Programming Exercise 2: Linear regression with multiple variables

In this exercise, you will implement linear regression with multiple variables to predict the price of houses. Let's assume 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 a city. The first column is the size of house in square feet, the second column is the number of bedrooms and the third column is the price of the house.

For working on this exercise, you will be given a starter code. You may need to use `cd` command to change to the right directory.

Following files are either complete or partially complete. If it is partially complete, you will have to complete it.

```
ex1_multi.m                        %%complete
featureNormalize.m                 %% partially complete
computeCostMulti.m                 %% partially complete
gradientDescentMulti.m             %% partially complete
normalEqn.m                        %% partially complete
```

You will be running `ex1_multi.m`. This file is almost complete. The code in `ex1_multi.m` will set up the dataset for the problem and make calls to functions that you will have to write. However, you will have to modify functions in other files according to the instructions that will be provided later in this document.

You will have to modify `ex1_multi.m` when you are trying to find the price for 1,650 square feet, 3 bedroom house. Remember about feature scaling when doing so.

1. Feature Normalization

`ex1_multi.m` script will start by loading and displaying some values from the 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` to

- 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). 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. At the time that `featureNormalize.m` is called, the extra column of 1's corresponding to x_0 = 1 has not yet been added to X (see `ex1_multi.m` for details).

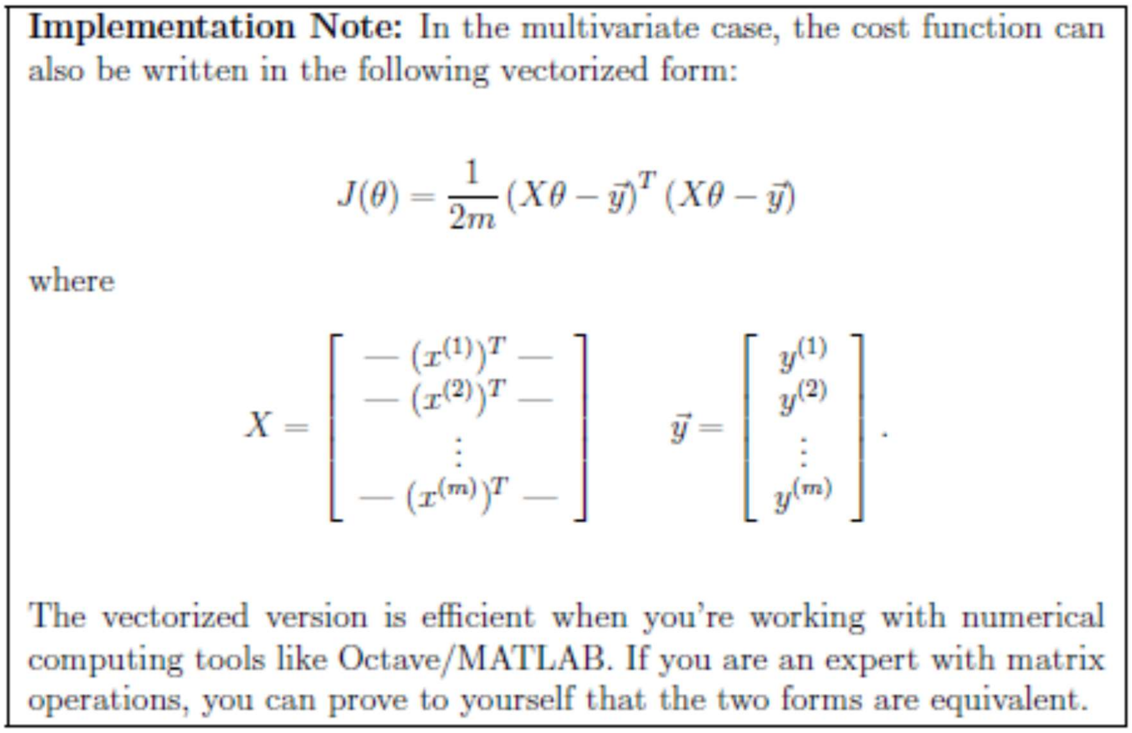
You will do this for all the features and **your code should work with datasets of all sizes** (any number of features / examples). Note that each column of the matrix `X` corresponds to one feature.

Note for Implementation
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. That is why it is important to store the values used for normalization—the *mean value* and the *standard deviation* used for computations.

2. 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.m` and `gradientDescentMulti.m` 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 '`size (X, 2)`' to find out how many features are present in the dataset.



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

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

The vectorized version is efficient when you're working with numerical computing tools like Octave/MATLAB. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

2(b)
Please also explain why

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

is equivalent to

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

2(c) Also explain why the first one is a better implementation?

<u>Gradient Descent Theory</u>

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent. Your goal is to find θ through gradient descent.
Update equations:

The objective of linear regression is to minimize the cost function

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

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

$h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6 + ..$

Remember 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 as follows:

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

However, for n features, we have to repeat ‘n’ times.

```
repeat until convergence: {
$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) . x_0^{(i)}$
$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) . x_1^{(i)}$
$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}$
...
}
```

(You have to update θ_j simultaneously 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 $J(\theta)$.

Selecting Learning Rates

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*** `ex1_multi.m` and changing the part of the code that sets the learning rate.

The next phase in `ex1_multi.m` will call your `gradientDescent.m` function and run gradient descent for about 50 iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector J. After the last iteration, the `ex1_multi.m` script plots the J values against the number of the iterations.
If you picked a learning rate within a good range, your plot should look similar to **Figure 1**. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on).

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.

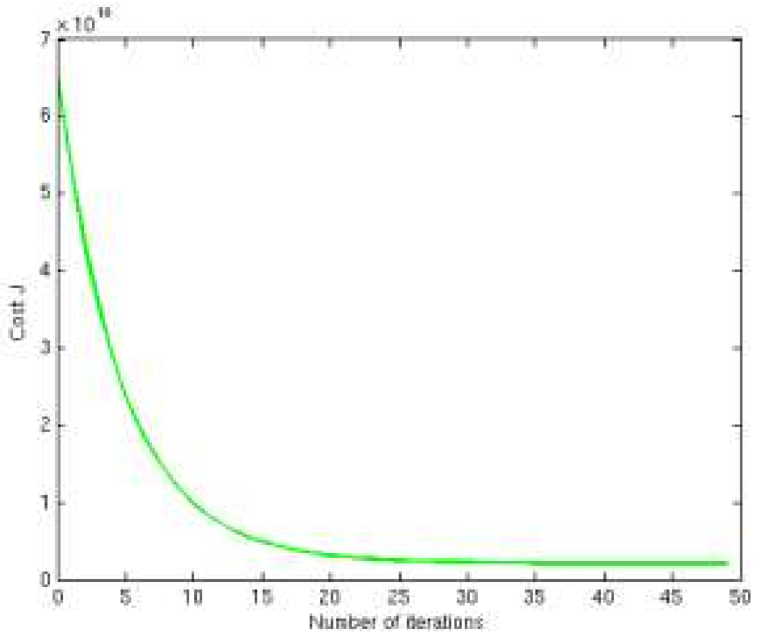


Figure 1

Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Using the best learning rate that you found, run the `ex1_multi.m` script to run gradient descent until convergence to find the final values of θ. Next, use this value of θ to predict the price of a house with area of 1650 square feet and 3 bedrooms. (You must modify `ex1_multi.m` code here). You will use value later to check your implementation of the normal equations**. Don't forget to normalize your features when you make this prediction!**

3. Normal Equations

Closed-form solution to linear regression is

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.
Complete the code in `normalEqn.m` to use the formula above to calculate θ. Remember that while

you don't need to scale your features, we still need to add a column of 1's to the X matrix to have an intercept term (θ_0).

The code in `ex1_multi.m` will add the column of 1's to X for you.

Now, once you have found θ using this method, use it to make a price prediction for a 1650-square-foot house with 3 bedrooms. You should find that this gives the same predicted price as the value you obtained using the model fit with gradient descent.

Distribution of points

Finish in the class, for 100% points.
Submit by midnight today, for 92% of points.
Submit by midnight tomorrow, for 88% of points.
Submit by midnight day after tomorrow, for 84% of points.

Problem	**Points**
featureNormalize	5
Select Learning rate	5
Modify `ex1_multi.m` for 1650 square feet, 3 bed house	5
ComputeCostMulti	10
GradientDescentMulti	15
Explanation Question 2(b), 2(c)	2.5+2.5