UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA TELECOMUNICAÇÕES E INFORMÀTICA

Machine Learning (2016/17) – Lab work 6

Objectives: Implement regularized linear regression and use it to study models with different bias-variance properties.

Files included in this lab work:
ex5.m - Octave/MATLAB script that steps you through the exercise
ex5data1.mat – Dataset
linearRegCostFunction.m - Regularized linear regression cost function **(complete this function)**
trainLinearReg.m - Trains linear regression model
learningCurve.m - Generates a learning curve
fmincg.m - Function minimization routine (similar to *fminunc*)
featureNormalize.m - Feature normalization function
polyFeatures.m - Maps data into polynomial feature space **(complete this function)**
plotFit.m - Plot a polynomial fit
validationCurve.m - Generates a cross validation curve **(complete this function)**

The task is to implement regularized linear regression to predict the amount of water owing out of a dam using the change of water level in a reservoir. You will examine the effects of bias versus variance.

1. Load and plot the data
File *ex5data1.mat* contains historical records on the change in the water level, x, and the amount of water owing out of the dam, y. The dataset is divided into the following parts:
• A training set (*X, y*) used to fit your model
• A cross validation set (*Xval, yval*) for determining the regularization parameter
• A test set (*Xtest, ytest*) for evaluating performance. These are examples which your model did not see during training.

Load the data and plot the training data (Fig.1). First, you will implement linear regression 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.

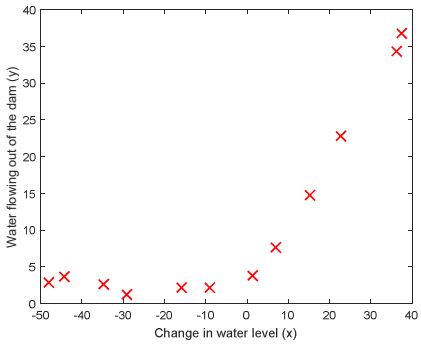

Fig.1 Training data

2. Linear regression regularized cost function and its gradient

Recall that *Lambda* is the regularization parameter which helps preventing overfitting. The regularization term puts a penalty on the overall cost *J*. As the magnitudes of the model parameters increase, the penalty increases as well. Note that you should not regularize θ_0 term. In Octave/MATLAB, θ_0 term is represented as theta(1) since indexing in Octave/MATLAB starts from 1. Complete the code in the function *linearRegCostFunction.m* to calculate the regularized linear regression cost function and its gradient (variable *grad*), consult function *CostFunctionReg.m* (lab 3) and *ComputeCost.m* (lab 2).

After that script will run the cost function using *theta* initialized at [1; 1]. You should expect to see the cost value *J*=303.993 and the gradient vector (i.e. the partial derivative of regularized linear regression's cost with respect to each theta) = [-15.30; 598.250].

3. Fitting linear regression

Once your cost function and gradient are working correctly, the main script will run the code in *trainLinearReg.m* to compute the optimal values of θ. This training function uses *fmincg* to optimize the cost function. In this part, we set regularization parameter = 0. Because our current implementation of linear regression is trying to fit a 2-dimensional θ, regularization will not be incredibly helpful for a θ of such low dimension. In the later parts of the exercise, you will be using polynomial regression with regularization. The best fit line will be plotted as shown in Fig. 2. The best fit line tells us that the model is not a good fit to the data because the data has a non-linear pattern.

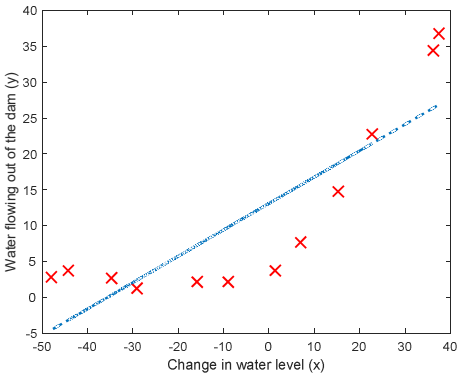

Fig.2 Linear fit

4. Bias-variance and learning curves

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 under-fit, while models with high variance over-fit to the training data. Now, you will plot training and test errors on a learning curve to diagnose bias-variance problems.

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, *learningCurve.m* use different subsets of the original training set X. Specifically, for a training set size of *i*, the first *i* examples (i.e., X(1:i,:) and y(1:i)) are used. After learning the parameters, the error on the training and cross validation sets are computed. The training error is defined as

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

Note that the training error does not include the regularization term. The regularized cost function is minimized during the learning process and the optimal vector theta is obtained. The training error is then computed by (1). Note that, the cross validation error is computed over the entire cross validation set. The computed errors are stored in the vectors *error_train* and *error_val*.

The learning curves are then plotted as shown in Fig. 3. You can observe that both the training error and cross validation error are high even 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 the dataset well).

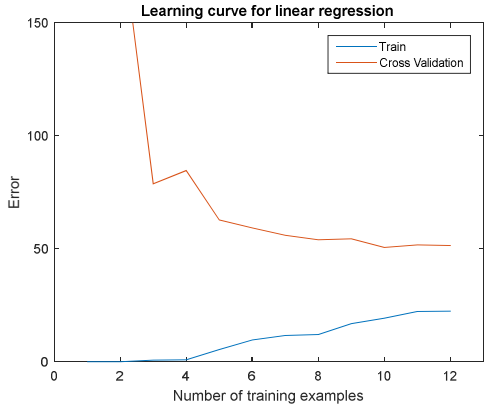

Fig.3 Linear Regression learning curve

5. Polynomial regression

The problem with our linear model was that it was too simple for the data and resulted in under-fitting (high bias). In this part of the exercise, you will address this problem by adding more features. To use polynomial regression, our hypothesis has now 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 we still have a linear regression model where the features are the various powers of the original feature (waterLevel). Now, you will add more features using the higher powers of the existing feature x in the dataset. Your task is to complete the code in *polyFeatures.m* so that the function maps the original training set *X* of size *mx1* into its higher powers. Specifically, when a training set *X* of size *mx1* is passed into the function, the function should return a *mxp* 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. Now you have a function that will map features to a higher dimension, and Part 6 of the script will apply it to the training set, the test set, and the cross validation set.

6. Learning Polynomial Regression

After you have completed *polyFeatures.m*, the 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 are simply new 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, you will be using a polynomial of degree 8. It turns out that if we run the training directly on the data, 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 = 6.5x10^12). Therefore, before learning the parameters for the polynomial regression, the script will first call the function *featureNormalize* (already implemented) and normalize the features of the training set, storing *mu* and *sigma* parameters separately. After learning the parameters, you should see two plots (Fig. 4,5) generated for polynomial regression with Lambda= 0.

From Fig. 4, 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 un-regularized (lambda=0) model, you can see that the learning curve (Fig. 5) shows the same effect where the training error is very low (almost equal to zero), but the cross validation error is high. There is a gap between the training and cross validation errors, indicating a high variance problem.

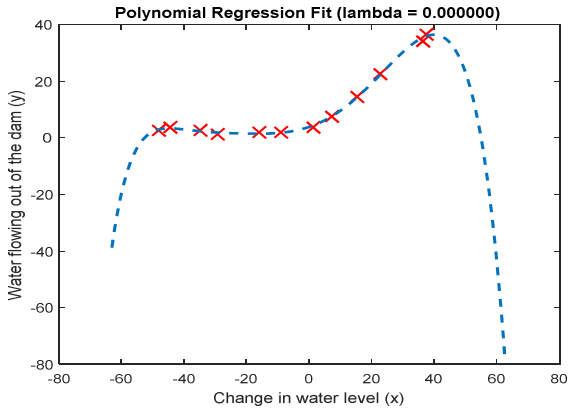

Fig. 4 Polynomial regression Lambda=0

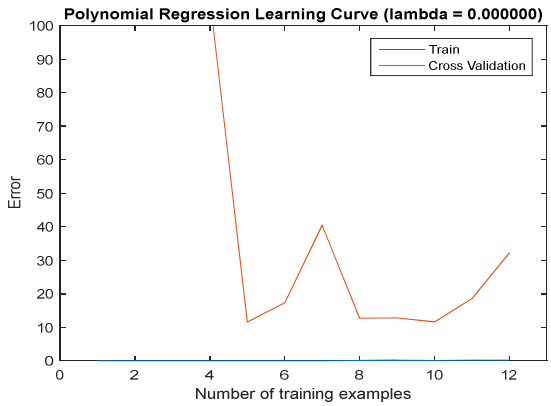

Fig. 5 Polynomial learning curve, Lambda=0

Repeat part 7 of the script for Lambda=1 and Lambda=100 to get similar curves as on Figs. 6-9 and make conclusions.

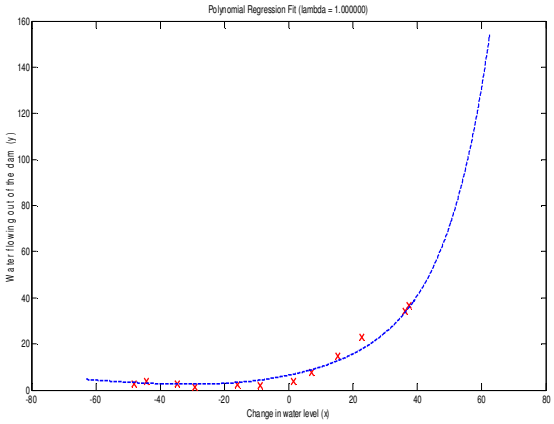

Fig. 6 Polynomial regression Lambda=1

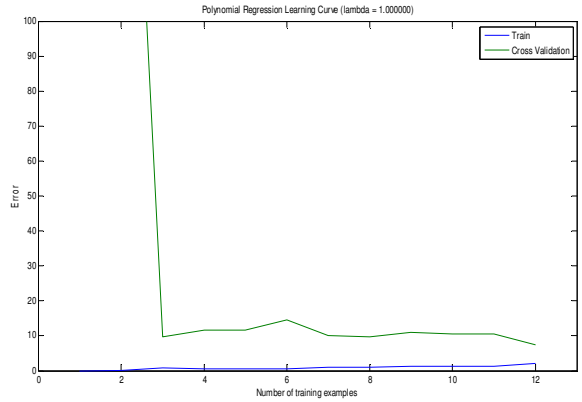

Fig. 7 Polynomial learning curve, Lambda=1

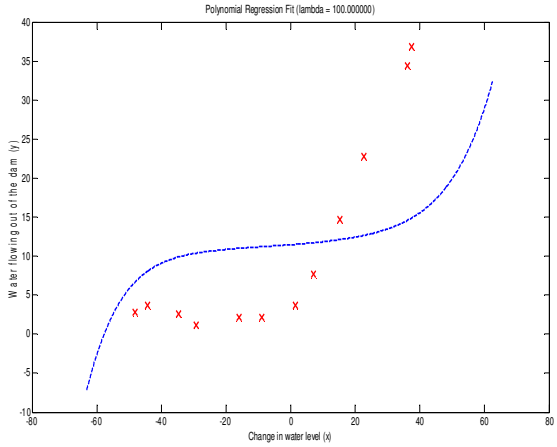

Fig. 8 Polynomial regression Lambda=100

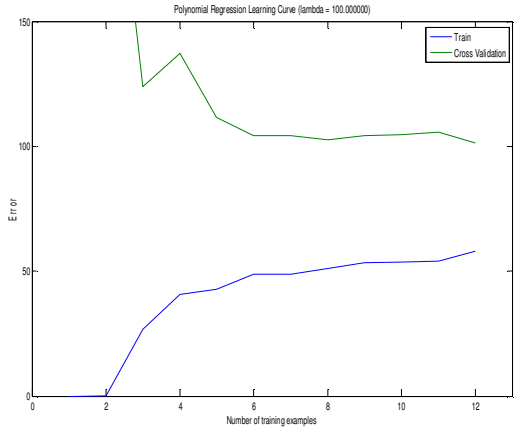

Fig. 9 Polynomial learning curve, Lambda=100

7. Selecting Lambda using a cross validation set

You observed that the value of *Lambda* 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. Now, you will implement an automated method to select the *Lambda* parameter.

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 *Lambda* and compute the training error and cross validation error. Try the following values for *Lambda* = [0; 0.001; 0.003; 0.01; 0.03; 0.1; 0.3; 1; 3; 10].

Function *validationCurve.m* is similar to the function *learningCurve.m*, however instead of loop *for* with respect to training examples here the loop *for* is with respect to *Lambda*. Use the code in function *learningCurve.m* to guide you.

After you have completed the code, the script will run *validationCurve* and plot the train and cross validation error curves versus *Lambda*. You should see a plot similar to Fig 10 and conclude that the best value of Lambda is around 3.

After selecting the best Lambda 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.

*For Lambda=3 y*ou are expected to get test error around 6.8, cross validation (CV) error around 6.69, training error around 4.9.

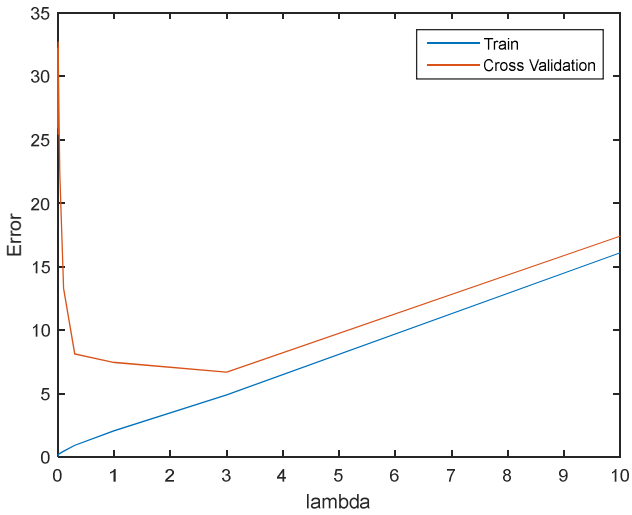

Fig. 10 Selecting Lambda using a cross validation set