```
'''
    Template for polynomial regression
    AUTHOR Eric Eaton, Xiaoxiang Hu
'''

import numpy as np
import scipy.stats as ss



#--------------------------------------------------------------------
#  Class PolynomialRegression
#--------------------------------------------------------------------

class PolynomialRegression:

    def __init__(self, degree=1, reg_lambda=1E-8):
        """
        Constructor
        """
        #TODO
        self.degree = degree
        self.regLambda = reg_lambda
        self.theta = None
        self.mean= None;
        self.sd= None;

    def polyfeatures(self, X, degree):
        """
        Expands the given X into an n * d array of polynomial features of
            degree d.

        Returns:
            A n-by-d numpy array, with each row comprising of
            X, X * X, X ** 3, ... up to the dth power of X.
            Note that the returned matrix will not include the zero-th power.

        Arguments:
            X is an n-by-1 column numpy array
            degree is a positive integer
        """

        # add 1s column
        X_ = np.copy(X)
        for  i in range(2,degree+1):
            X_ = np.c_[X_, np.power(X,i)]
        print(X_.shape)
        return X_


    def fit(self, X, y):
        """
            Trains the model
            Arguments:
                X is a n-by-1 array
                y is an n-by-1 array
            Returns:
                No return value
            Note:
                You need to apply polynomial expansion and scaling
                at first
        """
        #here
        X_ = self.polyfeatures(X,self.degree)
        print(X_[:,1:3])
        n, d = X_.shape
        self.mean = np.mean(X_,axis=0)
        self.sd= np.std(X_,axis=0)
        #end
        #####take care of  std==0
        if(self.sd.all() == 0 ):
            print("ecountered 0!")
```

```
            X_ = (X_ - self.mean)/self.sd
        else:
            print("sd suffices")
            X_ = (X_ - self.mean)/self.sd

        # add column of 1s
        X_= np.c_[np.ones([n,1]),X_]

        # construct reg matrix
        reg_matrix = self.regLambda * np.eye(d + 1)
        reg_matrix[0, 0] = 0

        # analytical solution (X'X + regMatrix)^-1 X' y
        self.theta = np.linalg.pinv(X_.T.dot(X_) + reg_matrix).dot(X_.T).dot(y)


    def predict(self, X):
        """
        Use the trained model to predict values for each instance in X
        Arguments:
            X is a n-by-1 numpy array
        Returns:
            an n-by-1 numpy array of the predictions
        """
        # TODO
        n = len(X)
        X_ = self.polyfeatures(X,self.degree)
        #X_ = (X_ - self.mean)/self.sd
        if(self.sd.all() == 0 ):
            print("ecountered 0!")
            X_ = (X_ - self.mean)/self.sd
        else:
            print("sd suffices")
            X_ = (X_ - self.mean)/self.sd
        # add 1s column
        X_ = np.c_[np.ones([n, 1]), X_]

        # predict
        return X_.dot(self.theta)


#---------------------------------------------------------------------
#  End of Class PolynomialRegression
#---------------------------------------------------------------------



def learningCurve(Xtrain, Ytrain, Xtest, Ytest, reg_lambda, degree):
    """
    Compute learning curve

    Arguments:
        Xtrain -- Training X, n-by-1 matrix
        Ytrain -- Training y, n-by-1 matrix
        Xtest -- Testing X, m-by-1 matrix
        Ytest -- Testing Y, m-by-1 matrix
        regLambda -- regularization factor
        degree -- polynomial degree

    Returns:
        errorTrain -- errorTrain[i] is the training accuracy using
        model trained by Xtrain[0:(i+1)]
        errorTest -- errorTrain[i] is the testing accuracy using
        model trained by Xtrain[0:(i+1)]

    Note:
        errorTrain[0:1] and errorTest[0:1] won't actually matter, since we start dis
playing the learning curve at n = 2 (or higher)
    """

    n = len(Xtrain)

    errorTrain = np.zeros(n)
```

```
    errorTest = np.zeros(n)
    polyReg = PolynomialRegression(degree,reg_lambda)
    #TODO -- complete rest of method; errorTrain and errorTest are already the corre
ct shape
        #X_curr_test=np.copy(Xtest)
        #Y_curr_test=np.copy(Ytest)
    for i in range(2,n):

        X_curr_train = np.copy(Xtrain[0:i+1])
        Y_curr_train = np.copy(Ytrain[0:i+1])


        polyReg.fit(X_curr_train,Y_curr_train)

        res_train =polyReg.predict(X_curr_train)
        res_test = polyReg.predict(Xtest)
        n_train = len(X_curr_train)
        n_test = len(Xtest)
        errorTrain[i]= 1/n_train * np.power(np.subtract(res_train,Y_curr_train),2).s
um();
        errorTest[i]= 1/n_test * np.power(np.subtract(res_test, Ytest),2).sum() ;


    return errorTrain, errorTest
```