```
"""
Author      : Matt Guillory
Class       : HMC CS 158
Date        : 2018 Aug 11
Description : Polynomial Regression
"""

# This code was adapted from course material by Jenna Wiens (UMichigan).

# python libraries
import os, time

# numpy libraries
import numpy as np

# matplotlib libraries
import matplotlib.pyplot as plt

######################################################################
# classes
######################################################################

class Data :

    def __init__(self, X=None, y=None) :
        """
        Data class.

        Attributes
        --------------------
            X       -- numpy array of shape (n,d), features
            y       -- numpy array of shape (n,), targets
        """

        # n = number of examples, d = dimensionality
        self.X = X
        self.y = y

    def load(self, filename) :
        """
        Load csv file into X array of features and y array of labels.

        Parameters
        --------------------
            filename -- string, filename
        """

        # determine filename
        dir = os.path.dirname(__file__)
        f = os.path.join(dir, '..', 'data', filename)

        # load data
        with open(f, 'r') as fid :
            data = np.loadtxt(fid, delimiter=",")

        # separate features and labels
        self.X = data[:,:-1]
        self.y = data[:,-1]

    def plot(self, **kwargs) :
        """Plot data."""

        if 'color' not in kwargs :
            kwargs['color'] = 'b'

        plt.scatter(self.X, self.y, **kwargs)
        plt.xlabel('x', fontsize = 16)
        plt.ylabel('y', fontsize = 16)
        plt.show()

# wrapper functions around Data class
def load_data(filename) :
    data = Data()
```

```
    data.load(filename)
    return data

def plot_data(X, y, **kwargs) :
    data = Data(X, y)
    data.plot(**kwargs)


class PolynomialRegression() :

    def __init__(self, m=1, reg_param=0) :
        """
        Ordinary least squares regression.

        Attributes
        --------------------
            coef_   -- numpy array of shape (d,)
                       estimated coefficients for the linear regression problem
            m_      -- integer
                       order for polynomial regression
            lambda_ -- float
                       regularization parameter
        """
        self.coef_ = None
        self.m_ = m
        self.lambda_ = reg_param


    def generate_polynomial_features(self, X) :
        """
        Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

        Parameters
        --------------------
            X       -- numpy array of shape (n,1), features

        Returns
        --------------------
            Phi     -- numpy array of shape (n,(m+1)), mapped features
        """

        n,d = X.shape

        # part b: modify to create matrix for simple linear model
        if self.m_ == 1:
            newX = np.append(np.ones([n,1]), X, 1)

        # part g: modify to create matrix for polynomial model
        if self.m_ > 1:
            newX = np.empty((n,self.m_+1))
            for i in range(n):
                for j in range(self.m_+1):
                    newX[i][j] = X[i] ** j

        Phi = newX

        return Phi


    def fit_SGD(self, X, y, eta=None,
                eps=1e-10, tmax=1000000, verbose=False) :
        """
        Finds the coefficients of a {d-1}^th degree polynomial
        that fits the data using least squares stochastic gradient descent.

        Parameters
        --------------------
            X       -- numpy array of shape (n,d), features
            y       -- numpy array of shape (n,), targets
            eta     -- float, step size (also known as alpha)
            eps     -- float, convergence criterion
            tmax    -- integer, maximum number of iterations
            verbose -- boolean, for debugging purposes
```

```
        Returns
        --------------------
            self    -- an instance of self
        """
        if self.lambda_ != 0 :
            raise Exception("SGD with regularization not implemented")

        if verbose :
            plt.subplot(1, 2, 2)
            plt.xlabel('iteration')
            plt.ylabel(r'$J(\theta)$')
            plt.ion()
            plt.show()

        X = self.generate_polynomial_features(X) # map features
        n,d = X.shape
        eta_input = eta
        self.coef_ = np.zeros(d)                 # coefficients
        err_list  = np.zeros((tmax,1))           # errors per iteration

        # SGD loop
        for t in xrange(tmax) :
            # part f: update step size
            # change the default eta in the function signature to 'eta=None'
            # and update the line below to your learning rate function
            if eta_input is None :
                eta = 0.0015 # change this line
            else :
                eta = eta_input

            # iterate through examples
            for i in xrange(n) :
                # part d: update theta (self.coef_) using one step of SGD
                # hint: you can simultaneously update all theta using vector math

                self.coef_ = self.coef_ - eta * (np.dot(self.coef_, X[i]) - y[i]) *
X[i]

                # track error
                # hint: you cannot use self.predict(...) to make the predictions
                y_pred = np.dot(X, self.coef_)
                err_list[t] = np.sum(np.power(y - y_pred, 2)) / float(n)

            # stop?
            if t > 0 and abs(err_list[t] - err_list[t-1]) < eps :
                break

            # debugging
            if verbose :
                x = np.reshape(X[:,1], (n,1))
                cost = self.cost(x,y)
                plt.subplot(1, 2, 1)
                plt.cla()
                plot_data(x, y)
                self.plot_regression()
                plt.subplot(1, 2, 2)
                plt.plot([t+1], [cost], 'bo')
                plt.suptitle('iteration: %d, cost: %f' % (t+1, cost))
                plt.draw()
                plt.pause(0.05) # pause for 0.05 sec

        print 'number of iterations: %d' % (t+1)

        return self


    def fit(self, X, y) :
        """
        Finds the coefficients of a {d-1}^th degree polynomial
        that fits the data using the closed form solution.

        Parameters
```

```
        --------------------
            X       -- numpy array of shape (n,d), features
            y       -- numpy array of shape (n,), targets

        Returns
        --------------------
            self    -- an instance of self
        """

        X = self.generate_polynomial_features(X) # map features

        # part e: implement closed-form solution
        # hint: use np.dot(...) and np.linalg.pinv(...)
        #       be sure to update self.coef_ with your solution
        self.coef_ =  np.dot(np.dot(np.linalg.pinv( np.dot(X.transpose(), X) ) , X.t
ranspose()) , y)
        # part j: include L_2 regularization



    def predict(self, X) :
        """
        Predict output for X.

        Parameters
        --------------------
            X       -- numpy array of shape (n,d), features

        Returns
        --------------------
            y       -- numpy array of shape (n,), predictions
        """
        if self.coef_ is None :
            raise Exception("Model not initialized. Perform a fit first.")

        X = self.generate_polynomial_features(X) # map features


        y = np.dot(X, self.coef_)


        return y


    def cost(self, X, y) :
        """
        Calculates the objective function.

        Parameters
        --------------------
            X       -- numpy array of shape (n,d), features
            y       -- numpy array of shape (n,), targets

        Returns
        --------------------
            cost    -- float, objective J(theta)
        """
        n,d = X.shape
        cost = 0

        # calculate cost using formula
        error = self.predict(X) - y
        for i in range(n):
            cost += error[i] ** 2
        cost *= 0.5

        return cost


    def rms_error(self, X, y) :
        """
        Calculates the root mean square error.
```

```
        Parameters
        --------------------
            X       -- numpy array of shape (n,d), features
            y       -- numpy array of shape (n,), targets

        Returns
        --------------------
            error   -- float, RMSE
        """
        # part h: compute RMSE
        error = (float(2 * self.cost(X,y))/float(len(y))) ** 0.5

        return error


    def plot_regression(self, xmin=0, xmax=1, n=50, **kwargs) :
        """Plot regression line."""
        if 'color' not in kwargs :
            kwargs['color'] = 'r'
        if 'linestyle' not in kwargs :
            kwargs['linestyle'] = '-'

        X = np.reshape(np.linspace(0,1,n), (n,1))
        y = self.predict(X)
        plot_data(X, y, **kwargs)
        plt.show()


######################################################################
# main
######################################################################

def main() :
    # toy data
    X = np.array([2]).reshape((1,1))          # shape (n,d) = (1L,1L)
    y = np.array([3]).reshape((1,))           # shape (n,) = (1L,)
    coef = np.array([4,5]).reshape((2,))      # shape (d+1,) = (2L,), 1 extra for bi
as

    # load data
    train_data = load_data('regression_train.csv')
    test_data = load_data('regression_test.csv')



    print 'Visualizing data...'
    plot_data(train_data.X, train_data.y, color='g')
    plot_data(test_data.X, test_data.y, color='r')


    # parts b-f: main code for linear regression
    print 'Investigating linear regression...'

    # model
    model = PolynomialRegression()

    # test part b -- soln: [[1 2]]
    print model.generate_polynomial_features(X)

    # test part c -- soln: [14]
    model.coef_ = coef
    print model.predict(X)

    # test part d, bullet 1 -- soln: 60.5
    print model.cost(X, y)

    # test part d, bullets 2-3
    # for eta = 0.01, soln: theta = [2.441; -2.819], iterations = 616
    start = time.time() # printing out time of different methods of optimization
    model.fit_SGD(train_data.X, train_data.y, 0.0015)
    print 'elapsed SGD: ' + str(time.time() - start)
    print 'sgd solution: %s' % str(model.coef_)
```

```
    # test part e -- soln: theta = [2.446; -2.816]
    start = time.time()
    model.fit(train_data.X, train_data.y)
    print 'elapsed closed form: ' + str(time.time() - start)
    print 'closed_form solution: %s' % str(model.coef_)



    # parts g-i: main code for polynomial regression
    print 'Investigating polynomial regression...'

    # toy data
    m = 2                                      # polynomial degree
    coefm = np.array([4,5,6]).reshape((3,))    # shape (3L,), 1 bias + 3 coefficients

    # test part g -- soln: [[1 2 4]]
    model = PolynomialRegression(m)
    print model.generate_polynomial_features(X)

    # test part h -- soln: 35.0
    model.coef_ = coefm
    print model.rms_error(X, y)

    # non-test code (YOUR CODE HERE)
    # part i -- Add  other values of m to generalize data
    #zeroCost = 0
    #for element in train_data.y:
    #    zeroCost += (1-element) ** 2
    #zeroCost *= 0.5
    #print "0 degree train error: ", ((float(2 * zeroCost))/float(len(train_data.y))
) ** 0.5
#
    #zeroCost = 0
    #for element in test_data.y:
    #    zeroCost += (1-element) ** 2
    #zeroCost *= 0.5
    #print "0 degree test error: ", (float(2 * zeroCost))/(float(len(test_data.y)))
** 0.5
    #for i in range(1,11):
    #    modelI = PolynomialRegression(i)
    #    modelI.fit(train_data.X, train_data.y)
    #    print i, " degree training error: ", modelI.rms_error(train_data.X, train_d
ata.y)
#
    #for i in range(1,11):
    #    modelJ = PolynomialRegression(i)
    #    modelJ.fit(train_data.X, train_data.y)
    #    print i, " degree test error: ", modelJ.rms_error(test_data.X, test_data.y)

    ### ========== TODO : END ========== ###





    ### ========== TODO : START ========== ###
    # parts j-k: main code for regularized regression
    print 'Investigating regularized regression...'

    # test part j -- soln: [3 5.24e-10 8.29e-10]
    # note: your solution may be slightly different
    #       due to limitations in floating point representation
    #       you should get something close to [3 0 0]
    model = PolynomialRegression(m=2, reg_param=1e-5)
    model.fit(X, y)
    print model.coef_

    # non-test code (YOUR CODE HERE)

    ### ========== TODO : END ========== ###
```

```
    print "Done!"

if __name__ == "__main__" :
    main()
```