Código

```
import numpy as np
import copy
import math
import public_tests as pt
import utils
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 sigmoid(z):
    """
    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


#########################################################################
# logistic regression
#
def compute_cost(X, y, w, b, lambda_=None):
```

```
    """
    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
    """

    total_cost = 0
    m = len(y)
    for i in range(m):
        aux = sigmoid(np.dot(w, X[i]) + b)
        total_cost += -y[i] * math.log(aux) - (1 - y[i]) * math.log(1 - aux)
    total_cost = total_cost / m

    return total_cost


def compute_gradient(X, y, w, b, lambda_=None):
    """
    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.
    """

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

    return dj_db, dj_dw


########################################################################
# regularized logistic regression
#
def compute_cost_reg(X, y, w, b, lambda_=1):
```

```
    """
    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 = 0
    m = len(y)
    for i in range(m):
        aux = sigmoid(np.dot(w, X[i]) + b)
        total_cost += -y[i] * math.log(aux) - (1 - y[i]) * math.log(1 - aux)
    total_cost = total_cost / m
    total_cost += lambda_ / (2 * m) * np.sum(w ** 2)

    return total_cost


def compute_gradient_reg(X, y, w, b, lambda_=1):
    """
    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.

    """

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

    return dj_db, dj_dw


#########################################################################
```

```
# gradient descent
#
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha,
num_iters, lambda_=None):
    """
    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
    """

    J_history = np.zeros(num_iters)
    m = len(y)
    for i in range(num_iters):
        dj_db, dj_dw = gradient_function(X, y, w_in, b_in, lambda_)
        w_in -= alpha * dj_dw
        b_in -= alpha * dj_db
        J_history[i] = cost_function(X, y, w_in, b_in, lambda_)

    return w_in, b_in, J_history


########################################################################
# predict
#
def predict(X, w, b):
    """
    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
    """
```

```python
    m = X.shape[0]
    p = np.zeros(m)
    for i in range(m):
        p[i] = sigmoid(np.dot(w, X[i]) + b)
    p = p > 0.5

    return p


def test():
    pt.sigmoid_test(sigmoid)
    pt.compute_cost_test(compute_cost)
    pt.compute_gradient_test(compute_gradient)
    pt.predict_test(predict)

    pt.compute_cost_reg_test(compute_cost_reg)
    pt.compute_gradient_reg_test(compute_gradient_reg)


def load_data(path):
    data = np.loadtxt(path, delimiter=',')
    X = data[:, 0:2]
    y = data[:, 2]
    return X, y


def partA():
    X, y = load_data('data/ex2data1.txt')
    X_norm, mu, sigma = zscore_normalize_features(X)

    w = np.zeros(X_norm.shape[1])
    b = 0
    lambda_ = 0.01
    alpha = 0.01
    num_iters = 10000
    w, b, J_history = gradient_descent(X_norm, y, w, b, compute_cost,
compute_gradient, alpha, num_iters, lambda_)
    # w = w / sigma
    # w = w - w * mu

    utils.plot_decision_boundary(w, b, X_norm, y)
    plt.show()

    res = predict(X_norm, w, b)
    print('Predict: ', np.sum(res) * 100 / res.size, '%', sep='')


def partB():
    X, y = load_data('data/ex2data2.txt')
    X_map = utils.map_feature(X[:, 0], X[:, 1])
    b = 0
    w = np.zeros(X_map.shape[1])
    lambda_ = 0.01
    alpha = 0.01
    num_iters = 10000
```

```
    w, b, J_history = gradient_descent(X_map, y, w, b, compute_cost_reg,
compute_gradient_reg, alpha, num_iters, lambda_)
    print('cost:', J_history[J_history.size - 1])

    utils.plot_decision_boundary(w, b, X_map, y)
    plt.show()

    res = predict(X_map, w, b)
    print('Predict: ', np.sum(res) * 100 / res.size, '%', sep='')


if __name__ == "__main__":
    test()
    partA()
    partB()
```