

Programming Ex.1

Tutorials

Compute Cost Tutorial

This is a step-by-step tutorial for how to complete the computeCost function portion of ex1. You will still have to do some thinking, because I'll describe the implementation, but you have to turn it into Octave script commands. All the programming exercises in this course follow the same procedure: you are provided a starter code template for a function that you need to complete. You never have to start a new script file from scratch. This is a mediated implementation. You're only going to write a few simple lines of code.

With a text editor (NOT a word processor), open up the computeCost.m file. Scroll down until you find the "===== YOUR CODE HERE =====" section. Below this section is where you're going to add your lines of code. Just skip over the area that starts with the "%%" sign. Those are two-line comments.

With either these three lines of code to implement the equation on Page 5 of ex1.pdf. The first line of code will compute a vector "Y" containing all of the hypothesis values, one for each training example i.e., for each row of X. The hypothesis value for the problem i is simply the product of X(i) and theta. So your first line of code is...



```
x - error = (the difference between x and y)
```

The third line of code will compute the square of each of these error terms (squared error terms).  
An example of using element-wise exponentiation - try this in your workspace command line to see how it works.

```
1 x = [2 3]
2
3 x_0 = x/2
```

So, now you should compute the squares of the error terms



Next, here's an example of how the sum function works (try this from your command line)

```
x = n * np.dot(z, X)
```

Now, we'll finish the last two steps all in one line of code. You need to compute the sum of the `error_sq` vector, and scale the result (`trace0`) by  $1/(2 * m)$ . That completed sum is the cost value.

 coursera 

$$1 - 2 \times (\text{multiply } 1/(2^*n) \text{ times the sum of the error\_sqr vector})$$

$x = \text{theta} - \text{theta}_0 / \text{learning\_rate}$

That's it. Since you're never indexing by  $m$  or  $n$ , this solution works identically for both `gradientDescent` and `gradientDescentWithSGD`.

**Feature Normalization Tutorial**

There are a couple of methods to accomplish this. The method here is one I use that doesn't rely on automatic broadcasting or the `broadcast` or `repeat2` functions.

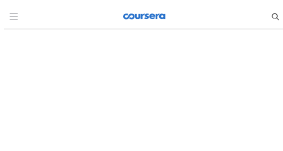
You can use the `mean2` and `std2` functions to get the mean and std deviation for each column of  $X$ . These are returned as row vectors ( $1 \times n$ ). Now you want to apply those values to each element in every row of the  $N$  matrix. One way to do this is to duplicate these vectors for each row in  $X$ , so they're the same size.

One method to do this is to create a column vector of ones (`ones(n, 1)`) and multiply it by the `mu` or `sigma` row vector ( $1 \times n$ ). One downside, `ones * 12` ( $1 \times n$ ) gives you a  $1n \times n$  matrix, and every row of the resulting matrix will be identical.

Now that  $X$ , `mu`, and `sigma` are all the same size, you can use elementwise operators to compute  $X$  normalizes.

Try these commands in your workspace:





```
1 K = [2 3 3; 4 5 6]
2
3 % creates a test matrix
4
5 m = ones(3)
6
7 % returns a row vector
8
9 sigma = ones(3)
10
11 % returns a row vector
12
13 n = size(X, 2)
14
15 % returns the number of rows in X
16
17 mu_matrix = ones(n, 2) * m
18
19 sigma_matrix = ones(n, 2) * sigma
```

Now you can subtract the  $\mu$  matrix from  $X$ , and divide element-wise by the sigma matrix, and arrive at  $Z$  (normalized).

You can do this much faster if you're using a Matlab or Octave version that supports automatic broadcasting. Then you can skip the "multiply by a column of 1's" part.

You can also use the built-in `repmat` functions. Be advised the built-in has a non-obvious syntax that I can never remember, and `repmat` runs rather slowly.

### Test Cases

#### computeCost:

```
>>computeCost([2 3 3; 4 5 6; 7 8 9], [0.1 0.2])
```

```
ans = 11.9450
```

```
----
```

```
>>computeCost([2 3 3; 4 5 6; 7 8 9], [0.1 0.2], [0.1 0.2])
```

```
ans = 7.0175
```

```
=====
```

#### gradientDescent:

```
% Test Case 1:
```

Coursera

```
1  z = theta * x; theta = gradientDescent([1 1; 1 1; 1 4; 1 1]; [1 4 2]; [0 0]'; 0.01, 1000);
2
3  % Show type in these variable names, to display the final results
4
5  z = theta * x
6
7  theta =
8
9  0.2168
10
11  -0.3723
12
13  z = 2.3415151
14
15  w = 5.40764
16
17  z = 2.341515088
18
19  w = 8.85426
```

For debugging, here are the final two theta values computed in the gradientDescent() for loop for this test case:

1	# First iteration
2	theta =
3	0.002086
4	0.002086
5	# second iteration
6	theta =
7	0.001675
8	0.001687
9	# third iteration
10	theta =
11	0.001675
12	0.001687
13	# Fourth iteration
14	theta =
15	0.001686
16	0.001686

The values can be inspected by adding the "breakpoint" command within your for loop. This puts the code in the debugger, where you can inspect the values. Use the "return" command to resume execution.

Test Case 2:

This test case is similar, but uses a non-zero initial theta value.

```
1 m = [theta_1,theta_0] = gradientDescent(1.5, 1, 2, [1 4]', [-5 -3]', 0.1, 100)
2 m = theta
3 theta =
4 -3.9896
5 -0.0128
6 -3.5445
7 -0.0117
8 -3.8853
9 -0.7539
10 -5.5475
11 -0.9861
12 -5.2298
13 -0.8773
14 -6.5265
15 -0.7861
16 -6.6489
17 -6.5317
```

formatFormalIn()

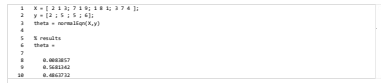
```
1 [m m] sigma = FeatureNormalizer([1 3 ; 1])
2 % result
3 m = 1
4 %
5 %
6 %
7 m = 2
8 sigma = 1
9 [m m] sigma = FeatureNormalizer(magic(3))
10 % result
11 m =
12 1.12188 -1.88888 0.27796
13 -0.75755 1.88888 0.75755
14 -0.27796 1.88888 -1.12188
15 %
16 %
17 sigma =
18 2.64518 0.88888 2.64518
19 %-----
20 [m m] sigma = FeatureNormalizer([ones(1,1); magic(3)])
21 % result
22 m =
23 1.12176 -1.88872 -1.12176
24 1.12176 -0.12173 0.47925
25 -0.12125 0.12124 0.88875
26 0.12125 1.12127 -0.88875
27 m =
28 1.12188 1.12188 1.12188
29 sigma =
30 2.64518 0.45127 3.0382
```

computeCostSub0



```
1 W = [[ 1 1 1; 7 1 1; 1 8 1; 1 7 8 ]];
2 w = [0 1 1; 1 1 1; 1 1 1];
3 [theta_2, J_hist] = gradientDescentMultiD(x, y, zeros(1,1), 0.01, 100);
4
5 % results
6
7 % theta =
8 theta =
9
10 0.22026
11 0.34248
12 0.31248
13
14 == 2.914512
15 err = -2.4200e
16
17 == 2.914512e01
18 err = 0.0027196
```

normates



The submit script, for all the programming assignments, does not report the line number and location of the error when it crashes. The following method can be used to make it do so which makes debugging easier.

Open `ex1\libsvm\libsvmConfiguration.m` and replace line:



```
1 print("I'll leave my agent alone, lol")
2
```

Output 20 with

```
1 printf("Error from FileIO: column has code %d\n", s.stack[1,1], FileIO.stack[1,1].name, s.stack[1,1].line );
2
```

That's the top. If Press by again later on again, instead of that, the bottom line will give the location and the number of the error. This change can be applied to all the programming assignments.

Note for OS X users:

If you are using OS X and get this error message when you run `ex1.m` and expect to see a plot figure:

```
1 gplot2<-set_theme(theme_minimal)+theme(plot.title=element_text(size=14),font.family="serif",font.size=12)
2
3 title(toupper("MACHINE LEARNING"))
4
```

... If you enter this command in the workspace console to change the default type:

```
1 return("MATHS")
```

How to check format of function arguments  
So that you may print the argument just by typing its name in the body of the function on a distinct line and call it later in OGCave.  
For example I may print the data argument in the "Compute cost for one variable" exercise by writing this in my computeCost.m file. Of course, it will fail because it is just random number, but it will show me the value of data.

```
1 function I = computeCost(X, y, theta)
2     m = length(X);
3     J = 0;
4     % TODO: Compute the cost function
5     J = 0; % I have added this line just to show that the argument you want to print doesn't have to be on the last line
6     end
7
```

Testing matrix operations in Octave

In our programming exercises, there are many complex matrix operations where I may not be clear what the result is. In this helps to check a few basic matrix and vector to test out my operations. The resources the following comments can be copied to a file to be used at any time for testing an operation.

The `bsxfun` is helpful for applying a function (limited to two arguments) in an element-wise fashion to rows of a matrix using a vector of source values. This is useful for feature normalization. An example you can enter at the Octave command line:

```
1  b=[x1 x2 x3 x4];
2  w=[1 1 1];
3  b=norm(b-w*w);
4  w=w-w;
5  w=w;
6  w=w;
7
```

If this case, the corresponding elements of  $w$  are subtracted from each row of  $Z$ . The `norm(b-w*w)` function is equivalent to computing  $\|b\|_2$  (other mathematical functions: `abs`, `sin`, `cos`).

If Octave  $\sim$  3.8.0 you can use `norm(w,w)` to abbreviate `norm(w,w,2)` (see <https://www.gnu.org/software/octave/doc/essentials/linear.html#Norms>).

1	$\mu = [x_1, x_2, x_3, x_4]$
2	$\sigma^2 = [x_1, x_2]$
3	$x = [x_1, x_2, x_3, x_4]$
4	$\mu = x$
5	$\mu = [x_1, x_2]$
6	$x = [x_1, x_2]$
7	$x = [x_1, x_2]$

A note regarding Feature Normalization when a feature is a constant - provided by a ML/DS student:

When I used the Feature Normalization routine we used in class I did not occur to me that some features of the training examples may have constant values, which means that the given vector has zeros for those features. Thus when dividing by sigma to normalize, the routine fails! I had in some code. This causes gradient descent to get lost wandering through a NaN wilderness, but never reporting why. The fix is simple. In FeatureNormalizer, after sigma is calculated but before the division takes place, insert



TA note: for the ML class exercises, you do not need this trick, because the scripts add the columns of bias units after the features are normalized. But for your use outside of the class exercises, this may be a useful technique.

The lecture notes "Week 2" under section Matrix Notation basically spells out one line solution to the problem.

When predicting prices using theta derived from gradient descent, do not forget to normalize input  $x$  or you'll get multimillion house value (wrong!)

I found that the line `"data = conread('ext-data2.txt')"` in `ext_multi.m` is not needed as we previously load this data via `"data = load('ext-data2.txt')"`.

Prior steps normalized  $X$ , this line sets  $X$  back to the original values. To have theta from gradient descent and from the normal equations to be comparable, run the normal equations using normalized features as well. Therefore do not reload  $X$ .

Comment: I think the point in reloading is to show that you actually get the same results even without doing anything with the data beforehand. Of course for this script it's not effective, but in a real application you would use only one of the approaches. Similar considerations would apply against feature normalization. Therefore do reload X.

