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 vectorized 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 lines that start with the '%' sign - those are instructive comments.

We'll write these three lines of code by inspecting the equation on Page 5 of ex1.pdf. The first line of code will compute a vector 'h' containing all of the hypothesis values - one for each training example (i.e. for each row of X). The hypothesis (also called the prediction) is simply the product of X and theta. So your first line of code is...

```
h = {multiply X and theta, in the proper order that the ....inner dimensions match}
```

Since X is size (m x n) and theta is size (n x 1), you arrange the order of operators so the result is size (m x 1).

The second line of code will compute the difference between the hypothesis and y - that's the error for each training example. Difference means subtract.

```
error = {the difference between h and y}
```

The third line of code will compute the square of each of those error terms (using element-wise exponentiation),

An example of using element-wise exponentiation - try this in your workspace command line so you see how it works.

```
v = [-2 3]
v_sqr = v.^2
```

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

```
error_sqr = {use what you have learned}
```

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

```
q = sum([1 2 3])
```

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

```
J = {multiply 1/(2*m) times the sum of the error_sqr vector}
```

That's it. If you run the ex1.m script, you should have the correct value for J. Then you should run one of the unit tests (available in the Forum).

Then you can run the submit script, and hopefully it will pass.

Be sure that every line of code ends with a semicolon. That will suppress the output of any values to the workspace. Leaving out the semicolons will surely make the grader unhappy.

Gradient Descent Tutorial - also applies to gradientDescentMulti() - includes test cases.

I use the vectorized method, hopefully you're comfortable with vector math. Using this method means you don't have to fuss with array indices, and your solution will automatically work for any number of features or training examples.

What follows is a vectorized implementation of the gradient descent equation on the bottom of Page 5 in ex1.pdf.

Reminder that 'm' is the number of training examples (the rows of X), and 'n' is the number of features (the columns of X). 'n' is also the size of the theta vector (n x 1).

Perform all of these steps within the provided for-loop from 1 to the number of iterations. Note that the code template provides you this for-loop - you only have to complete the body of the for-loop. The steps below go immediately below where the script template says "====== YOUR CODE HERE ======".

1 - The hypothesis is a vector, formed by multiplying the X matrix and the theta vector. X has size (m x n), and theta is (n x 1), so the product is (m x 1). That's good, because it's the same size as 'y'. Call this hypothesis vector 'h'.

2 - The "errors vector" is the difference between the 'h' vector and the 'y' vector.

3 - The change in theta (the "gradient") is the sum of the product of X and the "errors vector", scaled by alpha and 1/m. Since X is (m x n), and the error vector is (m x 1), and the result you want is the same size as theta (which is (n x 1)), you need to transpose X before you can multiply it by the error vector.

The vector multiplication automatically includes calculating the sum of the products.

When you're scaling by alpha and 1/m, be sure you use enough sets of parenthesis to get the factors correct.

4 - Subtract this "change in theta" from the original value of theta. A line of code like this will do it:

```
theta = theta - theta_change;
```

That's it. Since you're never indexing by m or n, this solution works identically for both gradientDescent() and gradientDescentMulti().

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 bsxfun() or repmat() functions.

You can use the mean() and sigma() functions to get the mean and std deviation for each column of X. These are returned as row vectors (1 x n)

Now you want to apply those values to each element in every row of the X 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 all-ones - size (m x 1) - and multiply it by the mu or sigma row vector (1 x n). Dimensionally, (m x 1) * (1 x n) gives you a (m x 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 element-wise operators to compute X_normalized.

Try these commands in your workspace:

```
X = [1 2 3; 4 5 6]

% creates a test matrix

mu = mean(X)

% returns a row vector

sigma = std(X)

% returns a row vector

m = size(X, 1)

% returns the number of rows in X

mu_matrix = ones(m, 1) * mu

sigma_matrix = ones(m, 1) * sigma
```

Now you can subtract the mu matrix from X, and divide element-wise by the sigma matrix, and arrive at X_normalized.

You can do this even easier 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 bsxfun() or repmat() functions. Be advised the bsxfun() has a non-obvious syntax that I can never remember, and repmat() runs rather slowly.

Test Cases

computeCost:

>>computeCost([1 2; 1 3; 1 4; 1 5], [7;6;5;4], [0.1;0.2])

ans = 11.9450

>>computeCost([1 2 3; 1 3 4; 1 4 5; 1 5 6], [7;6;5;4], [0.1;0.2;0.3])

ans = 7.0175

gradientDescent:

Test Case 1:

```
>>[theta J_hist] = gradientDescent([1 5; 1 2; 1 4; 1 5],[1 6 4 2]',[0 0]',0.01,1000);

% then type in these variable names, to display the final results

>>theta

theta =

5.2148

-0.5733

>>J_hist(1)

ans = 5.9794

>>J_hist(1000)

ans = 0.85426
```

For debugging, here are the first few theta values computed in the gradientDescent() for-loop for this test case:

```
% first iteration
theta =
  0.032500
  0.107500
% second iteration
theta =
  0.060375
  0.194887
% third iteration
theta =
  0.084476
  0.265867
% fourth iteration
theta =
  0.10550
  0.32346
```

The values can be inspected by adding the "keyboard" command within your for-loop. This exits the code to 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.

```
>> [theta J_hist] = gradientDescent([1 5; 1 2],[1 6]',[.5 .5]',0.1,10);
>> theta
theta =
1.70986
0.19229
>> J_hist
J_hist =
  5.8853
  5.7139
  5.5475
  5.3861
  5.2294
  5.0773
  4.9295
  4.7861
  4.6469
  4.5117
```

featureNormalize():

```
[Xn mu sigma] = featureNormalize([1 ; 2 ; 3])
% result
Xn =
  -1
  0
  1
mu =  2
sigma =  1
[Xn mu sigma] = featureNormalize(magic(3))
% result
Xn =
  1.13389 -1.00000 0.37796
  -0.75593 0.00000 0.75593
  -0.37796 1.00000 -1.13389
mu =
  5   5   5
sigma =
  2.6458   4.0000   2.6458
%--------------
[Xn mu sigma] = featureNormalize([-ones(1,3); magic(3)])
% results
Xn =
  -1.21725  -1.01472  -1.21725
  1.21725  -0.56373   0.67625
  -0.13525   0.33824   0.94675
  0.13525   1.24022  -0.40575
mu =
  3.5000   3.5000   3.5000
sigma =
  3.6968   4.4347   3.6968
```

computeCostMulti

```
X = [ 2 1 3; 7 1 9; 1 8 1; 3 7 4 ];
y = [2 ; 5 ; 5 ; 6];
theta_test = [0.4 ; 0.6 ; 0.8];
computeCostMulti( X, y, theta_test )
% result
ans =  5.2950
```

gradientDescentMulti

```
X = [ 2 1 3; 7 1 9; 1 8 1; 3 7 4 ];
y = [2 ; 5 ; 5 ; 6];
[theta J_hist] = gradientDescentMulti(X, y, zeros(3,1), 0.01, 100);

% results

>> theta
theta =

   0.23680
   0.56524
   0.31248

>> J_hist(1)
ans =  2.8299

>> J_hist(end)
ans =  0.0017196
```

normalEqn

```
X = [ 2 1 3; 7 1 9; 1 8 1; 3 7 4 ];
y = [2 ; 5 ; 5 ; 6];
theta = normalEqn(X,y)

% results
theta =

   0.0083857
   0.5681342
   0.4863732
```

Debugging Tip

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

Open ex1/lib/submitWithConfiguration.m and replace line:

```
fprintf('!! Please try again later.\n');
```

(around 28) with:

```
fprintf('Error from file:%s\nFunction:%s\nOn line:%d\n', e.stack(1,1).file,e.stack(1,1).name, e.sta
```

That top line says '!! Please try again later' on crash, instead of that, the bottom line will give the location and line 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:

```
gnuplot> set terminal aqua enhanced title "Figure 1" size 560 420  font "*,6" dashlength 1
                                                                   ^
         line 0: unknown or ambiguous terminal type; type just 'set terminal' for a list
```

... try entering this command in the workspace console to change the terminal type:

```
setenv("GNUTERM","qt")
```

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 submit() in Octave.

For example I may print the theta argument in the "Compute cost for one variable" exercise by writing this in my computeCost.m file. Of course, it will fail because 5 is just random number, but it will show me the value of theta:

```
function J = computeCost(X, y, theta)
    m = length(y);
    J = 0
    theta
    J = 5  % I have added this line just to show that the argument you want to print doesn't have t
end
```

Testing matrix operations in Octave

In our programming exercises, there are many complex matrix operations where it may not be clear what form the result is in. I find it helpful to create a few basic matrices and vectors to test out my operations. For instance the following commands can be copied to a file to be used at any time for testing an operation.

```
X = [1 2 3; 1 2 3; 1 2 3; 1 2 3; 1 5 6] % Make sure X has more rows than theta and isn't square
y = [1; 2; 3; 4; 5]
theta = [1; 1; 1]
```

With these basic matrices and vectors you can model most of the programming exercises. If you don't know what form specific operations in the exercises take, you can test it in the Octave shell.

One thing that got me was using formulas like theta' * x where x was a single row in X. All the notes show x as being a mX1 vector, but X(i,:) is a 1xm vector. Using the terminal, I figured out that I had to transpose x. It is very helpful.

Repeating previous operations in Octave

When using the great unit tests by Vinh, if your function doesn't work the first time -- after you to edit and save your function file, then in your Octave window - just type ctrl-p to back up to what you typed previously, then enter to run it. (once you've gone back, can use ctrl-n for next) (more info @ https://www.gnu.org/software/octave/doc/interpreter/Commands-For-History.html)

Warm up exercise

If you type "ex1.m" you will get an error - just use "ex1". Press 'Run' in Matlab editor.

Compute cost for one variable

theta is a matrix of size 2x1; first row is theta[0] and second one is theta[1] (I following index convention of videos here) Also fill arbitrary (non-zero) initial values to theta[0] and theta[1].

Gradient descent for one variable

See the 5th segment of Week 1 Video II ("Gradient Descent") for a key tip on simultaneous updates of theta.

Feature normalization

Use the zscore function to normalize: http://www.gnu.org/software/octave/doc/interpreter/Basic-Statistical-Functions.html#XREFzscore

repmat function can be used here.

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:

```
Z=[1 1 1; 2 2 2;];
v=[1 1 1];
bsxfun(@minus,Z,v);
ans =
    0   0   0
    1   1   1
```

In this case, the corresponding elements of v are subtracted from each row of Z. The minus(a,b) function is equivalent to computing (a-b).

(other mathematical functions: @plus, @rdivide)

In Octave >= 3.0.6 you can use broadcast feature to abbreviate:https://www.gnu.org/software/octave/doc/interpreter/Broadcasting.html#Broadcasting

```
Z=[1 1 1; 2 2 2;];
v=[1 1 1];
Z - v   % or Z .- v
ans =
   0   0   0
   1   1   1
```

A note regarding Feature Normalization when a feature is a constant: <provided by a ML-005 student>

When I used the feature normalization routine we used in class it did not occur to me that some features of the training examples may have constant values, which means that the sigma vector has zeroes for those features. Thus when I divide by sigma to normalize the matrix NaNs filled in some slots. This causes gradient descent to get lost wandering through a NaN wasteland, but never reporting why.The fix is easy. In featureNormalize, after sigma is calculated but before the division takes place, insert

```
sigma( sigma == 0 ) = 1;         % to keep away the NaN's and Inf's
```

Once this was done, gradient descent ran fine.

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

Gradient descent for multiple variables

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 one).

Normal Equations

I found that the line "data = csvread('ex1data2.txt');" in ex1_multi.m is not needed as we previously load this data via "data = load('ex1data2.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 close run the normal equations using normalized features as well. Therefor 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 its not effective, but in a real application you would use only one of the approaches. Similar considerations would argue against feature normalization. Therefore do reload X.