Entrega 2: regresión lineal de multivariable

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 2 y el proceso de la regresión lineal en variables múltiples.

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

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

Figura 0.1: Código de las bibliotecas usadas

1. Regresión lineal en variables múltiples

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 1000 iteraciones, 0.01 en α y 0 para iniciar w y b.

Si no hacemos una normalización de los datos, estos procesos darán pesos bastante dispares e incorrectos a los distintos campos de predicción, para ello usamos la función 'zscore_normalize_features' (figura 1.5) que normaliza los datos de la entrada dados.

```
def fun(x: np.ndarray, w: np.ndarray, b: float) -> float:
    """Returns a predicted w value for a linear regression with the given w and b values.

    Args:
        x (np.ndarray): x value for de function
        w (np.ndarray): array of w
        b (float): b value

    Returns:
        float: predicted y value
    """

    return np.dot(x, w) + b
```

Figura 1.1: Código de la función 'fun'

```python
def compute_cost(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float) -> float:
    """
    compute cost
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter
    Returns
      cost (scalar)    : cost
    """
    x_fun = fun(X, w, b)

    cost = np.sum((x_fun - y)**2)

    cost = cost / (2 * X.shape[0])

    return cost
```

Figura 1.2: Código de la función 'compute_cost'

```python
def compute_gradient(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float) -> tuple[float, np
    .ndarray]:
    """
    Computes the gradient for linear regression
    Args:
      X : (ndarray Shape (m,n)) matrix of examples
      y : (ndarray Shape (m,))  target value of each example
      w : (ndarray Shape (n,))  parameters of the model
      b : (scalar)              parameter of the model
    Returns
      dj_dw : (ndarray Shape (n,)) 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.
    """

    x_fun = fun(X, w, b)
    dj_dw = np.dot((x_fun - y), X) / X.shape[0]

    dj_db: float = np.sum(x_fun - y)

    dj_db /= X.shape[0]

    return dj_db, dj_dw
```

Figura 1.3: Código de la función 'compute_gradient'

```python
def zscore_normalize_features(X: np.ndarray) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    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
    """
    mu = np.mean(X, axis=0)
    sigma = np.std(X, axis=0)
    X_norm = (np.array(X) - mu) / sigma

    return (X_norm, mu, sigma)
```

Figura 1.5: Código de la función 'zscore_normalize_features'

```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) -> tuple[np.
    ndarray, float, list[float]]:
    """
    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)    matrix of examples
      y : (array_like Shape (m,))    target value of each example
      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
      gradient_function: function to compute the gradient
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run gradient descent
    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 = 0 + b_in
    J_history = [cost_function(X, y, w, b)]

    for i in range(0, num_iters):
        (dj_db, dj_dw) = gradient_function(X, y, w, b)

        w = w - (alpha * dj_dw)
        b = b - (alpha * dj_db)
        J_history.append(cost_function(X, y, w, b))

    return w, b, J_history
```

Figura 1.4: Código de la función 'gradient_descent'

2. Gráficas

La primera gráfica a generar es la que nos muestra los datos brutos (figura 2.5), viendo así como se relaciona cada característica con el precio de una vivienda, para ello usamos la función 'visualize_data' 2.1. Tras el entrenamiento del modelo podemos ver en la figura 2.6 la predicción de precios y los datos dados, este gráfico lo generamos con la función 'visualize_data_train' (figura 2.2).

También podemos observar la evolución de la función de coste en la figura 2.7 con la función 'visualize_J_history' (figura 2.4) y las regresiones parciales en la figura 2.8, producidas por la función 'visualize_partial_regression' (figura 2.3).

```python
def visualize_data(X_train: np.ndarray, y_train: np.ndarray, name: str) -> None:
    """Generates a scatter plot of the data

    Args:
        X_train (np.ndarray): training data
        y_train (np.ndarray): training data
        name (str): name of the plot
    """
    X_features = ['size(sqft)', 'bedrooms', 'floors', 'age']
    fig, ax = plt.subplots(1, 4, figsize=(25, 5), sharey=True)
    for i in range(len(ax)):
        ax[i].scatter(X_train[:, i], y_train)
        ax[i].set_xlabel(X_features[i])
        ax[0].set_ylabel("Price (1000's)")
    plt.savefig(f'./memoria/images/{name}.png', dpi=300)
```

Figura 2.1: Código de la función 'visualize_data'

```python
def visualize_data_train(X_train: np.ndarray, y_train: np.ndarray, prediction: np.ndarray) ->
    None:
    """Generates a scatter data with the original data and the predictions

    Args:
        X_train (np.ndarray): Train values
        y_train (np.ndarray): Train prices
        prediction (np.ndarray): Predicted prices
    """
    X_features = ['size(sqft)', 'bedrooms', 'floors', 'age']
    fig, ax = plt.subplots(1, 4, figsize=(25, 5), sharey=True)
    for i in range(len(ax)):
        ax[i].scatter(X_train[:, i], y_train)
        ax[i].scatter(X_train[:, i], prediction, color='orange')
        ax[i].set_xlabel(X_features[i])
        ax[0].set_ylabel("Price (1000's)")
    plt.savefig(f'./memoria/images/predicted_data.png', dpi=300)
```

Figura 2.2: Código de la función 'visualize_data_train'

```python
def visualize_partial_regressions(X_train: np.ndarray, y_train: np.ndarray, w: np.ndarray, b:
    float) -> None:
    """Draws partial regressions for each variable

    Args:
        X_train (np.ndarray): Train data
        y_train (np.ndarray): Train prices
        w (np.ndarray): Weights
        b (float): bias
    """
    (X_norm, _, _) = zscore_normalize_features(X_train)
    X_features = ['size(sqft)', 'bedrooms', 'floors', 'age']
    fig, ax = plt.subplots(1, 4, figsize=(25, 5), sharey=True)
    for i in range(len(ax)):
        ax[i].scatter(X_train[:, i], y_train)
        ax[i].plot(X_train[:, i], fun(
            X_norm[:, i], w[i], b), color='red')
        ax[i].set_xlabel(X_features[i])
        ax[0].set_ylabel("Price (1000's)")
    plt.savefig(f'./memoria/images/partail_regression.png', dpi=300)
```

Figura 2.3: Código de la función 'visualize_partial_regression'

```python
def visualize_J_history(J_history: np.ndarray) -> None:
    """Generates a plot with the evolution of the cost function

    Args:
        J_history (np.ndarray): Cost function over each iteration
    """
    plt.clf()
    plt.figure(figsize=(7, 5))
    plt.plot(J_history)
    plt.xlabel('Iterations')
    plt.xscale('log')
    plt.ylabel('Cost')
    plt.title('Cost function over iterations')
    plt.savefig(f'./memoria/images/cost_function.png', dpi=300)
```

Figura 2.4: Código de la función 'visualize_J_history'

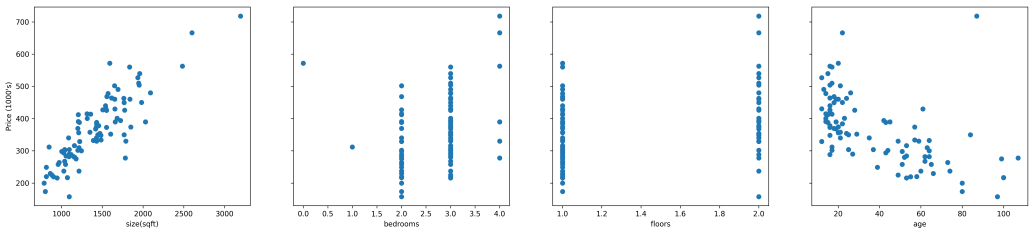

Figura 2.5: Visualización de los datos iniciales

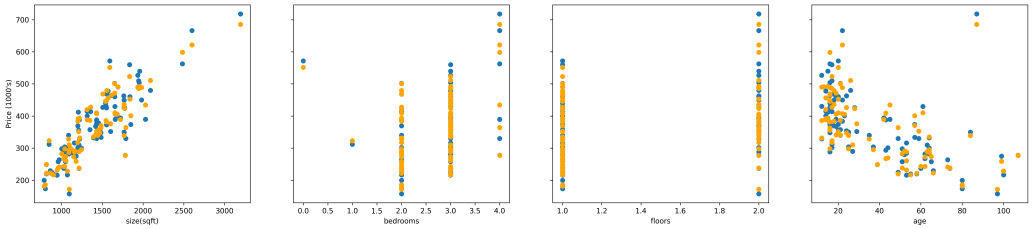

Figura 2.6: Visualización de los datos predichos

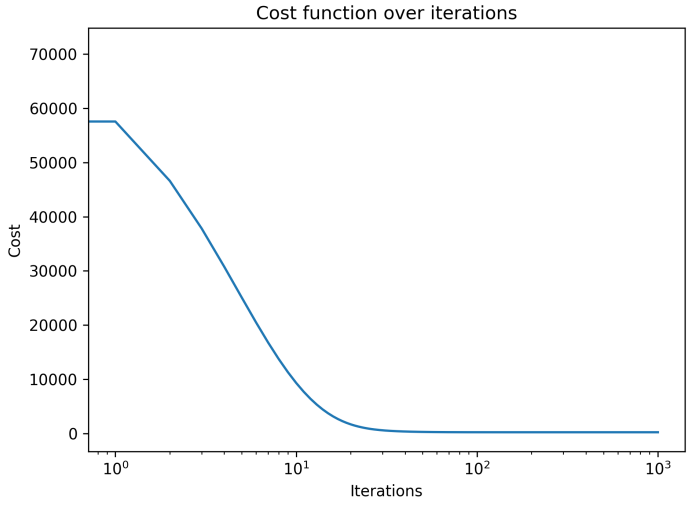

Figura 2.7: Evolución de la función de coste

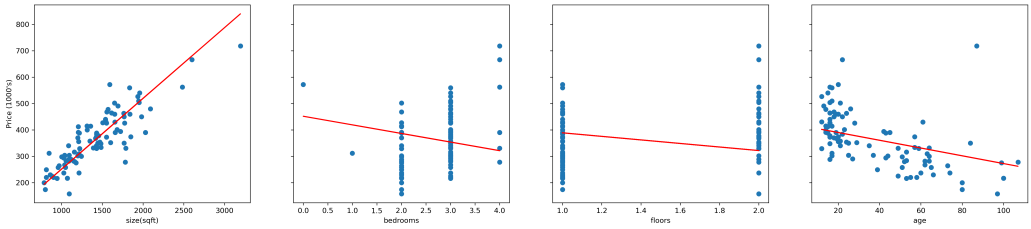

Figura 2.8: Regresiones parciales

3. conclusiones

Este modelo al tener más variables produce mejores predicciones que el modelo de la práctica anterior, ya que el anterior solo tenía en cuenta una variable arbitraria.

Para tener más presentes los resultados, usamos la función 'write_results' (figura 3.1) que nos escribe en un archivo csv los resultados de la predicción. Los resultados finales son:

Iteraciones	**J(w,b)**
100	71280.37305561663
200	221.21399549940065
300	219.20904207205783
400	219.2068268334026
500	219.20682438520015
600	219.20682438249446
700	219.20682438249148
800	219.2068243824914
900	219.2068243824914
1000	219.20682438249148

Cuadro 3.1: Evolución de J(w,b)

size(sqft)	**Bedrooms**	**floors**	**age**
1418.3737373737374	2.717171717171717	1.3838383838383839	38.38383838383838

Cuadro 3.2: Medias de cada atributo para la normalización

size(sqft)	**Bedrooms**	**floors**	**age**
411.6156289269652	0.6519652348729262	0.4863193178671001	25.777880687549874

Cuadro 3.3: Desviación estándar de cada atributo para la normalización

W. size(sqft)	**W. Bedrooms**	**W. floors**	**W. age**	**b**
110.56039755974356	-21.267150958179855	-32.707181391704225	-37.97015909100321	363.15608080808056

Cuadro 3.4: Pesos y bias finales para el modelo

```python
def write_results(J_history: np.ndarray, w: np.ndarray, b: float, mu: np.ndarray, sigma: np.ndarray) -> None:
    X_features = ['size(sqft)', 'bedrooms', 'floors', 'age']
    with open('./memoria/recursos/J_history_simplificado.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['Iteracion', 'J'])
        slicedHistory = J_history[0:-1:100]
        for i in range(1, len(slicedHistory) + 1):
            writer.writerow([i*100, slicedHistory[i - 1]])
    with open('./memoria/recursos/wb.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(X_features + ['b']),
        writer.writerow(np.concatenate((w, [b])))

    with open('./memoria/recursos/mean.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(X_features),
        writer.writerow(mu)
    with open('./memoria/recursos/sigma.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(X_features),
        writer.writerow(sigma)
```

Figura 3.1: Código de la función 'write_results'