Here is a brief tutorial for the linearRegCostFunction().

We last did a linear regression exercise back in ex1, so start with these two tutorials for computeCost() and gradientDescent(). Since they're vectorized, they work equally well for any multiple-variable linear regression.

[computeCost tutorial](https://www.coursera.org/learn/machine-learning/discussions/t35D1xn3EeWA7CIAC5WDNQ)

[gradientDescent tutorial](https://www.coursera.org/learn/machine-learning/discussions/-m2ng_KQEeSUBCIAC9QURQ)

You only need the first three steps of the gradientDescent() tutorial, plus scaling by 1/m (ignore the 'alpha' variable, it is not used in this exercise). That's gives us the unregularized gradients. Since we let fmincg() perform gradient descent for us, we just have to compute the cost and gradient. We don't use a for-loop over the number of iterations, or use any learning rate. The fmincg() function does that for us.

A special consideration in this cost function is that it must work correctly when there is only one example (row) in the "X" matrix. If your code for the gradients uses the sum() function, there is a good chance you'll be computing the gradients incorrectly (by summing the columns instead of summing the rows). If you use vector math to compute the gradients, the sum() function is not needed.

So now you've got unregularized cost J, and unregularized gradients'grad'.

For the cost regularization:

* Set theta(1) to 0.
* Compute the sum of all of the theta values squared. One handy way to do this is sum(theta.^2). Since theta(1) has been forced to zero, it doesn't add to the regularization term.
* Now scale this value by lambda / (2\*m), and add it to the unregularized cost.

For the gradient regularization:

* The regularized gradient term is theta scaled by (lambda / m). Again, since theta(1) has been set to zero, it does not contribute to the regularization term.
* Add this vector to the unregularized portion.

That's it. Here is a test case for this function:

<https://www.coursera.org/learn/machine-learning/discussions/O25D0QykEeWZSyIAC5bWOg>

==============

There are a couple of different methods that work for the polyFeatures() function.

One is to use the bsxfun() function, with the @power operator, like this:

X\_poly = bsxfun(@power, vector1, vector2)

... where vector1 is a column vector of the feature values 'X', and vector2 is a row vector of exponents from 1 to 'p'.

Other options involve using the element-wise exponent operator '.^', and converting both X and the vector of exponent values into equal-sized matrices by multiplying each by a vectors of all-ones.

This thread is the tutorial for the learningCurve() function.

The thread is closed to comments (to prevent issues with the Forum software over time). If you have questions, please post them in a new thread.

--------------

**Note:** Almost all of the code you need for this function is provided in the code examples and hints in the learningCurve.m script.

Step 1) Use a for-loop to iterate over the length of the training set. The "Hint" in learningCurve.m gives you the code to use.

Step 2) Create a subset of the "X" matrix and the 'y' vector, using the elements 1 through 'i'. The first "Note" in learningCurve.m gives you the code to use. This causes the training set size to increase by one for each iteration through the training set. You will use this subset for training (Step 3) and measuring the training set error (Step 4).

Step 3) Use the trainLinearReg() function to learn the theta vector for the current size of training set (see page 6 of ex5.pdf).

Step 4) Then use your cost function to compute the training set error. Do not include regularization. Store the training set cost in error\_train(i).

Step 5) Then use your cost function to compute the validation set error, using Xval and yval. Do not include regularization. Do not create any subsets of the validation set. Store the validation set error in error\_val(i).

Tips:

* Use the lambda parameter - from the learningCurve() parameter list - every time you call trainLinearReg().
* **do not** set lambda = 0 inside the learningCurve() function. You are going to experiment with different lambda values in ex5.m, and the submit grader doesn't use lambda = 0. So do not hard-code lambda = 0 inside the learningCurve() function.
* When you compute the training set error and the validation set error, use your cost function with a zero for the lambda parameter. We want to measure the error in the hypothesis, without including any additional penalties for the theta values.
* When you run the "ex5" script, you may get some "divide by zero" warnings. These are expected and normal. fmincg() generates "divide by zero" warnings whenever the training set has only one or two examples. Do not worry about it.