Práctica 4

Código

```
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt


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


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 = y.shape[len(y.shape) - 1]
    dj_dw = np.zeros(w.shape)
    dj_db = np.zeros(b.shape)
    for i in range(m):
        aux = sigmoid(np.dot(w, X[i]) + b) - y[:,i]
        dj_dw += aux[:, np.newaxis] * 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, 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
      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
    """

    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

    return w_in, b_in


#########################################################################
# one-vs-all
#
def oneVsAll(X, y, n_labels, lambda_):
    """
     Trains n_labels logistic regression classifiers and returns
     each of these classifiers in a matrix all_theta, where the i-th
     row of all_theta corresponds to the classifier for label i.

     Parameters
     ----------
     X : array_like
         The input dataset of shape (m x n). m is the number of
         data points, and n is the number of features.

     y : array_like
         The data labels. A vector of shape (m, ).
```

```
        n_labels : int
            Number of possible labels.

        lambda_ : float
            The logistic regularization parameter.

        Returns
        -------
        all_theta : array_like
            The trained parameters for logistic regression for each class.
            This is a matrix of shape (K x n+1) where K is number of classes
            (ie. `n_labels`) and n is number of features without the bias.
        """

    m, n = X.shape
    all_theta = np.zeros((n_labels, n + 1))
    X = np.hstack([np.ones((m, 1)), X])

    y_ = np.array([y == i for i in range(n_labels)])
    all_theta = gradient_descent(X, y_, all_theta, np.zeros(n_labels),
compute_gradient_reg, 0.01, 3000, lambda_)[0]

    return all_theta


def predictOneVsAll(all_theta, X):
    """
    Return a vector of predictions for each example in the matrix X.
    Note that X contains the examples in rows. all_theta is a matrix where
    the i-th row is a trained logistic regression theta vector for the
    i-th class. You should set p to a vector of values from 0..K-1
    (e.g., p = [0, 2, 0, 1] predicts classes 0, 2, 0, 1 for 4 examples) .

    Parameters
    ----------
    all_theta : array_like
        The trained parameters for logistic regression for each class.
        This is a matrix of shape (K x n+1) where K is number of classes
        and n is number of features without the bias.

    X : array_like
        Data points to predict their labels. This is a matrix of shape
        (m x n) where m is number of data points to predict, and n is number
        of features without the bias term. Note we add the bias term for X in
        this function.

    Returns
    -------
    p : array_like
        The predictions for each data point in X. This is a vector of shape (m,
).
    """

    m = X.shape[0]
    p = np.zeros(m)
```

```
    X = np.hstack([np.ones((m, 1)), X])
    a = sigmoid(np.dot(X, all_theta.T))
    p = np.argmax(a, axis=1)

    return p


#########################################################################
# NN
#
def predict(theta1, theta2, X):
    """
    Predict the label of an input given a trained neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size)

    X : array_like
        The image inputs having shape (number of examples x image dimensions).

    Return
    ------
    p : array_like
        Predictions vector containing the predicted label for each example.
        It has a length equal to the number of examples.
    """

    m = X.shape[0]
    p = np.zeros(m)
    X = np.hstack([np.ones((m, 1)), X])
    a = sigmoid(np.dot(X, theta1.T))
    a = np.hstack([np.ones((m, 1)), a])
    a2 = sigmoid(np.dot(a, theta2.T))
    p = np.argmax(a2, axis=1)

    return p


def partA():
    data = sio.loadmat('data/ex3data1.mat', squeeze_me=True)
    X = data['X']
    y = data['y']
    # rand_indices = np.random.choice(X.shape[0], 100, replace=False)
    # utils.displayData(X[rand_indices, :])
    # plt.show()

    lambda_ = 0.01
    num_labels = 10
    all_theta = oneVsAll(X, y, num_labels, lambda_)
```

```python
    p = predictOneVsAll(all_theta, X)
    print('Part A accuracy: ', np.sum(p == y) / p.size * 100, '%', sep='')
    print('Expected: 95%')


def partB():
    data = sio.loadmat('data/ex3data1.mat', squeeze_me=True)
    X = data['X']
    y = data['y']
    # rand_indices = np.random.choice(X.shape[0], 100, replace=False)
    # utils.displayData(X[rand_indices, :])
    # plt.show()

    weights = sio.loadmat('data/ex3weights.mat')
    theta1, theta2 = weights['Theta1'], weights['Theta2']
    p = predict(theta1, theta2, X)
    print('Part B accuracy: ', np.sum(p == y) / p.size * 100, '%', sep='')
    print('Expected: 97.5%')


if __name__ == '__main__':
    partA()
    partB()
```