Practica 2

Regresión Multivariable

**CÓDIGO FUENTE:**

def our\_test():

    data = np.loadtxt("./data/houses.txt", delimiter=',', skiprows=1)

    X\_train = data[:, :4]

    y\_train = data[:, 4]

    b\_init = 785.1811367994083

    w\_init = np.array([0.39133535, 18.75376741, -53.36032453, -26.42131618])

    X\_train\_norm, mu, sigma = ml.zscore\_normalize\_features(X\_train)

    iterations = 1500

    alpha = 0.01

    # #TRAINING

    w , b, history = ml.gradient\_descent(X\_train\_norm, y\_train, w\_init, b\_init, ml.compute\_cost,

                                         ml.compute\_gradient, alpha, iterations)

    X = np.array([1200, 3, 1, 40])

    X = (X - mu) / sigma

    test = np.sum((X @ w)) + b

    target\_value = 318.85441519816953

    assert np.isclose(test, target\_value,

                      rtol=1e-4), f"Case 1: prediction is wrong: {test} != {target\_value}"

    print("\033[92mTest prediction passed!")

    Y\_prediction = (X\_train\_norm @ w) + b

    X\_features = ['size(sqft)', 'bedrooms', 'floors', 'age']

    fig, ax = plt.subplots(1, 4, figsize=(25, 5), sharey=True)

    plt.title('Target versus prediction using z-score normalized model.')

    for i in range(len(ax)):

        ax[i].scatter(X\_train\_norm[:, i], y\_train, label = 'target')

        ax[i].scatter(X\_train\_norm[:, i], Y\_prediction, color = 'orange', label = 'prediction')

        ax[i].set\_xlabel(X\_features[i])

    plt.legend()

    ax[0].set\_ylabel("Price (1000's)")

    # plt.show()

    plt.savefig('linearRegression\_prediction.pdf')

def public\_Test():

    test.compute\_cost\_test(ml.compute\_cost)

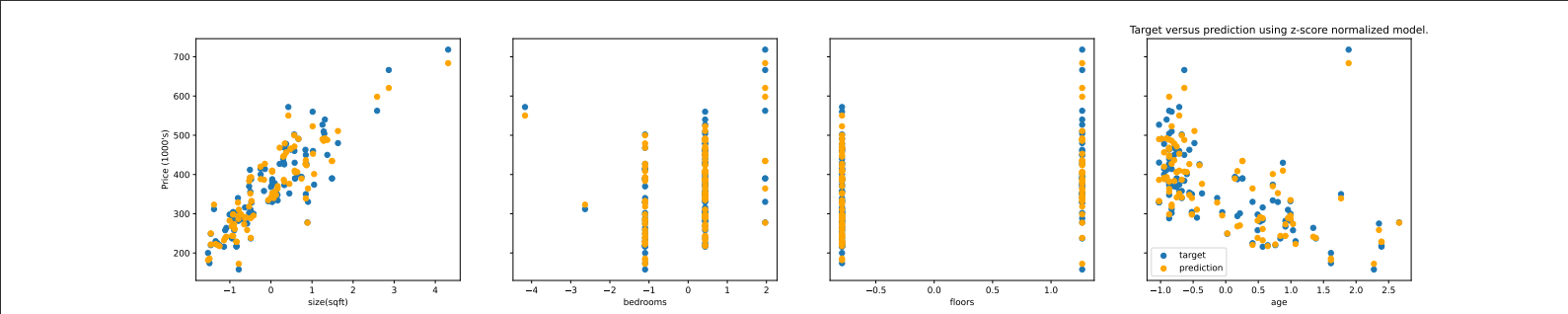
    test.compute\_gradient\_test(ml.compute\_gradient)

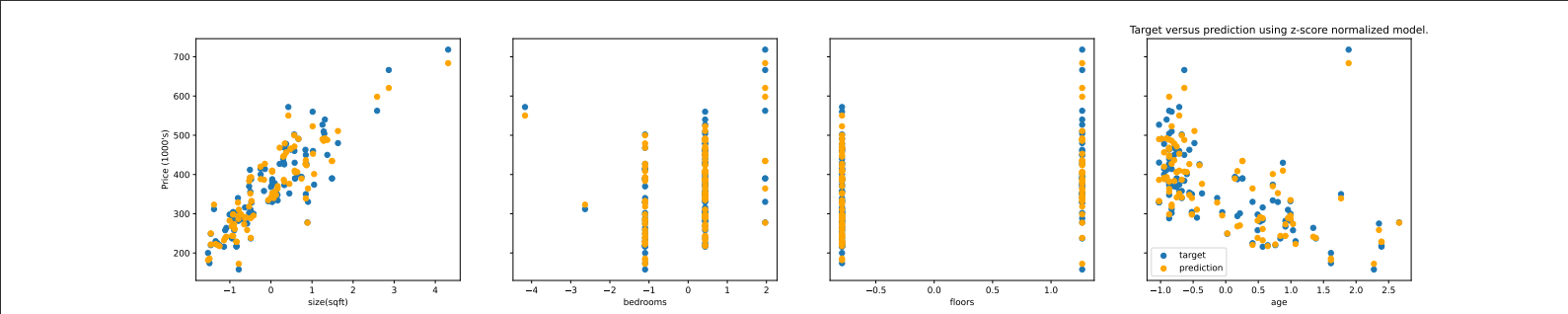
def main():

    public\_Test()

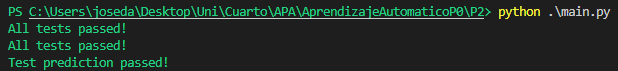
    our\_test()

Ejecutando los datos almacenados en el archivo “houses.txt”, obtenemos cuatro gráficas con sus datos representados, y una representación en naranja que predice los valores de dichos datos habiendo obtenido *w* y *b* una vez se ha realizado el entrenamiento.  
**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)