

[home](#) [articles](#) [quick answers](#) [discussions](#) [features](#) [community](#) [help](#)

Search for articles, questions, tips



Implementing Gradient Descent to Solve a Linear Regression Problem in Matlab



Ashkan Pourghasem, 20 Jul 2015

CPOL

★★★★★ 4.86 (20 votes)

Rate this:

Hands on tutorial of implementing batch gradient descent to solve a linear regression problem in Matlab

[Download Linear_Regression_With_One_Variable.zip - 1.9 KB](#)[Download Linear_Regression_With_Multiple_Variables.zip - 1.5 KB](#)

Introduction

This article is built around Andrew Ng's machine learning course on Coursera, I definitely recommend you check it out, it's very informative. In this article, I'll be focusing more on the programming part of the first section which is linear regression with one variable¹.

Artificial Intelligence is one of the most interesting fields in computer science nowadays, and one of its most useful areas is called machine learning.

Machine learning is definitely amongst the most broadly used forms of A.I. in today's technology, it is used for speech recognition, handwriting recognition, separating spam emails, showing you the most relevant websites when you search something on a search engine, and many other things.

Things You Need To Know

Knowing about matrices and vectors is necessary because matrix multiplication will be used in the code, but if you're not familiar with the concept, don't worry. I will explain each step as thoroughly as possible.

However, knowing a little bit of Matlab and being familiar with the concepts of coding is necessary for getting the most out of this article.

Concept

Now I could go ahead and tell you the theoretical definition of linear regression but because this is a hands-on tutorial, I'd rather explain it to you using an example:

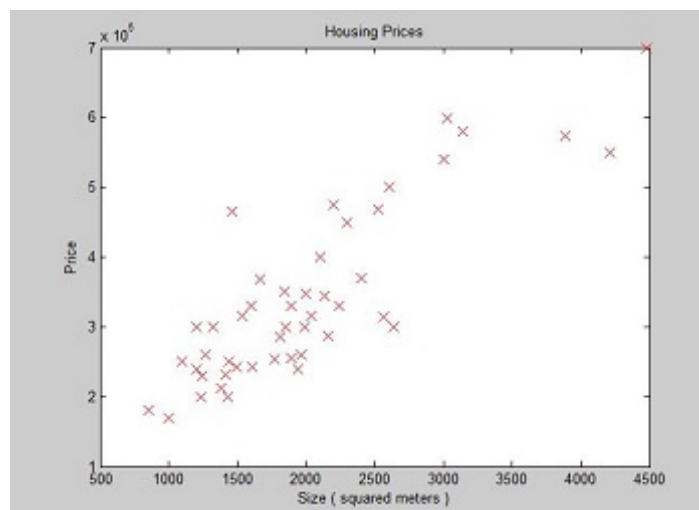
So assume that we have collected some data on the housing prices in our neighborhood, and this data contains the size of each house in squared meters and the price of the corresponding house, and we want to write a simple piece of software that takes this data and use it to train itself to predict the price of a house based on its size².

Now one of the ways of going about this problem is linear regression, which basically is an approach for finding a relationship between the dependent variable of the problem (in this case: price) and the independent variable of the problem (in this case: size). now in this problem we only have one independent variable or in other words we only have one feature of the house to deal with (which is the size of the house), in cases like this we'll use a form of linear regression which is called simple linear regression or univariate linear regression.

Linear Regression

The best way of learning how linear regression works is using an example:

First let's visualize our data set:



Now what we want to do is to find a straight line³, that is the best fit to this data, this line will be our hypothesis, let's define it's function like so :

$$h(x) = \theta_1 + \theta_2 x$$

- θ_1 is the intercept of our line
- θ_2 is the slope of our line
- x is the size of the house
- $h(x)$ is the predicted price of the house for the given size

Now in order to get our hypothesis as close as possible to the real value, we must get the right intercept and slope, and we can do that by minimizing our cost function. Now what is a cost function?

Cost Function

Our cost function (or squared error function) can be defined like this:

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

- θ is a two-dimensional vector containing our θ_1 and θ_2
- m is the count of our training examples
- x_i is the size of the i^{th} training example in our data set
- y_i is the price of the i^{th} training example in our data set

Now let me explain to you what this function is, first of all, we see the $(1/2m)$ in the beginning, this is just for the sake of making the derivation process a little easier⁴ so don't worry about that.

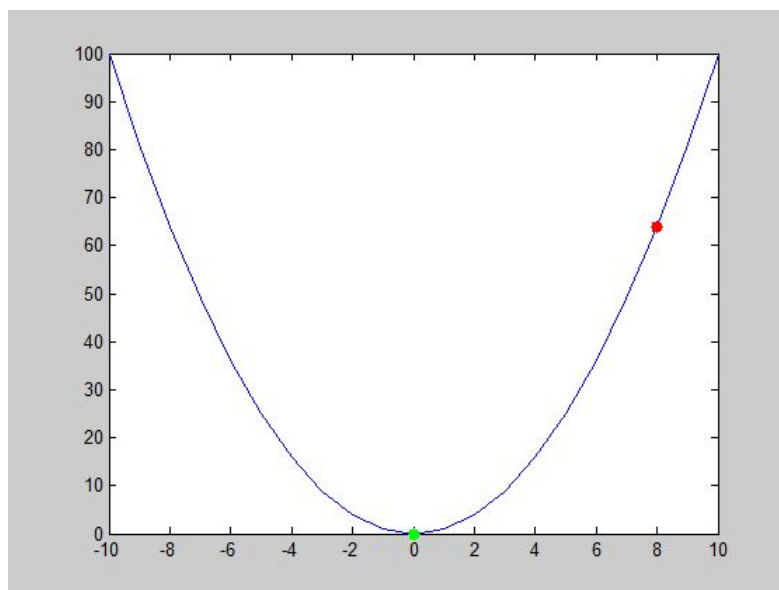
Next, we see the $(h_{\theta}(x_i) - y_i)$ now if you remember, $h(x)$ was the predicted price for the given size (x), and this part of the function is just calculating the difference between the real price of the i^{th} training example and the predicted price of the i^{th} training example (with respect to θ) so it's actually calculating the error of the prediction, the reason that we are squaring this value is to remove the sign before this error because it can be both positive and negative, and lastly the summation is for summing the errors for all the samples in our training set.

So the cost function is actually calculating how much our predictions were off, therefore the key to solving this problem is to minimize our cost function, in other words make it as small as possible, and in order to do that we have to select the right values for our θ_1 and θ_2 .

It turns out that one of the ways to do that is by using an algorithm called gradient descent.

Gradient Descent

To explain to you what gradient descent is and how it works, it helps to plot our cost function, so our cost function will probably look like something like that blue line:



Now that red circle is where you could end up depending on your data set and initial theta (2D vector containing intercept and slope), and that green circle is where you want to end up, because if you remember our cost function's job is to calculate how off our prediction is, so the minimum value for this function means getting the most accurate predictions. But the question remains, how do we get to that green circle?

Well, it turns out there is a useful algorithm called gradient descent, and this algorithm basically takes multiple steps towards the local minimum of a function, and it converges upon finding it.

So first let's see what gradient descent looks like:

Repeat until convergence {

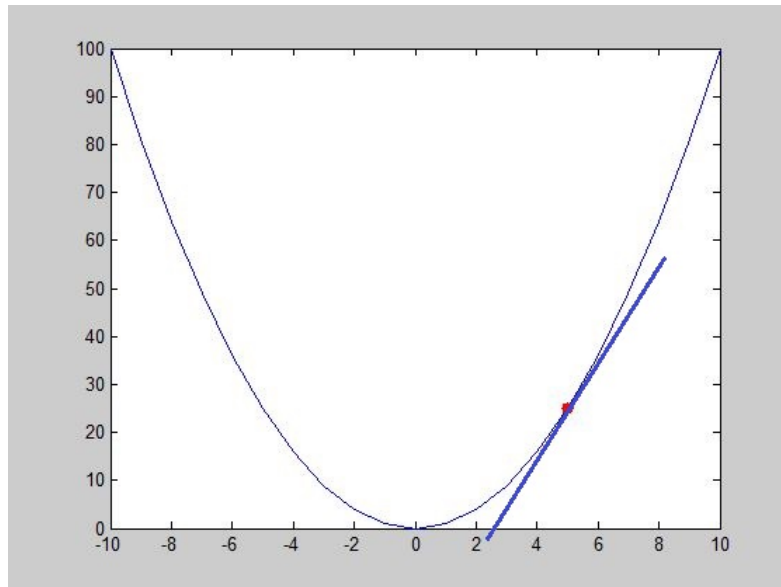
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

Now let's explain how it works, the formula seems to be decreasing θ_j by some value that is the partial derivative of our cost function of theta (with respect to θ_j) multiplied by alpha (which is something called a learning rate), and then assigning the result to θ_j . But this is very confusing and complicated so I'll take each element of this formula individually and explain it to you.

First of all, you should know that when I say parameters I mean our vector of theta which is a 2D vector that contains the intercept and slope of our hypothesis line.

Now let's start with the partial derivative of our cost function of theta (with respect to θ_j), that partial derivative is actually the slope of the line tangent to our cost function of theta, which looks like something like this:



In the picture above, that red circle is the value of cost function for a specific vector of theta, and the bold blue line is the tangent line of that circle and the slope of that line is the partial derivative of our cost function for a specific vector of theta. Now this is the smart thing about this algorithm that makes it useful for minimizing functions, because using the sign of the slope of this tangent line, it can decide whether to increase or decrease the value of our parameters (intercept and slope), and using the value of the slope of this tangent line, it can decide how much to change those values.

With the explanation above, it's now clear how this algorithm works. It simply takes each one of our parameters which is θ_j (j^{th} item in our vector of theta) and it increments/decrements it by the slope of the tangent line I explained above. If you repeat this process enough times, it modifies our parameters so that when we plug them in our cost function, we get a result as close to zero as possible, and that alpha is just a number called our learning rate which we'll choose to decide how big we want our gradient descent steps to be (We'll talk about it more in the programming section).

So I explained almost everything that there is to know in order to solve our problem, in the next section we will implement this algorithm in Matlab and solve the problem.

Implementing Gradient Descent

I thought it would be best to go through the codes file by file, so first I'm going to explain what each of these files does to give you an idea of how the program works.

main.m is the file that prepares all the data that is required for our algorithm, feeds this data to another function which actually has the implementation of the algorithm in it, and then shows us the results.

gradient.m is the file that has the gradient function and the implementation of gradient descent in it.

cost.m is a short and simple file that has a function that calculates the value of cost function with respect to its arguments.

main.m

So first of all, we load the data set that we are going to use to train our software.

Hide Copy Code

```
% Loading the dataset
dataSet = load('DataSet.txt');

% Storing the values in separate matrices
x = dataSet(:, 1);
y = dataSet(:, 2);
```

This block of code is basically loading the comma separated list of data stored in *DataSet.txt* to a variable called **dataSet**. **dataSet** is a matrix with 2 columns, the first column is sizes and the second column is prices. We're populating the variable **x** with the first column and the variable **y** with the second column.

Hide Copy Code

```
% Do you want feature normalization?
normalization = true;

% Applying mean normalization to our dataset
if (normalization)
    maxX = max(x);
    minX = min(x);
    x = (x - maxX) / (maxX - minX);
end
```

In this part, we're doing something called feature normalization. This is a trick that is used in machine learning a lot and it turns out it really helps gradient descent to reach convergence by normalizing the data, now what does that mean?

You see, often in machine learning, our data sets contain very large numbers, and this can cause many problems in many cases, even worse than that is when you have a set of features in our data set in the range of 1 and 5, and you have another set in the range of say 1000 and 200000, now that could cause many problems, so what do we do?

One of the ways of dealing with such problems is to use something called mean normalization. To apply mean normalization, you simply apply the formula below to every sample in your data set:

$$x_i = \frac{x_i - \max(x)}{\max(x) - \min(x)}$$

- x_i is the i^{th} training example
- $\max(x)$ is the biggest value in our data set
- $\min(x)$ is the smallest value in our data set

Now what we want to do is to apply this formula to every single one of values in our data set. Probably when I say those words, the first thing that comes to the mind of a programmer is some sort of loop, and you can do it using any kind of loop but Matlab supports vectors and matrices out of the box, so we can use them for a cleaner and more efficient code (which is what I did in the above code). Now let me explain how that code works⁵.

So the code is:

```
x = (x - maxX) / (maxX - minX);
```

[Hide](#) [Copy Code](#)

The variable **x** in the code above is a $n \times 1$ matrix that contains all of our house sizes, and the $\max()$ function simply finds the biggest value in that matrix, when we subtract a number from a matrix, the result is another matrix and the values within that matrix look like this:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \end{bmatrix} - c = \begin{bmatrix} x_1 - c \\ x_2 - c \\ x_3 - c \\ \dots \end{bmatrix}$$

Then we are dividing this matrix by another number which is the biggest value in our original vector $\max(x)$ minus the smallest value in our original vector $\min(x)$, the procedure of dividing a matrix by a number is exactly the same as the subtraction of them, but instead of subtracting we divide each entry by that number. One mistake that I have made in the past was normalizing the y matrix, you shouldn't normalize the dependent variable except in special cases and when you really know what you're doing.

You might be wondering why I stored the min and max of x in variables before using them, that's because later on we are gonna have to normalize the input value in order to get the correct prediction.

So now we have a variable called x that contain the normalized values of our data set and y that contain the prices of our houses.

```
% Adding a column of ones to the beginning of the 'x' matrix
x = [ones(length(x), 1) x];

% Plotting the dataset
figure;
plot(x(:, 2), y, 'rx', 'MarkerSize', 10);
xlabel('Size ( squared meters )');
ylabel('Price');
title('Housing Prices');
```

[Hide](#) [Copy Code](#)

**** If you are not familiar with the plot function, don't worry, it is not essential and is used only for showing our work and debugging ****

The first section of this code is just adding a column of ones to our x matrix, x is now a $n \times 2$ matrix with a first column of ones, later on I will explain why I did this.

In the second section, we are just plotting our data set to see what we're dealing with (essentially a debugging code).

```
% Running gradient descent on the data

% 'x' is our input matrix
% 'y' is our output matrix
% 'parameters' is a matrix containing our initial theta and slope

parameters = [0; 0];

learningRate = 0.1;

repetition = 1500;

[parameters, costHistory] = gradient(x, y, parameters, learningRate, repetition);
```

[Hide](#) [Copy Code](#)

Now this is where it all happens, we are calling a function called **gradient** that runs gradient descent on our data based on the arguments we send it, and it is returning two things first, **parameters** which is a matrix that contains the intercept and slope of the line that fits our data set best, and the second one is another matrix containing the value of our cost function on each iteration of gradient descent to plot the cost function later (another debugging step).

OK, let's explain the arguments, x is the $n \times 2$ matrix containing a column of ones and another column of normalized house sizes, y is the $n \times 1$ matrix containing the normalized house prices, parameters is a 2×1 matrix containing our initial intercept and slope values of the hypothesis line, *learningRate* is the alpha in our gradient descent algorithm, and *repetition* is the number of times we are going to run gradient descent on the data.

There is no definitive way to come up with the best learning rate and repetition values, you just have to tinker with it to find the best values for your data set, although plotting the cost function can help you with this because it lets you see how good (or bad) the algorithm did its job.

```
% Plotting our final hypothesis
figure;
plot(min(x(:, 2)):max(x(:, 2)), parameters(1) + parameters(2) * (min(x(:, 2)):max(x(:, 2))));
hold on;
```

[Hide](#) [Copy Code](#)

****** At this point, the intercept and slope of the hypothesis line have been modified by the algorithm, and the resulting line is the best fit to our data. ******

This part is simply plotting our hypothesis line, it may look complicated on the first look, but it's pretty self-explanatory, I'll explain what the functions do and you just have to read through it to understand it, this is also another debugging step.

- *parameters(1)* is the intercept of our hypothesis line or θ_1
- *parameters(2)* is the slope of our hypothesis line or θ_2

[Hide](#) [Copy Code](#)

```
% Plotting the dataset on the same figure
plot(x(:, 2), y, 'rx', 'MarkerSize', 10);
```

This part is just plotting our data set on the same window that we plotted our hypothesis line.

[Hide](#) [Copy Code](#)

```
% Plotting our cost function on a different figure to see how we did
figure;
plot(costHistory, 1:repetition);
```

And this is just plotting our cost function on a different window to see how we did (debugging purposes).

[Hide](#) [Copy Code](#)

```
% Finally predicting the output of the provided input
input = 120;

if (normalization)
    input = (input - maxX) / (maxX - minX);
end

output = parameters(1) + parameters(2) * input;
disp(output);
```

And finally we are predicting a price for a specific size (120) and displaying the result. Of course, this is just a dummy data set and the number of training examples is way too low to make an accurate prediction. Please note that if you normalize you're features you're gonna have to normalize this input variable in order to get the correct prediction.

gradient.m

This is the actual function that runs gradient descent on our data and adjusts the parameters of our hypothesis line to make it the best fit for our training set.

So first, we are going to declare this function like so:

[Hide](#) [Copy Code](#)

```
function [ parameters, costHistory ] = gradient( x, y, parameters, learningRate, repetition )
```

In the code above, we are simply declaring a function called `gradient` that takes five parameters and returns two values. I've explained these before but we're going to go through them again real quick:

parameters is the first value returned by this function which is a 2×1 matrix, that contains the intercept and slope of our hypothesis line. notice that there is another variable called **parameters** within our arguments which is essentially the same as this variable with only one difference, the one in the function's arguments hasn't been modified by the algorithm yet, so it contains the initial intercept and slope which is, in this case zero, the variable that is being returned by the function is that same variable but now it's modified by the algorithm and it contains the intercept and slope of the line that is potentially the best fit to our data.

costHistory is the second value returned by our function which is a $n \times 1$ matrix, that will contain the value of our cost function on every iteration of gradient descent, this will come in handy later when we are trying to find the best learning rate for an specific data set, or when we simply want to find out if our algorithm has worked correctly.

x is a $n \times 2$ matrix containing a column of ones and another column of our independent variables, in this case, the size of our houses.

y is a $n \times 1$ matrix containing our dependent variables, in this case, the price of our houses.

learningRate and **repetition** are self-explanatory, I've also explained them in the last section.

[Hide](#) [Copy Code](#)

```
% Getting the length of our dataset
m = length(y);

% Creating a matrix of zeros for storing our cost function history
costHistory = zeros(repetition, 1);
```

In the above code, we are storing the number of our training examples in a variable called *m*, and we are also pre-allocating a $n \times 1$ matrix called **costHistory** which I explained earlier. Notice that we are using our **repetition** variable to determine how many values are going to be stored in this vector, now the reason for that is because on every iteration of gradient descent we are going to have different values for intercept and slope of our hypothesis line, therefore on every iteration we are going to have a different value for our cost function.

[Hide](#) [Copy Code](#)

```
% Running gradient descent
for i = 1:repetition

    % Calculating the transpose of our hypothesis
    h = (x * parameters - y)';

    % Updating the parameters
    parameters(1) = parameters(1) - learningRate * (1/m) * h * x(:, 1);
    parameters(2) = parameters(2) - learningRate * (1/m) * h * x(:, 2);

    % Keeping track of the cost function
```

```
costHistory(i) = cost(x, y, parameters);
end
```

Now we get to the main part of our code, which is implementing gradient descent. First, we are going to create a **for** loop which I think is pretty self-explanatory.

Inside the **for** loop is where it all happens, first let me explain what formulas we're using, so we said that the formula for gradient descent is this:

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

If we plug in the definition of our cost function and take into consideration that we have two parameters (**intercept** and **slope**), we end up with an algorithm like this:

Repeat until convergence {

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} \left(\frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2 \right)$$

$$\theta_2 \leftarrow \theta_2 - \alpha \frac{\partial}{\partial \theta_2} \left(\frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2 \right)$$

}

Now let's calculate those partial derivatives to make the implementation easier:

Repeat until convergence {

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h(x_i) - y_i)$$

$$\theta_2 \leftarrow \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^m (h(x_i) - y_i) \cdot x_i$$

}

There are a ton of ways to implement this, and I recommend using a method that you understand best, but the method I'll be using is perhaps the shortest and the most efficient, so let's explain it line by line, first we're going to start with this line:

```
h = (x * parameters - y)';
```

Hide Copy Code

The whole point of this line of code is to calculate that summation that is in both of our formulas, the best way to explain this is using pictures so first I'm going to show you what $x * parameters$ will look like:

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \dots & \dots \end{bmatrix} \times \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} \theta_1 + x_1 \theta_2 \\ \theta_1 + x_2 \theta_2 \\ \theta_1 + x_3 \theta_2 \\ \dots \end{bmatrix}$$

**** If you don't know how to do matrix multiplication, just do a quick search, it's very easy ****

So after multiplying these matrices, we are left with another matrix that contains our predicted price or $h(x)$ for each one of the sizes in our training set, BTW this is why I added that column of ones to this matrix in *main.m*. Now we are going to have to subtract the real prices or y from these:

$$\begin{bmatrix} \theta_1 + x_1 \theta_2 \\ \theta_1 + x_2 \theta_2 \\ \theta_1 + x_3 \theta_2 \\ \dots \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \end{bmatrix} = \begin{bmatrix} (\theta_1 + x_1 \theta_2) - y_1 \\ (\theta_1 + x_2 \theta_2) - y_2 \\ (\theta_1 + x_3 \theta_2) - y_3 \\ \dots \end{bmatrix}$$

Now we almost have that entire summation part of our algorithm in a matrix, but in order to proceed and do the rest of our calculation, we have to transpose this matrix which in this case basically means converting it into a vector, you'll understand why I did this later:

$$\begin{bmatrix} (\theta_1 + x_1 \theta_2) - y_1 \\ (\theta_1 + x_2 \theta_2) - y_2 \\ \dots \end{bmatrix}^T = [(\theta_1 + x_1 \theta_2) - y_1 \quad (\theta_1 + x_2 \theta_2) - y_2 \quad \dots]$$

Now with this vector saved in a variable called h , we are ready to move on to the next part of the code:

Hide Copy Code

```
parameters(1) = parameters(1) - learningRate * (1/m) * h * x(:, 1);
parameters(2) = parameters(2) - learningRate * (1/m) * h * x(:, 2);
```

- $parameters(1)$ is θ_1
- $parameters(2)$ is θ_2

This part is actually very easy to understand, but nevertheless I'm going to illustrate the whole process. Now if you look at the original formula for gradient descent, you'll notice that there is a slight difference between modifying θ_1 (the intercept) and θ_2 (the slope), at the end of modifying θ_2 there is another multiplication and it's a part of the summation, so with θ_2 we are basically going to have to multiply every one of the objects in our h variable by their corresponding size or x_i , and once again we are going to use matrix multiplication to do this. but if you read the code you'll realize that we are multiplying by x on both of the lines, but the first line is multiplying only by the first column of our x matrix, which is in itself another matrix that consists of ones, so that multiplication is not going to have any effect on the values in our h vector, but if you do the multiplication you'll realize that it's actually is summing all of the entries in our h vector, and giving us a scalar value (a number) to work with, which is exactly what we want. also this is why we transposed the variable h :

$$\begin{bmatrix} \theta_1 + x_1\theta_2 - y_1 & \dots \end{bmatrix} \times \begin{bmatrix} 1 \\ \dots \end{bmatrix} = \begin{bmatrix} (\theta_1 + x_1\theta_2 - y_1) + \dots \end{bmatrix}$$

However in the second line of the code, we are multiplying the modified slope (θ_2) by the second column of our x matrix, which contains the sizes of the houses in our dataset, and that's exactly what we want according to the original formula of gradient descent. That multiplication is going to look like this:

$$\begin{bmatrix} \theta_1 + x_1\theta_2 - y_1 & \dots \end{bmatrix} \times \begin{bmatrix} x_1 \\ \dots \end{bmatrix} = \begin{bmatrix} (\theta_1 + x_1\theta_2 - y_1)x_1 + \dots \end{bmatrix}$$

The result would be a scalar value (a number), and if you replace the $\theta_1 + x_i\theta_2$ which is our hypothesis function with $h(x_i)$, you'll realize that this result is exactly the same as that summation at the end of the original formula for modifying θ_2 in gradient descent.

Now that we have both of the summations, the rest is pretty easy, we are just multiplying the summations by $1/m$ and our **learningRate** and then subtracting them from their corresponding thetas.

Alright now we have modified the intercept and slope of our hypothesis line once, and we are a little closer to local minimum of our cost function, it's time to write the code that's necessary to keep track of the cost function for debugging and choosing the right value for the **learningRate** and **repetition**, by calculating the return value of the cost function for our recently modified parameters and saving them into one of the slots of our pre-allocated vector which we created earlier:

```
costHistory(i) = cost(x, y, parameters);
```

[Hide](#) [Copy Code](#)

This line is self-explanatory, the code for the cost function itself is explained in the next section.

cost.m

```
function [ cost ] = cost( x, y, parameters )

% Calculates the cost function

cost = (x * parameters - y)' * (x * parameters - y) / (2 * length(y));

end
```

[Hide](#) [Copy Code](#)

The code for this section is very short and straightforward, so let's get into it.

The first line is defining a function which is returning a variable called *cost*, which obviously is going to contain the value of cost function for a specific dataset and set of parameters (intercept and slope), and it's expecting three arguments including:

- **x**: a $n \times 2$ matrix that consists of a column of ones and another column containing the sizes of the houses
- **y**: a $n \times 1$ matrix that contains all of our house prices
- **parameters**: which is a 2×1 matrix containing the θ_1 or intercept and the θ_2 or slope of our hypothesis line

Now let's see the cost function again just to remind ourselves how it looked like:

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

Alright now let's explain the code, the code consists of three parts:

1. $(x * parameters - y)'$
2. $(x * parameters - y)$
3. $(2 * length(y))$

The calculation of the first and second part have been explained thoroughly in the last section, here we're just going to see what they look like and what is the product of them.

The first part looks like this:

$$\begin{bmatrix} (\theta_1 + x_1\theta_2) - y_1 & (\theta_1 + x_2\theta_2) - y_2 & \dots \end{bmatrix}$$

And the second part looks like this:

$$\begin{bmatrix} (\theta_1 + x_1\theta_2) - y_1 \\ (\theta_1 + x_2\theta_2) - y_2 \\ (\theta_1 + x_3\theta_2) - y_3 \\ \dots \end{bmatrix}$$

And the result of their multiplication is going to look like this:

$$[(\theta_1 + x_1\theta_2 - y_1)^2 + (\theta_1 + x_2\theta_2 - y_2)^2 + \dots]$$

Which (once again) is a scalar value (a number), and if you replace the $\theta_1 + x_i\theta_2$ which is our hypothesis function with $h(x_i)$, you'll realize that it's exactly the same as that summation in the original cost function.

We're lastly going to divide the whole thing by the third part of the code which is `2*length(y)` or double the number of values in our y matrix, and now we have the value of the cost function for those specific arguments.

Testing To See If It Works!

Ok, so now we are gonna actually test this program to see how well it works. First I'm gonna use another dummy data set that has values in it that are a little bit more predictable, just to make sure everything is working correctly, and then we are gonna move on to the housing prices data set.

the TestDataSet.txt is in the rar file attached to the article, and here is how it looks like:

[Hide](#) [Copy Code](#)

```
1,2
2,4
2,5
3,6
3,7
4,8
5,10
6,12
6,13
6,13
8,16
10,19
```

So every "y" is almost "x*2", now we are gonna feed this data to our program.

This data set doesn't really need normalizing but we are gonna try with and without normalization to make sure both approaches work.

One thing you have to note is that you are gonna need different values for the "learningRate" and "repetition" relevant to your data set and whether or not you're normalizing your features. You can find suitable values for these variables by experimenting and looking at the graph of your cost function, for example if you use large values for these you get Inf's (Infinities) as your thetas and therefore the program will fail.

After experimenting with this data set I found out that without normalizing the following values will work fine:

[Hide](#) [Copy Code](#)

```
learningRate = 0.01;
repetition = 1500;
```

And with normalizing you can use these values:

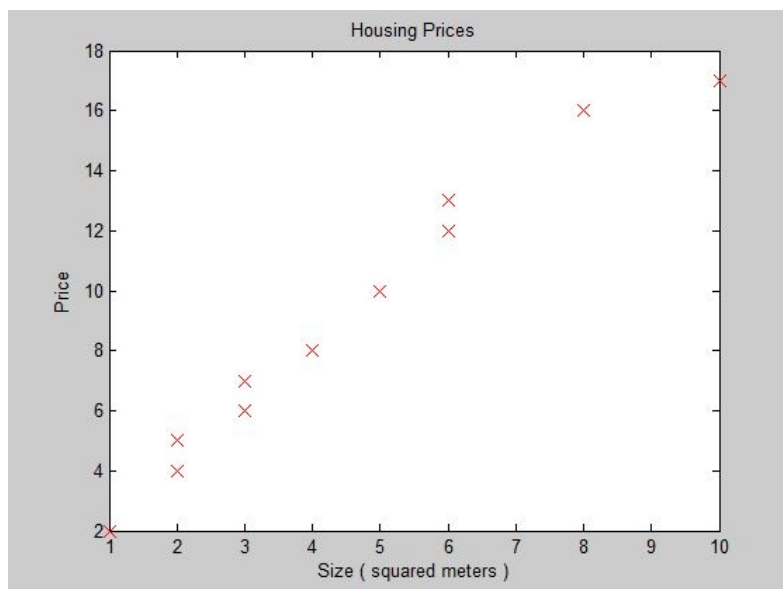
[Hide](#) [Copy Code](#)

```
learningRate = 1;
repetition = 100;
```

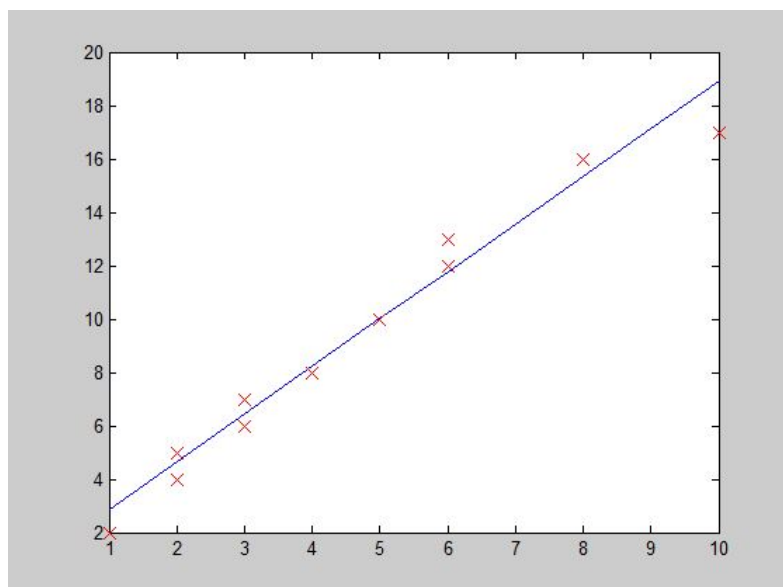
Now look at the drastic difference between the repetition values, you see that with normalization it takes the program one fifteenth of that repetition to converge, and that makes the program converge way faster.

Ok so these are the results without using feature normalization:

This is the plot of our data set:

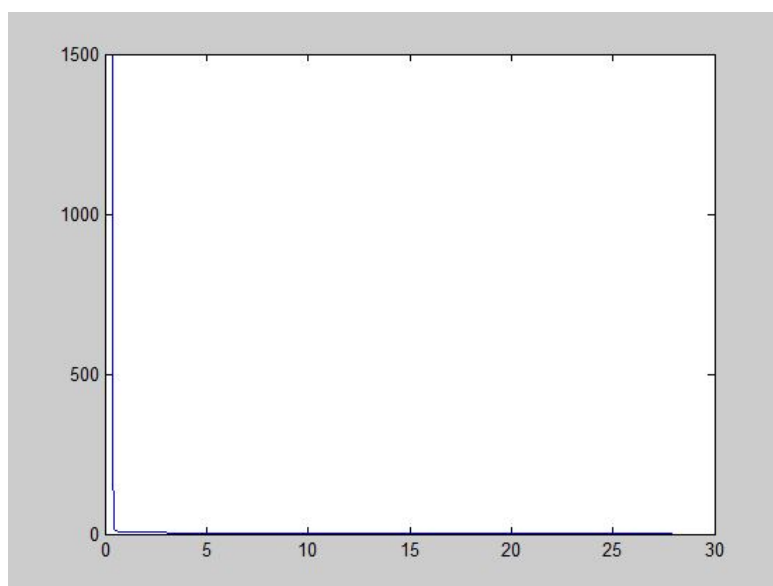


This is the hypothesis line that our program came up with:



Now as you can see it came up with a good hypothesis line that covers most of our samples and it's very likely to give us accurate predictions.

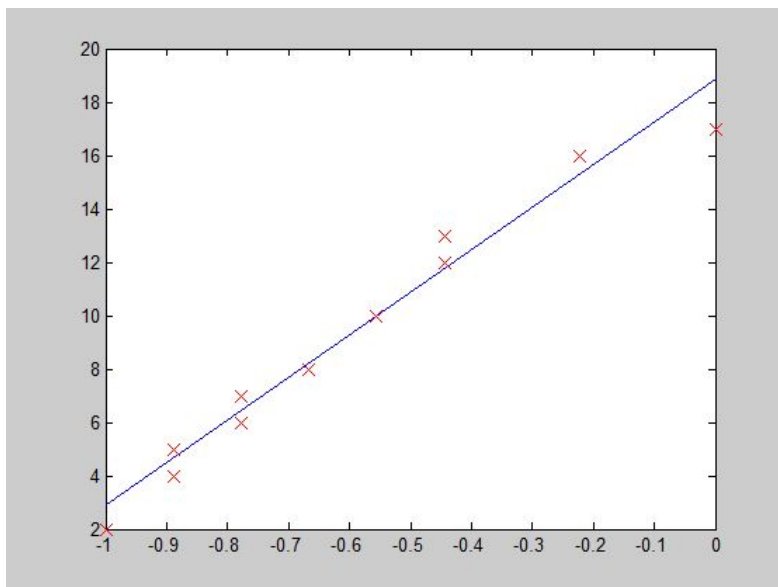
And this is the plot of the cost function:



Notice how it started out from the top and ended at almost zero.

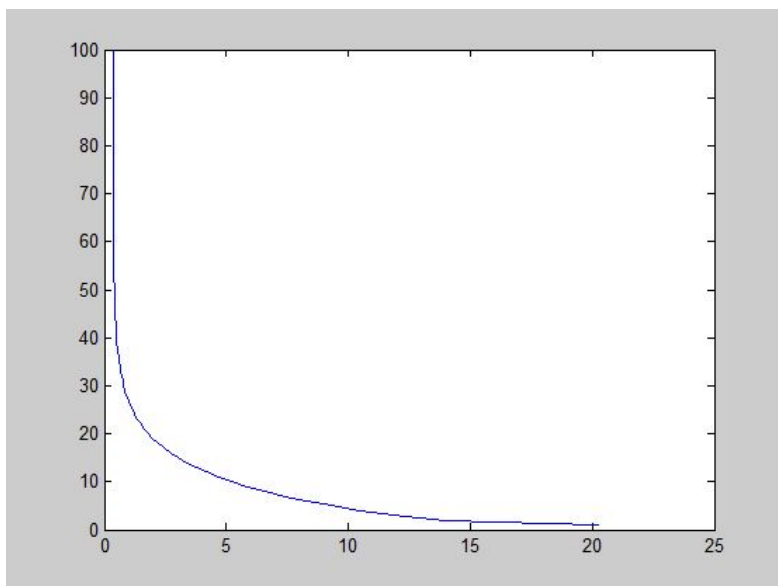
Now we are gonna give normalization a roll:

This is the hypothesis line that the program came up with for our normalized feature:



As you can see even though the algorithm ran 100 times instead of 1500 times, we almost got an identical hypothesis line.

And this is the plot of cost function with feature normalization:

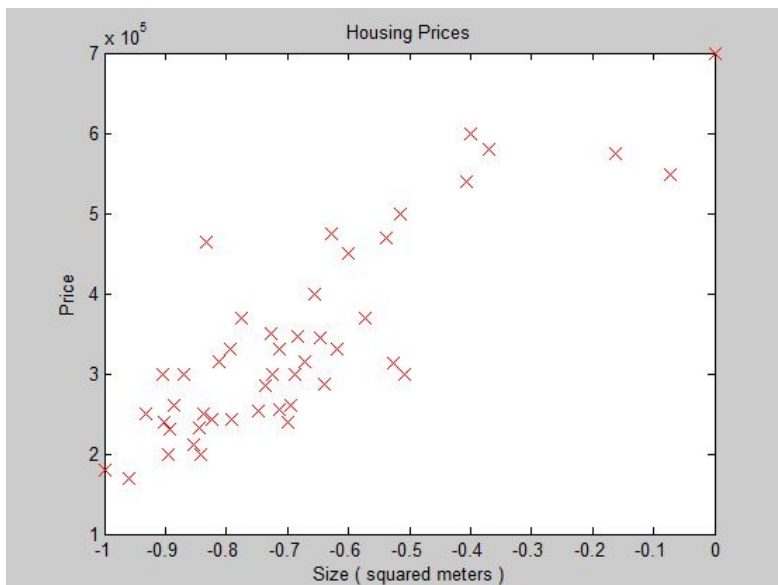


And this plot clearly demonstrates how good the program was able to minimize the cost function to almost zero.

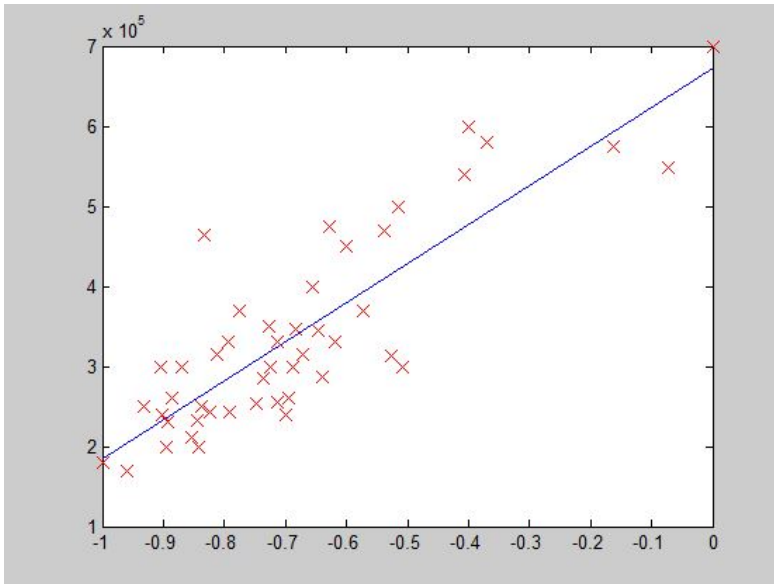
With both normalization on and off the program came up with ≈ 10 as a prediction for input = 5, which is what we were hoping to get.

Now with the housing prices data set, I highly recommend using feature normalization, I was able to get the cost almost zero without normalization but with very highly inefficient learningRate (≈ 0.0000001) and repetition (≈ 50000), but with normalization it converges with the learningRate set to 1.3 and the repetition set to 500.

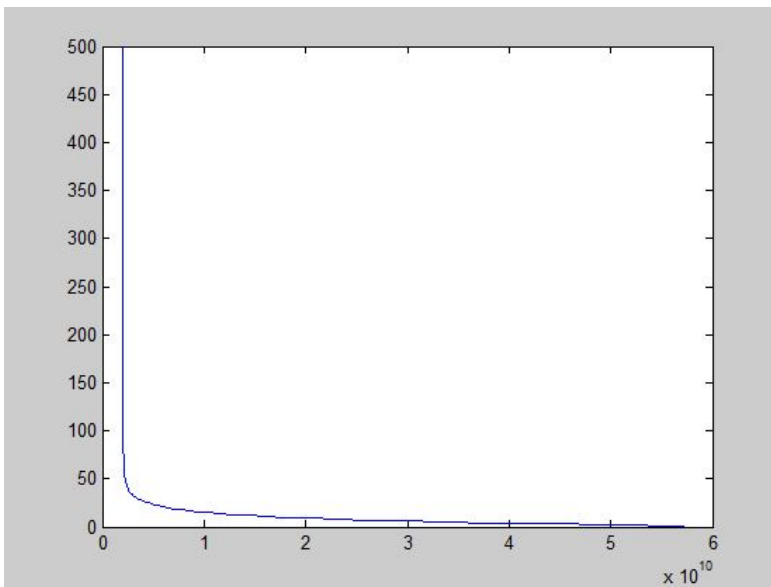
This is what the data looks like on a plot:



Now because the data in this data set is very outspread, linear regression is probably not such a good idea and we are not gonna get accurate results but we're still gonna try:



This is the hypothesis line that our program came up with, this looks like it would give us some average predictions, but in order to see how well the program did its job, and whether or not it can do any better, we have to look at the cost function plot:



Now as you can see at the end of algorithm cost is almost zero, so the line above is as good as it gets with linear regression, I plugged in some inputs that were present in the data set itself and in most cases it came up with a good prediction of the real price plus or minus ≈ 30 which is pretty good considering the data set wasn't well suited for linear regression.

Conclusion

That's all the information you are going to need to implement gradient descent in Matlab to solve a linear regression problem.

Please keep in mind that in this example we are using univariate linear regression with a very limited data set so the results are not going to be very accurate but if you apply these techniques and use a better data set, you are going to end up with pretty satisfying results.

Also I've implemented gradient descent to solve a multivariate linear regression problem in Matlab too and the link is in the attachments, it's very similar to univariate, so you can go through it if you want, this is actually my first article on this website, if I get good feedback, I may post articles about the multivariate code or other A.I. stuff.

I hope the content of this article was helpful for you, If you have any questions or suggestions, please do leave a comment.

References

1. The linear regression problem and the data set used in this article is also from Coursera.
2. The house can be present or not present in our training set.
3. The line doesn't have to be straight but because a straight line is the simplest case, we're going to go with it.
4. We need to get the partial derivative of this function for our main algorithm.
5. The first part of that code is just for being able to switch between normalizing and not normalizing easier, you can ignore it.

License


This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Share

About the Author



Ashkan Pourghasem

Student Freelance Web Developer
Iran (Islamic Republic of) 

Just a high school student who is very interested in programming and computer science, especially artificial intelligence.

You may also be interested in...

[An Algorithm for Weighted Linear Regression](#)[Visual COBOL New Release: Small point. Big deal](#)[Gradients made easy](#)[Sony's N – The Smart Wireless Headset that You Can Program](#)[10 Ways to Boost COBOL Application Development](#)[SAPrefs - Netscape-like Preferences Dialog](#)

Comments and Discussions

You must **Sign In** to use this message board.

[First](#) [Prev](#) [Next](#)

Question on the Linear Regression with One Variable 

Excellent explanation... Thanks a lot. 

Re: Excellent explanation... Thanks a lot. 

Why normalize between [-1,0] instead of [0,1]? 

Thank you- excellent explanation.. 


Re: Thank you- excellent explanation.. 

Awesome! 

Re: Awesome! 

My vote of 5 

Re: My vote of 5 

Re: My vote of 5 

Terrific Tutorial! 

Re: Terrific Tutorial! 

Nice Exposition 

Re: Nice Exposition 

Query on MATLAB code 

Re: Query on MATLAB code 

Linear regression 📌
Re: Linear regression 📌
NaN !!! 📌
Re: NaN !!! 📌
Re: NaN !!! 📌
Very nice, I did not understando why of that matrix of ones 📌
Re: Very nice, I did not understand why of that matrix of ones 📌
Erronious code for plotting hypothesis without normalisation 📌
Last Visit: 26-Feb-17 5:17 Last Update: 26-Feb-17 12:30 Refresh 1 2 Next »

 General  News  Suggestion  Question  Bug  Answer  Joke  Praise  Rant  Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

