Práctica 3: Regresión logística

Alberto Muñoz Fernández

Óscar García Castro

```
from ctypes import sizeof
import numpy as np
import copy
import math
import public_tests as tests
import utils

def read_data():
  data=np.loadtxt("./data/ex2data2.txt",delimiter=',',skiprows=0)
  X_train=data[:,:2]
  y_train=data[:,2]
  return X_train, y_train

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*-1))

    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
    """

    cost = 0
    m = len(X)

    cost = np.sum(-y * np.log(sigmoid(X @ w + b)) - (1-y) * np.log(1 -
sigmoid(X @ w + b)))

    return cost/m


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.
    """
    dj_db = 0
    dj_dw = np.zeros(len(X[0]))
    m = len(X)

    dj_db = np.sum(sigmoid(X @ w + b) - y)
    dj_dw = (sigmoid(X @ w + b) - y) @ X

    return dj_db / m, dj_dw / m


##########################################################################
# 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
    """
    cost = 0
    w_total = np.sum((w**2))

    m = len(X)

    cost = np.sum(-y * np.log(sigmoid(X @ w + b)) - (1-y) * np.log(1 -
sigmoid(X @ w + b)))

    total_cost = (cost/m) + ((lambda_*w_total)/(2*m))
    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.

    """
    dj_db = 0
    dj_dw = np.zeros(len(X[0]))
    m = len(X)

    dj_db = np.sum(sigmoid(X @ w + b) - y)
    dj_dw = (sigmoid(X @ w + b) - y) @ X

    return dj_db / m, (dj_dw / m) + (np.dot(np.divide(lambda_, m), w))


##########################################################################
# 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 = []
    w = copy.deepcopy(w_in)
    b = b_in

    for i in range(num_iters):
        gb, gw = gradient_function(X, y, w, b)
        w -= alpha*gw
        b -= alpha*gb
        J_history += [cost_function(X, y, w, b)]

    return w, b, 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
    """

    p = np.zeros(len(X))

    for i in range(len(X)) :
      if(sigmoid(np.dot(w, X[i]) + b) > 0.5) :
        p[i] = 1.0

    return p

def main() :
  # tests.sigmoid_test(sigmoid)
  # tests.compute_cost_test(compute_cost)
  # tests.compute_gradient_test(compute_gradient)
  # tests.predict_test(predict)
  # tests.compute_cost_reg_test(compute_cost_reg)
  # tests.compute_gradient_reg_test(compute_gradient_reg)

  # Test de la prediccion

  X, y = read_data()
  X = utils.map_feature(X[:, 0], X[:, 1])
  w = np.zeros(len(X[0]))
  w, b, J_history = gradient_descent(X, y, w, 1, compute_cost_reg,
compute_gradient_reg, 0.01, 10000, 0.01)
  p_Y = predict(X, w, b)
  utils.plot_decision_boundary(w, b, X,y)

  num_correct = 0
  for i in range(len(y)):
    if p_Y[i] == y[i]:
        num_correct += 1

  print(num_correct/len(y)*100)

main()
```