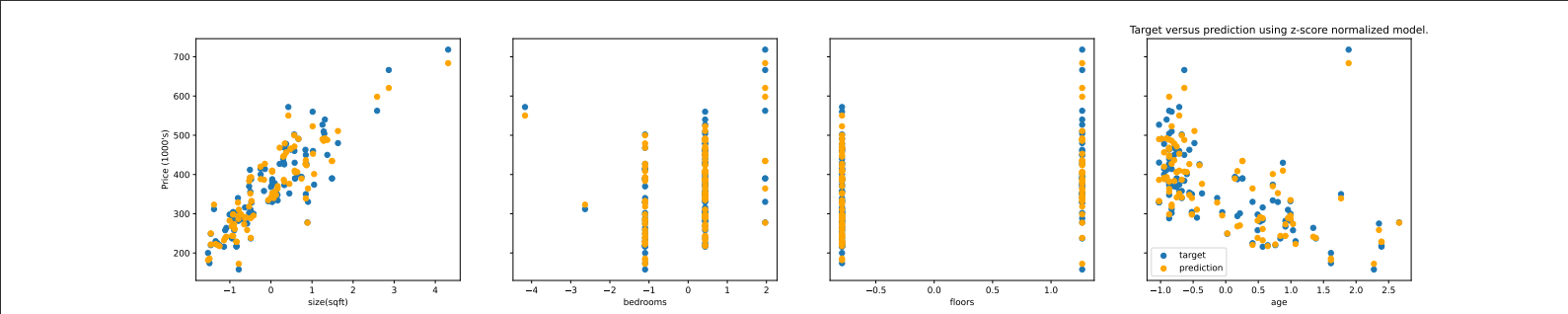
    plt.legend()

    plt.show()

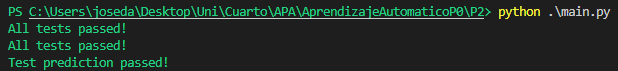
Ejecutando los datos almacenados en el archivo “ex2data1.txt”, obtenemos la gráfica de las notas con la recta de regresión.

**GRÁFICA:**





Para comprobar que los métodos programados funcionan correctamente, pasamos estos por las funciones de testeo proporcionadas en el archivo public\_test.py.



**IMPLEMENTACIÓN**

Los métodos implementados para el Descenso de Gradiente son:

**Calcular el costo de la función**

def compute\_cost(X, y, w, b):

    m = y.shape[0]

    return  np.sum(((X @ w + b) - y ) \*\* 2) / (2 \* m )

**Calcular el gradiente**

def compute\_gradient(X, y, w, b):

    m = y.shape[0]

    fun = X @ w + b

    e = fun - y

    dj\_db = np.sum(e / m)

    dj\_dw = (X.T @ e) / m

    return dj\_db , dj\_dw

**Descenso de Gradiente**

def gradient\_descent(X, y, w\_in, b\_in, cost\_function,

                     gradient\_function, alpha, num\_iters):

    J\_history = []

    w = copy.deepcopy(w\_in)

    b = b\_in

    for i in range(num\_iters):

      dj\_db, dj\_dw = gradient\_function(X, y, w, b)

      w -= alpha \* dj\_dw

      b -= alpha \* dj\_db

      if i < 100000:

        cost = cost\_function(X,y,w,b)

        J\_history.append(cost)

    return w, b, J\_history

Y por último, el método que nos permite normalizar los valores de entrada a datos más manejables (pequeños), ya que de lo contrario tendríamos números tan grandes que complicarían la manipulación con ellos.

**Normalización**

def zscore\_normalize\_features(X):

    X\_norm = 0

    X\_norm = np.empty((X.shape[0], X.shape[1]))

    # if(len(X.shape) > 1):

    # else:

    #   X\_norm = np.empty((X.shape[0]))

    mu = np.mean(X , axis = 0)

    sigma = np.std(X , axis = 0)

    # X\_norm = (X - mu) / sigma

    for i in range (len(X)):

      X\_norm[i] = (X[i] - mu) / sigma

    return (X\_norm, mu, sigma)