UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA TELECOMUNICAÇÕES E INFORMÀTICA

Machine Learning (2016/17) – Lab work 2

PART 1

Objectives: Implement linear regression with one variable and get to see it work on data.

First, you need to download the starter code to the directory where you wish to complete the exercise.

Files included in this exercise:
ex1.m - Octave/MATLAB script that steps you through part 1 (the main program)
ex1data1.txt - Dataset for linear regression with one variable
plotData.m - Function to display the dataset
computeCost.m - Function to compute the cost of linear regression
gradientDescent.m - Function to run gradient descent

In this part of the lab work you will implement linear regression with one variable to predict profits for a food truck (a large vehicle equipped to cook and sell food). 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.

1. Load and plot the Data

The file *ex1data1.txt* contains the dataset for our linear regression problem. The first column is the population of a city (values are scaled: number of people/10000 and profit in dolars/10000) and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss. Load the data into variables X (the first column) and y (the second column).
Complete the *plotData.m* function to create a scatter plot of data (Fig.1).

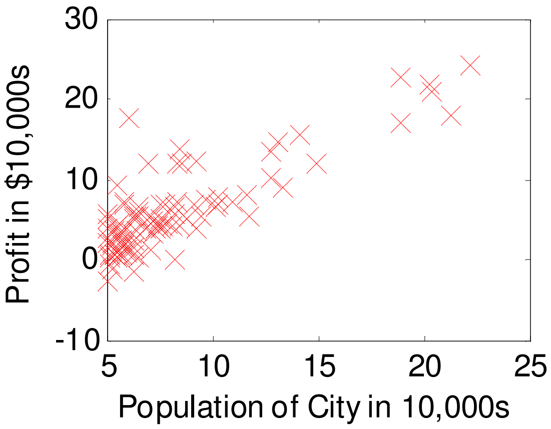

Fig. 1

2. Gradient Descent

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

2.1 Update Equations

The objective of linear regression is to minimize the cost function

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

where the hypothesis *h* is given by the linear model :

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

θ_j are the values you will adjust to minimize the cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs simultaneously update θ_j for all j.

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

With each step of gradient descent, the parameters θ_j come closer to the optimal values that will achieve the lowest cost J.

2.2 Implementation

Each example is stored as a row in the X matrix. To take into account the intercept term (θ_0), you need to add an additional first column to X and set it to all ones. This allows us to treat θ_0 as simply another `feature'. Initialize the fitting parameters to 0 and the learning rate *alpha*= 0.01 and choose 1500 iterations.

2.3 Computing the cost $J(\theta)$

Your task is to complete the code in the file *computeCost.m*,. Remember that the variables X and y are not scalar values, X is a matrix with dimension (mx2), y is vector with dimension (mx1), their m rows represent the examples from the training set. After you completed the function, call *computeCost.m* using θinitialized to zeros, and see the cost printed to the screen. You should expect to see a cost of 32.07.

2.4 Gradient descent

Next, you will implement gradient descent in the function *gradientDescent.m*. The loop structure is written, and you only need to supply the updates to θ within each iteration. After you finish *gradientDescent.m* the main program will use the final parameters (theta(1) = -3.630291, theta(2)=1.166362) to plot the linear fit (Fig.2).

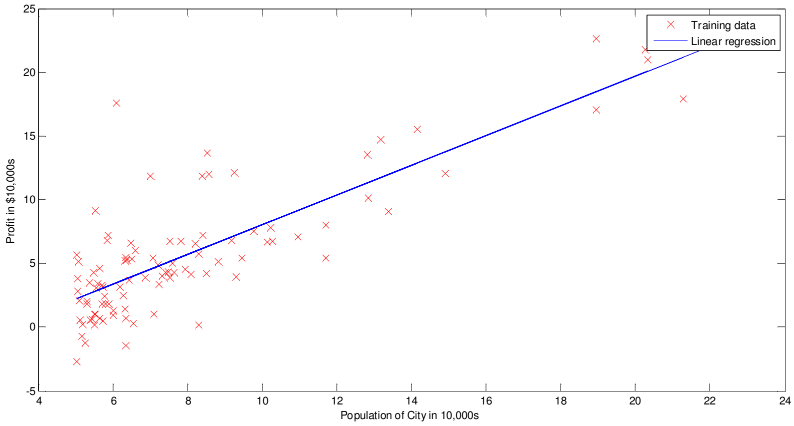

Fig. 2

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. Function *gradientDescent.m* calls function *computeCost.m* on every iteration and saves the costs over the iterations. The value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. Plot the gradient history and get a curve similar to Fig.3.

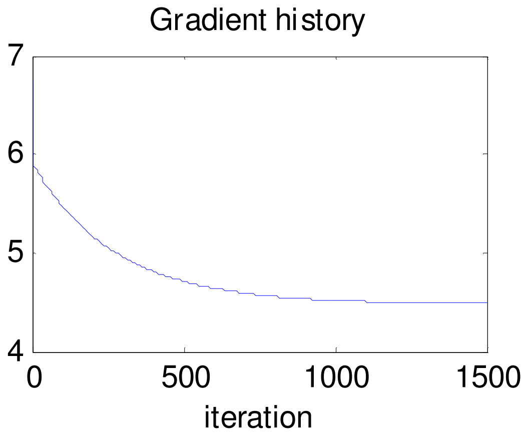

Fig.3

3. Predict profit

Use the final values for θ to make predictions on profits in areas of 35,000 and 70,000 people. Note that you need to scale the numbers properly !

Answer: For population = 35,000, predicted profit of 4519.77

For population = 70,000, predicted profit of 45342.45

4. Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1. In the next step there is code set up to calculate the 2-D array of $J(\theta)$ over a grid of values using the computeCost function. After that *surf* and *contour* commands will produce the surface and contour plots. These graphs show how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function has a global minimum. This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

Remarks

• Octave/MATLAB array indices start from one, not zero. If you're storing θ_0 and θ_1 in a vector called theta, the values will be theta(1) and theta(2).

• If you are seeing many errors at runtime, inspect your matrix operations to make sure that you're adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the command *size* will help you debug.

• By default, Octave/MATLAB interprets math operators to be matrix operators. This is a common source of size incompatibility errors. If you don't want matrix multiplication, you need to add the \dot" notation. For example, A*B does a matrix multiply, while A.*B does an element-wise multiplication.

PART 2

Objectives: Implement linear regression with multiple variables (features) and get to see it work on data.

Files included in this exercise:
ex1_multi.m - Octave/MATLAB script that steps you through the exercise (the main program)
ex1data2.txt - Dataset for linear regression with multiple variables
featureNormalize.m - Function to normalize features
computeCost.m - Cost function (you have already implemented in Part1)
gradientDescent.m - Gradient descent (you have already implemented in Part1)

In this lab work 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 ex1_multi.m script has been set up to help you step through this exercise.

1. Load the Data

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. Load the data into variables X (the first and the second columns) and y (the third column).
First 10 examples from the dataset:

```
 x = [2104 3], y = 399900
 x = [1600 3], y = 329900
 x = [2400 3], y = 369000
 x = [1416 2], y = 232000
 x = [3000 4], y = 539900
 x = [1985 4], y = 299900
 x = [1534 3], y = 314900
 x = [1427 3], y = 198999
 x = [1380 3], y = 212000
 x = [1494 3], y = 242500
```

2. Feature Normalization

The ex1_multi.m script will display 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.m* :

- 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. In Octave/MATLAB, you can use the *std* function to compute the standard deviation. For example, inside *featureNormalize.m*, the quantity X(:,1) contains all the values of x1 (house sizes) in the training set, so std(X(:,1)) computes the standard deviation of the house sizes. 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.

This normalization is an alternative to normalizing by making the absolute values < 1 (i.e. dividing by max_value-min_value.
After normalizing the features add an extra column of 1's corresponding to $x_0 = 1$.

Remark: 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 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. 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. If you have implemented the vectorized form of the code in *computeCost.m* and *gradientDescen.m* it will work for linear regression with any number of features. You can use size(X, 2) to find out how many features are present in the dataset.

4. Selecting learning rates

Now, you will try different learning rates for the dataset and find a learning rate that makes the cost *J* converging quickly, for example try for *alpha*=[0.01 0.03 0.1 0.3]. You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the *J* cost curve.

Remarks: If the learning rate is too large, the cost *J* can diverge and `blow up', resulting in values which are too large for computer calculations. In these situations, Octave/MATLAB will return NaNs. NaN stands for `not a number' and is often caused by undefined operations that involve +/- infinity. For example try with *alpha*=1.4 and *iter*=50. To compare how different learning rates affect convergence, it is helpful to plot *J* for several learning rates on the same figure such as Fig. 4. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value.

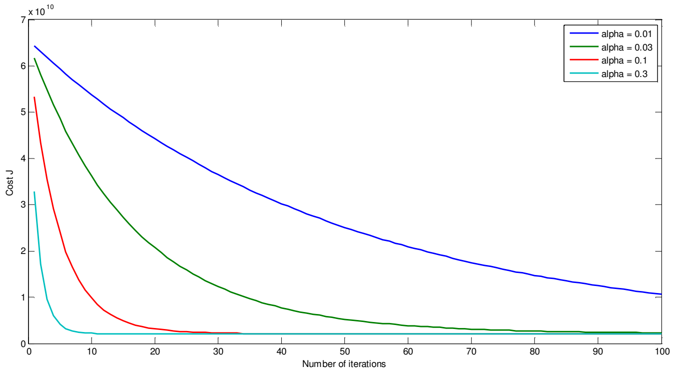

Fig. 4

5. Predict the house price

Using the best learning rate that you found, run gradient descent until convergence to find the final values of theta. Next, use this value of to predict the price of a house with 1650 square feet and 3 bedrooms. Don't forget to normalize your features when you make this prediction!
Answer: alpha=3; iter=50; price=293090 USD.