Práctica 2

Código Completo

```
import numpy as np
import copy
import math
import public_tests as pt
import matplotlib.pyplot as plt

def zscore_normalize_features(X):
    """
    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.zeros(X.shape[1])
    sigma = np.zeros(X.shape[1])
    X_norm = np.zeros(X.shape)
    for i in range(X.shape[1]):
        mu[i] = np.mean(X[:, i])
        sigma[i] = np.std(X[:, i])
        X_norm[:, i] = (X[:, i] - mu[i]) / sigma[i]

    return (X_norm, mu, sigma)


def compute_cost(X, y, w, b):
    """
    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
    """

    m = X.shape[0]
    cost = 0
    for i in range(m):
        cost += (y[i] - np.dot(w, X[i]) - b)**2
    cost = cost / (2 * m)

    return cost
```

```python
def compute_gradient(X, y, w, b):
    """
    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.
    """

    m = X.shape[0]
    dj_dw = np.zeros(w.shape)
    dj_db = 0
    for i in range(m):
        aux = y[i] - np.dot(w, X[i]) - b
        dj_dw += aux * X[i]
        dj_db += aux
    dj_dw = -dj_dw / m
    dj_db = -dj_db / m

    return dj_db, dj_dw


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 : (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 = w_in
    b = b_in
    J_history = np.zeros(num_iters)
```

```
    for iter in range(num_iters):
        J_history[iter] = cost_function(X, y, w, b)
        dj_db, dj_dw = gradient_function(X, y, w, b)

        w -= alpha * dj_dw
        b -= alpha * dj_db

    return w, b, J_history


def load_data():
    data = np.loadtxt("./data/houses.txt", delimiter=',', skiprows=1)
    X_train = data[:, :4]
    y_train = data[:, 4]

    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)")

    return X_train, y_train

def main():
    X, y_train = load_data()
    X_norm, mu, sigma = zscore_normalize_features(X)

    b = 0
    alpha = 0.1
    num_iters = 1000
    w = np.zeros(X.shape[1])
    w, b, J_history = gradient_descent(X_norm, y_train, w, b, compute_cost,
compute_gradient, alpha, num_iters)

    print(f"w = {w}")
    print(f"b = {b}")

    price = sum(w * (([1200, 3, 1, 40] - mu) / sigma)) + b
    price = price * 1000
    print(f"Price for a 1200 sqft, 3 bedrooms, 1 floor, 40 years old house is
{price}$")


def test():
    pt.compute_cost_test(compute_cost)
    pt.compute_gradient_test(compute_gradient)


if __name__ == "__main__":
    test()
    main()
```

Output:

```
All tests passed!
All tests passed!
w = [110.56039756 -21.26715096 -32.70718139 -37.97015909]
b = 363.15608080808056
Price for a 1200 sqft, 3 bedrooms, 1 floor, 40 years old house is 318709.0923199992$
```