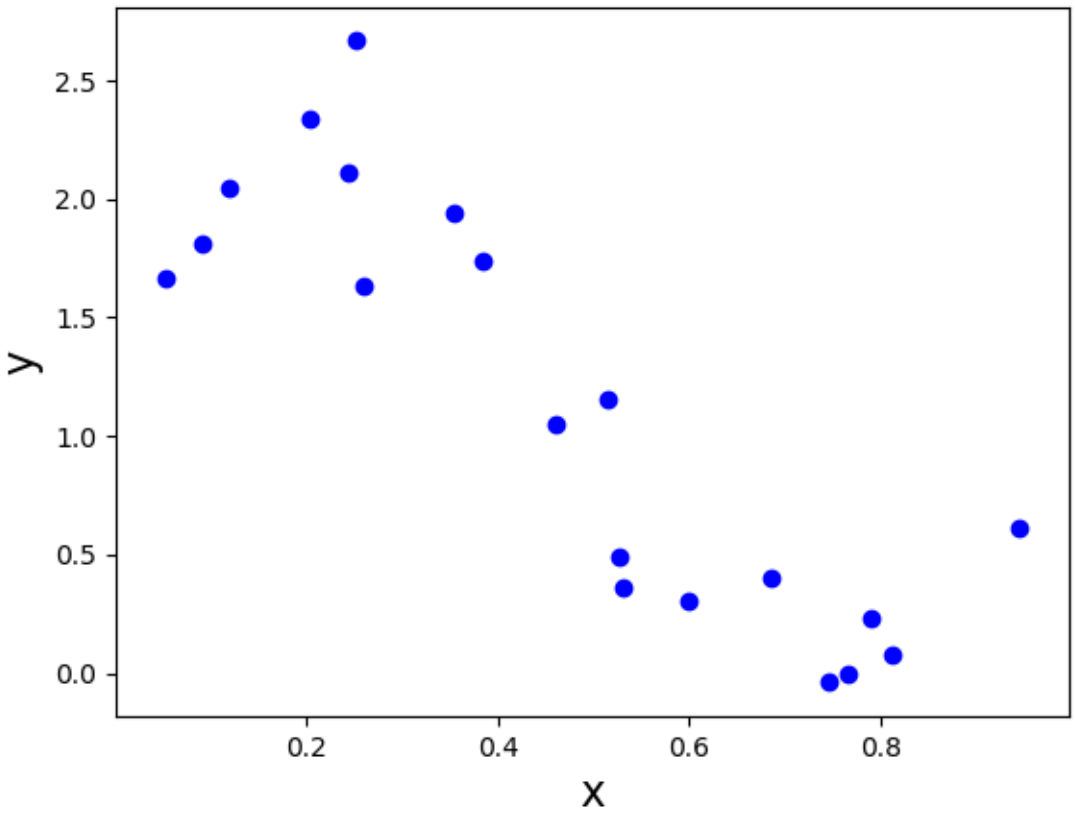
M146 HW2 P4

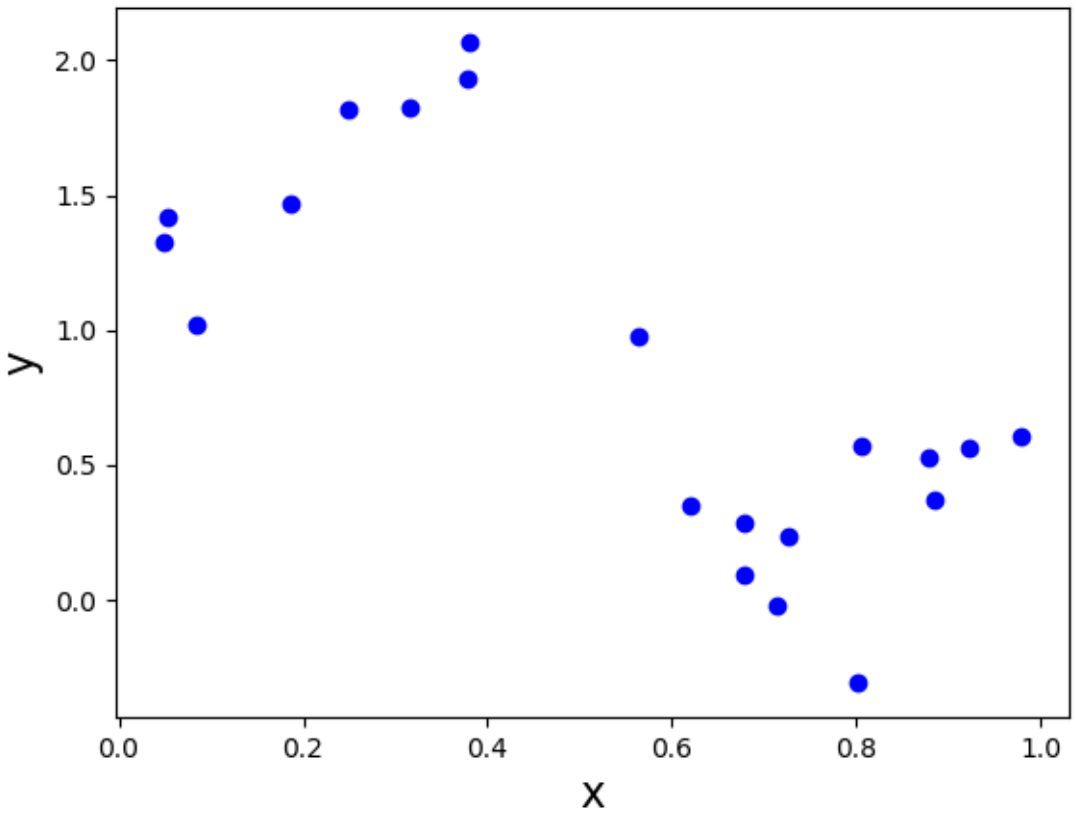
Zheyi Wang 705147852

October 28 2018

(a)

```
### ========== TODO : START ========== ###
# part a: main code for visualizations
print ('Visualizing data...')
train_data.plot()
test_data.plot()
### ========== TODO : END ========== ###
```





The data closely fit to a second order polynomial. Using linear regression to predict the data will cause a very great training error and test error

(b)

```
### ========== TODO : START ========== ###
# part b: modify to create matrix for simple linear model
# part g: modify to create matrix for polynomial model
if d != self.m_ + 1 :
    if self.m_ == 1:
        Phi = np.ones((n, 1))
        Phi = np.column_stack((Phi, X))
    else:
        Phi = np.ones((n, 1))
        for i in range(self.m_):
            Phi = np.column_stack((Phi, X**(i+1)))
else:
    Phi = X

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

This part of code includes both the modifications for part b and part g

(c)

```
    ### ========== TODO : START ========== ###
    # part c: predict y
    y = np.dot(X, self.coef_)
    ### ========== TODO : END ========== ###
```

(d)
(i)

```
  ### ========== TODO : START ========== ###
  # part d: compute J(theta)
  if self.coef_ is None :
      raise Exception("Model not initialized. Perform a fit first.")
  X_mapped = self.generate_polynomial_features(X)
  cost = np.sum(((np.dot(X_mapped, self.coef_) - y)**2))
  ### ========== TODO : END ========== ###

  ### ========== TODO : START ========== ###
  # parts b-f: main code for linear regression
  print ('Investigating linear regression...')
  model = PolynomialRegression()
  model.coef_ = np.zeros(2)
  cost = model.cost(train_data.X, train_data.y)
  print ('    cost: ', cost)
```

```
Investigating linear regression...
    cost:  40.233847409671
```

The cost value is same as what mentioned in the homework manual
(ii)

```
    ### ========== TODO : START ========== ###
    # part d: update theta (self.coef_) using one step of GD
    # hint: you can write simultaneously update all theta using vector math

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

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

(iii)

```
print ('Gradient descent...')
model1 = PolynomialRegression()
for learnrate in [0.0407, 0.01, 0.001, 0.0001]:
    print('Learning Rate:', learnrate)
    start_time = time.time()
    model1.fit_GD(train_data.X, train_data.y, eta=learnrate)
    elapsed_time = time.time() - start_time
    print('    coefficent:        ', model1.coef_)
    print('    cost:              ', model1.cost(train_data.X, train_data.y))
    print('    execution time:    ', elapsed_time)
```

```
Gradient descent...
Learning Rate: 0.0407
number of iterations: 10000
    coefficent:          [-9.40470931e+18 -4.65229095e+18]
    cost:                2.710916520014386e+39
    execution time:      0.33980607986450195
Learning Rate: 0.01
number of iterations: 765
    coefficent:          [ 2.44640703 -2.81635347]
    cost:                3.912576405791486
    execution time:      0.0319828987121582
Learning Rate: 0.001
number of iterations: 7021
    coefficent:          [ 2.4464068 -2.816353 ]
    cost:                3.9125764057919437
    execution time:      0.24187397956848145
Learning Rate: 0.0001
number of iterations: 10000
    coefficent:          [ 2.27044798 -2.46064834]
    cost:                4.0863970367957645
    execution time:      0.3088381290435791
```

learning rate	0.0001	0.001	0.01	0.0407
coefficient	2.27, -2.46	2.45, -2.82	2.45, -2.82	-9.41e+18, -4.65e+18
iterations	10000	7021	765	10000
cost	4.086	3.913	3.913	2.711

From the result, we can see that when the learning rate is too large or too small, it is hard to approach to the solution. Larger learning rate makes gradient descent hard to find the exact minimum of the function while smaller learning rate makes gradient descent take too much iterations to approach to the final solution. Both runs large learning rate and small learning rate didn't converge after 10000 iteration so the result is not accurate. The other two runs get almost the same coefficient and final loss. With the larger effective learning rate the algorithm converge faster.

(e)

```
### ========== TODO : START ========== ###
# 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.inv(np.dot(X.T, X)),X.T),y)

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

```
print('Closed form solution...')
model2 = PolynomialRegression()
start_time = time.time()
model2.fit(train_data.X, train_data.y)
elapsed_time = time.time() - start_time
print('    coefficent:        ', model2.coef_)
print('    cost:              ', model2.cost(train_data.X, train_data.y))
print('    execution time:    ', elapsed_time)
```

```
Closed form solution...
    coefficent:         [ 2.44640709 -2.81635359]
    cost:               3.912576405791464
    execution time:     0.0
```

The coefficients and the cost is almost the same as the result get from convergent gradient descent. Comparing the execution time, we can see that the closed form solution runs much faster than GD since it only need to calculate the solution once.

(f)

```
### ========== TODO : START ========== ###
# 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 = 1/(1+t) # change this line
else :
    eta = eta_input
### ========== TODO : END ========== ###
```

```
print('1/(k+1) learning rate GD...')
model3 = PolynomialRegression()
start_time = time.time()
model3.fit_GD(train_data.X, train_data.y)
elapsed_time = time.time() - start_time
print('    coefficent:        ', model3.coef_)
print('    cost:              ', model3.cost(train_data.X, train_data.y))
print('    execution time:    ', elapsed_time)
```

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

```
1/(k+1) learning rate GD...
number of iterations: 1719
    coefficent:         [ 2.44640672 -2.81635282]
    cost:               3.9125764057922674
    execution time:     0.06597042083740234
```

It took 1719 iterations to converge which is faster than GD with learning rate 0.001 but slower than GD with learning rate 0.01 in terms of the given dataset.

(g)

```
### ========== TODO : START ========== ###
# part b: modify to create matrix for simple linear model
# part g: modify to create matrix for polynomial model
if d != self.m_ + 1 :
    if self.m_ == 1:
        Phi = np.ones((n, 1))
        Phi = np.column_stack((Phi, X))
    else:
        Phi = np.ones((n, 1))
        for i in range(self.m_):
            Phi = np.column_stack((Phi, X**(i+1)))
else:
    Phi = X

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

(h)

```
### ========== TODO : START ========== ###
# part h: compute RMSE
if self.coef_ is None :
    raise Exception("Model not initialized. Perform a fit first.")
n,d = X.shape
X_mapped = self.generate_polynomial_features(X)
cost = np.sum(((np.dot(X_mapped, self.coef_) - y)**2))
error = (cost/n)**0.5
### ========== TODO : END ========== ###
```

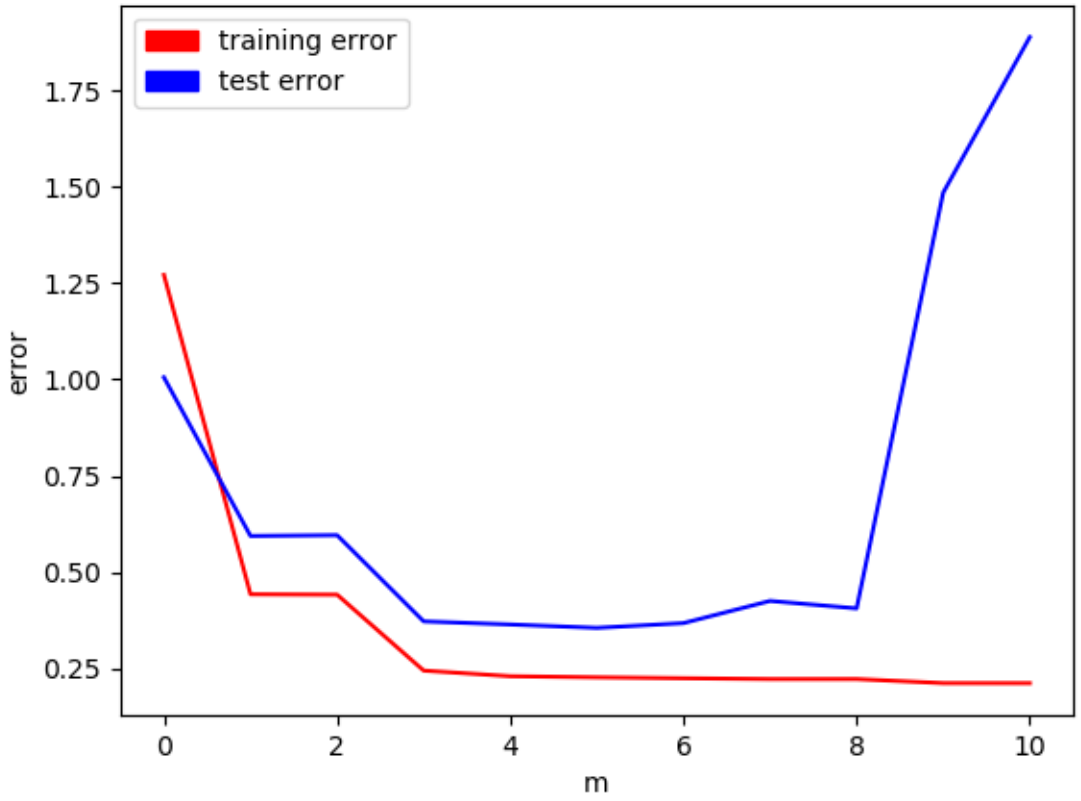
There are two main reason why $j(w)$ is not a good metric. The first reason is that $j(w)$ will keep increasing when n is increasing. The second reason is that $j(w)$ is a sum of square of $y_pred - y$, so the unit of $j(w)$ is the square of the unit of y. We can simply solve these two issue by using RMSE as metric. Therefore, we might prefer RMSE as metric over $j(w)$.

(i)

```
    ### ========== TODO : START ========== ###
    # parts g-i: main code for polynomial regression
    print ('Investigating polynomial regression...')

    train = []
    test = []
    ms = []
    for m in range(11):
        model4 = PolynomialRegression(m=m)
        model4.fit(train_data.X, train_data.y)
        train.append(model4.rms_error(train_data.X, train_data.y))
        test.append(model4.rms_error(test_data.X, test_data.y))
        ms.append(m)
    bestmtest = np.argmin(test)
    bestmtrain = np.argmin(train)

    print ('    best m for training data: ', bestmtrain)
    print ('    best m for test data:     ', bestmtest)

    plt.plot(ms, train, 'r', ms, test, 'b')
    red_patch = mpl.patches.Patch(color='red', label='training error')
    blue_patch = mpl.patches.Patch(color='blue', label='test error')
    plt.xlabel('m')
    plt.ylabel('error')
    plt.legend(handles=[red_patch, blue_patch])
    plt.show()


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

```
Investigating polynomial regression...
    best m for training data:  10
    best m for test data:      5
```

I think $m = 5$ fits best for the data. $m = 4$ and $m = 6$ also fit very well on the data. Higher order of polynomial provides a better fit on training data, but it might cause overfitting. Lower order of polynomial provides a simpler fit on training data, but it might cause underfitting the data. Though $m = 10$ and $m = 9$ fit better on training data, from the plot we can see that the test error significantly increases when m reached 9, so they are both overfitting the training data. When $m = 0, 1, 2, 3$, we can see that the both test error and training error are higher, so they are all underfitting the data.



regression.py

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

# python libraries
import os

# numpy libraries
import numpy as np

# matplotlib libraries
import matplotlib.pyplot as plt
import matplotlib as mpl
import time

##################################################################################
# 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

        ### ========== TODO : START ========== ###
        # part b: modify to create matrix for simple linear model
        # part g: modify to create matrix for polynomial model
```

```
    if d != self.m_ + 1 :
        if self.m_ == 1:
            Phi = np.ones((n, 1))
            Phi = np.column_stack((Phi, X))
        else:
            Phi = np.ones((n, 1))
            for i in range(self.m_):
                Phi = np.column_stack((Phi, X**(i+1)))
    else:
        Phi = X

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

    return Phi


def fit_GD(self, X, y, eta=None,
           eps=0, tmax=10000, verbose=False) :
    """
    Finds the coefficients of a {d-1}^th degree polynomial
    that fits the data using least squares batch gradient descent.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), features
        y       -- numpy array of shape (n,), targets
        eta     -- float, step size
        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("GD 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

# GD loop
for t in range(tmax) :
    ### ========== TODO : START ========== ###
    # 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 = 1/(1+t) # change this line
    else :
        eta = eta_input
    ### ========== TODO : END ========== ###

    ### ========== TODO : START ========== ###
    # part d: update theta (self.coef_) using one step of GD
    # hint: you can write simultaneously update all theta using vector math

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

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

    # 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, l2regularize = None ) :
    """
    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
        l2regularize    -- set to None for no regularization. set to positive double for L2

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

    X = self.generate_polynomial_features(X) # map features

    ### ========== TODO : START ========== ###
    # 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.inv(np.dot(X.T, X)),X.T),y)

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


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

    ### ========== TODO : START ========== ###
    # part c: predict y
```

```
    y = np.dot(X, self.coef_)
    ### ========== TODO : END ========== ###

    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)
    """
    ### ========== TODO : START ========== ###
    # part d: compute J(theta)
    if self.coef_ is None :
        raise Exception("Model not initialized. Perform a fit first.")
    X_mapped = self.generate_polynomial_features(X)
    cost = np.sum(((np.dot(X_mapped, self.coef_) - y)**2))
    ### ========== TODO : END ========== ###
    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
    """
    ### ========== TODO : START ========== ###
    # part h: compute RMSE
    if self.coef_ is None :
        raise Exception("Model not initialized. Perform a fit first.")
    n,d = X.shape
    X_mapped = self.generate_polynomial_features(X)
```

```
        cost = np.sum(((np.dot(X_mapped, self.coef_) - y)**2))
        error = (cost/n)**0.5
        ### ========== TODO : END ========== ###
        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() :
    # load data
    train_data = load_data('regression_train.csv')
    test_data = load_data('regression_test.csv')



    ### ========== TODO : START ========== ###
    # part a: main code for visualizations
    print ('Visualizing data...')
    train_data.plot()
    test_data.plot()
    ### ========== TODO : END ========== ###



    ### ========== TODO : START ========== ###
    # parts b-f: main code for linear regression
    print ('Investigating linear regression...')
    model = PolynomialRegression()
    model.coef_ = np.zeros(2)
    cost = model.cost(train_data.X, train_data.y)
    print ('    cost: ', cost)

    print ('Gradient descent...')
```

```
model1 = PolynomialRegression()
for learnrate in [0.0407, 0.01, 0.001, 0.0001]:
    print('Learning Rate:', learnrate)
    start_time = time.time()
    model1.fit_GD(train_data.X, train_data.y, eta=learnrate)
    elapsed_time = time.time() - start_time
    print('    coefficent:        ', model1.coef_)
    print('    cost:              ', model1.cost(train_data.X, train_data.y))
    print('    execution time:    ', elapsed_time)

print('Closed form solution...')
model2 = PolynomialRegression()
start_time = time.time()
model2.fit(train_data.X, train_data.y)
elapsed_time = time.time() - start_time
print('    coefficent:        ', model2.coef_)
print('    cost:              ', model2.cost(train_data.X, train_data.y))
print('    execution time:    ', elapsed_time)

print('1/(k+1) learning rate GD...')
model3 = PolynomialRegression()
start_time = time.time()
model3.fit_GD(train_data.X, train_data.y)
elapsed_time = time.time() - start_time
print('    coefficent:        ', model3.coef_)
print('    cost:              ', model3.cost(train_data.X, train_data.y))
print('    execution time:    ', elapsed_time)

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



### ========== TODO : START ========== ###
# parts g-i: main code for polynomial regression
print ('Investigating polynomial regression...')

train = []
test = []
ms = []
for m in range(11):
    model4 = PolynomialRegression(m=m)
    model4.fit(train_data.X, train_data.y)
    train.append(model4.rms_error(train_data.X, train_data.y))
    test.append(model4.rms_error(test_data.X, test_data.y))
    ms.append(m)
bestmtest = np.argmin(test)
bestmtrain = np.argmin(train)
```

```
    print ('    best m for training data: ', bestmtrain)
    print ('    best m for test data:     ', bestmtest)

    plt.plot(ms, train, 'r', ms, test, 'b')
    red_patch = mpl.patches.Patch(color='red', label='training error')
    blue_patch = mpl.patches.Patch(color='blue', label='test error')
    plt.xlabel('m')
    plt.ylabel('error')
    plt.legend(handles=[red_patch, blue_patch])
    plt.show()


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


    print ("Done!")

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