School Of Electrical Engineering And Computer Science
National University Of Science and Technology

Assignment No.2
Linear Regression and optimization Functions

Prepared by: Huzaifa Imran
himran.msee18seecs@seecs.edu.pk
Course Instructor: Dr. Wahjahat Hussain

1 Introduction

In this exercise, you will implement linear regression and get to see it work on data. Before starting on this programming exercise, we strongly recommend clearing concepts of linear regression, gradient descent and feature normalization. To get started with the exercise, you will need to download the starter code and unzip its contents to the folder. Files incuded are:

- **ex2P1.ipyb:** Google Colab Notebook (gradient descent) script that steps you through the exercise
- **ex2P2.ipyb:** Google Colab Notebook (Optimization Function) script that steps you through the exercise
- **ex2data1.txt:** Dataset Text file for linear regression with one variable

Throughout the exercise, you will be using the scripts ex2P1.ipyb and ex2P2.ipyb. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions , by following the instructions in this assignment.

Where to get help?

We also strongly encourage using the online Forums and WhatsApp Group to discuss exercises with other students. However, **do not look at any source code written by others or share your source code with others.**

For the first part of this exercise you will be using ex2P1.ipynb script you need to upload ex2P1.ipynb and ex1data1.txt into your Google drive and open ex2P1 in Google Colab.

1.1 Simple Python function

The first task in ex2P1.ipyb gives you practice with function syntax. In the the warmUpExercise, you will find a predefined function in given space to return a 5x5 or any other value identity matrix you should see output similar to the following:

```
1 print("Matrix a : \n", iden(5))
```

Output

```
Matrix a :
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

2 Linear Regression In One variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities, you have data for profits and populations from the cities.

You would like to use this data to help you select which city to expand to next. The file **ex2data1.txt** contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss.
The **ex2P1.ipyb** script has already been set up to load this data for you.

2.1 Plotting Data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). (Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.)
In ex2P1.ipyb, the dataset is loaded from the ex2data.txt into the variables X and Y:

```
# Read comma separated data
data = np.loadtxt(os.path.join('Data', path ), delimiter=',')
X, Y = data[:, 0], data[:, 1]
```

Next, the script calls the ***plotdata(X, Y)*** function to create a scatter plot of the data. Your job is to complete ***plotdata(X, Y)*** to draw the plot.
Now, when you run the plotdata, our end result should look like Figure 1 Below, with the same red "x" markers and axis labeled

2.2 Gradient Descent

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent.

2.2.1 Update Equations

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1}$$

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1 \tag{2}$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$.One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

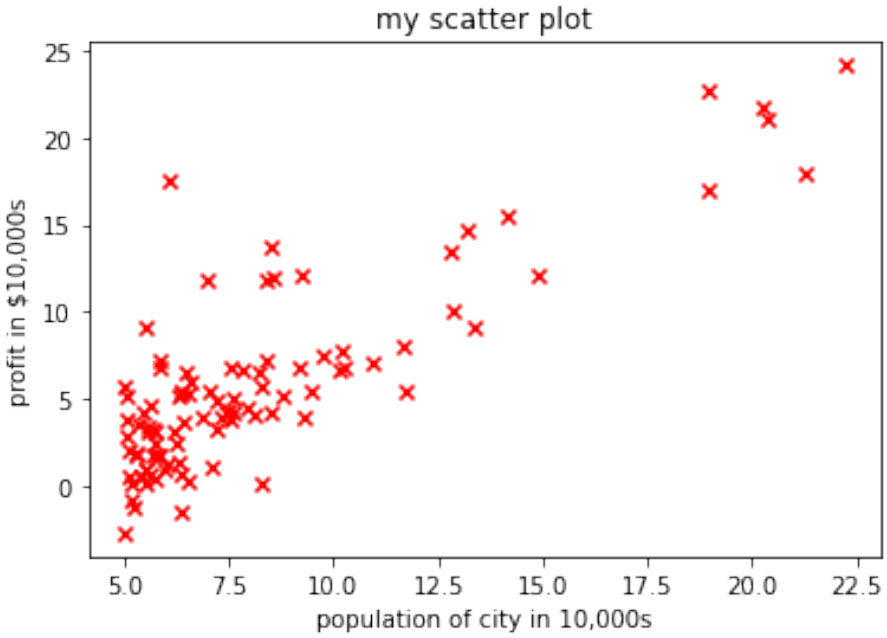

Figure 1: Scatter plot of training data

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all j)} \tag{3}$$

With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost J(θ). Another method is to use predefined function that optimizes or minimizes the error function.

2.2.2 Implementation

In ex2P1, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term. **Do NOT execute this cell more than once.**

```
m = Y.size # number of training examples
X = np.stack([np.ones(m), X], axis=1) # it used to convert X in to (97x2) i.e Add a
    column of ones to x
print(X.shape)
```

Output

```
(97, 2)
```

2.2.3 Computing cost

As you perform gradient descent to learn minimize the cost function J(θ), it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate J(θ) so you can check the convergence of your gradient descent implementation. Your next task is to complete the code in the ***computeCost*** function, which is a function that computes J(θ). As you are doing this, remember that the variables X and y are not scalar values, but matrices whose rows represent the examples from the training set. Once you have completed the function, run the block and with θ initialized to zeros,You should expect to see a cost of **32.07**.

```
def computeCost(X,y , theta):
  m = y.size
  J = 0 # you should return this parameter correctly
  h = np.dot(X, theta)
  ############ YOUR COST FUNCTION J HERE ############



 ################################################
  return J

J = computeCost(X, Y, theta=np.array([0.0, 0.0]))
print('With theta = [0, 0] \nCost computed =', J)
print('Expected cost value (approximately) 32.07\n')
```

2.2.4 Gradient descent

Next, you will complete a function which implements gradient descent. The loop structure has been written for you, and you only need to supply the updates to θ, within each iteration. As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost J(θ) is parameterized by the vector θ, not X and y. That is, we minimize the value of J(θ) by changing the values of the vector θ, not by changing X or y. Refer to the equations in this handout and to the lectures slides if you are uncertain 3.

A good way to verify that gradient descent is working correctly is to look at the value of J(θ) and check that it is decreasing with each step. The starter code for the function ***gradientDescent*** calls ***computeCost*** on every iteration and saves the cost to a python list. Assuming you have implemented gradient descent and computeCost correctly, your value of J(θ) should converge to a steady value by the end of the algorithm and you should see the plotted training data and fitted line through training data. 2

```
def gradientDescent(X, y, theta, alpha, num_iters):
  m = y.shape[0]
  theta = theta.copy()
  J_history = []
```

```
5
6     for i in range(num_iters):
7       ########### YOUR GRADIENT DESCENT "theta" HERE ###########
8
9
10
11
12
13      ############################################################
14      J_history.append(computeCost(X, y, theta))
15    return theta, J_history
16
17  # initialize fitting parameters
18  theta = np.zeros(2)
19
20  # some gradient descent settings
21  iterations = 1500
22  alpha = 0.01
23
24  theta, J_history = gradientDescent(X ,Y, theta, alpha, iterations)
25
26  # plot the linear fit
27  plotdata(X[:, 1],Y)
28  plt.plot(X[:, 1], np.dot(X, theta))
29
30  plt.legend([ 'Linear regression','Training data',]);
```

2.3 Visualization

To understand the cost function J(θ) better, you will now plot the cost over a 2 dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.
In the next cell, the code is set up to calculate J(θ) over a grid of values using the ***computeCost*** function that you wrote. After executing the following cell, you will have a 2D array of J(θ) values. Then, those values are used to produce surface and contour plots of J(θ) using the matplotlib plot surface and contour functions . The plots should look something like the figure 3.

The purpose of these graphs is to show you how J(θ) varies with changes in θ_0 and θ_1. The cost function J(θ)is bowl shaped and has a global minimum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

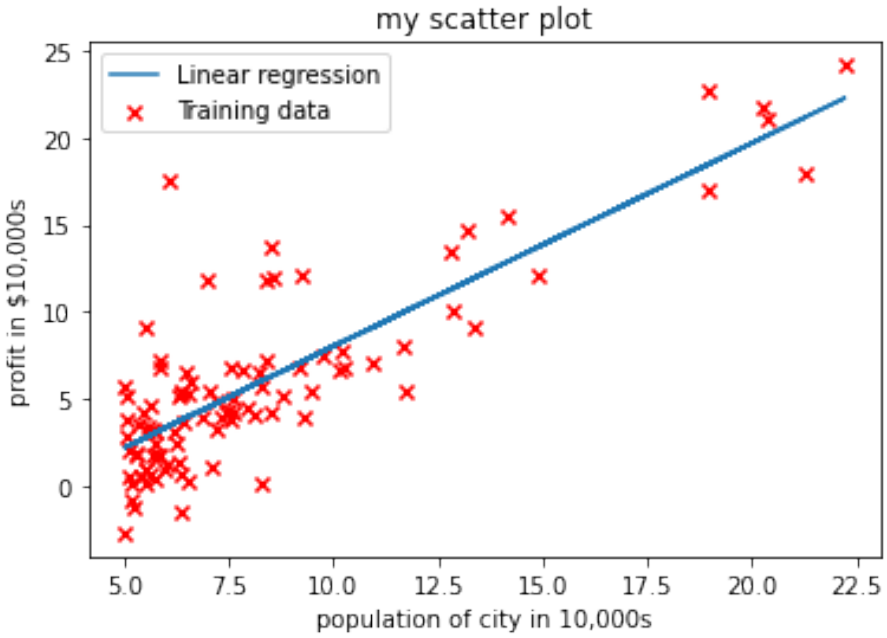

Figure 2: Training data with linear regression fit

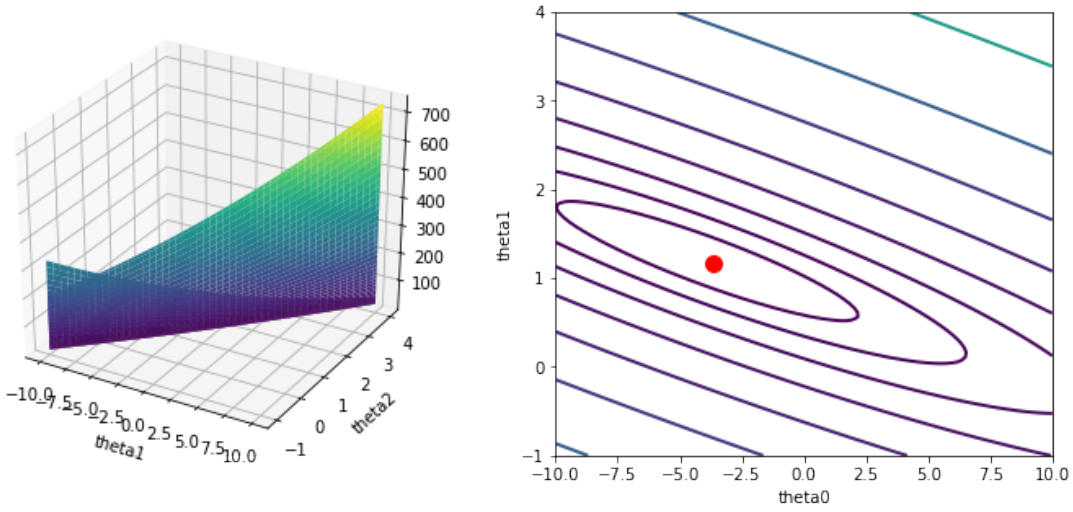

Figure 3: Surface and contour plots showing minimum cost

2.4 Feature Normalization

sometime in your data set you have features with mighty difference in magnitudes for example house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

The next section in ex2P1 will take you through this exercise. Your task here is to complete the code in ***featureNormalize*** function:

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations."

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature this is an alternative to taking the range of values (max-min). In numpy, you can use the std function to compute the standard deviation.

```
def  featureNormalize(X):

########### YOUR CODE HERE ###########



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

    return X_norm, mu, sigma

X, mu, sigma = featureNormalize(X)
Y, mu, sigma = featureNormalize(Y)

X = np.stack([np.ones(m), X], axis=1)
```

2.5 Learning rate

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying the following code and changing the part of the code that sets the learning rate.

```

#CHANGE THE VALUES of ALPHAS, 5 VALUES OF ALPHA
#PLOT LEARNING RATES FOR FOLLOWING FOR ALPHAS, NO NEED TO CHANGE THE CODE ONLY
    REQUIRE "gradientDescent" TO BE DEFINED CORRECTLY
# some gradient descent settings
iterations = 500
alpha = [] ########### ENTER YOUR LEARNING RATES #############
costs=[]

for i in range(5):
  theta = np.zeros(2)
  theta, J_history = gradientDescent(X ,Y, theta, alpha[i], iterations)
          # initialize fitting parameters
  costs.append(J_history)
# Plot the convergence graph

for i in range(5):
```

```
  plt.plot(np.arange(len(costs[i])), costs[i], label=str(alpha[i]))
plt.xlabel('Number of iterations')
plt.ylabel('Cost J')
plt.legend()
```

Use your implementation of gradient Descent function and run gradient descent for about 500 iterations at the chosen learning rate. The function should also return the history of J(θ) values in a vector J. After the last iteration, plot the J values against the number of the iterations. If you picked a learning rate within a good range, your plot look similar as the following Figure 4.

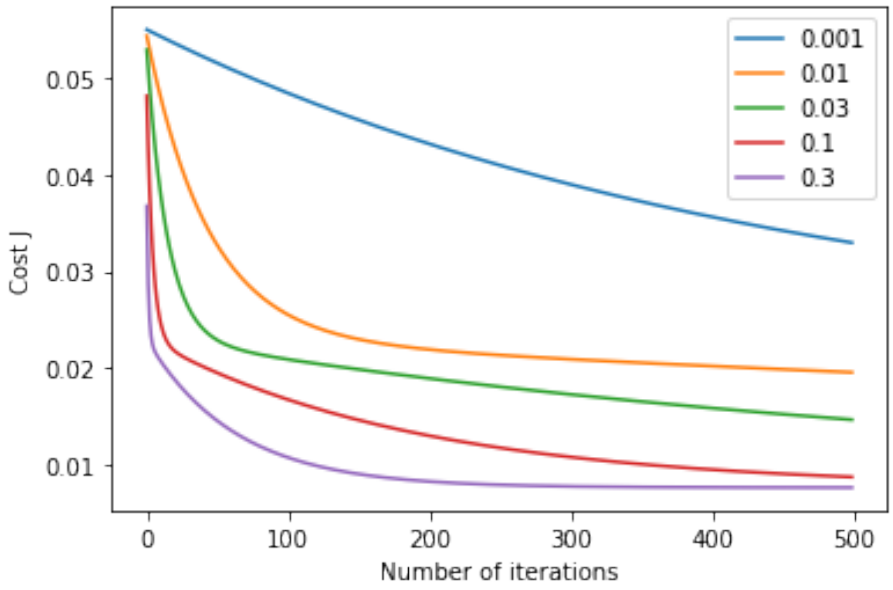

Figure 4: Number of Iterations vs Cost for different learning rates α

If your graph looks very different, especially if your value of J(θ) increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α (alpha) on a log-scale, at multiplicative steps of about 3 times the previous value **(0.3, 0.1, 0.03, 0.01 and 0.001)**. You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

3 Part 2: Optimization Functions

We will use the **fmin** and **fmin_cg** function of SciPy in **ex2P2.ipyb**. DATA initialization will be same as above.

```
res1 = optimize.fmin_cg(J, x0, fprime=gradf,args=args)
```

Read more about fmin_cg *here*.

```
res2 = optimize.fmin(J, x0, args=args)
```

Read more about fmin *here*

Cost function 4 minimization until it gets the optimal value, calls the gradient and finds error for each corresponding value of data.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{4}$$

```
#-------- COST FUNCTION-------------
def J(t,x,y):
  theta=t
  ########### YOUR COST FUNCTION CODE HERE ###########


  ################################################
  lr2.append(J)
  return J
```

Now you need to fill in the code for ***gradf*** function for fmin_cg which only requires 5

$$\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all j)} \tag{5}$$

```

# -----------GRADIENT ONLY FUNCTION-----------
def gradf(t,y, *args):
  theta =t
  ########### GRADIENT ONLY CODE HERE ###########


 ################################################
  #lr2.append(J)
  return theta
```

The Final Result should resemble the figure 5 with given labels. since both optimize equally the result will be same:

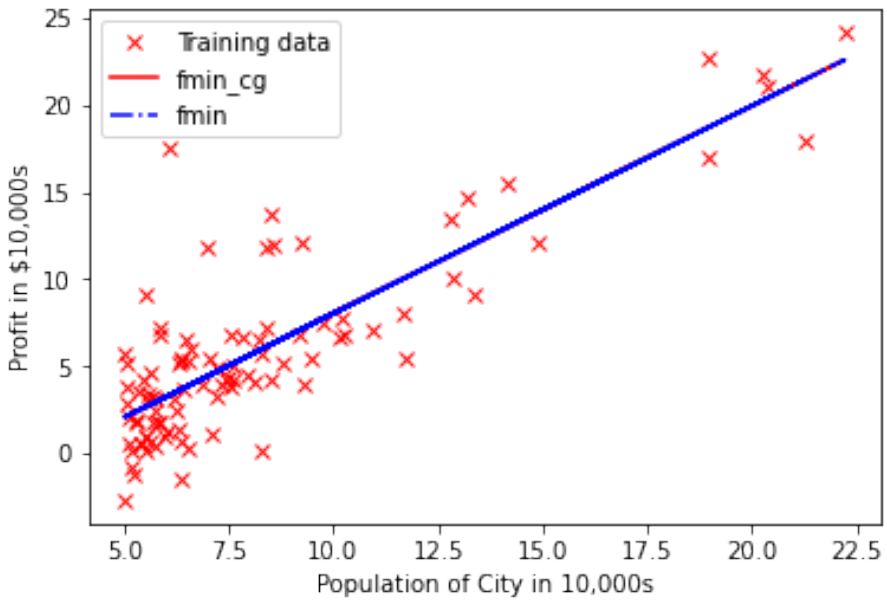

Figure 5: Linear Regression fit by optimization functions

Output

```
Optimization terminated successfully.
         Current function value: 4.476971
         Iterations: 8
         Function evaluations: 19
         Gradient evaluations: 19
Optimization terminated successfully.
         Current function value: 4.476971
         Iterations: 90
         Function evaluations: 172
```

3.1 Learning Rates

Since these function do not require alpha, single learning will be plotted by code given in the next cell.Your learning rates should look like figure 6

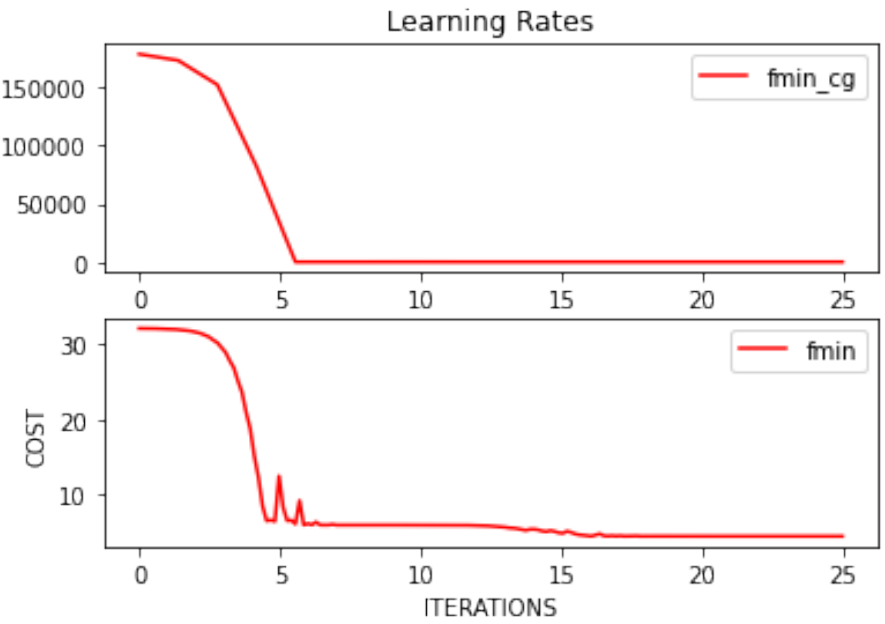

Figure 6: Learning Rates

3.2 Fitting Second order polynomial

The Next block of code in "ex2P2" will fit the 2nd order polynomial through the data.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 \tag{6}$$

Previously, you implemented cost function ***computeCost*** and ***gradientDescent*** on a univariate regression problem. The only difference now is that there is one more feature in the matrix X.The batch gradient descent update rule remain unchanged.Now matrix "X" has 'n' number of features i.e X can be m x (n+1) matrix. And you need n+1 θ_s.
If your ***computeCost*** is vectorized already and supports multiple variables, you can same cost function here too.The Gradient Descent should be generalized so that it can minimize the cost and update n+1 thetas simultaneously after every iteration. This can be done by vectorize implementation of Gradient descent function in ***gradientDescentVectorize*** function. By the end of loop, you should get optimum **theta vector** of order 1 x (n+1) where n are number of features. Your task is to implement vectorize implementations in ***computeCostVectorize*** and ***gradientDescentVectorize*** functions

```
1
2  def computeCostVectorize(X,y , theta):
3    m = y.size
4    J = 0   # You need to return this parameter correctly
5    h = np.dot(X, theta)
6    ############ YOUR Vectorize COST FUNCTION J HERE ###########
7    # Use vectorize implementation. (without using loop)
8
9
```

```

 ############################################################
  return J
```

```

def gradientDescentVectorize(X, y, theta, alpha, num_iters):
  m = y.shape[0]
  theta = theta.copy()
  J_history = []

  for i in range(num_iters):
    ########### YOUR Vectorize GRADIENT DESCENT "theta" HERE ###########
    #vactorize implementation (without looping through training data)









    ############################################################
    J_history.append(computeCostVectorize(X, y, theta))
  return theta, J_history
```

Make sure your code supports any number of features and is well-vectorized. The end plot should look like figure 7

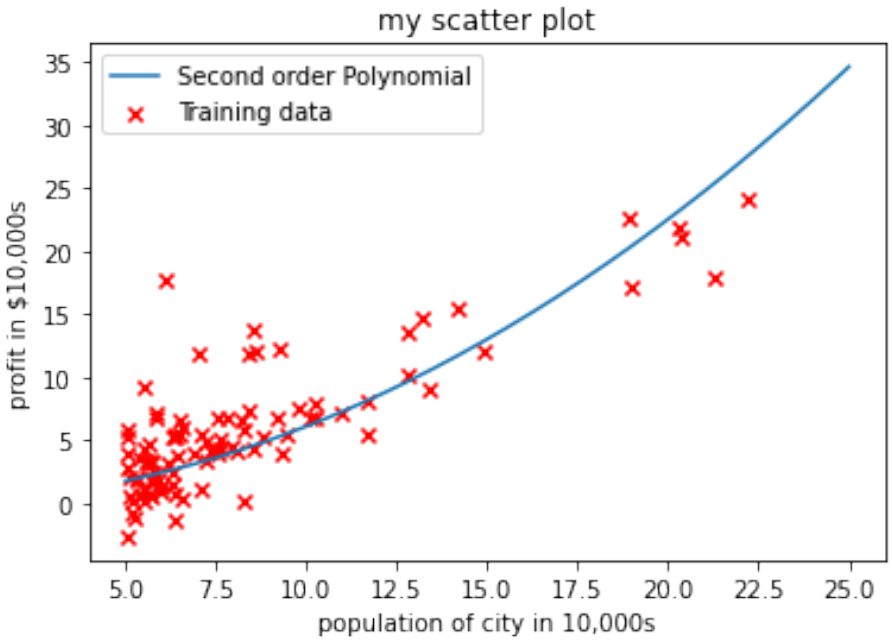

Figure 7: Fitting Second order polynomial

Similarly you can fit third order polynomial through data. Figure 8

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3 \tag{7}$$

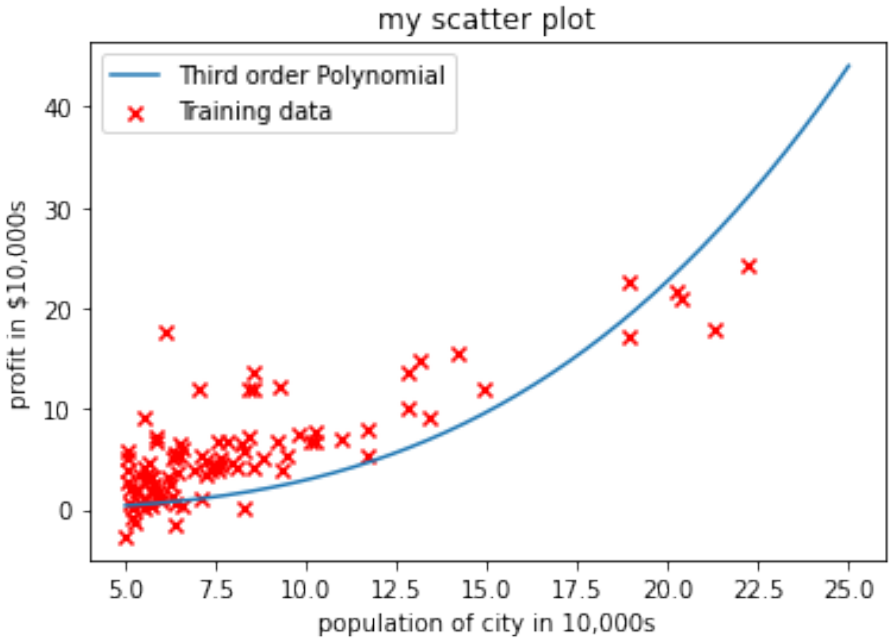

Figure 8: Fitting third order polynomial