Memoria P2. Regresión Lineal Multivariable

En esta práctica se ha implementado un ***modelo de regresión lineal multivariable*** por lotes, que nos permite derivar información a partir de un conjunto de datos independientes y un conjunto de datos dependientes.

Normalización de características

La función **zscore_normalize_features** se utiliza para normalizar las características de un conjunto de datos, tomando como parámetro:

- **X**: conjunto de datos a normalizar.

X es una matriz de tamaño **m*n**, donde **m** es el número de ejemplos y **n** es el número de características. Primero calculamos la media y la desviación estándar de **X** y guardamos estos valores en las variables **mu** y **sigma**.

A continuación utilizamos estas variables para normalizar los valores de **X**, sustrayendo a **X** la media y dividiendo el resultado entre la desviación estándar. Esta función devuelve los valores normalizados (**X_norm**), la media (**mu**) y la desviación estándar (**sigma**).

```
def zscore_normalize_features(X):

    mu = np.mean(X)
    sigma = np.std(X)
    X_norm = (X - mu) / sigma

    return X_norm, mu, sigma
```

Cómputo del costo

La función **compute_cost** se utiliza para calcular el coste del modelo de regresión lineal multivariable, y recibe los siguientes parámetros:

- **X**: matriz de ejemplos.
- **y**: valor de cada ejemplo.
- **w** y **b**: parámetros del modelo.

La función comienza definiendo una variable **m** del tamaño del conjunto **X**. Calculamos el valor predecido por el modelo de regresión lineal mediante una multiplicación matricial y después calculamos el coste para los elementos del conjunto **X** como la diferencia entre este valor predecido y el valor real, elevado al cuadrado.

Una vez que se ha recorrido todo el conjunto **X**, se calcula el costo total como el cociente entre **cost** y el doble del número de elementos del conjunto **X**. Este valor es el que se devuelve como resultado de la función.

```
def compute_cost(X, y, w, b):
    m = X.shape[0]

    # unoptimized (iterative)
    # cost = 0
    # for i in range(m):
    #     f_wb = np.dot(w, X[i]) + b
    #     cost_i = (f_wb - y[i]) ** 2
    #     cost += cost_i

    # optimized (vectorized)
    f_wb = X @ w + b
    cost = np.sum((f_wb - y) ** 2)

    cost /= 2 * m

    return cost
```

Cómputo del gradiente

La función **compute_gradient** se utiliza para calcular el gradiente del costo del modelo de regresión lineal multivariable, y recibe los siguientes parámetros:

- **X**: La matriz de ejemplos.
- **y**: El valor de cada ejemplo.
- **w** y **b**: Parámetros del modelo.

La función comienza definiendo una variable m del tamaño del conjunto **X**. A continuación, se calcula el gradiente de la función de costo para cada elemento del conjunto **X,** respecto a **w** y **b**. Estos gradientes se guardan en las variables **dj_dw** y **dj_db**.

Tras esto se divide cada uno de los gradientes por el número de elementos del conjunto **X**. Los valores obtenidos son los que se devuelven como resultado de la función.

```python
def compute_gradient(X, y, w, b):
    m = X.shape[0]

    # unoptimized (iterative)
    # dj_dw = 0
    # dj_db = 0
    # for i in range(m):
    #     f_wb = np.dot(w, X[i]) + b
    #     dj_dw_i = np.dot((f_wb - y[i]), X[i])
    #     dj_db_i = (f_wb - y[i])

    #     dj_dw += dj_dw_i
    #     dj_db += dj_db_i

    # optimized (vectorized)
    f_wb = X @ w + b
    dj_dw = X.T @ (f_wb - y)
    dj_db = np.sum(f_wb - y)

    dj_dw /= m
    dj_db /= m

    return dj_dw, dj_db
```

Descenso de gradiente

La función **gradient_descent** realiza el descenso de gradiente por lotes (Batch gradient descent), y recibe los siguientes parámetros:

- **X**: matriz de ejemplos.
- **y**: valor de cada ejemplo
- **w_in** y **b_in**: calores iniciales de los parámetros del modelo.
- **cost_function**: función de coste del modelo.
- **gradient_function**: función del gradiente.
- **alpha**: tasa de aprendizaje del algoritmo.
- **num_iters**: número de iteraciones a realizar.

La función comienza definiendo una lista **J_history** que se utilizará para almacenar el valor del costo en cada una de las iteraciones del algoritmo. También se definen las variables **w** y **b**, que contienen los valores iniciales de los parámetros del modelo de regresión lineal.

A continuación, iteramos **num_iter** veces. En cada iteración, se calcula el gradiente de la función de costo en el punto actual utilizando la función **gradient_function**. Después, se actualizan los valores de los parámetros **w** y **b** restando a cada uno de ellos el producto de **alpha** y el gradiente correspondiente.

Una vez que se han realizado todas las iteraciones, la función devuelve los valores actualizados de los parámetros **w** y **b**, así como la lista **J_history** con los valores del costo en cada una de las iteraciones.

```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function,
alpha, num_iters):
    J_history = []
    w = copy.deepcopy(w_in)
    b = copy.deepcopy(b_in)

    for _ in range(num_iters):
        dj_dw, dj_db = gradient_function(X, y, w, b)

        w -= alpha * dj_dw
        b -= alpha * dj_db

        cost = cost_function(X, y, w, b)
        J_history.append(cost)

    return w, b, J_history
```

Cómputo de la regresión multilineal

La función **compute_multi_linear_reg_descent** realiza el cómputo de la regresión lineal, y recibe el parámetro **data**, el cual contiene el conjunto de valores casa/coste de las ciudades. Data se divide en **X** (el tamaño, número de habitacion, plantas y edad de la casa), e **y** (el precio de la casa).

Definimos los valores iniciales de los parámetros del modelo de regresión lineal, **w_in** y **b_in**, en cero (siendo **w_in** un array de Os del tamaño de los datos de entrada). También se define la tasa de aprendizaje **alpha** y el número de iteraciones **num_iters** que se realizan en el algoritmo de descenso del gradiente.

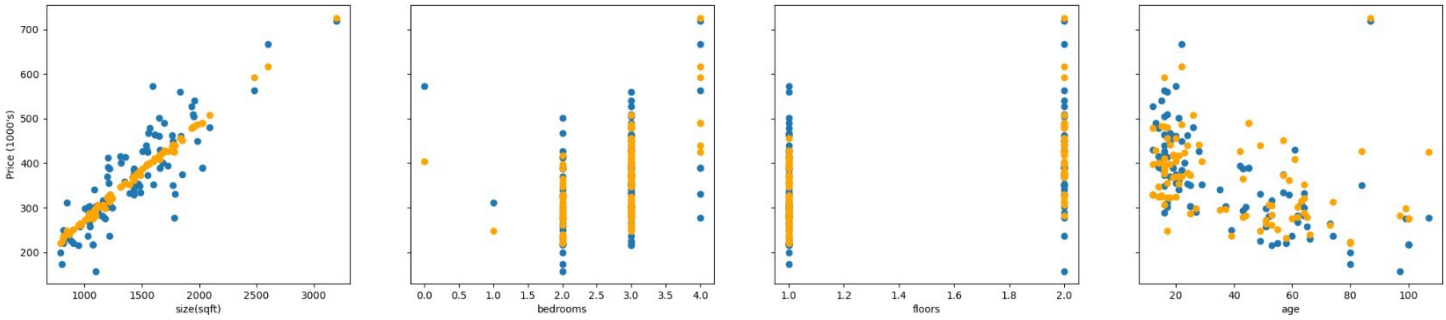
A continuación, llamamos a la función **gradient_descent**, que es la encargada de realizar el proceso de optimización por descenso del gradiente.

Por último, devolvemos los valores de los parámetros **w** y **b** obtenidos tras el proceso de optimización, así como una lista de los valores del costo en cada una de las iteraciones realizadas.

```
def compute_multi_linear_reg_descent(data):
    # initial data and parameters
    x_train, y_train = data[:,:-1], data[:,-1]
    w_in, b_in = np.zeros(x_train.shape[1]), 0

    alpha = 0.01
    num_iters = 10000

    # we obtain w and b here
    w, b, costs = mlr.gradient_descent(x_train, y_train, w_in, b_in,
mlr.compute_cost, mlr.compute_gradient, alpha, num_iters)

    return w, b, costs
```

Resultado

Tras cargar los datos y hacer el cómputo de la regresión multilineal, obtenemos una gráfica que muestra la predicción obtenida:



Además, con el test realizado con un valor particular de entrada, se obtiene este resultado:

```
Price for a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old:
$317078.0592424978
```