Práctica 1: Regresión Linear

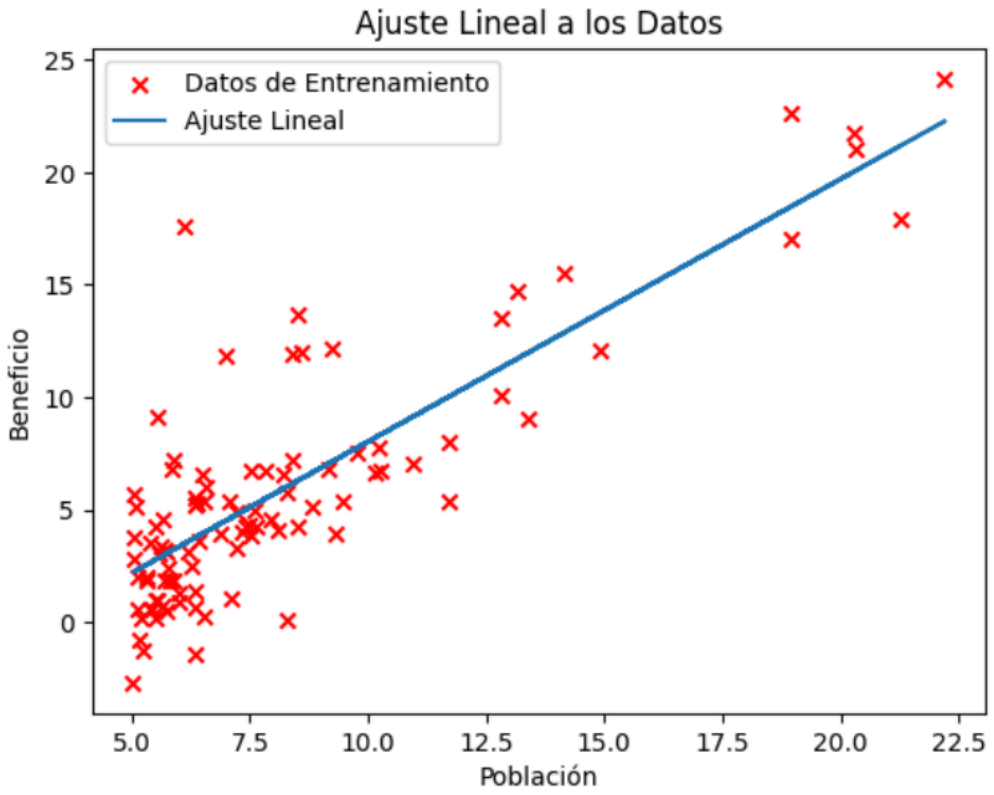
Esther Babon Arcauz

Primero he implementado las funciones de coste y gradiente siguiendo las fórmulas especificadas en la descripción de la práctica. Comprobando mediante los test que estaban implementados correctamente.

La función de descenso de gradiente recibe como entrada el conjunto de datos x y y, los valores iniciales de los parámetros w y b, la función de coste, la función de gradiente, la tasa de aprendizaje, y el número de iteraciones.Itera num_iters veces y en cada iteración calcula los gradientes usando la función de gradiente, actualiza los parámetros w y b utilizando la tasa de aprendizaje y los gradientes, y registra el valor del costo en cada iteración.

En resumen, este código implementa el algoritmo de descenso de gradiente para encontrar los parámetros w y b que mejor ajustan una regresión lineal a un conjunto de datos dado, y luego utiliza estos parámetros para hacer predicciones y visualizar el ajuste lineal.

Como comenta el enunciado de la práctica, podemos usar los parámetros finales para dibujar el ajuste lineal y para predecir el beneficio de áreas de 35.000 y 70.000 personas.




```
Beneficios predichos para 35,000 y 70,000 personas: [37.19239082 78.01507308]
```

```
##############################################################################
# Cost function
#
def compute_cost(x, y, w, b):
    """
    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
    """

    total_cost = 0
    for i in range(len(x)):
        # Compute predicted value
        predicted = x[i] * w  + b
        # Compute squared difference and add to total cost
        total_cost += (predicted - y[i]) ** 2

    # Divide by 2*m to get the average cost
    total_cost /= (2 * len(x))

    return total_cost


##############################################################################
# Gradient function
#
def compute_gradient(x, y, w, b):
    """
    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
     """

    dj_dw = 0
    dj_db = 0

    for i in range(len(x)):
        # Compute predicted value
        predicted = x[i] * w  + b
        # Update gradients
        dj_dw += (predicted - y[i]) * x[i]
        dj_db += (predicted - y[i])

    # Divide by m to get average gradient
    dj_dw /= len(x)
    dj_db /= len(x)

    return dj_dw, dj_db


##########################################################################
# gradient descent
#
def gradient_descent(x, y, w_in, b_in, cost_function, gradient_function,
alpha, num_iters):
    """
    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
    """

    J_history = np.zeros(num_iters)

    for i in range(num_iters):
        dj_dw, dj_db = gradient_function(x, y, w_in, b_in)
        w_in -= alpha * dj_dw
        b_in -= alpha * dj_db
        J_history[i] = cost_function(x, y, w_in, b_in)

    return w_in, b_in, J_history


##########################################################################
# plot & predictions
#
# Datos de entrenamiento
x,y = load_data()

# Parámetros iniciales y configuración del descenso de gradiente
w_in = 0
b_in = 0
alpha = 0.01
num_iters = 1500

# Ejecutar descenso de gradiente
w, b, J_history = gradient_descent(x, y, w_in, b_in, compute_cost,
compute_gradient, alpha, num_iters)

# Graficar el ajuste lineal
plt.scatter(x, y, color='red', marker='x', label='Datos de Entrenamiento')
plt.plot(x, w*x + b, label='Ajuste Lineal')
plt.xlabel('Población')
plt.ylabel('Beneficio')
plt.title('Ajuste Lineal a los Datos')
plt.legend()
```

```
plt.show()

# Predicciones para poblaciones de 35.000 y 70.000
x_pred = np.array([3.5, 7]) * 10
predictions = w * x_pred + b
print("Beneficios predichos para 35,000 y 70,000 personas:", predictions)
```