Practica01: Regresión Lineal

Somos los científicos de datos de una empresa que se dedica a asesorar a estudios indies para conseguir buenas valoraciones en sus juegos. Para disponer de datos se ha realizado un estudio de mercado sobre el rating y el user score de videojuegos ya publicados. Los datos han sido almacenados en formato CSV en el fichero "data/games-data.csv".

Debemos crear un modelo de regresión lineal que nos permita predecir el user score de un juego en base al score obtenido por la prensa especializada. Nuestra hipótesis es que como las reviews de la prensa especializad **score** se conoce previamente, esta influye de alguna forma en la percepción y la valoración que los usuarios dan a los juegos.

Se pide:

Primera parte: Regresión Lineal

Ejericio 1:

Limpiad los datos, convertidlos a float y poned en la misma escala los datos de **score** y **user score**. La entrada será **score** y la variable a predecir **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 LinearReg:** del fichero **LinearRegression.py** en su versión vectorial (Es decir, usando numpy)

Ejecutad los test de prueba que vienen en **public_test.py** y en el fichero **practica01.py** y comprobado que funcionan correctamente. Implementad todo lo necesario en **practica01.py** La clase LinearReg consta de los siguientes métodos:

Constructor:

```
    """
    Computes the cost function for linear regression.

    Args:
        x (ndarray): Shape (m,) Input to the model
        y (ndarray): Shape (m,) the real values of the prediction
        w, b (scalar): Parameters of the model
    """
    def __init__(self, x, y,w,b):
```

```
        #(scalar): Parameters of the model
        return #delete this return
```

función de regresión lineal f_w_b:

$$Y = w * x + b => f(w,b) = w * x + b$$

```
    """
    Computes the linear regression function.

    Args:
        x (ndarray): Shape (m,) Input to the model

    Returns:
        the linear regression value
    """

    def f_w_b(self, x):
        return self.w * x + self.b
```

función de coste compute_cost:

$$MSE(Y,Y') = \frac{1}{2m} \cdot \sum_{i=1}^{m} (y_i - y_i')^2$$

```
    """
    Computes the cost function for linear regression.

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear regression
               to fit the data points in x and y
    """
    def compute_cost(self):
        total_cost = 0


        return total_cost
```

función de gradiente compute_gradient:

$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m} \sum_{i=1}^{m} (y_i\prime - y_i) * x_i$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (y_i\prime - y_i)$$

```
    """
    Computes the gradient for linear regression
    Args:

    Returns
      dj_dw (scalar): 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
     """
    def compute_gradient(self):
        dj_dw = 0
        dj_db = 0
        return dj_dw, dj_db
```

función de gradiente descendente gradient_descent:

Repetir hasta el número de iteraciones (Epocs).

$$w = w - \alpha \frac{\partial J(w,b)}{\partial w}$$

$$b = b - \alpha \frac{\partial J(w,b)}{\partial b}$$

```
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run gradient descent
    Returns
      w : (ndarray): Shape (1,) 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
      w_initial : (ndarray): Shape (1,) initial w value before running gradient descent
      b_initial : (scalar) initial b value before running gradient descent
    """
    def gradient_descent(self, alpha, num_iters):
        # An array to store cost J and w's at each iteration - primarily for graphing later
        J_history = []
        w_history = []
        w_initial = copy.deepcopy(self.w)  # avoid modifying global w within function
```

```
    b_initial = copy.deepcopy(self.b)  # avoid modifying global b within function

    return self.w, self.b, J_history, w_initial, b_initial
```

Ejercicio 3 (opcional) :

Un punto extra a tu hucha de puntos adicionales. Implementar una versión iterativa y medir ambos modelos para ver cual es el más eficiente, si la vectorial o la iterativa.

Segunda Parte: regresión multivariable.

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 número de usuarios que han realziado las críticas.

Se pide:

Ejericio 1:

el método está pero cuándo lo llamo

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
```

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 primera parte siguen pasando con las modificaciones realizadas. **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 para el calculo del coste se se calcula de la siguiente forma:

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

```
"""
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 de la siguiente forma:

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

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 are data scientists at a company that advises indie studios on how to get good ratings for their games. To obtain data, we conducted market research on the ratings and user scores of already released video games. The data has been stored in CSV format in the file 'data/games-data.csv'.

We must create a linear regression model that allows us to predict the user score of a game based on the score obtained by the specialised press. Our hypothesis is that, as the reviews of the specialised press **score** are known in advance, this influences in some way the perception and rating that users give to the games.

Required:

Part One: Linear Regression

Clean the data, convert it to float, and put the **score** and **user score** data on the same scale. The input will be **score** and the variable to predict will be **user score**.

You can use Jupiter Notebook for visualisation (submit it as part of the exercise), but you must put the cleaning in the **cleanData** function of the **utils.py** file so that you can use it in later exercises.

Exercise 2:

Implement the **class LinearReg:** class from the **LinearRegression.py** file in its vector version (i.e., using numpy).

Run the test cases in **public_test.py** and in the **practica01.py** file and check that they work correctly. Implement everything necessary in **practica01.py** The LinearReg class consists of the following methods:

Constructor:

```
"""
Computes the cost function for linear regression.

Args:
    x (ndarray): Shape (m,) Input to the model
    y (ndarray): Shape (m,) the real values of the prediction
    w, b (scalar): Parameters of the model
"""
def __init__(self, x, y,w,b):
    #(scalar): Parameters of the model
    return #delete this return
```

Linear Regression function f_w_b:

$$Y = w * x + b => f(w,b) = w * x + b$$

```
"""
Computes the linear regression function.

Args:
    x (ndarray): Shape (m,) Input to the model

Returns:
    the linear regression value
"""

def f_w_b(self, x):
    return self.w * x + self.b
```

Cost function compute_cost:

$$MSE(Y,Y') = \frac{1}{2m} \cdot \sum_{i=1}^{m} (y_i - y_i')^2$$

```
"""
Computes the cost function for linear regression.

Returns
    total_cost (float): The cost of using w,b as the parameters for linear regression
```

```
            to fit the data points in x and y
    """
    def compute_cost(self):
        total_cost = 0


        return total_cost
```

Gradient function compute_gradient:

$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(y_i\prime - y_i) * x_i$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(y_i\prime - y_i)$$

```
    """
    Computes the gradient for linear regression
    Args:

    Returns
      dj_dw (scalar): 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
     """
    def compute_gradient(self):
        dj_dw = 0
        dj_db = 0
        return dj_dw, dj_db
```

Gradient descent function gradient_descent:

Repeat until number of iterations (Epocs)

$$w = w - \alpha\frac{\partial J(w,b)}{\partial w}$$

$$b = b - \alpha\frac{\partial J(w,b)}{\partial b}$$

```
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      alpha : (float) Learning rate
```

```
    num_iters : (int) number of iterations to run gradient descent
  Returns
    w : (ndarray): Shape (1,) 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
    w_initial : (ndarray): Shape (1,) initial w value before running gradient descent
    b_initial : (scalar) initial b value before running gradient descent
  """
  def gradient_descent(self, alpha, num_iters):
      # An array to store cost J and w's at each iteration - primarily for graphing later
      J_history = []
      w_history = []
      w_initial = copy.deepcopy(self.w)  # avoid modifying global w within function
      b_initial = copy.deepcopy(self.b)  # avoid modifying global b within function

      return self.w, self.b, J_history, w_initial, b_initial
```

Exercise 3 (optional):

An extra point for your bonus points piggy bank. Implement an iterative version and measure both models to see which is more efficient, the vectorial or the iterative one.

Part Two: Multivariate regression.

We continue with our market analysis. After the previous study, we thought that there might be 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 written reviews.

Required:

Exercise 1:

Clean the data, convert it to float64, and normalise the data with **ZScore** from the input fields (score, critics, and users), leaving the **user score** on a scale of 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
```

Exercise 2:

Implement the **class LinearRegMulti:** class from the **LinearRegression-Multi.py** file in its vector version only. It must inherit from the vector version of LinearReg that you have implemented. The functions to be implemented are the same, but **you must try to make the most of the code already implemented in the single-variable version**. You can modify the base LinearReg class, but you must ensure that the tests from the first part still pass with the modifications made. **To do this, you will need to make proper use of object-oriented programming**.

To multiply matrices, you can use the @ operator. This operator allows you to multiply two matrices or a matrix and a vector, provided they are compatible. That is, (MxN) @ (N,K) = (M,K) must be true. In other words, the columns of the matrix on the left of @ must match the rows of the matrix or vector on the right of the @ operator.

Exercise 3

Add an additional field called lambda (lambda_ to avoid using the reserved word lambda) to the constructor, which is the regularisation parameter. Implement L2 regularisation in two separate methods called _regularisationL2Cost and _regularisationL2Gradient 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 that the L2 adjustment for calculating the cost is calculated as follows:

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

```
    """
    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 that the L2 regularisation gradient is calculated as follows:

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

Run the tests in the practica02.py file:

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

And check that they pass correctly.