Memoria P3. Regresión Logística

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

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

Parte A

Función sigmoide

La función **sigmoid** calcula la función sigmoide para un número o un conjunto de números/valores **z.**

```
def sigmoid(z):

    g = 1 / (1 + (np.e ** -z)) # np.exp(-z)

    return g
```

Cómputo del costo

La función **compute_cost** se utiliza para calcular el coste del modelo de regresión logística, 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 del sigmoide y después calculamos el coste o pérdida para los elementos del conjunto **X** en la variable **loss**.

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

```python
def compute_cost(X, y, w, b, lambda_=None):
    m = X.shape[0]

    f_wb = sigmoid(X @ w + b)
    loss = (-y @ np.log(f_wb)) - ((1 - y) @ np.log(1 - f_wb))
    cost = np.sum(loss) / m

    return cost
```

Cómputo del gradiente

La función **compute_gradient** se utiliza para calcular el gradiente del costo del modelo de regresión logística, 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**. 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, lambda_=None):
    m = X.shape[0]

    dj_dw = X.T @ (sigmoid(X @ w + b) - y)
    dj_db = np.sum((sigmoid(X @ w + b) - y))

    dj_dw /= m
    dj_db /= m

    return dj_dw, dj_db
```

Predicción

La función **predict** utiliza el modelo de regresión logística que hemos entrenado para predecir si la etiqueta de los datos es 0 o 1.

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

La función comienza calculando el valor de la función sigmoidal para cada uno de los ejemplos de **X** utilizando la función **sigmoid**. A continuación, se redondea el valor obtenido (pues esto generará o bien 0 ó 1) y se devuelve como predicción.

```
def predict(X, w, b):
    f_wb = sigmoid(X @ w + b)
    predict = np.round(f_wb)    # 0 or 1 values

    return predict
```

Parte B

Cómputo del costo para regresión logística regularizada

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

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

La función comienza definiendo una variable ***m*** del tamaño del conjunto **X**. Calculamos el valor del factor de regularización **reg_factor** en base al **lambda_** proporcionado, y después calculamos el coste total **cost** utilizando la función **compute_cost** anteriormente explicada. Finalmente, sumámos estos valores para obtener el costo total, y devolvemos el coste calculado.

```
def compute_cost_reg(X, y, w, b, lambda_=1):
    m = X.shape[0]

    reg_factor = (lambda_ / (2 * m)) * np.sum(w ** 2)
    cost = compute_cost(X, y, w, b) + reg_factor

    return cost
```

Cómputo del gradiente para regresión logística regularizada

La función **compute_gradient_reg** se utiliza para calcular el gradiente del costo del modelo de regresión logística regularizada, y recibe los siguientes parámetros:

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

La función comienza definiendo una variable m del tamaño del conjunto **X**. A continuación, calculamos el valor del factor de regularización **reg_factor.**

Tras esto, se calcula el gradiente mediante la función **compute_gradient**, la cual devuelve **dj_dw** y **dj_db**. Por último, antes de retornar estos parámetros, se suma el factor de regularización a los elementos del conjunto de gradientes **dj_dw**.

```
def compute_gradient_reg(X, y, w, b, lambda_=1):
    m = X.shape[0]

    reg_factor = (lambda_ * w / m)
    dj_dw, dj_db = compute_gradient(X, y, w, b)
    dj_dw += reg_factor

    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**: 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.
- **lambda_**: factor de regularizacion.

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 logística.

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.

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, lambda_=None):
    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, lambda_)

        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 logística

La función **compute_logistic_reg_gradient_descent** realiza el cómputo de la regresión logística, y recibe los parámetros:

- **X_data**: La matriz de ejemplos.
- **y_data**: La etiqueta asignada a cada ejemplo.
- **lambda_**: Constante de regularización.

La función comienza definiendo los parámetros iniciales, tales como el valor inicial de los parámetros del modelo **w_in** y **b_in**, el valor de la tasa de aprendizaje **alpha**, y el número de iteraciones que se ejecutará el algoritmo de descenso de gradiente **num_iters**.

Después determinamos si se utilizaran las funciones de regresión logística sin regularización, si **lambda_** no ha sido especificada, o las funciones de regresión logística regularizada en el caso contrario.

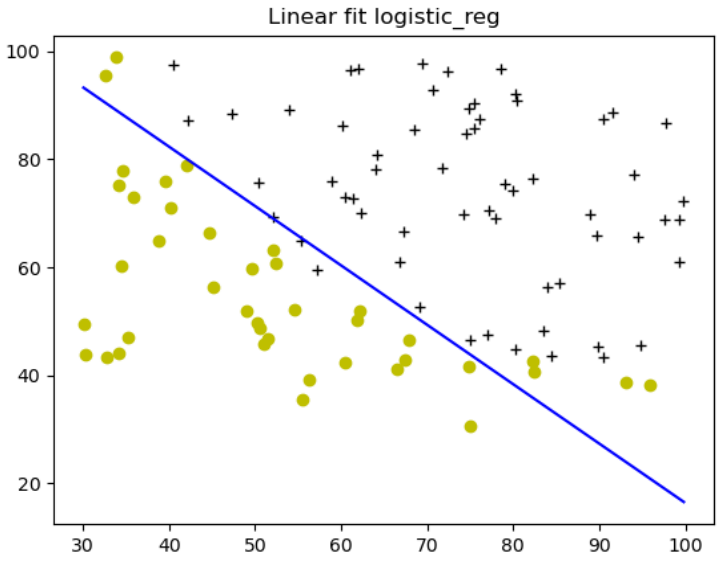
Por último obtenemos los valores de los parámetros **w** y **b**, y el conjunto de costes **costs** mediante la función **gradient_descent**.

```
def compute_logistic_reg_gradient_descent(X_data, y_data, lambda_=None):
    # initial data and parameters
    X_train, y_train = X_data, y_data
    w_in, b_in = np.zeros(X_train.shape[1]), -8 if lambda_ is None else 1

    alpha = 0.001 if lambda_ is None else 0.01
    compute_cost, compute_grad = (lgr.compute_cost, lgr.compute_gradient) if
lambda_ is None else (lgr.compute_cost_reg, lgr.compute_gradient_reg)
    num_iters = 10000

    # we obtain w and b here
    w, b, costs = lgr.gradient_descent(X_train, y_train, w_in, b_in,
compute_cost, compute_grad, alpha, num_iters, lambda_)

    return w, b, costs
```

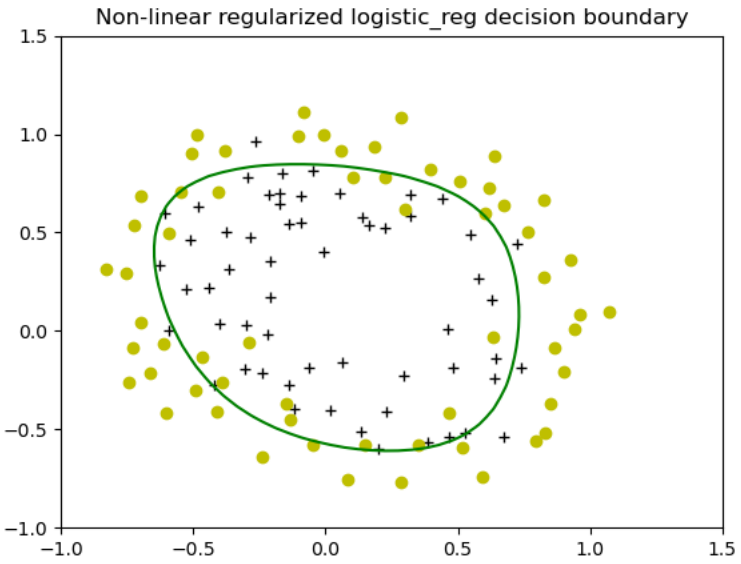
Resultado

Tras cargar los datos y hacer el cómputo de la regresión logística, obtenemos una gráfica que muestra la predicción obtenida, junto con el ajuste logístico:

Parte A



Parte B


Non-linear regularized logistic_reg decision boundary
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5