Memoria P1. Regresión Lineal

Aarón Nauzet Moreno Sosa (aarmor01@ucm.es)
Tomás López Antón (tomalope@ucm.es)

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

En el caso de ejemplo, el conjunto de datos independientes es la población de varias ciudades, y el conjunto de casos dependientes es el beneficio obtenido al abrir una franquicia en cada una de estas ciudades.

A partir de estos datos iniciales, podemos predecir el posible beneficio a obtener si abrimos una franquicia en una nueva ciudad, conociendo la población de la ciudad.

Cómputo del costo

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

- **X**: Población de las ciudades.
- **y**: Beneficios obtenidos al establecer una franquicia en dichas ciudades.
- **w** y **b**: Parámetros del modelo.

La función comienza definiendo una variable **m** del tamaño del conjunto **X**. Después calculamos el coste para los elementos del conjunto **X** como la diferencia entre el valor predecido por el modelo de regresión lineal 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):
    # the shape attribute returns the dimensions of the array [x.shape=(n,m)]
    m = X.shape[0]

    # unoptimized (iterative)
    # cost = 0
    # for i in range(m):
    #     f_wb = w * x[i] + b
    #     cost_i = (f_wb - y[i]) ** 2
    #     cost += cost_i
```

```
    # optimized (vectorized)
    f_wb = w * X + 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, y recibe los siguientes parámetros:

- **X**: población de las ciudades.
- **y**: beneficios obtenidos al establecer una franquicia en dichas ciudades.
- **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.

```
def compute_gradient(X, y, w, b):
    # the shape attribute returns the dimensions of the array [x.shape=(n,m)]
    m = X.shape[0]

    # unoptimized (iterative)
    # dj_dw = 0
    # dj_db = 0
    # for i in range(m):
    #     f_wb = w * x[i] + b
    #     dj_dw_i = (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 = w * X + b
    dj_dw = np.sum((f_wb - y) * X)
    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**: población de las ciudades.
- **y**: beneficios obtenidos al establecer una franquicia en dichas ciudades.
- **w_in** y **b_in**: valores 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 comienzan como los valores iniciales de los parámetros del modelo de regresión lineal.

A continuación, iteramos **num_iters** 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.

Por último, se calcula el costo en el punto actual utilizando la función **cost_function** y se añade este valor a la lista **J_history**.

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 lineal

La función **compute_linear_reg_gradient_descent** realiza el cómputo de la regresión lineal, y recibe el parámetro **data**, el cual contiene el conjunto de valores población/beneficio de las ciudades. Data se divide en **X**, la población de las ciudades, e **y**, el beneficio de cada ciudad.

Definimos los valores iniciales de los parámetros del modelo de regresión lineal, **w_in** y **b_in**, en cero. 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_linear_reg_gradient_descent(data):
    # initial data and parameters
    X, y = data[0], data[1]
    w_in = b_in = 0

    alpha = 0.01
    num_iters = 1500
```

```
    # we obtain w and b here
    w, b, costs = lr.gradient_descent(X, y, w_in, b_in, lr.compute_cost,
lr.compute_gradient, alpha, num_iters)

    return w, b, costs
```

Resultado

Tras cargar los datos y hacer el cómputo de la regresión lineal, obtenemos una gráfica que muestra el ajuste lineal obtenido: