Homework 06

Regularized Linear Regression and Bias v.s. Variance Section 2

In this exercise, you will implement regularized linear regression and use it to study models with different bias-variance properties. In the first half of the exercise, you will implement regularized linear regression to predict the amount of water owing out of a dam using the change of water level in a reservoir. In the next half, you will go through some diagnostics of debugging learning algorithms and examine the effects of bias v.s. variance. The provided script, **hw6.m**, will help you step through this exercise.

Part 1-4: See Regularized Linear Regression and Bias v.s. Variance Section 1 (Homework 5)

Part 5: Learning Curves (20 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 to the training data. In this part of the exercise, you will plot training and test errors on a learning curve to diagnose bias-variance problems.

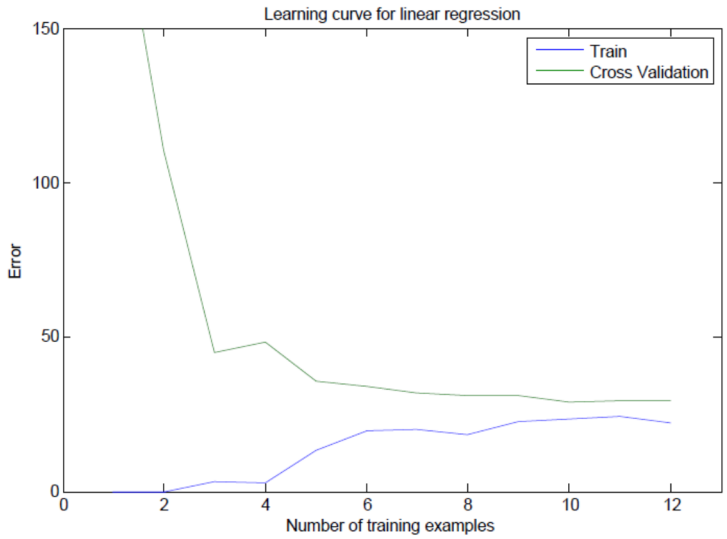
You will now implement code to generate the learning curves that will be useful in debugging learning algorithms. Recall that a learning curve plots training and cross validation error as a function of training set size. Your job is to fill in **learningCurve.m** so that it returns a vector of errors for the training set and cross validation set. To plot the learning curve, we need a training and cross validation set error for different training set sizes. To obtain different training set sizes, you should use different subsets of the original training set X. Specifically, for a training set size of i, you should use the first i examples (i.e., X(1:i,:) and y(1:i)).

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

$$J_{\text{train}}(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2\right]$$

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 λ to 0 only when using it to compute the training error and cross validation error. When you are computing the training set error, make sure you compute it on the training subset (i.e., X(1:n,:) and y(1:n)) (instead of the entire training set). However, for the cross validation error, you should compute it over the entire cross validation set. You should store the computed errors in the vectors **error_train** and **error_val**. When you are finished, hw5.m will print the learning curves and produce a plot similar to the following one. In this Figure, 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. In the next section, you will implement polynomial regression to ift a better model for this dataset.



Part 5: Polynomial Regression (10 points)

The problem with our linear model was that it was too simple for the data and resulted in underfitting (high bias). In this part of the exercise, you will address this problem by adding more features. For use polynomial regression, our hypothesis has the form:

$$h_\theta(x) = \theta_0 + \theta_1 * (\text{waterLevel}) + \theta_2 * (\text{waterLevel})^2 + \cdots + \theta_p * (\text{waterLevel})^p$$
$$= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_p x_p.$$

Notice that by defining $x_1 = (waterLevel), x_2 = (waterLevel)^2, \ldots, x_p = (waterLevel)^p$, we obtain a linear regression model where the features are the various powers of the original value (waterLevel). Now, you will add more features using the higher powers of the existing feature x in the dataset. Your task in this part 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 X.^2, column 3 holds the values of X.^3, and so on. Note that you don't have to account for the zero-eth power in this function. Now you have a function that will map features to a higher dimension, and Part 6 of hw6.m will apply it to the training set, the test set, and the cross validation set (which you haven't used yet).

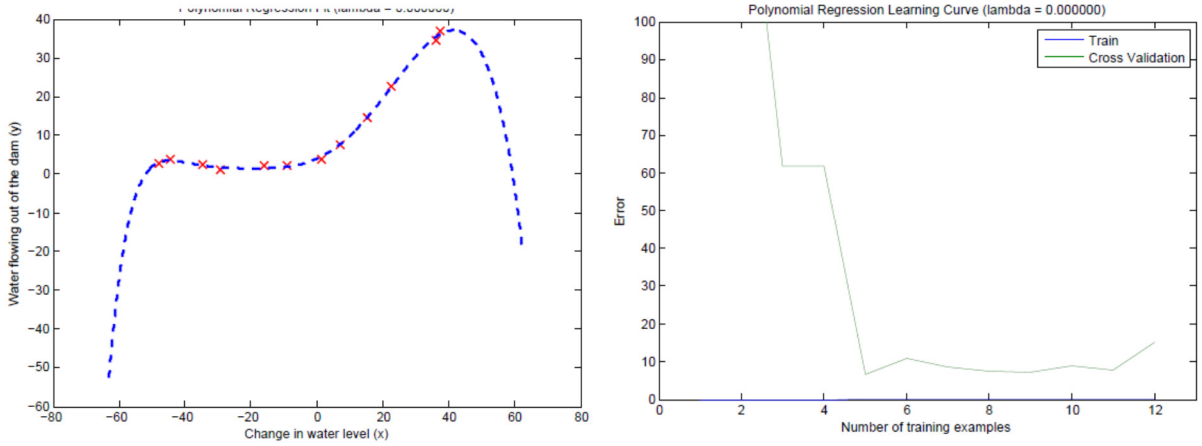
Part 6: Learning Polynomial Regression (0 points)

After you have completed **polyFeatures.m**, the hw6.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. We are using the same cost function and gradient that you wrote for the earlier part of this exercise.

For this part of the exercise, you will be using 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 parameters θ for the polynomial regression, hw6.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 and it is the same function from the first exercise. After learning the parameters θ, you should see two plots generated for polynomial regression with $\lambda = 0$.



From the above Figures, you should see that the polynomial fit is able to follow the data points 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 (right Figure) shows the same effect where the low training error is low, but the cross validation error is high. There is a gap between the training and cross validation errors, indicating a high variance problem. One way to combat the overfitting (high-variance) problem is to add regularization to the model. You may try different parameters λ to see how regularization can lead to a better model.

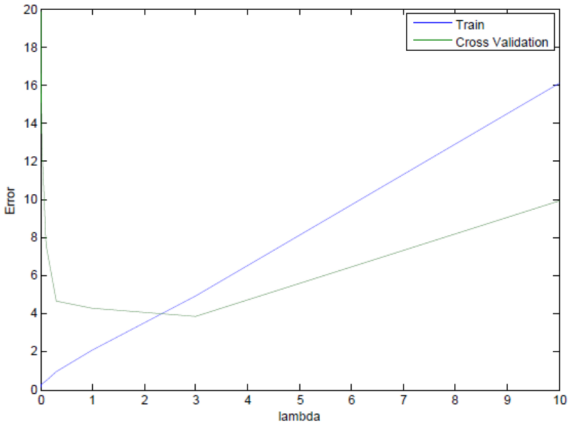
Part 7: Selecting Regularization Parameter using a Cross Validation Set (20 points)

The value of λ can significantly affect the results of regularized polynomial regression on the training and cross validation set. In particular, a model without regularization ($\lambda = 0$) fits the training set well, but does not generalize. Conversely, a model with too much regularization ($\lambda = 100$) does not fit the training set and testing set well. A good choice of λ (e.g., $\lambda = 1$) can

provide a good fit to the data. In this section, you will implement an automated method to select the λ parameter. Concretely, you will use a cross validation set to evaluate how good each λ value is. After selecting the best λ value using the cross validation set, we can then evaluate the model on the test set to estimate how well the model will perform on actual unseen data.

Your task is to complete the code in **validationCurve.m**. Specifically, you should use the trainLinearReg function to train the model using different values of λ and compute the training error and cross 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, the next part of hw6.m will run your function can plot a cross validation curve of error v.s. λ that allows you select which λ parameter to use. You should see a plot similar to a Figure below. In this figure, we can see that the best value of λ is around **3**. Due to randomness in the training and validation splits of the dataset, the cross validation error can sometimes be lower than the training error.



Submission:

To submit, turn in the following files on Canvas:

- **learningCurve.m**
- **polyFeatures.m**
- **validationCurve.m**