Entrega 3: regresión lógica

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 3 y el proceso de la regresión lógica. Esta práctica se divide en 2 apartados: regresión lógica y regresión lógica regularizada. Para ambos apartados se usan *datasets*, el primero siendo un listado de notas de dos exámenes distintos de estudiantes y el otro resultado de tests de QA hechos a distintos chips para comprobar si son válidos o no.

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

```
1 import numpy as np
2 import utils
3 import matplotlib.pyplot as plt
4 import public_tests
5 import copy
```

Figura 0.1: Código de las bibliotecas usadas

También se usarán los siguientes strings estáticos durante toda la ejecución del programa (figura 0.2).

```
1 plot_folder: str = './memoria/imagenes/'
2 csv_folder: str = './memoria/csv/'
3 ex_folder: str = './data/'
```

Figura 0.2: Código de los strings estáticos

1. Parte A: Regresión lógica

En esta parte haremos una regresión lógica simple sobre el primer dataset para predecir si un estudiante va a pasar la asignatura en base a sus dos primeras notas (distribución visible en la figura 1.1). Para ello usaremos la función $f_{w,b}(x^{(i)} = g(w * x^{(i)} + b))$ (implementación en la figura 1.3), siendo g el sigmoide implementado en la función *sigmoid* (figura 1.4).

Para seguir con el aprendizaje del modelo tendremos que implementar las funciones de coste y de gradiente, que se implementan en las figuras 1.5 y 1.7 respectivamente. La función de coste se expresa como

$$J(w,b) = \frac{1}{m}\sum_{i=0}^{m-1}[loss(f_{w,b}(x^{(i)}), y^{(i)})]$$

siendo *loss* la función de pérdida que se implementa en la figura 1.6 y cuya fórmula matemática es

$$loss(f_{w,b}(x^i), y^i) = (-y^{(i)} log(f_{w,b}(x^i)) - (1 - y^i) log(1 - f_{w,b}(x^i)))$$

La función de gradiente tiene dos partes, una para w y otra para b, cuya fórmula matemática se representa como

$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} * (f_{w,b}(x^{(i)}) - y^{(i)})$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

respectivamente.

Para estas funciones tenemos la opción de pasarle un argumento *lambda* que por ahora solo estará ahí como placeholder para no interferir con las funciones del segundo apartado.

Para terminar esta parte. se implemente un descenso de gradiente (implementado en la figura 1.8) por número de iteraciones donde w y b se actualizan en cada iteración según la fórmula

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

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

siendo α la tasa de aprendizaje. Siguiendo los valores del ejemplo dado en el enunciado ($b = -8$, $\vec{w} = 0$ y $\alpha = 0{,}001$) se obtiene que $\vec{w} = [0{,}07125349, 0{,}06482881]$, y $b = -8{,}188614567810179$ con $J(w,b) = 0{,}3018682223181452$ tras 10000 iteraciones. Podemos ver la barrera de decisión en la figura 1.2.

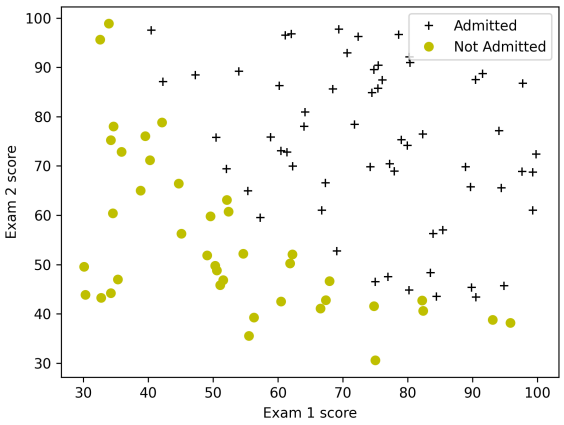

Figura 1.1: Distribución de notas de estudiantes

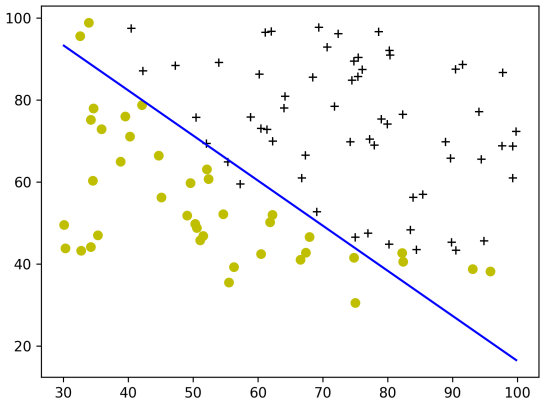

Figura 1.2: Barrera de decisión

```python
def function(x: np.ndarray, w: np.ndarray, b: float) -> np.ndarray:
    """Function using ''sigmoid'' to calculate the value of y to the given x, w and b

    Args:
        x (np.ndarray): X data
        w (np.ndarray): w data
        b (float): b data

    Returns:
        np.ndarray: final value after the sigmoid
    """
    return sigmoid(np.dot(x, w) + b)
```

Figura 1.3: Función de regresión lógica

```python
def sigmoid(z: np.ndarray) -> np.ndarray:
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    g = 1/(1+np.exp(-z))

    return g
```

Figura 1.4: Función sigmoide

```python
def compute_cost(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_=None) -> float
    :
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : scalar Values of bias parameter of the model
      lambda_: unused placeholder
    Returns:
      total_cost: (scalar)         cost
    """
    # apply the loss function for each element of the x and y arrays
    loss_v = loss(X, y, function, w, b)
    total_cost = np.sum(loss_v)
    total_cost /= X.shape[0]

    return total_cost
```

Figura 1.5: Función de coste

```python
def loss(X: np.ndarray, Y: np.ndarray, fun: np.ndarray, w: np.ndarray, b: float) -> float:
    """loss function for the logistic regression

    Args:
        X (np.ndarray): X values
        Y (np.ndarray): Expected y results
        fun (np.ndarray): logistic regression function
        w (np.ndarray): weights
        b (float): bias

    Returns:
        float: total loss of the regression
    """
    return (-Y * np.log(fun(X, w, b))) - ((1 - Y) * np.log(1 - fun(X, w, b)))
```

Figura 1.6: Función de perdida

```python
def compute_gradient(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_=None) ->
    tuple[float, np.ndarray]:
    """
    Computes the gradient for logistic regression

    Args:
        X : (ndarray Shape (m,n)) variable such as house size
        y : (array_like Shape (m,1)) actual value
        w : (array_like Shape (n,1)) values of parameters of the model
        b : (scalar)                 value of parameter of the model
        lambda_: unused placeholder
    Returns
        dj_db: (scalar)                The gradient of the cost w.r.t. the parameter b.
        dj_dw: (array_like Shape (n,1)) The gradient of the cost w.r.t. the parameters w.
    """

    func = function(X, w,b)

    dj_dw = np.dot(func -y, X)
    dj_dw /= X.shape[0]

    dj_db = np.sum(func - y)
    dj_db /= X.shape[0]

    return dj_db, dj_dw
```

Figura 1.7: Función de gradiente

```python
def gradient_descent(X: np.ndarray, y: np.ndarray, w_in: np.ndarray, b_in: float,
    cost_function: float, gradient_function: float, alpha: float, num_iters: int, lambda_:
    float = None) -> tuple[np.ndarray, float, np.ndarray, np.ndarray]:
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X :    (array_like Shape (m, n)
      y :    (array_like Shape (m,))
      w_in : (array_like Shape (n,))  Initial values of parameters of the model
      b_in : (scalar)                 Initial value of parameter of the model
      cost_function:                  function to compute cost
      alpha : (float)                 Learning rate
      num_iters : (int)               number of iterations to run gradient descent
      lambda_ (scalar, float)         regularization constant

    Returns:
      w : (array_like Shape (n,)) 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 = copy.deepcopy(w_in)
    b = b_in
    predict_history = [predict_check(X, y, w, b)]
    J_history = [cost_function(X, y, w, b, lambda_)]

    for i in range(num_iters):
        dj_db, dj_dw = gradient_function(X, y, w, b, lambda_)
        w = w - (alpha * dj_dw)
        b -= alpha * dj_db
        J_history.append(cost_function(X, y, w, b, lambda_))
        predict_history.append(predict_check(X, y, w, b))

    return w, b, np.array(J_history), predict_history
```

Figura 1.8: Descenso de gradiente

2. Parte B: Regresión lógica regularizada

En esta parte se implementa una regresión lógica regularizada para predecir si un chip va a ser valido basándonos en distintos datos de QA (distribución en la figura 2.1). Para este apartado seguiremos usando la función sigmoide implementada en la figura 1.4 y la función de regresión de la figura 1.3. La función de coste y de gradiente se implementan en las figuras 2.3 y 2.4 respectivamente. La función de coste se expresa como

$$J(w,b) = \frac{1}{m}\sum_{i=0}^{m-1}[loss(f_{w,b}(x^{(i)}), y^{(i)})] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

siendo λ el parámetro de regularización y n el número de características. La función de gradiente tiene dos partes, una para w y otra para b, cuya fórmula matemática se representa como

$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} * (f_{w,b}(x^{(i)}) - y^{(i)}) + \frac{\lambda}{m}w$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})$$

respectivamente. Para este apartado usaremos el descenso de gradiente del apartado anterior (figura 1.8) dándole uso a la variable lambda_.

Utilizando las métricas dadas en el enunciado ($b = 1$, $\vec{w} = 0$, $\lambda = 0{,}01$ y $\alpha = 0{,}01$), obtenemos $J(w,b) =$ 0,449111111581372, $b =$ 1,3575493852806007 y w:

w
0.72259293
1.37503464
-2.26623624
-0.93538886
-1.43093541
0.10544778
-0.38027447
-0.38596694
-0.21958157
-1.67262795
-0.09567878
-0.65568869
-0.27805803
-1.34114809
-0.30618779
-0.23477687
-0.07613709
-0.29039658
-0.30685795
-0.6307361
-1.21790115
-0.00397421
-0.32015246
-0.00392708
-0.35180518
-0.14316118
-1.15539835

Cuadro 2.1: Tabla de pesos

Podemos ver la barrera de decisión en la figura 2.2.

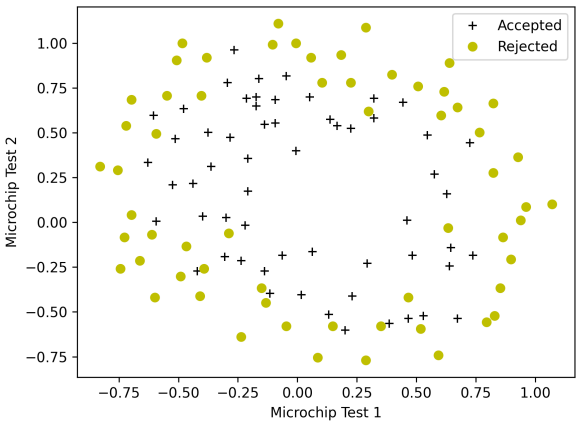

Figura 2.1: Distribución de datos de QA de chips

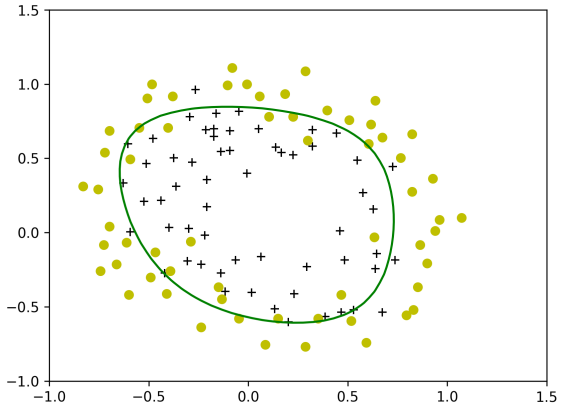

Figura 2.2: Barrera de decisión

```python
def compute_cost_reg(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_: float =
    1) -> float:
    """
    Computes the cost over all examples
    Args:
      X : (array_like Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : (array_like Shape (n,)) Values of bias parameter of the model
      lambda_ : (scalar, float)    Controls amount of regularization
    Returns:
      total_cost: (scalar)         cost
    """

    total_cost = compute_cost(X, y, w, b)
    total_cost += (lambda_ / (2 * X.shape[0])) * np.sum(w**2)

    return total_cost
```

Figura 2.3: Función de coste regularizada

```python
def compute_gradient_reg(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lambda_: float
    = 1) -> tuple[float, np.ndarray]:
    """
    Computes the gradient for linear regression

    Args:
      X : (ndarray Shape (m,n))   variable such as house size
      y : (ndarray Shape (m,))    actual value
      w : (ndarray Shape (n,))    values of parameters of the model
      b : (scalar)                value of parameter of the model
      lambda_ : (scalar,float)    regularization constant
    Returns
      dj_db: (scalar)             The gradient of the cost w.r.t. the parameter b.
      dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t. the parameters w.

    """
    dj_db, dj_dw = compute_gradient(X, y, w, b)
    dj_dw += (lambda_ / X.shape[0]) * w

    return dj_db, dj_dw
```

Figura 2.4: Función de gradiente regularizada

3. Predicciones, test de funciones y ejecución del programa

Para comprobar estos modelos, hacemos una predicción de cada modelo con la función *predict* (figura 3.5) y comprobamos su precisión con la función *predict_check* (figura 3.6). Para cargar todos los datos del dataset usamos la función *load_data* (figura 3.7), y para ejecutar ambos modelos usamos las funciones *run_student_sim* (figura 3.8) y *run_chip_sim* (figura 3.9).

Antes de ejecutar ambos modelos, se ejecutan las pruebas para ver que todo funciona usando la función *run_tests* (figura 3.10). Para visualizar mejor los datos usaremos la función *plot_ex_data* (figura 3.11) para los datos de ejemplo y *plot_linear_data* (figura 3.12) para la evolución del coste y de la predicción en cada iteración.

Tras las ejecuciones de ambas simulaciones, obtenemos un porcentaje de acierto del 92 % en el primer modelo y un 83,05084745762711 % en el segundo y podemos ver las siguientes gráficas:

- Evolución del coste (figura 3.1) y de la predicción (figura 3.2) para el primer dataset.
- Evolución del coste (figura 3.3) y de la predicción (figura 3.4) para el segundo dataset.

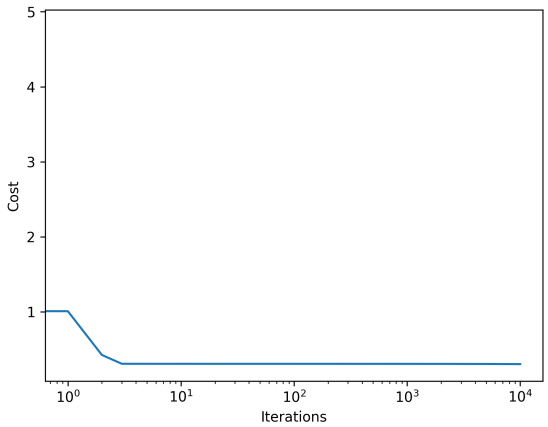

Figura 3.1: Evolución del coste para el primer dataset

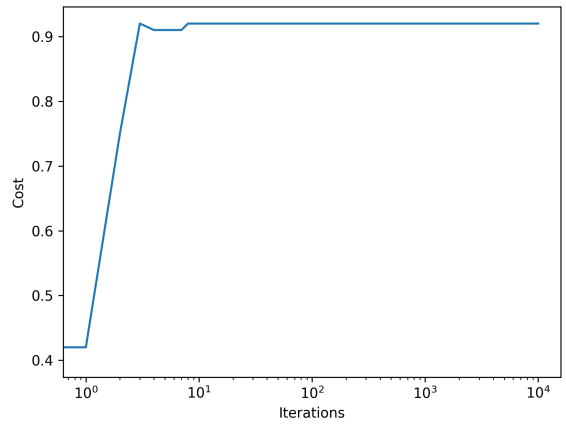

Figura 3.2: Evolución de la predicción para el primer dataset

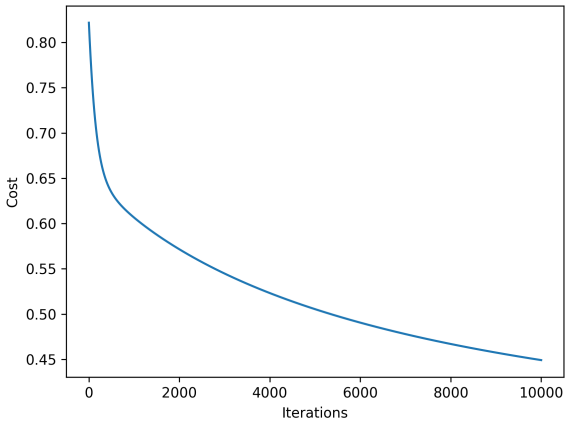

Figura 3.3: Evolución del coste para el segundo dataset

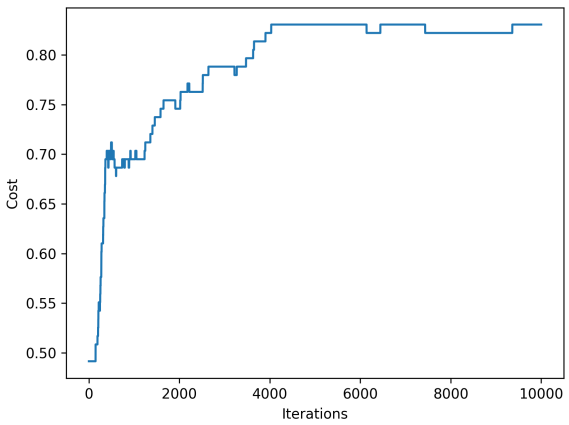

Figura 3.4: Evolución de la predicción para el segundo dataset

```python
def predict(X, w, b)->np.ndarray:
    """
    Predict whether the label is 0 or 1 using learned logistic
    regression parameters w and b

    Args:
    X : (ndarray Shape (m, n))
    w : (array_like Shape (n,))      Parameters of the model
    b : (scalar, float)              Parameter of the model

    Returns:
    p: (ndarray (m,1))
        The predictions for X using a threshold at 0.5
    """

    p = np.vectorize(lambda x: 1 if x > 0.5 else 0)(
        function(X, w, b))
    return p
```

Figura 3.5: Función de predicción

```python
def predict_check(X, Z, w, b) -> float:
    """Gives a percentage of the accuracy of the prediction

    Args:
        X (_type_): X train data
        Z (_type_): expected values
        w (_type_): weights
        b (_type_): bias

    Returns:
        float: percentage of accuracy
    """
    p = predict(X, w, b)
    return np.sum(p == Z) / Z.shape[0]
```

Figura 3.6: Función de comprobación de predicción

```python
def load_data(filename:str) -> tuple[np.ndarray, np.ndarray]:
    """Loads the train data

    Args:
        filename (str): dataset filename

    Returns:
        tuple[np.ndarray, np.ndarray]: Train data and expected values
    """
    data = np.loadtxt(filename, delimiter=',')
    X = data[:, 0:2]
    Z = data[:, 2]
    return X, Z
```

Figura 3.7: Función de carga de datos

```python
def run_student_sim() -> None:
    """Trains the model for the first dataset and plots the results
    """
    (X, Z) = load_data(f'{ex_folder}ex2data1.txt')
    w = np.zeros(X.shape[1])
    b = -8
    alpha = 0.001
    num_iters = 10000
    w, b, J_history, predict_history = gradient_descent(
        X, Z, w, b, compute_cost, compute_gradient, alpha, num_iters)
    plt.clf()
    utils.plot_decision_boundary(w, b, X, Z)
    plt.savefig(f'{plot_folder}muestreo1_sim.png', dpi=300)
    plt.clf()
    plot_linear_data(J_history, range(num_iters + 1), 'muestreo1_cost.png', 'log')
    plot_linear_data(predict_history, range(num_iters + 1), 'muestreo1_accuracy.png', 'log')
    print(f'Cost: {J_history[-1]}, w: {w}, b: {b}')
    print(f'Accuracy rate: {predict_history[-1] * 100}%')
```

Figura 3.8: Función de ejecución de modelo de estudiantes

```python
def run_chip_sim() -> None:
    """Trains the model for the second dataset and plots the results
    """
    (X, Z) = load_data(f'{ex_folder}ex2data2.txt')
    X = utils.map_feature(X1=X[:, 0], X2=X[:, 1])
    w = np.zeros(X.shape[1])
    b = 1
    alpha = 0.01
    lambda_ = 0.01
    num_iters = 10000
    w, b, J_history, predict_history = gradient_descent(
        X, Z, w, b, compute_cost_reg, compute_gradient_reg, alpha, num_iters, lambda_)
    utils.plot_decision_boundary(w, b, X, Z)
    plt.savefig(f'{plot_folder}muestreo2_sim.png', dpi=300)
    plt.clf()
    plot_linear_data(J_history, range(num_iters + 1), 'muestreo2_cost.png')
    plot_linear_data(predict_history, range(num_iters + 1), 'muestreo2_accuracy.png')
    print(f'Cost: {J_history[-1]}, w: {w}, b: {b}')
    print(f'Accuracy rate: {predict_history[-1] * 100}%')
```

Figura 3.9: Función de ejecución de modelo de chips

```python
def run_test() -> None:
    """Runs the given tests
    """
    test_methods = [
        (public_tests.sigmoid_test, sigmoid),
        (public_tests.compute_cost_test, compute_cost),
        (public_tests.compute_gradient_test, compute_gradient),
        (public_tests.predict_test, predict),
        (public_tests.compute_cost_reg_test, compute_cost_reg),
        (public_tests.compute_gradient_reg_test, compute_gradient_reg)
    ]
    for test, method in test_methods:
        print('\033[37m')
        print(f'Running test: {test.__name__}')
        test(method)
```

Figura 3.10: Función de ejecución de pruebas

```python
def plot_ex_data() -> None:
    """Plots the given datasets into a graph
    """
    (X, Z) = load_data(f'{ex_folder}ex2data1.txt')

    utils.plot_data(X, Z)
    plt.legend(['Admitted', 'Not Admitted'], loc='upper right')
    plt.xlabel('Exam 1 score')
    plt.ylabel('Exam 2 score')

    plt.savefig(f'{plot_folder}muestreo1.png', dpi=300)

    plt.clf()

    (X2, Z2) = load_data(f'{ex_folder}ex2data2.txt')

    utils.plot_data(X2, Z2)
    plt.legend(['Accepted', 'Rejected'], loc='upper right')
    plt.xlabel('Microchip Test 1')
    plt.ylabel('Microchip Test 2')
    plt.savefig(f'{plot_folder}muestreo2.png', dpi=300)
    plt.clf()
```

Figura 3.11: Función de visualización de datos de ejemplo

```python
def plot_linear_data(X: np.ndarray, Y: np.ndarray, filename: str, scale:str = 'linear') ->
    None:
    """Plots the given data into a graph

    Args:
        X (np.ndarray): X data
        Y (np.ndarray): Y data
        filename (str): file to store the graph
    """
    plt.clf()
    plt.plot(Y, X)
    plt.xscale(scale)
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.savefig(f'{plot_folder}{filename}', dpi=300)
    plt.clf()
```

Figura 3.12: Función de visualización de datos de evolución de coste y predicción