Tutorial Ex 1

ex1 Tutorial for computeCost

Tom MosherMentorWeek 2 · 2 years ago · Edited

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...

1

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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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.

1

error = {the difference between h and y}

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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.

1
2

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

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

1

error_sqr = {use what you have learned}

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

1

q = sum([1 2 3])

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
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.

1

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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
That's it. If you run the ex1.m script (by entering the command "ex1" in the console), you should have the correct value for J. Then you should test further by running the additional Test Cases (available via the Resources menu).
Important Note: You cannot test your computeCost() function by simply entering "computeCost" or "computeCost()" in the console. The function requires that you pass it three data parameters (X, y, and theta). The "ex1" script does this for you.
Then you can run the "submit" script, and hopefully it will pass.
Note: 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.
==========

==========

keywords: tutorial computeCost

Ex I Tutorial and test case for gradientDescent()

Tom MosherMentorWeek 2 · 2 years ago · Edited

Here is a tutorial on implementing gradientDescent() and gradientDescentMulti().

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:

```
1

theta = theta - theta_change;
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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

There is a test case below (or use this link):
https://www.coursera.org/learn/machine-learning/discussions/-m2ng_KQEeSUBCIAC9QURQ/replies/jCkbzfQsEeSkXCIAC4tJTg

==========
==========
Keywords: ex1 tutorial gradientdescent gradientdescentmulti gradient

ex1 tutorial for featureNormalize()

Tom MosherMentorWeek 2 · 2 years ago · Edited

This tutorial discusses how to implement the featureNormalize() function using vectorization.

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:

```
1
2
3
4
5
6
```

```
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
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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.

END