Entrega 1: regresión lineal de una variable

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 1 y el proceso de la regresión lineal

Para esta práctica se usarán los siguientes *imports* vistos en la figura 0.1.

```
import numpy as np
import public_tests
import utils
import matplotlib.pyplot as plt
from matplotlib import cm
import csv
```

Figura 0.1: Código de las bibliotecas usadas

1. Regresión lineal

Para esta regresión definimos la función de pendiente como en la figura 1.1 como métrica, también se usan las funciones de coste 1.2 y de gradiente 1.3 según descritas en los apuntes.

Para hacer el gradiente descendente usamos 1.4 con 1500 iteraciones, 0.01 en α y 0 para iniciar w y b.

```
def fun(x: float, w: float, b: float) -> float:
    """Slope function

    Args:
        x (float): x coordinate
        w (float): slope
        b (float): starting point

    Returns:
        float: predicted y
    """
    return (w*x)+b
```

Figura 1.1: Código de la función 'fun'

```
def compute_cost(x: np.ndarray, y: np.ndarray, w: float, b: float) -> float:
    """
    Computes the cost function for linear regression.

    Args:
        x (ndarray): Shape (m,) Input to the model (Population of cities)
        y (ndarray): Shape (m,) Label (Actual profits for the cities)
        w, b (scalar): Parameters of the model

    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
    """
    m: int = x.size
    x_fun: np.ndarray = np.vectorize(fun, otypes=[dtype])(x, w, b)
    total_cost = (1.0/(2 * m)) * \
        np.sum((np.subtract(x_fun, y)) ** 2)

    return total_cost
```

Figura 1.2: Código de la función 'compute_cost'

```python
def compute_gradient(x: np.ndarray, y: np.ndarray, w: float, b: float) -> list[float, float]:
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray): Shape (m,) Input to the model (Population of cities)
      y (ndarray): Shape (m,) Label (Actual profits for the cities)
      w, b (scalar): Parameters of the model
    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
     """

    v_func = np.vectorize(fun, otypes=[dtype])
    x_fun = v_func(x, w, b)
    dj_dw = (1/x.size) * np.sum((x_fun - y)*x)
    dj_db = (1/x.size) * np.sum(x_fun - y)

    return dj_dw, dj_db
```

Figura 1.3: Código de la función 'compute_gradient'

```python
def gradient_descent(x: np.ndarray, y: np.ndarray, w_in: float, b_in: float, cost_function:
    float, gradient_function: float, alpha: float = 0.01, num_iters: int = 1500) -> list[list[
    float], list[float], list[float]]:
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      x :    (ndarray): Shape (m,)
      y :    (ndarray): Shape (m,)
      w_in, b_in : (scalar) Initial values of parameters of the model
      cost_function: function to compute cost
      gradient_function: function to compute the gradient
      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
    """
    m = x.size
    w = [w_in]
    b = [b_in]
    J_history = [cost_function(x, y, w[-1], b)]
    for i in range(1, num_iters):
        (dj_dw, dj_db) = gradient_function(x, y, w[-1], b[-1])
        total_cost = cost_function(x, y, w[-1], b[-1])

        w.append(w[-1] - (alpha * dj_dw))
        b.append(b[-1] - (alpha * dj_db))

        J_history.append(total_cost)

    return w, b, J_history
```

Figura 1.4: Código de la función 'gradient_descent'

2. Gráficas

Para hacer los gráficos primero tendremos que hacer un *grid* con la función 'make_grid' (figura 2.1). Tras esto podemos hacer el contorno (2.2) visto en la figura 2.5 y el valle en 3D con 'show_mesh' (figure 2.3) visto en la figura 2.6.

Para ver la evolución de J(w,b) tenemos el gráfico 2.7, pero como en los últimos pasos evoluciona lentamente

podemos ver la figura 2.8 con otra escala, hechos con la función 'show_J_history' 2.4.

```python
def make_grid(w_range: list[float], b_range: list[float], X: np.ndarray, Y: np.ndarray, step: float = 0.1) -> list[np.ndarray]:
    """Makes a grid for future graphs using the given data

    Args:
        w_range (list[float]): range of the w value
        b_range (list[float]): range of the b value
        X (np.ndarray): X values of the given data
        Y (np.ndarray): Y values of the given data
        step (float, optional): steps for each range. Defaults to 0.1.

    Returns:
        list[np.ndarray]: W, B, Cost as 2D arrays
    """
    W = np.arange(w_range[0], w_range[1], step)
    B = np.arange(b_range[0],  b_range[1], step)
    W, B = np.meshgrid(W, B)
    cost = np.empty_like(W)

    for ix, iy in np.ndindex(W.shape):
        cost[ix, iy] = compute_cost(X, Y, W[ix, iy], B[ix, iy])

    return [W, B, cost]
```

Figura 2.1: Código de la función 'make_grid'

```python
def show_contour(data: np.ndarray, wb: list[float], x: np.ndarray, y: np.ndarray) -> None:
    """Saves the contour graph of the program

    Args:
        data (np.ndarray): W, B, cost
        wb (list[float]): final w and b values
        x (np.ndarray): X values given
        y (np.ndarray): Y values given
    """
    plt.clf()
    plt.contour(data[0], data[1], data[2], np.logspace(-2, 3, 20))
    plt.plot(wb[0], wb[1], 'rx')
    annotate_cost = compute_cost(x, y, wb[0], wb[1])
    plt.annotate(f'{annotate_cost}', (wb[0] + 0.2, wb[1] + 0.2), bbox=dict(
        facecolor='white', edgecolor='black', boxstyle='round,pad=0.5'))
    plt.xlabel('w')
    plt.ylabel('b')
    plt.savefig("./memoria/imagenes/contour_plot.png", dpi=300)
    plt.clf()
```

Figura 2.2: Código de la función 'show_contour'

```python
def show_contour(data: np.ndarray, wb: list[float], x: np.ndarray, y: np.ndarray) -> None:
    """Saves the contour graph of the program

    Args:
        data (np.ndarray): W, B, cost
        wb (list[float]): final w and b values
        x (np.ndarray): X values given
        y (np.ndarray): Y values given
    """
    plt.clf()
    plt.contour(data[0], data[1], data[2], np.logspace(-2, 3, 20))
    plt.plot(wb[0], wb[1], 'rx')
    annotate_cost = compute_cost(x, y, wb[0], wb[1])
    plt.annotate(f'{annotate_cost}', (wb[0] + 0.2, wb[1] + 0.2), bbox=dict(
        facecolor='white', edgecolor='black', boxstyle='round,pad=0.5'))
    plt.xlabel('w')
    plt.ylabel('b')
    plt.savefig("./memoria/imagenes/contour_plot.png", dpi=300)
    plt.clf()
```

Figura 2.3: Código de la función 'show_mesh'

```python
def show_J_history(J_history: np.ndarray) -> None:
    """Show J_history data in two graphs

    Args:
        J_history (ndarray): J history
    """
    plt.clf()
    plt.plot(range(0, len(J_history)), J_history)

    plt.xlabel('Num. Iteraciones')
    plt.ylabel('J(w, b)')
    plt.xscale('log')

    plt.savefig('./memoria/imagenes/J_history_log.png')
    plt.xscale('linear')
    plt.savefig('./memoria/imagenes/J_history_linear.png')
    plt.clf()
```

Figura 2.4: Código de la función 'show_J_history'

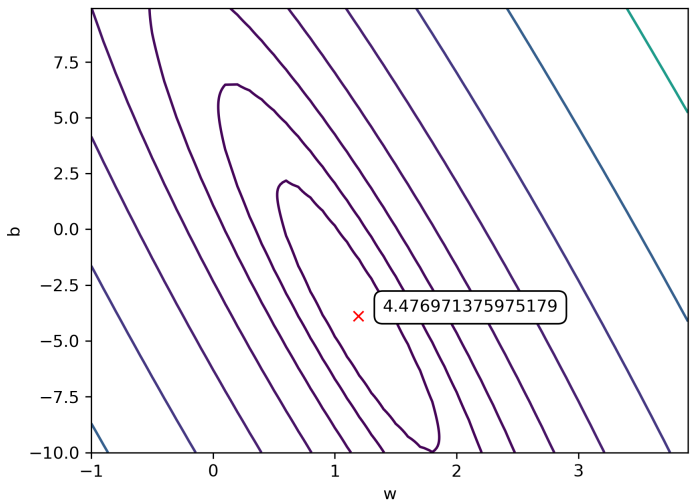

Figura 2.5: Gráfico de contorno

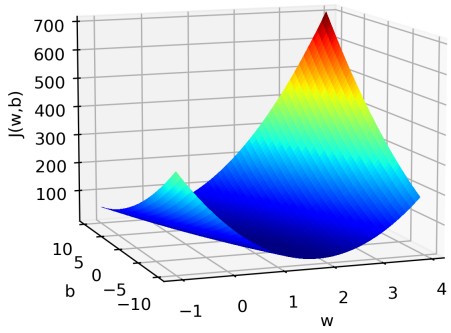

Figura 2.6: Gráfico de Valle

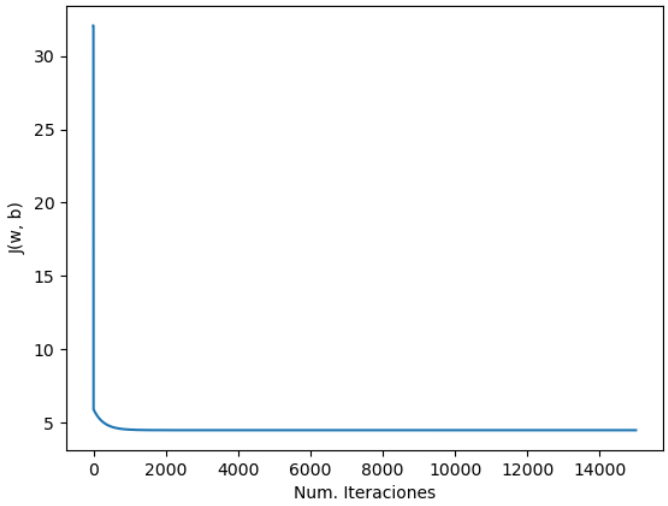

Figura 2.7: Gráfico de J_history linear

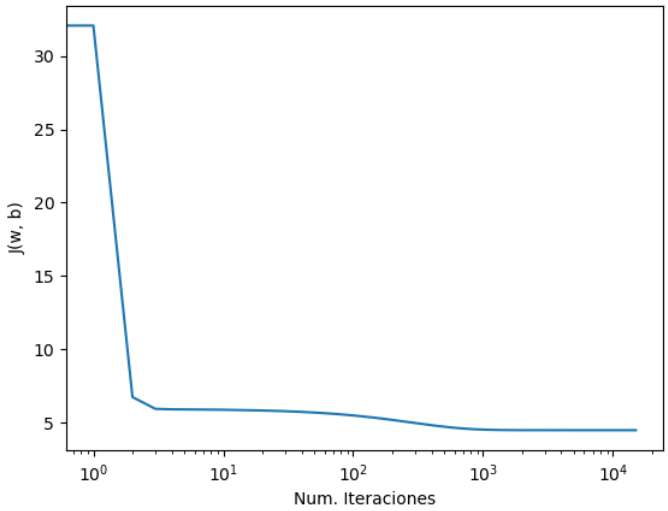

Figura 2.8: Gráfico de J_history log

3. conclusiones

Tras todas las iteraciones de 'gradient_descent', nos quedamos con el gráfico 3.1 producido por la función 3.2. Podemos ver la línea azul representando la predicción de precios y los puntos rojos representando los datos dados.

Para visualizar los datos de manera numérica usamos la función 3.3, la cual nos produce un historial simplificado de J(w,b) (tabla 3.2) y las predicciones para 35K personas y 70K personas (tabla 3.1).

Tam Población	**Prediccion**
350	37.860396668306244
700	79.6165742149173

Cuadro 3.1: Predicciones para 35K personas y 70K personas

Iteraciones	**J(w,b)**
1000	32.072733877455676
2000	4.516096429262984
3000	4.478031428126646
4000	4.477000096972836
5000	4.476972154140389
6000	4.476971397058746
7000	4.476971376546415
8000	4.476971375990655
9000	4.476971375975597
10000	4.476971375975189
11000	4.476971375975179
12000	4.476971375975179
13000	4.476971375975178
14000	4.476971375975178
15000	4.476971375975178

Cuadro 3.2: Evolución de J(w,b)

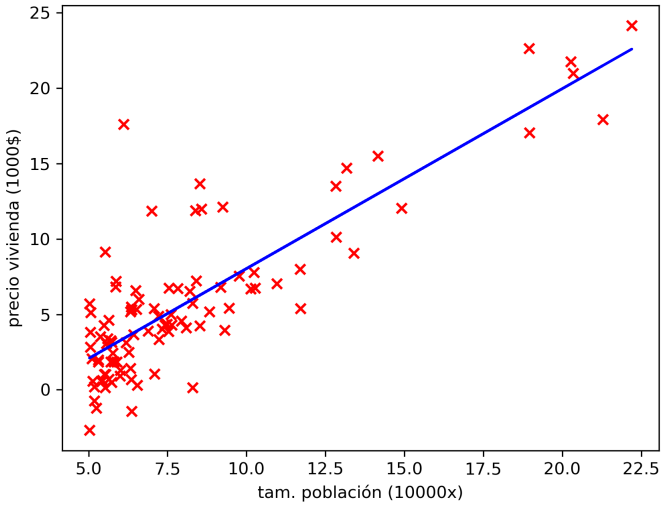

Figura 3.1: Gráfico de la predicción (azul) y los datos dados (rojo)

```python
def show_scatter_line(x: np.ndarray, y: np.ndarray, wb: list[float]) -> None:
    """Scatters the given data and plot the prediction line

    Args:
        x (np.ndarray): X given data
        y (np.ndarray): Y given data
        wb (list[float]): final values of w and b
    """
    plt.clf()
    plt.scatter(x, y, color='red', marker='x', label='observados')
    f_vec = np.vectorize(fun)
    y_vec = f_vec(x, wb[0], wb[1])

    # plt.scatter(35, fun(35, wb[0], wb[1]), marker='o', color='green')
    # plt.scatter(70, fun(70, wb[0], wb[1]), marker='o', color='green')

    plt.plot(x, y_vec, color='blue', label='prediccion')

    plt.xlabel('tam. población (10000x)')
    plt.ylabel('precio vivienda (1000$)')

    plt.savefig('./memoria/imagenes/scatter.png', dpi=300)
    plt.clf()
```

Figura 3.2: Código de la función 'show_scatter_line'

```python
def write_results(J_history: np.ndarray, w: np.ndarray, b: np.ndarray) -> None:
    """Writes simplified results to csv for data visualization

    Args:
        J_history (np.ndarray): Cost history of the linear regression
        w (np.ndarray): w history
        b (np.ndarray): b history
    """
    with open('./memoria/recursos/J_history_simplificado.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['Iteracion', 'J'])
        slicedHistory = J_history[0:-1:1000]
        for i in range(1, len(slicedHistory) + 1):
            writer.writerow([i*1000, slicedHistory[i - 1]])

    with open('./memoria/recursos/predicciones.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['Tam_poblacion', 'coste_predecido'])
        writer.writerow([350, fun(35, w[-1], b[-1])])
        writer.writerow([700, fun(70, w[-1], b[-1])])
```

Figura 3.3: Código de la función 'write_results'