Machine Learning	**Programming Assignment 1**
Comp540 Spring 2015	**due 23 January 2015 at 8 pm**

Introduction

In this two-part assignment, you will first implement linear regression and get experience working with it on a classical data set for predicting median home values at the census tract level in the Boston suburbs. Initially you will explore linear regression on one variable, and then you will extend your work to cover multiple variables. In the second part, you will implement regularized linear regression and use it to study models with different bias-variance properties. The code base `pa1.zip` for the assignment is on Owlspace under the Resources tab. When you unzip the archive, you will see the following two folders `part1` and `part2`. The material for the first part of the assignment is in `part1` and the files in this folder are shown in Table 1. Complete the problems in part 1 before moving on to part 2.

Part 1: Implementing linear regression

Name	**Edit?**	**Read?**	**Description**
computeCost.m	Yes	Yes	Cost function for the first problem
gradientDescent.m	Yes	Yes	Function that implements gradient descent for the first problem
computeCostMulti.m	Yes	Yes	Cost function for the second problem
gradientDescentMulti.m	Yes	Yes	Gradient descent for the second problem
featureNormalize.m	Yes	Yes	Function to normalize features
normalEqn.m	Yes	Yes	Function to compute the normal equations
ex1.m	Yes	Yes	Matlab script that will run your functions for the first problem (edit only where indicated)
ex1_multi.m	Yes	Yes	Matlab script that will run your functions for the second problem (edit only where indicated)
housing.data.txt	No	Yes	The Boston housing data set
housing.data.names	No	Yes	Description of the variables in the Boston housing data set
plotData.m	No	Yes	Function to visualize data for the first problem
pa1_2015.pdf	No	Yes	this document

Table 1: Files in folder `part1` for the first part of the assignment.

Problem 1: Linear regression with one variable (20 points)

You will implement linear regression with one variable to predict the median value of a home in a census tract in the Boston suburbs from the percentage of the population in the census tract that is of lower economic status. The file `housing.data.txt` contains the data for our linear regression problem. The thirteenth column is the percentage of the population of lower economic status and the fourteenth column is the median home value in a census tract (in $10000s). The `ex1.m` script has already been set up to load this data for you.

Plotting the data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two features to plot (percentage of population of lower economic status and median home value). Many other problems that you will encounter in real life are multi-dimensional and cannot be plotted on a 2-d plot. In `ex1.m`, the dataset is loaded from the data file into the variables `X` and `y`. You will see the plot in Figure 1 generated by `plotData.m`.

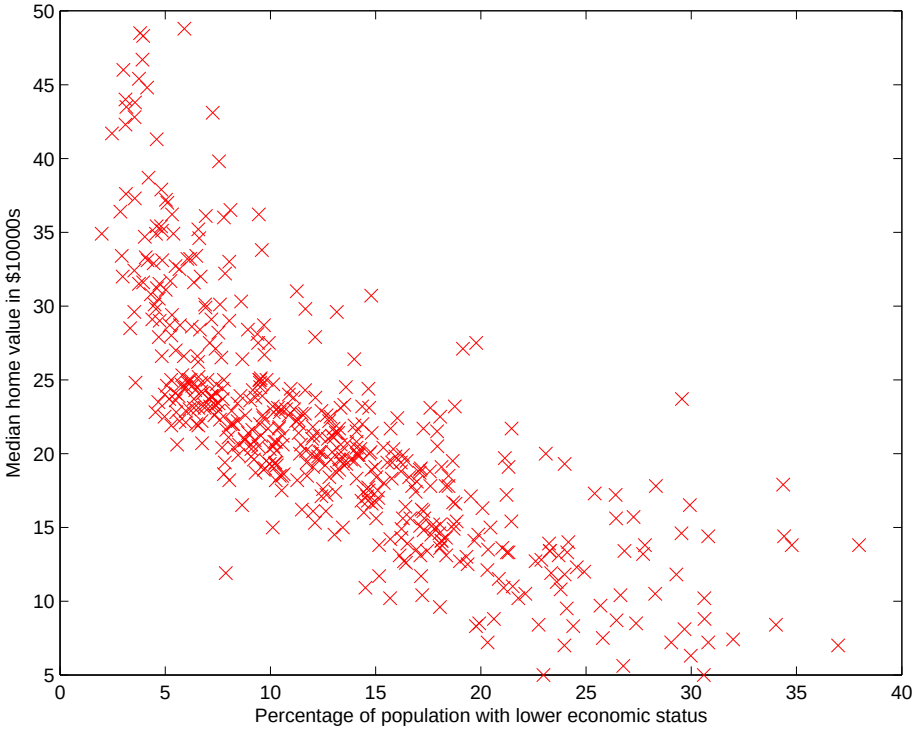

Figure 1: Scatter plot of training data

Gradient descent

You will implement gradient descent to fit the parameter θ to the housing data. Recall that the objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x$$

The gradient descent algorithm computes the value of θ to minimize the cost function $J(\theta)$. We will use batch gradient descent which performs the following update on each iteration.

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) {x_j}^{(i)}$$

where all the θ_j's are updated simultaneously. The parameter α is the learning rate. With each step of gradient descent, the parameters θ_j come closer to the optimal values that achieve the lowest cost $J(\theta)$. In `ex1.m`, we have already set the learning rate (0.01), number of iterations (1500) as well as the initial values for the parameter θ (all zeros).

We store each example as a row in the the `X` matrix in Matlab. To take into account the intercept term (θ_0), we add an additional first column to `X` and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

Computing the cost function $J(\theta)$ (5 points)

As you perform gradient descent to minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. You will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation. Your should complete the code in the file `computeCost.m`. Remember that the variables `X` and `y` are not scalar values, but matrices whose rows represent the examples from the training set.

Once you have completed the function, the next step in `ex1.m` will run `computeCost` once using θ initialized to zeros, and you will see the cost printed to the screen. You should expect to see a cost of approximately 296.07.

Implementing gradient descent (10 points)

Next, you will implement gradient descent in the file `gradientDescent.m`. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration. Recall that we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing `X` or `y`. A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. The starter code for `gradientDescent.m` calls `computeCost` on every iteration and prints the cost. Assuming you have implemented gradient descent and

`computeCost` correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. After you are finished, `ex1.m` will use your final parameters to plot the linear fit. The result should look something like Figure 2. Your final values for θ will also be used to make predictions on median home values for census tracts where the percentage of the population of lower economic status is 5% and 50%. Try to use code vectorization in Matlab to make your functions really compact.

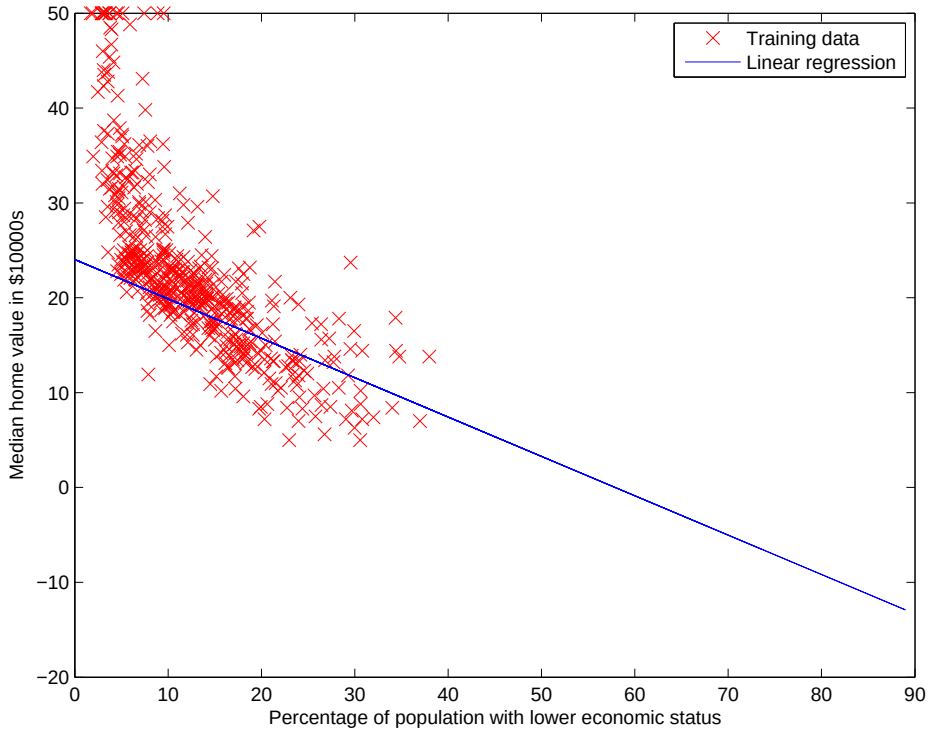

Figure 2: Fitting a linear model to the data in Figure 1

Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, we plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images. In `ex1.m`, we calculate $J(\theta)$ over a grid of (θ_0, θ_1) values using the `computeCost` function that you wrote. The 2-D array of $J(\theta)$ values is plotted using the surf and contour commands of Matlab. The plots should look something like Figure 3.

The purpose of these plots is to show you how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function is bowl-shaped and has a global mininum. This is easier to see in the contour plot than in the 3D surface plot. This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

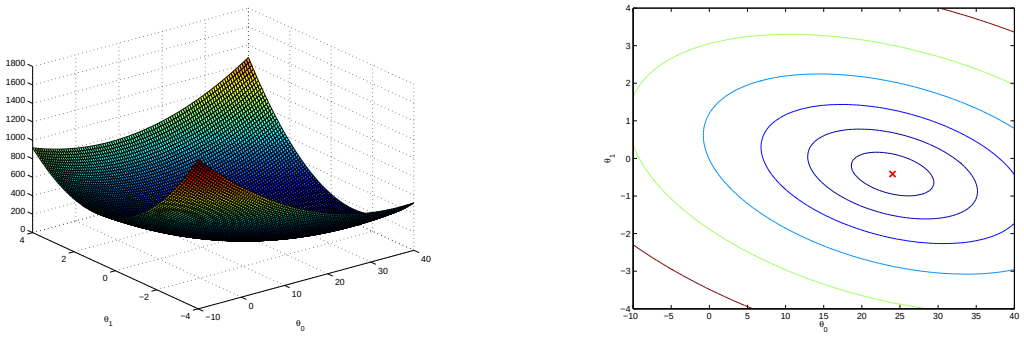

Figure 3: Surface and contour plot of cost function J

Making predictions on unseen data (5 points)

Your final parameter values for θ will now be used to make predictions on median home values in census tracts where the percentage of the population that is lower status is 5% and 50%. Complete the calculation in `ex1.m` in the indicated lines in that file. Now run the script `ex1.m` to see what the predictions are.

Problem 2: Linear regression with multiple variables (30 points)

Feature normalization (5 points)

The `ex1_multi.m` script will start by loading and displaying some values from the full Boston housing dataset with thirteen features of census tracts that are believed to be predictive of the median home price in the tract (see `housing.names.txt` for a full description of these features). By looking at the values, you will note that the values of some of the features are about 1000 times the values of others. When features differ by orders of magnitude, feature scaling becomes important to make gradient descent converge quickly. Your task here is to complete the code in `featureNormalize.m`. First, subtract the mean value of each feature from the dataset. Second, divide the feature values by their respective standard deviations. The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within two standard deviations of the mean). You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix `X` corresponds to one feature. When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameter θ, we want to predict the median home prices for new census tracts. Given the thirteen characteristics `x` of a new census tract, we must first normalize `x` using the mean and standard deviations that we had previously computed from the training set. Then, we take the dot product of `x` (with a 1 prepended (to account for the intercept term) with the parameter vector θ to make a prediction.

Note that in `ex1_multi.m`, when `featureNormalize.m` is called, the extra column of 1's has not yet been added to `X`.

Gradient descent (10 points)

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there are more features in the matrix `X`. The hypothesis function and the batch gradient descent update rule remain unchanged. You should complete the code in `computeCostMulti.m` and `gradientDescentMulti.m` to implement the cost function and gradient descent for linear regression with multiple variables. Make sure your code supports any number of features and that it is vectorized. You should see the cost $J(\theta)$ converge as shown in Figure 4.

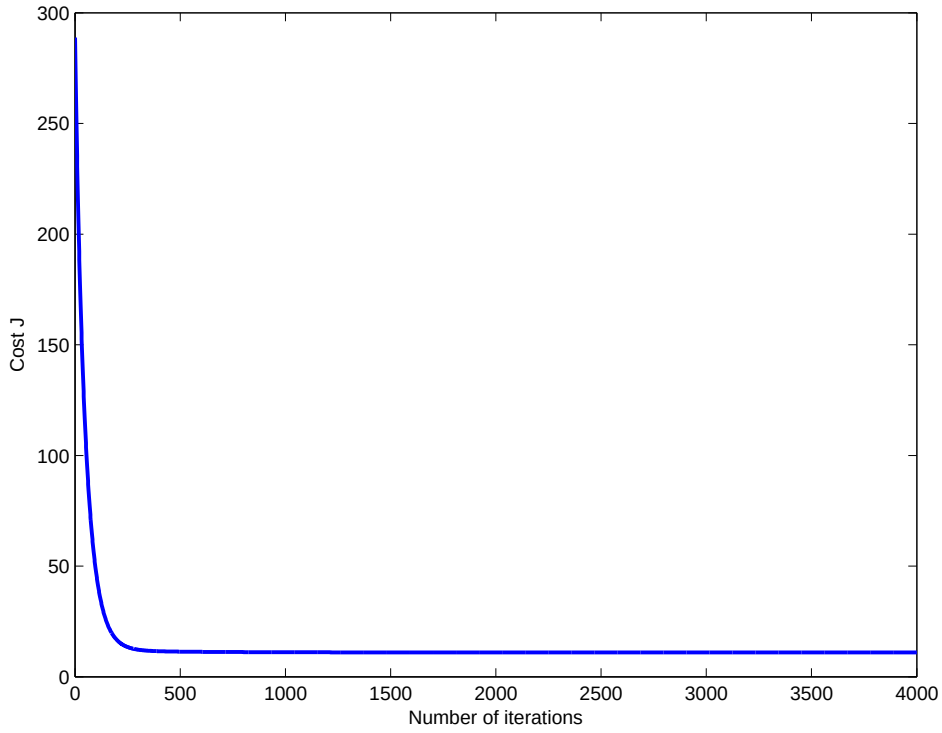

Figure 4: Convergence of gradient descent

Making predictions on unseen data (5 points)

Your final parameter values for θ will now be used to make predictions on median home values for an average census tract, characterized by average values for all the thirteen features. Complete the calculation in `ex1_multi.m` in the indicated lines in that file. Now run the script `ex1_multi.m` to see what the prediction of median home value for an average tract is. Remember to scale the features correctly for this prediction.

Normal equations (5 points)

The closed form solution for θ is

$$\theta = (X^T X)^{-1} X^T y$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no loop until convergence as in gradient descent. Complete the code in `normalEqn.m` to use the formula above to calculate θ. Now make a prediction for the average census tract (same example as in the previous problem). Do the predictions match up? Remember that while you do not need to scale your features, you still need to add a 1 to the example to have an intercept term (θ_0).

Exploring convergence of gradient descent (5 points)

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying `ex1_multi.m` and changing the part of the code that sets the learning rate (the assignment to the variable `num_iters`). The next phase in `ex1_multi.m` will call your `gradientDescent.m` function and run gradient descent for `num_iter` iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector `J`. After the last iteration, the `ex1_multi.m` script plots the `J` values against the number of the iterations. If you picked a learning rate within a good range, your plotshould look similar to Figure 4. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve. Present plots of J as a function of the number of iterations for different learning rates. What are good learning rates and number of iterations for this problem?

Part 2: Implementing regularized linear regression

In this part, you will implement regularized linear regression and use it to study models with different bias-variance properties. To get started, look at the code in the folder `part2`, which has the files shown in Table 2.

Problem 1: Regularized linear regression (10 points)

In this problem, you will implement regularized linear regression to predict the amount of water flowing out of a dam using the change of water level in a reservoir.

Visualizing the dataset

We will begin by visualizing the dataset containing historical records on the change in the water level x, and the amount of water y, flowing out of the dam. This dataset is divided into three parts:

- A training set that you will use to learn the model: `X`, `y`.
- A validation set for determining the regularization parameter: `Xval`, `yval`.

Name	Edit?	Read?	Description
linearRegCostFunction.m	Yes	Yes	Cost function for the regularized linear regression
learningCurve.m	Yes	Yes	Function that generates a learning curve
polyFeatures.m	Yes	Yes	Function to map data into a polynomial feature space
validationCurve.m	Yes	Yes	Function to generate a cross-validation plot
ex2.m	No	Yes	Matlab script that will run your functions
featureNormalize.m	No	Yes	Function to normalize features
trainLinearReg.m	No	Yes	trains linear regression model using your cost function
fmincg.m	No	Yes	Function minimization routine
plotFit.m	No	Yes	function to plot a polynomial fit
ex2data1.mat	No	No	Dataset for this assignment
pa1_2015.pdf	No	Yes	this document

Table 2: The files in folder `part2` for the second part of the assignment.

- A test set for evaluating the performance of your model: `Xtest, ytest`. These are unseen examples that were not used during the training of the model.

Run the script `ex2.m` and it will plot the training data as shown in Figure 5.

Next you will implement regularized linear regression and use it to fit a straight line to the data and plot learning curves. Following that, you will implement polynomial regression to find a better fit to the data.

Regularized linear regression cost function (5 points)

Regularized linear regression has the following cost function:

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}(y^{(i)} - h_\theta(x^{(i)})^2\right) + \frac{\lambda}{2m}\left(\sum_{j=1}^{n}{\theta_j}^2\right)$$

where λ is a regularization parameter which controls the degree of regularization (thus, help preventing overfitting). The regularization term puts a penalty on the overall cost $J(\theta)$. As the magnitudes of the model parameters θ_j increase, the penalty increases as well. Note that you should not regularize the θ_0 term. You should now complete the code in the file `linearRegCostFunction.m` to calculate $J(\theta)$. If possible, try to vectorize your code and avoid writing loops. The next part of `ex2.m` will run your cost function using theta initialized at `[1;1]`. You should expect to see an output of 303.993.

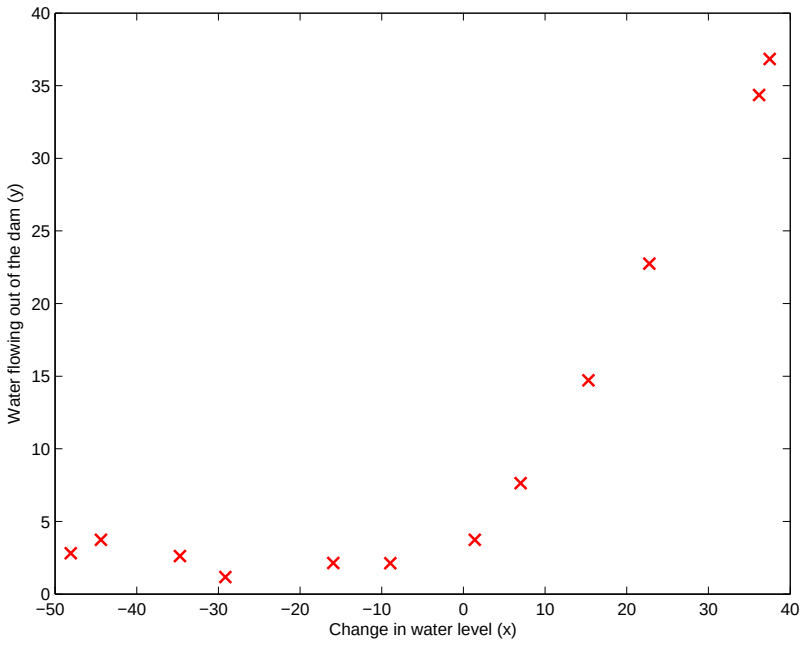

Figure 5: The training data

Gradient of the Regularized linear regression cost function (5 points)

Correspondingly, the partial derivative of regularized linear regressions cost function with respect to θ_j is defined as:

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_0} &= \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}){x_j}^{(i)} \\ \frac{\partial J(\theta)}{\partial \theta_j} &= \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}){x_j}^{(i)}\right) + \frac{\lambda}{m}\theta_j \quad \text{for } j \geq 1\end{aligned}$$

In `linearRegCostFunction.m`, add code to calculate the gradient, returning it in the variable `grad`. Then, run the script `ex2.m` using `theta` initialized at `[1; 1]`. You should expect to see a gradient of `[-15.30; 598.250]`.

Learning linear regression model

Once your cost function and gradient are working correctly, the script `ex2.m` will run the code in `trainLinearReg.m` to compute the optimal values of θ. This training function uses `fmincg` to optimize the cost function. Here we have set the regularization parameter λ to zero. Because we are

trying to fit a line on to data that is clearly non-linear, regularization will not be incredibly helpful. In the next problem, you will use polynomial regression and see the impact of regularization.

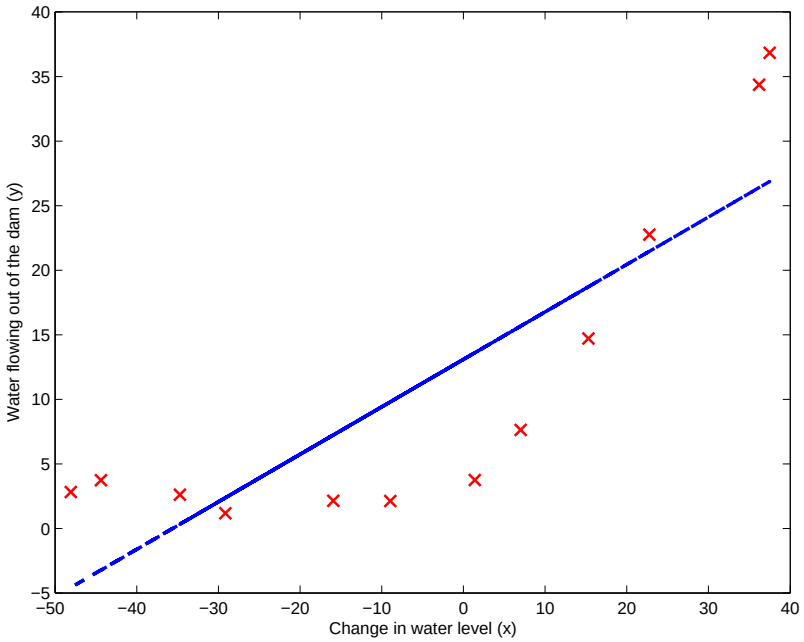

Figure 6: The best fit line for the training data

Finally, the `ex2.m` script plots the best fit line, resulting in a plot like the one shown in Figure 6. The best fit line tells us that the model is not a good fit to the data because the data is non-linear. While visualizing the best fit as shown is one possible way to debug your learning algorithm, it is not always easy to visualize the data and model. In the next problem, you will implement a function to generate learning curves that can help you debug your learning algorithm even if it is not easy to visualize the data.

Problem 2: Bias and Variance (30 points)

An important concept in machine learning is the bias-variance tradeoff. Models with high bias are not complex enough for the data and tend to underfit, while models with high variance overfit the training data. In this problem, you will plot training and test errors on a learning curve to diagnose bias-variance problems.

Learning curves (5 points)

A learning curve plots training and cross validation error as a function of training set size. You will complete the code in `learningCurve.m` so that it returns a vector of errors for the training set and validation set. To obtain different training set sizes, use different subsets of the original training set `X`. Specifically, for a training set size of i, you should use the first i examples.

You can use the `trainLinearReg` function to find the parameter `theta`. Note that `lambda` is passed as a parameter to the `learningCurve` function. After learning the `theta` parameter, you should compute the error on the training and validation sets. Recall that the training error for a dataset is defined as:

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}(y^{(i)} - h_\theta(x^{(i)})^2\right)$$

In particular, note that the training error does not include the regularization term. One way to compute the training error is to use your existing cost function and set `lambda` to 0 only when using it to compute the training error and validation error. When you are computing the training set error, make sure you compute it on the training subset instead of the entire training set. However, for the validation error, you should compute it over the entire validation set. You should store the computed errors in the vectors `error_train` and `error_val`. When you are finished, `ex2.m` will print the learning curves and produce a plot similar to Figure 7.

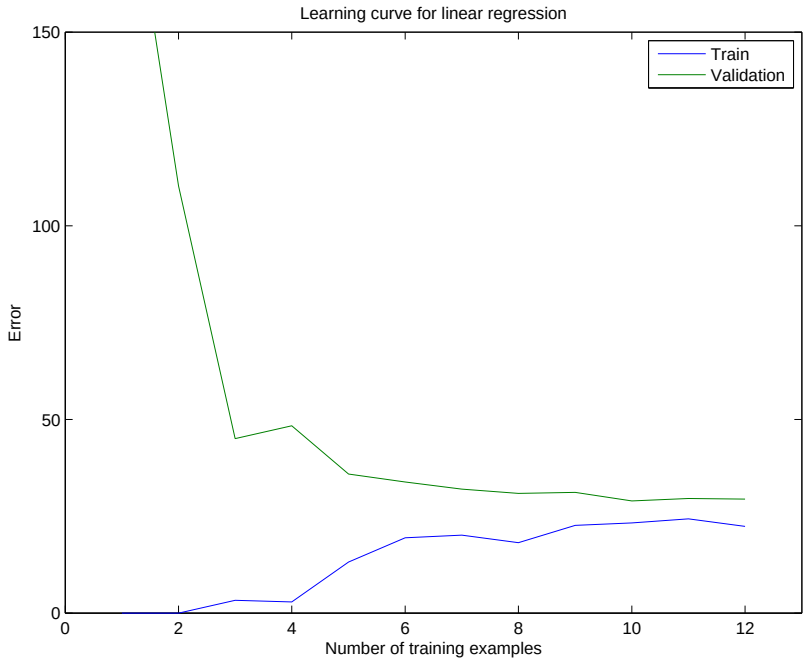

Figure 7: Learning curves

In Figure 7, you can observe that both the train error and cross validation error are high when the number of training examples is increased. This reflects a high bias problem in the model the linear regression model is too simple and is unable to fit our dataset well. Next, you will implement polynomial regression to fit a better model for this dataset.

Polynomial regression (5 points)

The problem with our linear model was that it was too simple for the data and resulted in underfitting (high bias). In this problem, you will address this issue by adding more features. In particular, you will consider hypotheses of the form

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_p x^p$$

This is still a linear model from the point of view of the parameter space. We have augmented the features with powers of x.

Your task is to complete the code in `polyFeatures.m` so that the function maps the original training set `X` of size $m \times 1$ into its higher powers. Specifically, when a training set `X` of size $m \times 1$ is passed into the function, the function should return a $m \times p$ matrix `X_poly`, where column 1 holds the original values of x, column 2 holds the values of the squares of x, column 3 holds the values of the cubes of x, and so on. Note that you do not have to account for the zeroth power in this function.

Learning polynomial regression models

After you have completed `polyFeatures.m`, the `ex2.m` script will proceed to train polynomial regression using your linear regression cost function. Keep in mind that even though we have polynomial terms in our feature vector, we are still solving a linear regression optimization problem. The polynomial terms have simply turned into features that we can use for linear regression.

You will use a polynomial of degree 8. It turns out that if we run the training directly on the projected data, it will not work well as the features would be badly scaled (e.g., an example with $x = 40$ will now have a feature $x^8 = 40^8 = 6.5 \times 10^{12}$). Therefore, you will need to use feature normalization. Before learning the parameter θ for the polynomial regression, `ex2.m` will first call `featureNormalize` and normalize the features of the training set, storing the `mu`, `sigma` parameters separately. We have already implemented this function for you. After learning the parameter θ, you should see two plots (Figures 8 and 9) generated for polynomial regression with $\lambda = 0$.

From Figure 8, you should see that the polynomial fit is able to follow the datapoints very well - thus, obtaining a low training error. However, the polynomial fit is very complex and even drops off at the extremes. This is an indicator that the polynomial regression model is overfitting the training data and will not generalize well.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve (Figure 9) shows the same effect where the training error is low, but the error on the validation set is high. There is a gap between the training and validation errors, indicating a high variance problem. One way to combat the overfitting (high-variance) problem is to add regularization to the model. Next, you will get to try different λ values to see how regularization can lead to a better model.

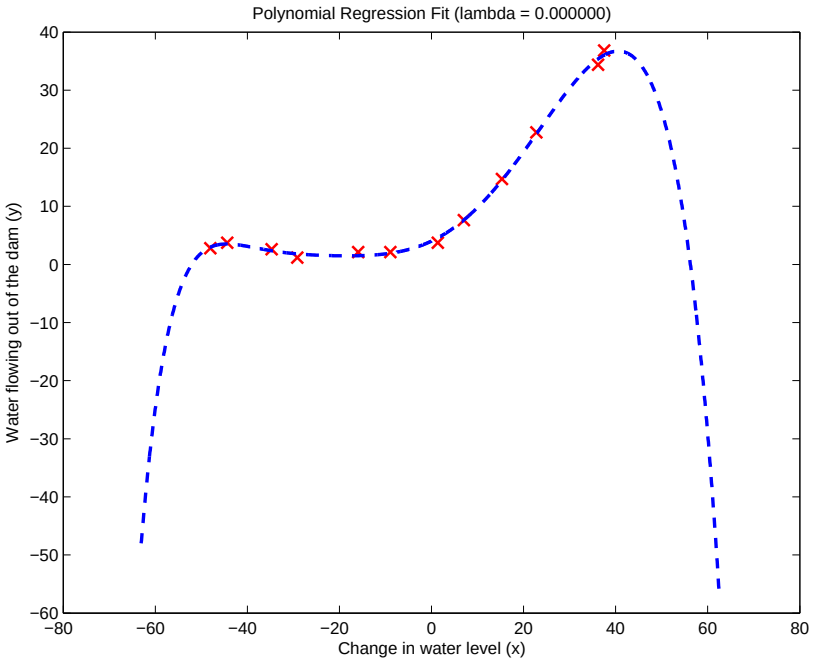

Figure 8: Polynomial fit for `lambda` $= 0$.

Adjusting the regularization parameter (5 points)

You will now explore how the regularization parameter affects the bias-variance of regularized polynomial regression. You should now modify the `lambda` parameter in the script `ex2.m` and try $\lambda = 1, 10, 100$. For each of these values, the script will generate a polynomial fit to the data and also a learning curve. Submit two plots for each value of `lambda`: the fit as well as the learning curve. Comment on the impact of the choice of `lambda` on the quality of the learned model.

Selecting λ using a validation set (5 points)

You have now observed that the value of λ can significantly affect the results of regularized polynomial regression on the training and validation set. In particular, a model without regularization fits the training set well, but does not generalize. Conversely, a model with too much regularization does not fit the training set and testing set well. A good choice of λ can provide a good fit to the data.

You will implement an automated method to select the λ parameter. Concretely, you will use a validation set to evaluate how good each λ value is. After selecting the best λ value using the validation set, we can then evaluate the model on the test set to estimate how well the model will perform on actual unseen data. Complete the code in `validationCurve.m`. Specifically, you should

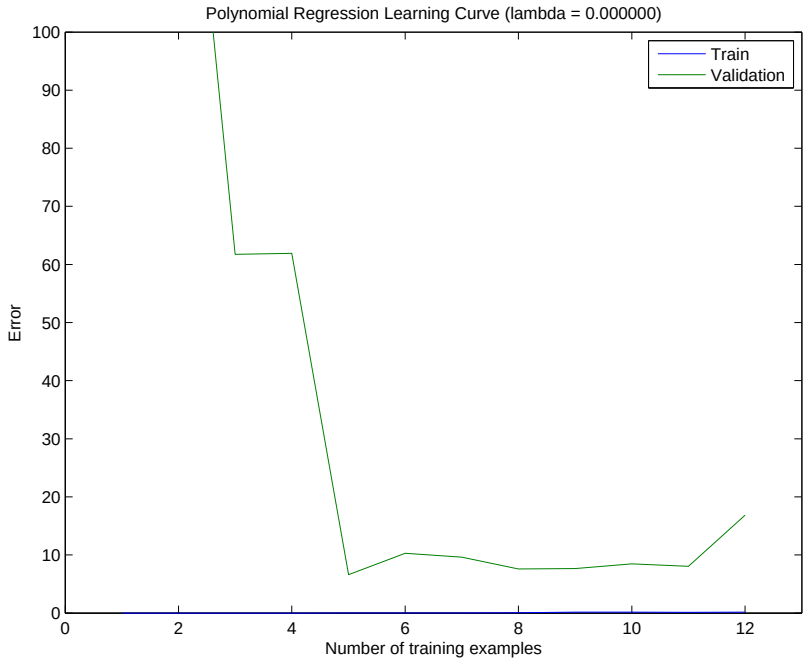

Figure 9: Learning curve for `lambda` $= 0$.

should use the `trainLinearReg` function to train the model using different values of λ and compute the training error and validation error. You should try λ in the following range: $\{0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10\}$.

After you have completed the code, run `ex2.m` to plot a validation curve of λ vs the error that allows you select which λ value to use. Due to randomness in the training and validation splits of the dataset, the cross validation error can sometimes be lower than the training error. Submit a pdf version of this plot in your report. Comment on the best choice of λ for this problem.

Computing test set error (5 points)

To get a better indication of a model's performance in the real world, it is important to evaluate the final model on a test set that was not used in any part of training (that is, it was neither used to select the regularization parameter, nor to learn the model parameters). Calculate the error of the best model that you found with the previous analysis and report it. You can add code at the end of `ex2.m` to print out this value.

Plotting learning curves with randomly selected examples (5 points)

In practice, especially for small training sets, when you plot learning curves to debug your algorithms, it is often helpful to average across multiple sets of randomly selected examples to determine the training error and validation error. Concretely, to determine the training error and cross validation error for i examples, you should first randomly select i examples from the training set and i examples from the validation set. You will then learn the model parameters using the randomly chosen training set and evaluate the parameters on the randomly chosen training set and validation set. The above steps should then be repeated multiple times (say 50) and the averaged error should be used to determine the training error and cross validation error for i examples. Implement the above strategy for computing the learning curves. For reference, Figure 10 shows the learning curve we obtained for polynomial regression with $\lambda = 1$. Your figure may differ slightly due to the random selection of examples. Write a new function `learningCurve_Averaged` in Matlab to generate compute and generate this plot. Call this function at the end of `ex2.m` and plot the averaged learning curve.

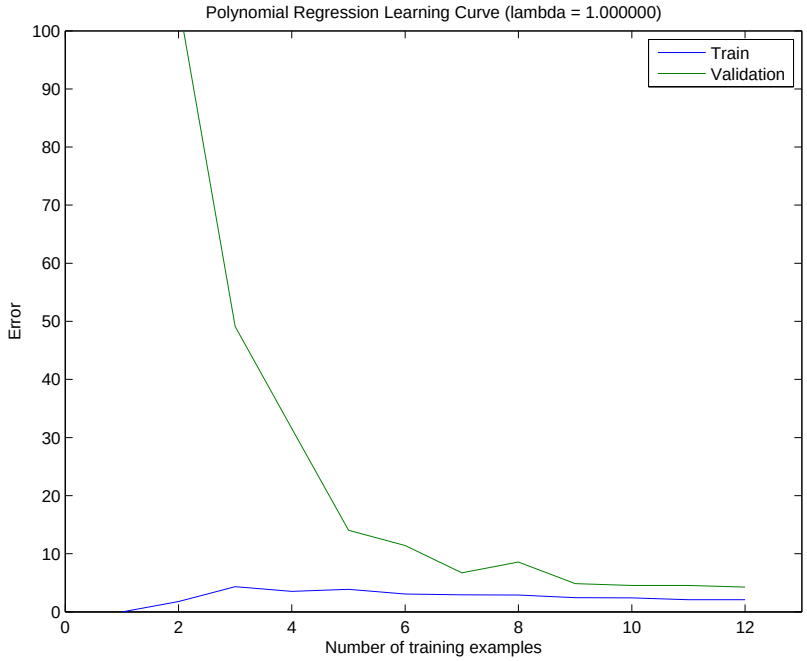

Figure 10: Averaged Learning curve for `lambda` = 1.

Problem 3: Building regularized models for the Boston housing data set (25 points)

Perform a bias variance analysis of the Boston housing data set with the thirteen predictors, following the steps in Problem 2. Use Matlab's `dividerand` function to split the data into training, validation and test sets. What is the lowest achievable error on the test set with $\lambda = 0$? Select the best value for λ and report the test set error with the best λ. Use the technique of adding features to extend each column of the Boston data set with powers of the values in the column. Repeat the bias-variance analysis with quadratic and cubic features. What is the test set error with quadratic features with the best λ chosen with the validation set? What is the test set error with cubic features with the best λ chosen with the validation set? Put your analysis code in a separate Matlab script called `bostonexpt.m`. Present your results analytically with plots to support your findings. Discuss the impact of regularization for building good models for the Boston housing data set.

What to turn in

Please zip up all the files in the archive (including files that you did not modify) and submit it as `pa1_netid`.zip on Owlspace before the deadline. Include a PDF file in the archive that presents your plots and discussion of results from the two parts above.

Acknowledgment

This assignment is adapted from Andrew Ng's exercise on linear regression.