Practica01: Regresión Lineal Parte I

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:

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**

Dibujad usando mathplotlib la distribución de los datos para ver si hay alguna correlación entre ambas variables.

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 iterativa que 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:

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

```
"""
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 parámetros.

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

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

```
    """
    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:

Implementar una versión de la misma clase, pero esta vez en forma vectorial usando las capacidades de numpy. Importante consultar la función **np.sum()** en la documentación de numpy que probablemente necesitéis en la realización de este ejercicio.

Ejercicio 4:

Ejecutad los test de prueba que vienen en **public_test.py** y en el fichero **practica01.py** y comprobado que ambas versiones (iterativa y vectorial) funcionan correctamente. Implementad todo lo necesario en **practica01.py**

Ejercicio 5:

Medir los tiempos de ambos modelos y comprobad cuál es más eficiente.

English verison

We are the data scientists of a company that advises indie studios to get good ratings for their games. In order to have data available, a market study has been carried out on the rating and user score of already published videogames. The data has been stored in CSV format in the file "data/games-data.csv".

We have to create a linear regression model that allows us to predict the user score of a game based on the score obtained by the specialized press. Our hypothesis is that as the reviews of the specialized press **score** is previously known, it influences in some way in the perception and the valuation of the users.

We ask:

Exercise 1:

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

Draw using mathplotlib the distribution of the data to see if exist correlation between both variables.

You can use Jupiter Notebook for the visualization (you give it to me as part of the practice) but you must put the cleanup code in the **cleanData** function (is a function of file **utils.py**) to be able to use it in the rest of the practice.

Exercise 2:

Implement the class **class LinearReg:** from the file **LinearRegression.py** in iterative mode (using a loop). This class have 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 model 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:

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

```
"""
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 interations.

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

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

```
"""
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:

Implement a version of the same class, but this time in vector form using numpy's capabilities. Important to consult the **np.sum()** function in the numpy documentation that you will probably need to solve this exercise.

Exercise 4:

Run the test writed in **public_test.py** and in **practice01.py** and checked that both versions (iterative and vectorial) work correctly. Implement everything needed in **practice01.py**.

Exercise 5:

Measure the times of both models and check which one is more efficient.