```python
'''
Implementation of polynomal fit.
'''

import numpy as np
from numpy import *
import matplotlib.pyplot as plt

# fit a polynomial of degree to t and y data
def polyfit_D(t, y, degree):
   n = len(y)

   # make a [N x (Degree+1)] matrix
   X = np.zeros((n, degree+1))
   for i in range(n):
      for j in range(degree+1):
         X[i][j] = (t[i])**j
   #print X

   # using ordinary least squares
   # compute (X^T*X)^(-1) * X^T * y
   XtX = np.dot(np.transpose(X),X)
   inv_XtX = np.linalg.inv(XtX)
   Xty = np.dot(np.transpose(X),y)
   coeff = np.dot(inv_XtX, Xty)

   return coeff

# returns MSE values for polynomial fitting with degree = [deg_min, deg_max]
# adjusted parameter determines if to add (sigma^2)*DLog(n)/n to MSE value
def polyfit_D_range(t,y,y_tilde,deg_min,deg_max,adjusted):
   i = 0

   MSE_vals = np.zeros(deg_max-deg_min)
   MSE_vals_ytilde = np.zeros(deg_max-deg_min)
   D_vals = np.zeros(deg_max-deg_min)
   # fit the data with a D degree polynomial
   for D in range(deg_min, deg_max):
      # compute coefficients for polynomial of degree D
      coeffs = polyfit_D(t,y,D)
      if(D == 3):
         print coeffs
      # reverse order of coefficients for poly1d function
      rev_coeffs = np.fliplr([coeffs])[0]

      # construct the polynomial for graphing
      polyn = np.poly1d(rev_coeffs)
      # compute mean squared error
      MSE_vals[i] = MSE(t,y,polyn,D)
      MSE_vals_ytilde[i] = MSE(t,y_tilde,polyn,D)
      if(adjusted):
         sigma_2 = 0.25**2
         MSE_vals[i] += (sigma_2*D)*np.log(len(y))/len(y)
```

```python
        D_vals[i] = D
        i += 1
    return (D_vals, MSE_vals, MSE_vals_ytilde)

# plot MSE vs degree for R(D) and F(D)
def plot_MSE2(D_vals, MSE_vals_y1, MSE_vals_ytilde1):
    # visualize degree vs. MSE
    plt.plot(D_vals, MSE_vals_y1, 'o-b', D_vals, MSE_vals_ytilde1, 'o-g')
    plt.title('Plot of the the mean-squared error vs. degree of polynomial')
    plt.legend(['R(D)', 'R_tilde(D)'])
    plt.xlabel('D values')
    plt.ylabel('Mean-squared error')
    plt.show()

# compute mean squared error for estimated polynomial of degree D
def MSE(t, y, polyn, D):
    n = len(y)
    sum = 0.0
    for i in range(0,n):
        sum += (y[i] - polyn(t[i])) ** 2

    return sum/n

if __name__ == "__main__":
    t = np.loadtxt('data_problem2.1/t.dat')
    y_orig = np.loadtxt('data_problem2.1/y.dat')
    y_fresh = np.loadtxt('data_problem2.1/yfresh.dat')

    # choose a y data source
    y = y_orig
    n = len(y) # 9 in this case

    # R(D), R_tilde(D)
    (D_vals1, MSE_vals_y1, MSE_vals_ytilde1) = polyfit_D_range(t, y_orig, y_fresh, 2, 10,
        0);
    # F(D), F_tilde(D)
    (D_vals2, MSE_vals_y2, MSE_vals_ytilde2) = polyfit_D_range(t, y_orig, y_fresh, 2, 10,
        1);

    plot_MSE2(D_vals1, MSE_vals_y1, MSE_vals_ytilde1)
    plot_MSE2(D_vals2, MSE_vals_ytilde1, MSE_vals_y2)
```

```python
"""
Implementation of stochastic gradient descent.
"""
import numpy as np
from numpy import *
import matplotlib.pyplot as plt

def log_loss(X,y,theta):
     sum = 0
     for i in range(m):
         sum += log(1+np.exp(-y[i]*((theta.T)*X[i] + b)))
```

```
    sum = -sum

def stoch_gd(X, y, numIter, stepSize, epsilon):
    (m,n) = np.shape(X)
    theta = np.random.rand(n)

    # begin iterations
    for i in range(numIter):
        I = np.random.randint(0,m)

        # extheta = np.exp((theta.T)*X[I])
        # compute the gradient at the current location
        g = y[I]*X[I] - X[I]*1.0/(1.0+np.exp(-(theta.T)*X[I]))

        # step in the direction of the gradient
        theta2 = theta + (1.0/(i+1.0))*g

        if(np.dot(theta2-theta, theta2-theta) < epsilon):
            return theta
        else:
            theta = theta2

    # return the solution
    return theta


if __name__ == '__main__':
    Xone = np.loadtxt('data_problem2.4/Xone.dat')
    yone = np.loadtxt('data_problem2.4/yone.dat')

    Xtwo = np.loadtxt('data_problem2.4/Xtwo.dat')
    ytwo = np.loadtxt('data_problem2.4/ytwo.dat')

    (m,n) = np.shape(Xone)
    # take number of iterations to be number of examples
    numIter = 10000
    epsilon = 0.0000000000001
    stepSize = 0.01

    # data set #1
    theta_hat1 = stoch_gd(Xone, yone, numIter, stepSize, epsilon)
    e_yxtheta1 = np.exp(-np.dot(Xone,theta_hat1))
    p_one = 1.0/(1.0+e_yxtheta1)

    # data set #2
    theta_hat2 = stoch_gd(Xtwo, ytwo, numIter, stepSize, epsilon)
    e_yxtheta2 = np.exp(-np.dot(Xtwo,theta_hat2))
    p_two = 1.0/(1.0+e_yxtheta2)

    print theta_hat1
    print theta_hat2

    bins = np.linspace(0, 1, 40)
```

```
    plt.title("Histogram of probabilities based on theta_hat")
    plt.xlabel("Probability: P(y_i | x_i, theta_hat)")
    plt.ylabel("Number of samples with given probability")
    #plt.hist(p_one, bins)
    plt.hist(p_two, bins, facecolor='green')
    plt.show()
```

```
"""
Implementation of 2-component GMM.
"""
import numpy as np
from numpy import *
import matplotlib.pyplot as plt
from sklearn import mixture
import pylab

def run_gmm(X, num_components):
    gmm = mixture.GMM(n_components=num_components, covariance_type='full')
    gmm.fit(X)
    print gmm.means_
    colors = ['r' if i==0 else 'b' for i in gmm.predict(X)]
    p = plt.gca()
    p.scatter(X[:,0], X[:,1], c=colors)
    plt.title("2-component GMM for Xone and Xtwo datasets")
    plt.show()

#returns two matrices, one with all one labels, and other with 0 labels
def parse_data(X, y):
    X_1 = []
    X_0 = []

    (m,n) = np.shape(X)
    for i in range(m):
        if y[i] == 1.0:
            X_1.append(X[i])
        else:
            X_0.append(X[i])

    return (X_1, X_0)

def perpendicular(a) :
    b = np.empty_like(a)
    b[0] = -a[1]
    b[1] = a[0]
    return b

def plot_gmm(X1, X0, m1, m0, theta_hat):
    diff = m0 - m1
    perp = perpendicular(diff)

    diff2 = perp - theta_hat
    slope = perp[1]/perp[0]
    b = theta_hat[1] - slope*theta_hat[0]
```

```
    x1 = 6
    y1 = slope*x1 + b

    x2 = -1
    y2 = slope*x2 + b

    xplot = [x1, x2]
    yplot = [y1, y2]
    plt.plot([x[0] for x in X1], [x[1] for x in X1], 'ob', [x[0] for x in X0], [x[1] for
        x in X0], 'og')
    plt.plot(m1[0], m1[1], 'or', m0[0], m0[1], 'or')
    plt.plot(xplot, yplot, '-r')
    plt.show()

def get_mean_cov(X):
    m = np.mean(X, axis=0)
    cov = np.cov(np.transpose(X))
    return (m, cov)

if __name__ == '__main__':
    Xone = np.loadtxt('data_problem2.4/Xone.dat')
    Xtwo = np.loadtxt('data_problem2.4/Xtwo.dat')
    yone = np.loadtxt('data_problem2.4/yone.dat')
    ytwo = np.loadtxt('data_problem2.4/ytwo.dat')

    theta_hat1 = [0.03971547, -1.69052142]
    theta_hat2 = [-0.01176046, -0.02374169]

    (Xone1, Xone0) = parse_data(Xone, yone)
    (Xtwo1, Xtwo0) = parse_data(Xtwo, ytwo)

    print "Xone results"
    (m1, c1) = get_mean_cov(Xone1)
    (m0, c0) = get_mean_cov(Xone0)

    plot_gmm(Xone1, Xone0, m1, m0, theta_hat1)

    print "Xtwo results"
    (m21, c21) = get_mean_cov(Xtwo1)
    (m20, c20) = get_mean_cov(Xtwo0)

    plot_gmm(Xtwo1, Xtwo0, m21, m20, theta_hat2)
```