Practica02: Regresión Lineal Parte 2

Continuamos con nuestro análisis de mercado. Después del estudio anterior, hemos pensado que quizás haya más parámetros que afecten al **user score** más allá del score de metacritic.

Vamos a incorporar los campos "critics" y "users" que son respectivamente el número de criticas y el numero de usuarios que han realziado las críticas.

Se pide:

Ejericio 1:

Limpiad los datos, convertidlos a float64 y normalizar los datos con **ZScore** de los campos de entrada (score, critics y users), dejando el **user score** en escala 0 a 10 como aparece en el dataset.

Para ello debéis implementar la función **zscore_normalize_features** que se ecnuentra en el fichero utils.py

```
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
    """

    return X_norm, mu, sigma
```

Dibujad usando mathplotlib la distribución de los datos para cada variable, con respecto a **user score**.

Podéis usar para la visualización Jupiter Notebook (me lo entregáis como parte de la práctica) pero la limpieza debéis ponerla en la función **cleanData** del fichero **utils.py** para poder posteriormente aprovecharla en los ejercicios posteriores.

Ejericio 2:

Implementar la clase **class LinearRegMulti:** del fichero **LinearRegression-Multi.py** en su versión vectorial solamente. Deberá heredar de la versión vectorial de LinearReg que hayáis implementado. Las funciones a implementar son las mismas, pero hay que intentar aprovechar al máximo el código ya implementado en la versión de una variable. Se puede modificar la clase LinearReg

base, pero debeis aseguraros que de los test de la práctica 1 se siguen pasando. Para ello habrá que hacer un correcto uso de la programación orientada a objetos.

Para multiplicar matrices, podéis usar el operador @, este operador permite multiplcar dos matrices o una matriz y un vector siempre que sean compatibles. Es decir que se cumpla que (MxN) @ (N,K) = (M,K). En otras palabras, que las columnas de la matriz de la izquierda de @ deben coincidir con las filas de la matriz o vector de la derecha del operador @.

Ejercicio 3

Añadir al constructor un campo adicional denominado lambda (lambda_ para evitar usar la palabra reservada lambda) que es el parámetro de regularización. Implementar la regularización L2 en dos métodos separados denominados _regularizationL2Cost y _regularizationL2Gradient y añadirlo al cálculo del coste y del gradiente.

```
    """
    Compute the regularization cost (is private method: start with _ )
    This method will be reuse in the future.

    Returns
        _regularizationL2Cost (float): the regularization value of the current model
    """

    def _regularizationL2Cost(self):
        return reg_cost_final
```

Recordemos que la regularización L2 se calcula como:

$$L2 = \frac{\lambda}{2m} \cdot \sum_{j=1}^{n} (w_j^2)$$

```
    """
    Compute the regularization gradient (is private method: start with _ )
    This method will be reuse in the future.

    Returns
        _regularizationL2Gradient (vector size n): the regularization gradient of the curre
    """
    def _regularizationL2Gradient(self):
        return reg_gradient_final
```

Recordemos que el gradiente de la regularización L2 se calcula como:

$$\frac{\partial L2}{\partial w_j} = \frac{\lambda}{m} \cdot w_j$$

Ejercicio 4

Ejecutad los test que están en el fichero practica02.py:

- test_cost(x_train, y_train)
- test_gradient(x_train, y_train)
- test_gradient_descent(x_train, y_train)

Y comprobad que se pasan correctamente.

English verison

We continue with our market analysis. After the previous study, we thought that maybe there are more parameters that affect the **user score** beyond the metacritic score.

We are going to incorporate the fields "critics" and "users" which are respectively the number of reviews and the number of users who have made the reviews.

We ask:

Exercise 1:

Clean the data, convert it to float64 and normalize the data with ZScore of the input fields (score, critics and users), leaving the **user score** scaled 0 to 10 as it appears in the dataset.

To do this you must implement the **zscore_normalize_features** function found in the utils.py file.

```
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
    """
```

```
return X_norm, mu, sigma
```

Draw using mathplotlib the distribution of the data for each variable, with respect to **user score**.

You can use Jupiter Notebook for the visualization (you give it to me as part of the practice) but you must put the cleaning in the function **cleanData** of the file **utils.py** to be able to take advantage of it later in the subsequent exercises.

Exercise 2:

Implement the class **class LinearRegMulti:** from the file **LinearRegression-Multi.py** in its vector version only. It must inherit from the vector version of LinearReg that you have implemented. The functions to implement are the same, but try to make the most of the code already implemented in the one-variable version. You can modify the base LinearReg class, but you must make sure that the tests of practice 1 are still passed. To do so, you will have to make a correct use of object-oriented programming.

To multiply matrices, you can use the @ operator, this operator allows to multiply two matrices or a matrix and a vector as long as they are compatible. That is to say that (MxN) @ (N,K) = (M,K). In other words, the columns of the matrix to the left of @ must be equal with the rows of the matrix or vector to the right of the @ operator.

Exercise 3

Add to the constructor an additional field named lambda (lambda_ to avoid using the reserved word lambda) which is the regularization parameter. Implement the L2 regularization in two separate methods named _regularizationL2Cost and _regularizationL2Gradient and add it to the cost and gradient calculation.

```
"""
Compute the regularization cost (is private method: start with _ )
This method will be reuse in the future.

Returns
    _regularizationL2Cost (float): the regularization value of the current model
"""

def _regularizationL2Cost(self):
    return reg_cost_final
```

Remember the L2 regularization is calculate as:

$$L2 = \frac{\lambda}{2m} \cdot \sum_{j=1}^{n} (w_j^2)$$

```
    """
    Compute the regularization gradient (is private method: start with _ )
    This method will be reuse in the future.

    Returns
        _regularizationL2Gradient (vector size n): the regularization gradient of the curre
    """
    def _regularizationL2Gradient(self):
        return reg_gradient_final
```

Remember the L2 gradkient regularization is calculate as:

$$\frac{\partial L2}{\partial w_j} = \frac{\lambda}{m} \cdot w_j$$

Ejercicio 4

Run the following test in file practica02.py:

- test_cost(x_train, y_train)
- test_gradient(x_train, y_train)
- test_gradient_descent(x_train, y_train)

And check that they all pass correctly