Introduction to Machine Learning

Assignment 1

Important Instructions:

- This is an **individual assignment**. Each student must work alone, without help from any other person.
- Plagiarism is NOT allowed. Copying material from the web or from documents and submitting it for this assignment is plagiarism. The copied assignment will be marked ZERO.
- You need to write a report which will contain all your outputs and you also need to explain them properly in your report. (Be concise and precise in your answers)
- A bad presentation usually earns bad credits

Linear Regression:

In this exercise, you will implement linear regression and get to see it work on data.

Throughout the exercise, you will be using the scripts **ex1.m** and **ex1_multi.m**. 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.

1 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.m** script has already been set up to load this data for you.

1.1 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.m**, the dataset is loaded from the data file into the variables X and y:

```
data = load('ex1data1.txt');        % read comma separated data
X = data(:, 1); y = data(:, 2);
m = length(y);                      % number of training examples
```

Next, the script calls the plotData function to create a scatter plot of the data. Your job is to complete **plotData.m** to draw the plot; modify the file and fill in the following code:

```
plot(x, y, 'rx', 'MarkerSize', 10);        % Plot the data
ylabel('Profit in $10,000s');              % Set the y-axis label
xlabel('Population of City in 10,000s');   % Set the x-axis label
```

Now, when you continue to run **ex1.m**, our end result should look like Figure 1, with the same red "x" markers and axis labels.

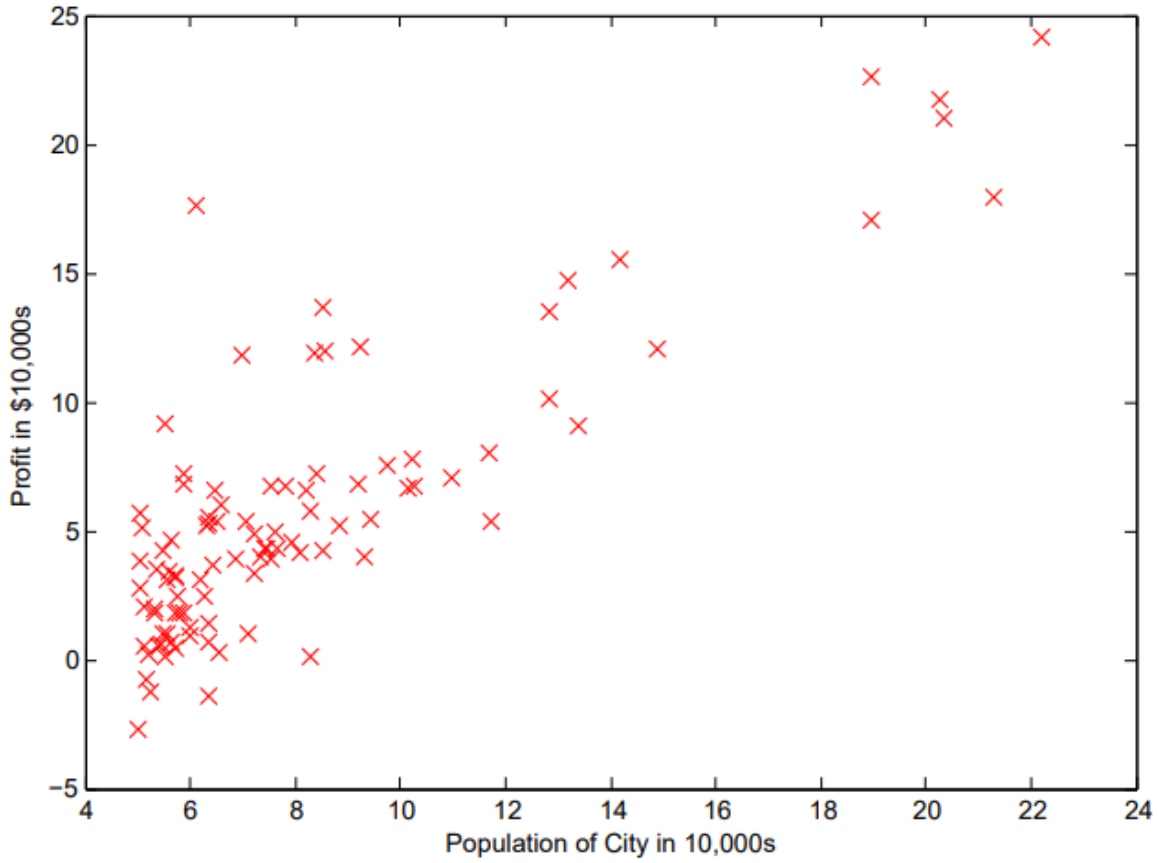


Figure 1: Scatter plot of training data

1.2 Gradient Descent

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

1.2.1 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 hθ(x) is given by the linear model

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

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

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

With each step of gradient descent, your parameters θj come closer to the optimal values that will achieve the lowest cost J(θ).

Implementation Note: We store each example as a row in the the X matrix in Octave/MATLAB. To take into account the intercept term (θ_0), we add an additional first column to X and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

1.2.2 Implementation

In **ex1.m**, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ0 intercept term. We also initialize the initial parameters to 0 and the learning rate alpha to 0.01.

```
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
theta = zeros(2, 1); % initialize fitting parameters

iterations = 1500;
alpha = 0.01;
```

1.2.3 Computing the cost J(θ)

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 file computeCost.m, 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, the next step in ex1.m 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.

1.2.4 Gradient descent

Next, you will implement gradient descent in the file gradientDescent.m. 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.
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 gradientDescent.m calls computeCost on every iteration and prints the cost. Assuming you have implemented gradient descent and computeCost correctly, your value of J(θ) should never increase, and should converge to a steady value by the end of the algorithm.
After you are finished, ex1.m will use your final parameters to plot the linear fit. The result should look something like Figure 2

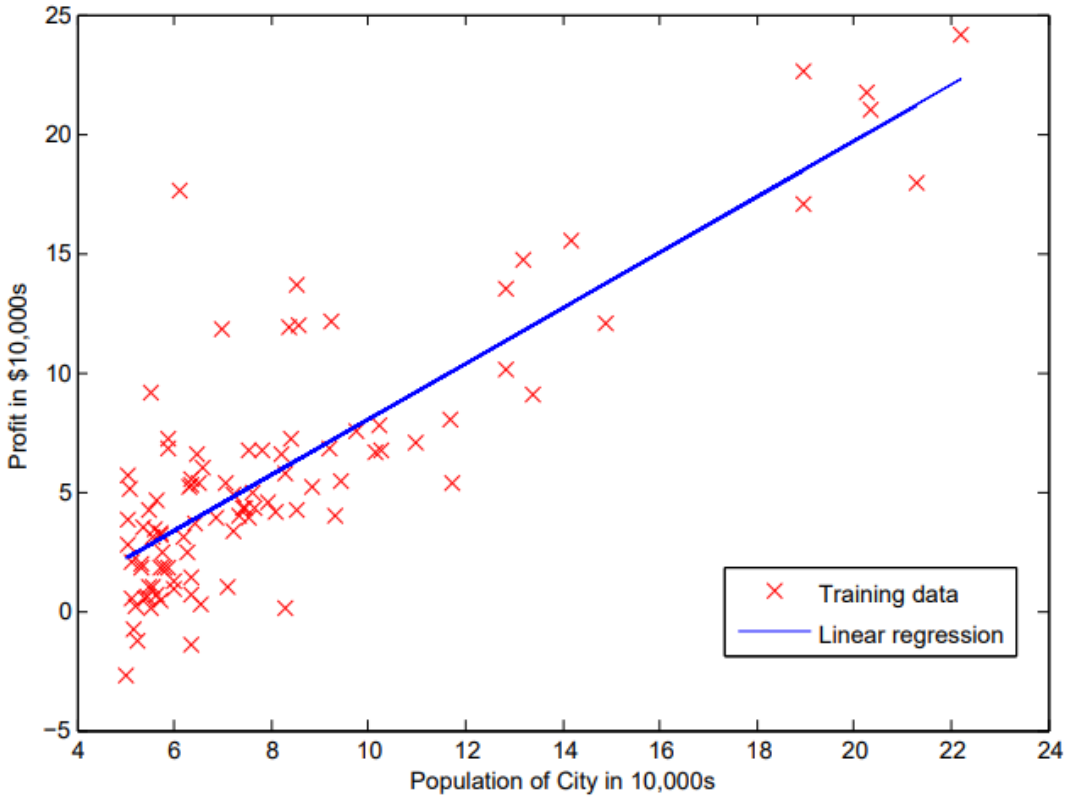


Figure 2: Training data with linear regression fit

1.3 Debugging

Here are some things to keep in mind as you implement gradient descent:

- Octave/MATLAB array indices start from one, not zero. If you're storing θ_0 and $\theta 1$ 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 size command 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 to specify this to Octave/MATLAB. For example, A*B does a matrix multiply, while A.*B does an element-wise multiplication.

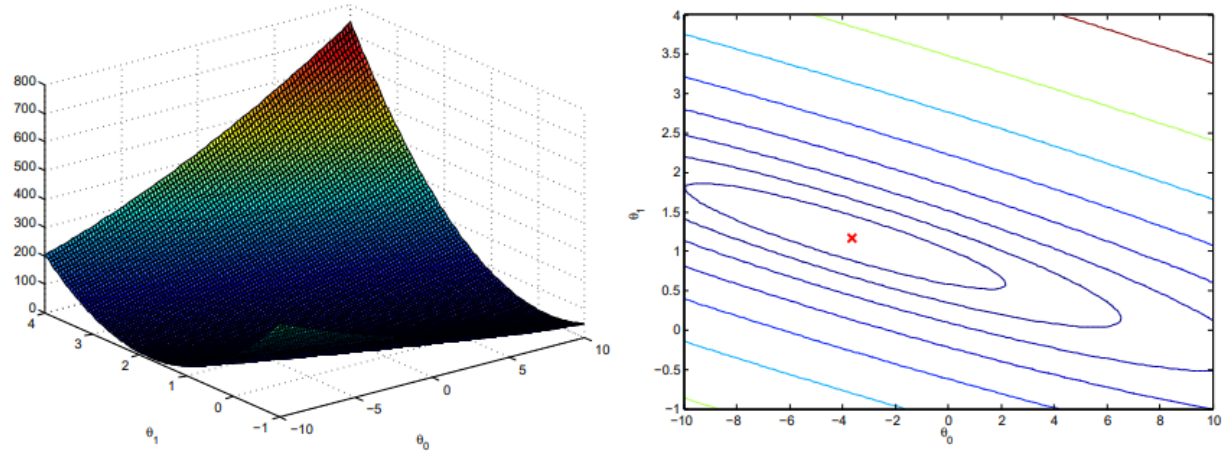
1.4 Visualizing J(θ)

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 step of ex1.m, there is code set up to calculate J(θ) over a grid of values using the computeCost function that you wrote.

```
% initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1:length(theta0_vals)
    for j = 1:length(theta1_vals)
       t = [theta0_vals(i); theta1_vals(j)];
       J_vals(i,j) = computeCost(x, y, t);
    end
end
```

After these lines are executed, you will have a 2-D array of J(θ) values. The script ex1.m will then use these values to produce surface and contour plots of J(θ) using the surf and contour commands. The plots should look something like Figure 3:



(a) Surface (b) Contour, showing minimum

Figure 3: Cost function $J(\theta)$

The purpose of these graphs is to show you that how J(θ) varies with changes in θ0 and θ1. The cost function J(θ) is bowl-shaped and has a global mininum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ0 and θ1, and each step of gradient descent moves closer to this point.

2 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.m** script has been set up to help you step through this exercise.

2.1 Feature Normalization

The **ex1_multi.m** 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.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 x_1 (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.

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.

2.2 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:

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

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

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.

2.2.1 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(θ) 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 look similar Figure 4. 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 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.

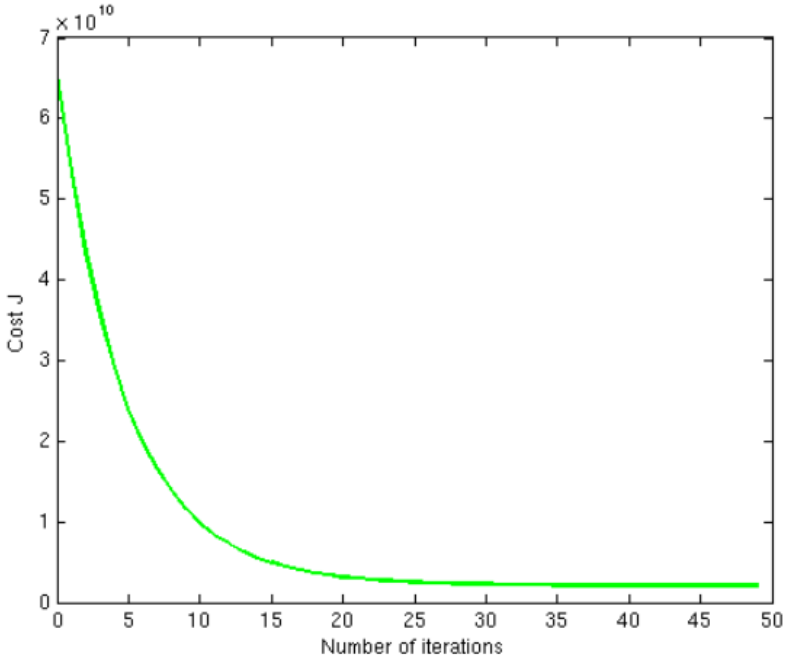


Figure 4: Convergence of gradient descent with an appropriate learning rate

Implementation Note: If your learning rate is too large, J(θ) can diverge and 'blow up', resulting in values which are too large for computer calculations. In these situations, Octave/MATLAB will tend to return NaNs. NaN stands for 'not a number' and is often caused by undefined operations that involve −∞ and +∞.

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 1650 square feet and 3 bedrooms. 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!

2.3 Normal Equations

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

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.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 gives the same predicted price as the value you obtained using the model fit with gradient descent (in Section 3.2.1).

3 Linear regression with multiple variables and regularization:

In this task, you will be following **ex1_multi_reg.m** for regularization. You need to make changes in the **costFunctionReg.m** and implement the following equation in it.

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

You need to change the **λ** (weight of regularization term) to see how it affects overfitting and underfitting and explain them in your report.

4 Using Different Cost functions:

Your task is to search online for different cost functions and implement them in the first part (Linear Regression with a single variable) and see what difference they make with the predictions. And explain them within your report. For the code you need to make a new "computeCost.m" file for each cost function with different naming conventions, also change the gradient file when using a different cost function. Compare them with each other and explain which one performs better.

(For the cost functions, you need to find two functions online and make one on your own. You also need to describe how they are working.)

Important: While changing the cost function you also need to find its gradient accordingly for gradient descent.

Good Luck