Programming Exercise 1: Linear Regression

1 Machine Learning 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 watching the video lectures and completing the review questions for the associated topics. Files included in this exercise

- ex1.py - Script that will help step you through the exercise
- ex1_multi.py - Script for the later parts of the exercise
- ex1data1.txt - Dataset for linear regression with one variable
- ex1data2.txt - Dataset for linear regression with multiple variables
- plotData.py - Function to display the dataset
- computeCost.py - Function to compute the cost of linear regression
- gradientDescent.py - Function to run gradient descent
- computeCostMulti.py - Cost function for multiple variables
- gradientDescentMulti.py - Gradient descent for multiple variables
- featureNormalize.py - Function to normalize features
- normalEqn.py - Function to compute the normal equations

Throughout the exercise, you will be using the scripts ex1.py and ex1_multi.py. 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 in other files, by following the instructions in this assignment.

2 Linear regression with 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 and 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 ex1data1.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 ex1.py script has already been set up to load this data for you.

2.1 Plotting the 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 ex1.py, the dataset is loaded from the data file into the variables X and y. Next, the script calls the plotData function to create a scatter plot of the data. Your job is to complete plotData.py to draw the plot; modify the file and fill in the following code: Now, when you continue to run ex1.py, our end result should look like Figure 1, with the same red "x" markers and axis labels.

2.2 Gradient Descent

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

2.3 Update Equations

The objective of linear regression is to minimize the cost function

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

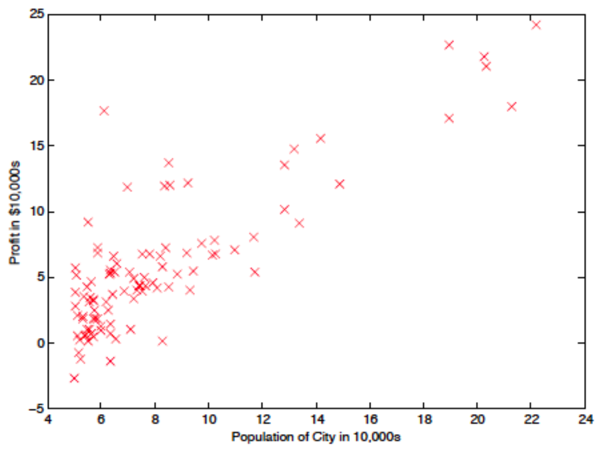

Figure 1: Figure 1: Scatter plot of training data

where the hypothesis $h(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x1$$

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

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

(simultaneously update θ_j for all j). With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost $J(\theta)$.

2.4 Implementation

In ex1.py, we have already already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the 0 intercept term. We also initialize the initial parameters to 0 and the learning rate alpha to 0.01.
X = np.array([[1.0, x] for x in X]) # Add a column of ones to x
theta = np.zeros((2, 1)) # initialize fitting parameters

```
# Some gradient descent settings
iterations = 1500
alpha = 0.01
```

2.5 Computing the cost $J(\theta)$

As you perform gradient descent to learn minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation. Your next task is to complete the code in the file computeCost.py, which is a function that computes $J(\theta)$. 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, the next step in ex1.py will run computeCost once using θ initialized to zeros, and you will see the cost printed to the screen. You should expect to see a cost of 32.07.

2.6 Gradient descent

Next, you will implement gradient descent in the file gradientDescent.py. 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 $J(\theta)$ is parameterized by the vector θ not X and y. That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing X or y. Refer to the equations in this handout and to the video lectures if you are uncertain. A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. The starter code for gradientDescent.py calls computeCost on every iteration and prints the cost. Assuming you have implemented gradient descent and computeCost correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. After you are finished, ex1.py will use your final parameters to plot the linear fit. The result should look something like Figure 2: Your final values for θ will also be used to make predictions on profits in areas of 35,000 and 70,000 people.

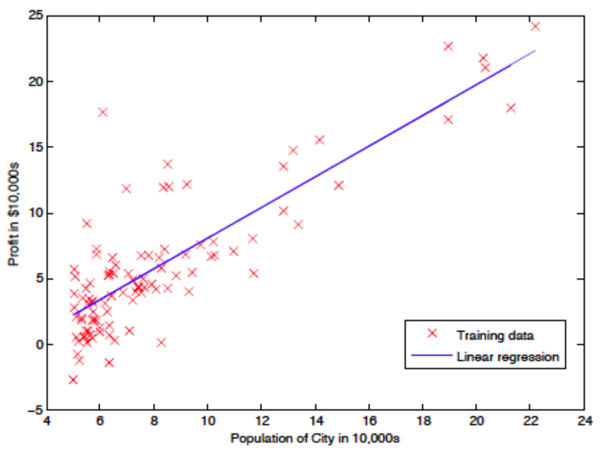

Figure 2: Figure 2: Training data with linear regression fit

3 Multi-variable case

If you have successfully completed the material above, congratulations! You now understand linear regression and should able to start using it on your own datasets.

3.1 Linear regression with multiple variables

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to first collect information on recent houses sold and make a model of housing prices. The file ex1data2.txt contains a training set of housing prices in Portland, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house. The ex1_multi.py script has been set up to help you step through this exercise.

3.2 Feature Normalization

The ex1_multi.py script will start by loading and displaying some values from this dataset. By looking at the values, note that 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.

- Your task here is to complete the code in featureNormalize.py
- 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 (most data points will lie within 2 standard deviations of the mean); this is an alternative to taking the range of values (max-min). For example, inside featureNormalize.py, the quantity X[:,0] contains all the values of x1 (house sizes) in the training set, so np.std(X[:,0]) computes the standard deviation of the house sizes. At the time that featureNormalize.py is called, the extra column of 1's corresponding to $x_0 = 1$ has not yet been added to X (see ex1_multi.py for details). You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix X corresponds to one feature. You should now submit feature normalization.

Implementation Note: When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new x value (living room area and number of bedrooms), we must first normalize x using the mean and standard deviation that we had previously computed from the training set.

3.3 Gradient Descent

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there is one more feature in the matrix X. The hypothesis function and the batch gradient descent update rule remain unchanged. You should complete the code in computeCostMulti.py and gradientDescentMulti.py to implement the cost function and gradient descent for linear regression with multiple variables. If your code in the previous part (single variable) already supports multiple variables, you can use it here too. Make sure your code supports any number of features and is well-vectorized. You can use X.shape[1] to find out how many features are present in the dataset. You should now submit compute cost and gradient descent for linear regression with multiple variables.

Implementation Note: In the multivariate case, the cost function can also be written in the following vectorized form:

$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

where

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

.