Práctica 1.2: Regresión Lineal Multivariable

Esther Babon Arcauz

El primer paso ha sido implementar las funciones de normalización de datos de entrada, coste, gradiente y descenso de gradiente.

Respecto a la función de normalización he utilizado el z-score ya que utiliza la media y la desviación estándar de cada característica.

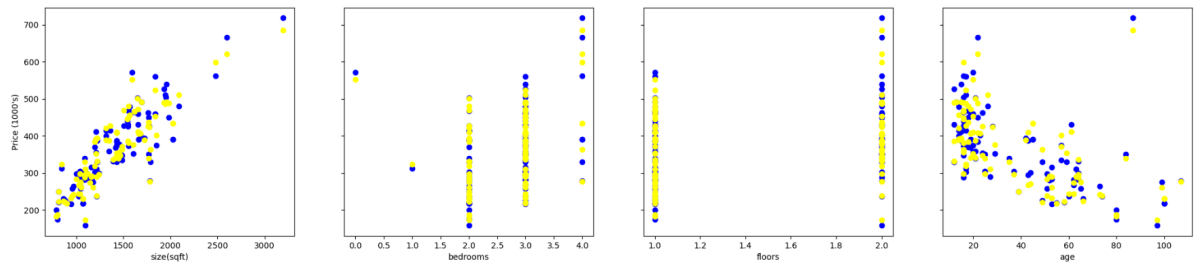
Respecto a las funciones de coste, gradiente y coste de gradiente he aplicado las fórmulas proporcionadas usando operaciones vectoriales.

Tras usar las funciones de test para ver que todo estaba en orden, he sacado los datos de entrenamiento de houses.txt y los he normalizado, guardando los valores de media y desviación estándar para utilizarlos más tarde. He inicializado las variables y he entrenado el modelo.

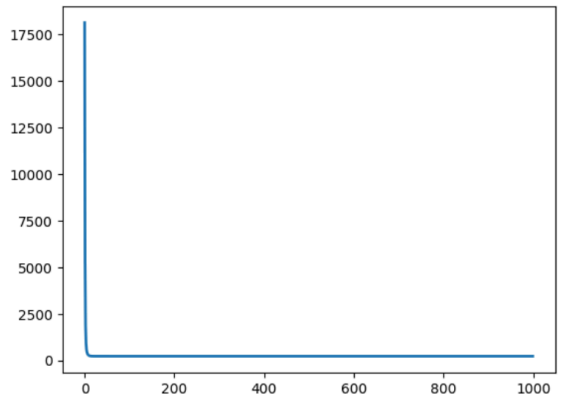
Después he normalizado los valores sobre los que quería predecir usando los valores previamente guardados de media y desviación estándar, y he predicho el coste de la casa de ejemplo proporcionada.

```
Predicted Price for the new house: [318.93635904]
```

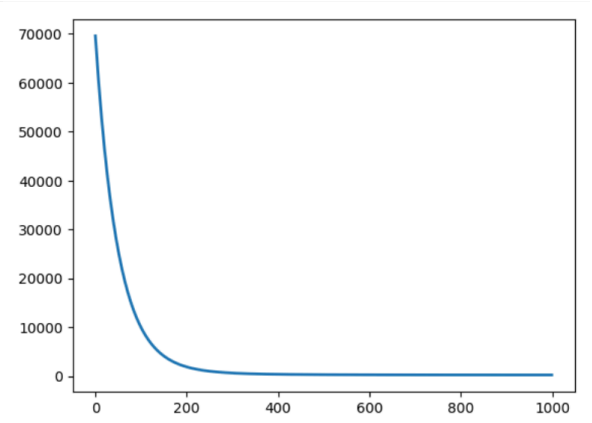
Una vez visto que el resultado estaba dentro de los valores esperados, he ejecutado el modelo sobre los datos de entrenamiento, para así dibujar la gráfica inferior. Los puntos azules son los valores reales de entrenamiento y los amarillos los que ha predicho el modelo.



Para ver si las variables de tasa de aprendizaje y número de iteraciones son las ideales dibujo la gráficas.

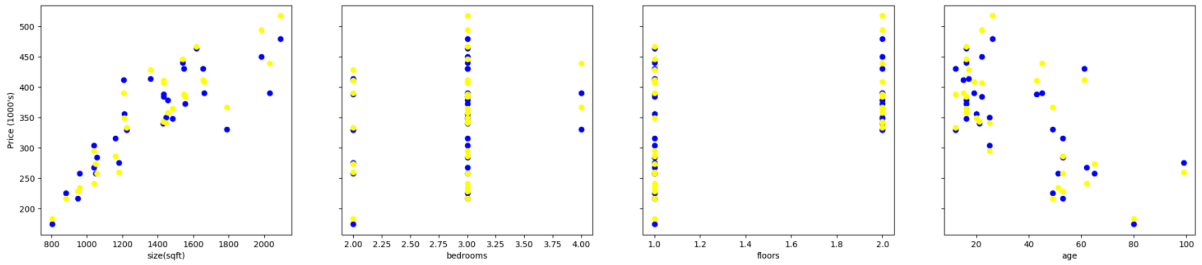


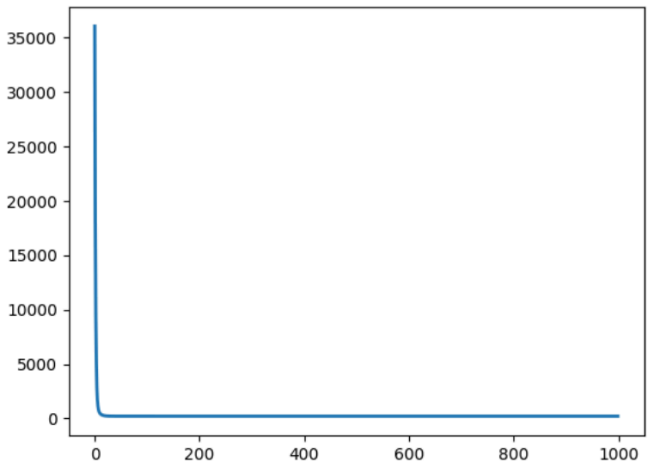
Si cambio el valor de la tasa de aprendizaje a $\propto = 0.01$ la gráfica de coste por número de iteraciones queda de la siguiente manera. Vemos que hacen falta más iteraciones para converger.



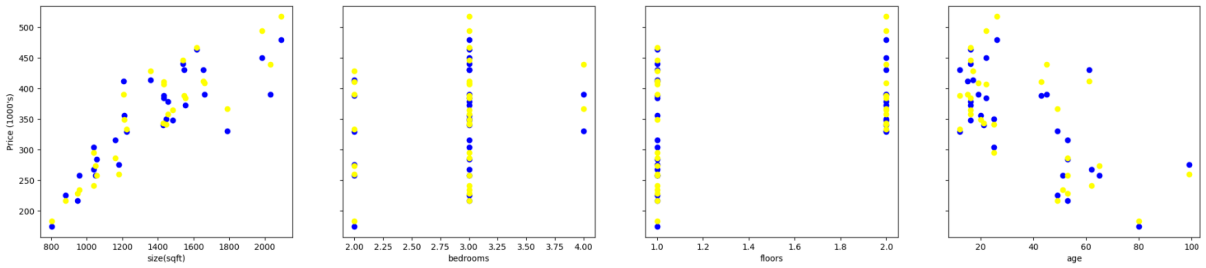
Después de intentarlo con $\propto = 0.3$ vemos que la gráfica es muy similar a la de $\propto = 0.1$, pero seguramente esté sobre aprendiendo de los datos de entrenamiento.
Para ello podemos dividir el conjunto de entrenamiento dado en train y test (70/30). Y ver cual de las dos tasas de aprendizaje es la más adecuada.

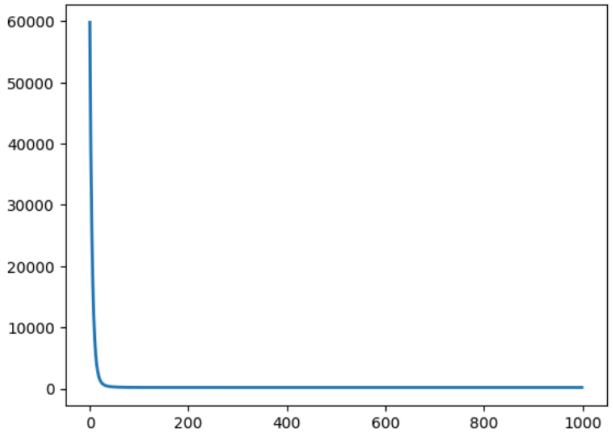
Sobre el número de iteraciones no voy a comentar mucho, ya que es un modelo rápido y podemos permitirnos hacer 1000 iteraciones.





$\propto = 0.3$





$\propto = 0.1$

Vemos que no hay mucha diferencia entre las predicciones (en amarillo) y los valores reales (en azul) usando las tasas de aprendizaje 0.3 y 0.1, asi que podemos afirmar que el modelo funciona bien con las dos tasas de aprendizaje, aunque la de 0.3 consigue converger en menos iteraciones.

```
import numpy as np
import matplotlib.pyplot as plt
import copy
import math


def zscore_normalize_features(X):
    """
    computes  X, zcore normalized by column

    Args:
      X (ndarray (m,n))     : input data, m examples, n features

    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (n,))  : standard deviation of each feature
    """
    #calculamos la media  y la deviacion estandar para el input data
    mu = np.mean(X, axis=0)
    sigma = np.std(X, axis=0)

    #aplicamos la formula de z-score normalization
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)


def compute_cost(X, y, w, b):
    """
    compute cost
    Args:
      X (ndarray (m,n)): Data, m examples with n features
```

```
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter
    Returns
      cost (scalar)    : cost
    """

    #calculamos las predicciones de manera vectorial
    fw_b = np.dot(X, w) + b

    #calculamos el coste segun la funcion de coste dada
    cost = (1/(2*len(y))) * np.sum(np.square(fw_b - y))

    return cost


def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X : (ndarray Shape (m,n)) matrix of examples
      y : (ndarray Shape (m,))  target value of each example
      w : (ndarray Shape (n,))  parameters of the model
      b : (scalar)              parameter of the model
    Returns
      dj_dw : (ndarray Shape (n,)) The gradient of the cost w.r.t. the
parameters w.
      dj_db : (scalar)             The gradient of the cost w.r.t. the
parameter b.
    """
    #calculamos las prediccionesde manera vectorial
    fw_b = np.dot(X, w) + b


    #calculamos los gradientes de las funciones de coste respecto a los
parametros w y b
    dj_dw = (1/len(y)) * np.dot((fw_b - y), X)
    dj_db = (1/len(y)) * np.sum(fw_b - y)


    return dj_db, dj_dw
```

```python
def gradient_descent(X, y, w_in, b_in, cost_function,
                     gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X : (array_like Shape (m,n)    matrix of examples
      y : (array_like Shape (m,))    target value of each example
      w_in : (array_like Shape (n,)) Initial values of parameters of the model
      b_in : (scalar)                Initial value of parameter of the model
      cost_function: function to compute cost
      gradient_function: function to compute the gradient
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run gradient descent
    Returns
      w : (array_like Shape (n,)) Updated values of parameters of the model
          after running gradient descent
      b : (scalar)                Updated value of parameter of the model
          after running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each iteration,
          primarily for graphing later
    """
    #inicializamos la lista
    J_history = []

    for a in range(num_iters):

        #calculamos el gradiente con la funcion que nos pasan por parametros
        dj_db, dj_dw = gradient_function(X, y, w_in, b_in)
        #actualizamos los parametros
        w_in -= alpha * dj_dw
        b_in -= alpha * dj_db

        #calculamos el coste con la funcion que nos pasan por parametros
        cost = cost_function(X, y, w_in, b_in)

        #añadimos el coste de cada iteracion a la lista
        J_history.append(cost)

    return w_in, b_in, J_history
```

```
def load_data_multi():
    data = np.loadtxt("houses.txt", delimiter=',')
    X = data[:, [0,1,2,3]]  # Tomar todas las columnas excepto la última para
X
    y = data[:, 4]   # Tomar la última columna para y
    return X, y

#datos de entrenamiento
X_train, y_train = load_data_multi()
#normalizamos los datos
x_train_norm, mu, sigma = zscore_normalize_features(X_train)

#inicializamos variables
w_init = np.zeros(4)
b_init = 0
alpha = 0.1
num_iters = 1000

#entrenamos el modelo
w_trained, b_trained, J_history = gradient_descent(x_train_norm, y_train,
w_init, b_init, compute_cost, compute_gradient, alpha, num_iters)

#predecimos para el ejemplo dado
x_pred = np.array([[1200, 3, 1, 40]])
#normalizamos los valores de entrada con los mismos valores con los que lo
hicimos con los datos de entrenamiento
x_new_norm =  (x_pred - mu) / sigma
#predecimos de manera vectorial con los datos entrenados
predicted_price = np.dot(x_new_norm, w_trained) + b_trained
print("Predicted Price for the new house:", predicted_price)



#A dibujar
x_t, y_t = load_data_multi()

#predecimos sobre el dato de entrenamiento
normX = (x_t - mu) / sigma
predicted = np.dot(normX, w_trained) + b_trained
```

```
#dibujamos los datos de entrenamieto reales en azul con las predicciones en
amarillo
X_features = ['size(sqft)', 'bedrooms', 'floors', 'age']
fig, ax = plt.subplots(1, 4, figsize=(25, 5), sharey=True)
for i in range(len(ax)):
  ax[i].scatter(x_t[:, i], y_t, color='blue')
  ax[i].scatter(x_t[:,i], predicted, color='yellow')
  ax[i].set_xlabel(X_features[i])
  ax[0].set_ylabel("Price (1000's)")



#Dibujamos grafica de Jhistory y num iteraciones
fig, ax = plt.subplots()
ax.plot(J_history, linewidth=2.0)
plt.show()



#input data 70/30 train test
data_train = np.loadtxt("houses.txt", delimiter=',')
xtrain = data_train[:70, :4]
ytrain = data_train[:70, 4]

xtest = data_train[70:, :4]
ytest = data_train[70:, 4]
#normalizamos los datos
x_train_norm, mu, sigma = zscore_normalize_features(xtrain)

#inicializamos variables
w_init = np.zeros(4)
b_init = 0
alpha = 0.3
num_iters = 1000

#entrenamos el modelo
w_trained, b_trained, J_history = gradient_descent(x_train_norm, ytrain,
w_init, b_init,
                                                   compute_cost,
compute_gradient, alpha, num_iters)
```

```
#predecimos sobre el dato de entrenamiento
normxtest = (xtest - mu) / sigma
predicted = np.dot(normxtest, w_trained) + b_trained

#dibujamos los datos de tes reales en azul con las predicciones en amarillo
X_features = ['size(sqft)', 'bedrooms', 'floors', 'age']
fig, ax = plt.subplots(1, 4, figsize=(25, 5), sharey=True)
for i in range(len(ax)):
  ax[i].scatter(xtest[:,i], ytest, color='blue')
  ax[i].scatter(xtest[:,i], predicted, color='yellow')
  ax[i].set_xlabel(X_features[i])
  ax[0].set_ylabel("Price (1000's)")
```