UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA TELECOMUNICAÇÕES E INFORMÀTICA

Machine Learning (2018/19) – Lab work 6

Part 1 Model selection and validation. Bias – Variance

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.

1.1. 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 the model.
• A cross validation set (*Xval, yval*) for determining the regularization parameter.
• A test set (*Xtest, ytest*) for evaluating performance. These are examples which the 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.

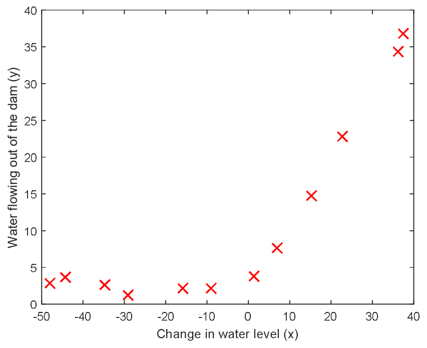

Fig.1 Training data

1.2. Linear regression cost function with regularization and its gradient

Recall that λ 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, the main 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 the regularized linear regression cost with respect to each element of vector θ) = [-15.30; 598.250].

1.3. Fitting linear regression

Once the cost function and the gradient are computed 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 the regularization parameter λ = 0. Because the 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 (similar to the one in Fig.2) tells that the model is not a good fit to the data.

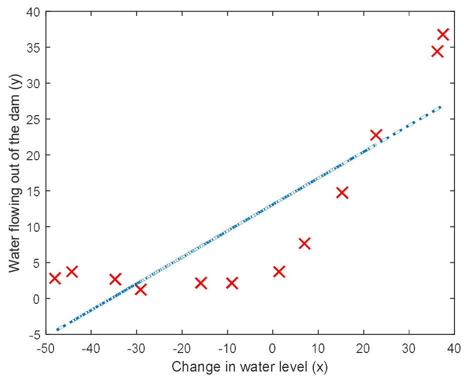

Fig.2 Linear fit

1.4. 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 training and cross validation errors 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 training and cross validation errors are computed. The training error is defined as

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

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). The cross validation error is computed over the entire cross validation set. The computed errors are stored in vectors *error_train* and *error_val*.

The learning curves are then plotted as shown in Fig. 3. You can observe that both the training and the cross validation errors are high even when the number of training examples increases. This reflects a high bias problem in the model.

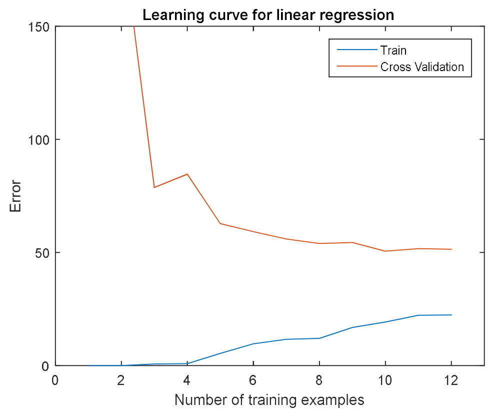

Fig.3 Linear Regression learning curve

1.5. Polynomial regression

The linear model is too simple for this data and resulted in under-fitting (high bias). Now, you will address this problem by adding more features using higher powers of the original feature (waterLevel):

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

Keep in mind that even though we have polynomial terms in the model hypothesis (2) , we are still solving a linear regression optimization problem. The polynomial terms are simply new features that we can use for linear regression. Your task is to complete the function *polyFeatures.m* so that it 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. Function *polyFeatures.m* is applied to the training, cross validation and test sets.

1.6. Learning Polynomial Regression

After you have completed *polyFeatures.m*, the main script will proceed to train the model (2). It uses the same linear regression cost function and gradient that you wrote for the earlier part of this exercise. Let assume a polynomial of degree 8. If we run the training directly on the 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 = 6.5x10^12). Therefore, before learning the parameters, the features of the training set are first normalized (function *featureNormalize* is already implemented*)*, storing the mean (*mu*) and the standard deviation (*sigma)* parameters. After training is over, you should see two plots (Figs. 4 and 5) generated for $\lambda = 0$.
See that the polynomial fit is able to follow the data points very well (Fig.4) and therefore the training error is low (almost equal to zero in Fig.5). However, there is a gap between the training and cross validation errors (Fig.5), indicating a high variance problem due to the lack of regularization (λ =0).

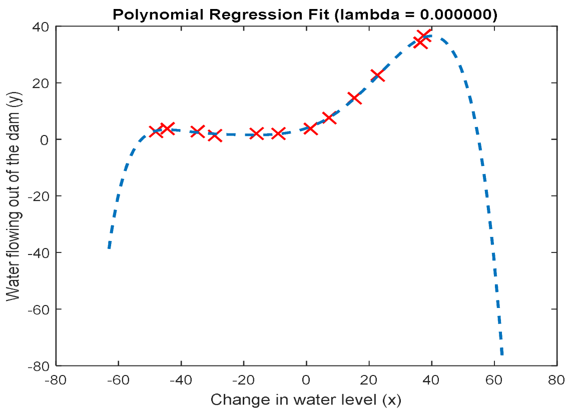

Fig. 4 Polynomial regression, 𝜆 =0

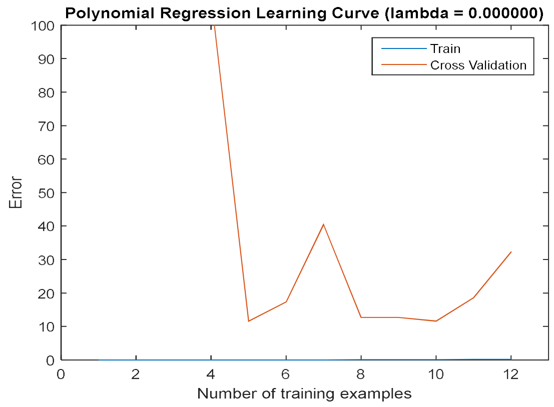

Fig. 5 Polynomial learning curve, 𝜆 =0

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

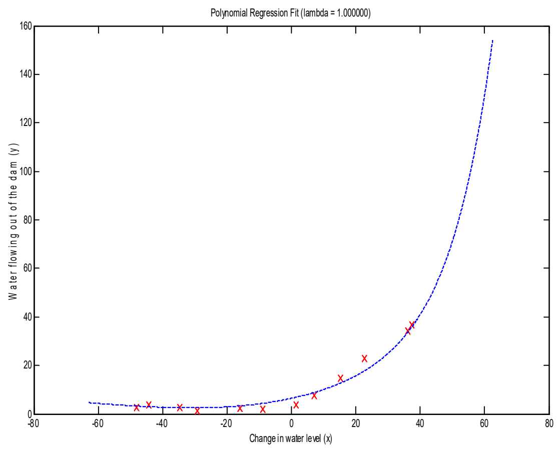

Fig. 6 Polynomial regression 𝜆 =1

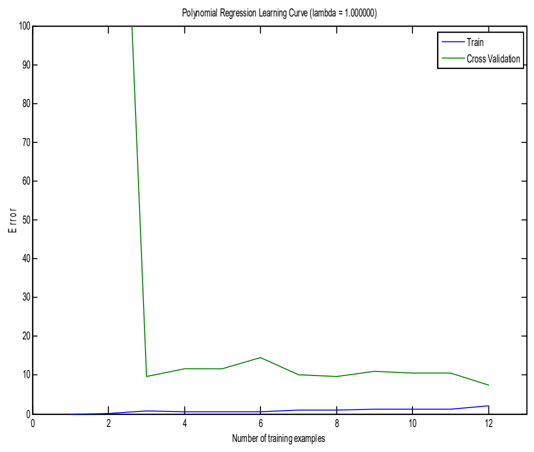

Fig. 7 Polynomial learning curve, 𝜆 =1

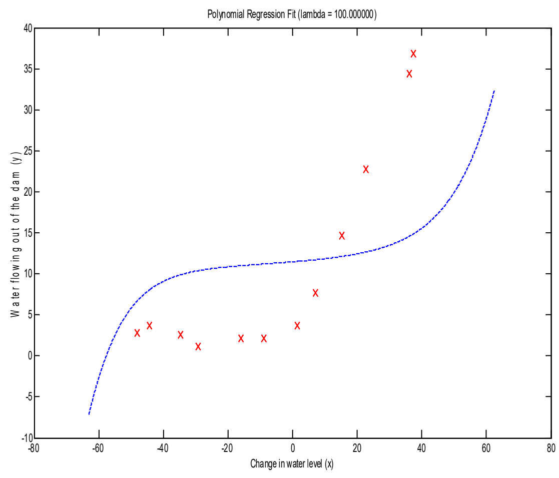

Fig. 8 Polynomial regression 𝜆 =100

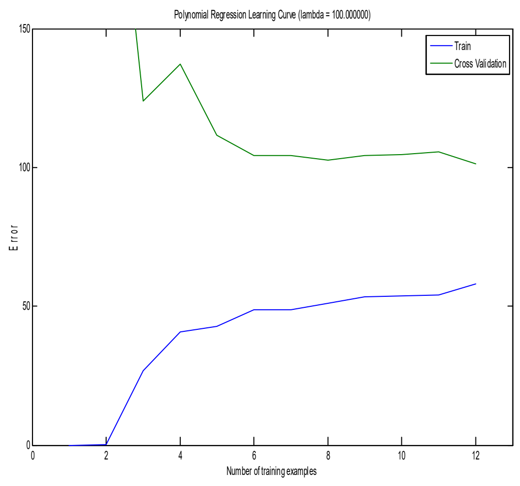

Fig. 9 Polynomial learning curve, 𝜆 =100

1.7. Selecting λ using the cross validation (CV) data set

You observed that a model without regularization (λ = 0) fits the training set well, but does not generalize. Now, you will implement an automated method to select the best value for λ. Your task is to complete function *validationCurve.m*. Specifically, you should use *trainLinearReg* function to train the model using different values of λ (for example λ = [0; 0.001; 0.003; 0.01; 0.03; 0.1; 0.3; 1; 3; 10]) and compute the training and CV errors. Function *validationCurve.m* is similar to function *learningCurve.m*, however instead of loop *for* with respect to training examples here the loop *for* is with respect to λ. Use function *learningCurve.m* to guide you.

After you complete the function and run the main script you should see a plot similar to Fig 10. The best value of λ is around 3. After selecting the best λ using the CV set, we should evaluate the model on the test set to estimate how well it will perform on actual unseen data.

For λ =3 you are expected to get test error around 6.8, CV error around 6.69, training error around 4.9.

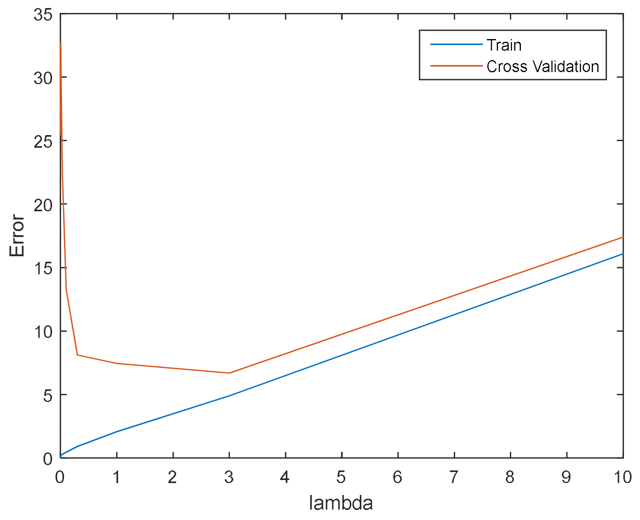

Fig. 10 Selecting λ using CV data set

Part 2 Spam Filter

Objectives: Apply Support Vector Machine (SVM) to build a spam classifier.

Files included in this lab work:

ex6 spam.m - Octave/MATLAB script (complete this file)
spamTrain.mat - Spam training set
spamTest.mat - Spam test set
emailSample1.txt - Sample email 1
emailSample2.txt - Sample email 2
spamSample1.txt - Sample spam 1
spamSample2.txt - Sample spam 2
vocab.txt - Vocabulary list
getVocabList.m - Load vocabulary list
porterStemmer.m - Stemming function
readFile.m - Reads a file into a character string
processEmail.m - Email preprocessing (**complete this function**)
emailFeatures.m - Feature extraction from emails (**complete this function**)

You will train a SVM classifier to classify whether a given email is spam (y = 1) or non-spam (y = 0). First, each email needs to be converted into a feature vector *x*. The dataset used in this exercise is based on a subset of SpamAssassin Public Corpus (http://spamassassin.apache.org/publiccorpus/). You will only use the body of the email (excluding the email headers).

2.1 Preprocessing emails

```
> Anyone knows how much it costs to host a web portal ?
>
Well, it depends on how many visitors youre expecting.  This can be
anywhere from less than 10 bucks a month to a couple of $100.  You
should checkout http://www.rackspace.com/ or perhaps Amazon EC2 if
youre running something big..

To unsubscribe yourself from this mailing list, send an email to:
groupname-unsubscribe@egroups.com
```

Fig.11 Sample email

Fig. 11 shows a sample email that contains a URL, an email address (at the end), numbers, and dollar amounts. While many emails contain similar types of entities (e.g., numbers, URLs, email addresses), the specific entities (e.g., the specific URL or specific dollar amount) will be different in almost every email. Therefore, one method often employed in processing emails is to “normalize" these values, so that all URLs are treated the same, all numbers are treated the same, etc. For example, we will replace each URL in the email with the unique string “httpaddr" to indicate that a URL was present. This has the effect of letting the spam classifier make a classification decision based on whether any URL was present, rather than whether a specific URL was present. This typically improves the performance of a spam classifier, since spammers often randomize the URLs, and thus the odds of seeing any particular URL again in a new piece of spam is very small. In *processEmail.m*, the following email preprocessing and normalization steps are implemented:

• **Lower-casing**: The entire email is converted into lower case, so that capitalization is ignored (e.g., IndIcaTE is treated the same as Indicate).

• **Stripping HTML**: All HTML tags are removed from the emails. Many emails often come with HTML formatting; we remove all the HTML tags, so that only the content remains.

• **Normalizing URLs**: All URLs are replaced with the text “httpaddr".

• **Normalizing Email Addresses**: All email addresses are replaced with the text “emailaddr".

• **Normalizing Numbers**: All numbers are replaced with the text “number".

• **Normalizing Dollars**: All dollar signs ($) are replaced with the text “dollar".

• **Word Stemming**: Words are reduced to their stemmed form. For example, “discount", “discounts", “discounted" and “discounting" are all replaced with “discount". Sometimes, the Stemmer actually strips off additional characters from the end, so “include", “includes", ”included", and “including" are all replaced with “includ".

• **Removal of non-words**: Non-words and punctuation have been removed. All white spaces (tabs, newlines, spaces) have all been trimmed to a single space character.

The result of these preprocessing steps is shown in Fig. 12. While preprocessing has left word fragments and non-words, this form turns out to be much easier to work with for performing feature extraction.

```
anyon know how much it cost to host a web portal well it depend on how
mani visitor your expect thi can be anywher from less than number buck
a month to a coupl of dollarnumb you should checkout httpaddr or perhap
amazon ecnumb if your run someth big to unsubscrib yourself from thi
mail list send an email to emailaddr
```

Fig.12 Preprocessed sample email

```
1 aa
2 ab
3 abil
...
86 anyon
...
916 know
...
1898 zero
1899 zip
```

Fig. 13 Vocabulary list

```
86 916 794 1077 883
370 1699 790 1822
1831 883 431 1171
794 1002 1893 1364
592 1676 238 162 89
688 945 1663 1120
1062 1699 375 1162
479 1893 1510 799
1182 1237 810 1895
1440 1547 181 1699
1758 1896 688 1676
992 961 1477 71 530
1699 531
```

Fig. 14 Word indices for sample email

2.2 Vocabulary List

After preprocessing the emails, we have a list of words for each email. The next step is to choose which words we would like to use in our classifier and which we would want to leave out. For this exercise, we have chosen only the most frequently occurring words as our set of words considered (the vocabulary list). Since words that occur rarely in the training set are only in a few emails, they might cause the model to overfit the training set. The complete vocabulary list is in the file vocab.txt and also shown in Fig. 13. Our vocabulary list was selected by choosing all words which occur at least a 100 times in the spam corpus, resulting in a list of 1899 words. In practice, a vocabulary list with about 10,000 to 50,000 words is often used. Given the vocabulary list, we can now map each word in the preprocessed emails (Fig. 12) into a list of word indices that contains the index of the word in the vocabulary list. Fig. 14 shows the mapping for the sample email. Specifically, in the sample email, the word "anyone" was first normalized to "anyon" and then mapped onto the index 86 in the vocabulary list. Complete the function *processEmail*.m to perform the mapping. In this function, you are given a string *str* which is a single word from the processed email. You should look up the word in the vocabulary list *vocabList* and find if the word exists in the vocabulary list. If the word exists, you should add the index of the word into the word indices variable. If the word does not exist, and is therefore not in the vocabulary, you can skip the word. Once you have implemented processEmail.m, the main script will run your code on the email sample and you should see an output similar to Fig. 12 & 14.

Remark: In Octave/MATLAB, you can compare two strings with *strcmp* function. For example, *strcmp(str1, str2)* will return 1 only when both strings are equal. In the provided starter code, vocabList is a "cell-array" containing the words in the vocabulary. In Octave/MATLAB, a cell-array is just like a normal array (i.e. vector), except that its elements can also be strings. To get the word at index i, you can use *vocabList{i}*. You can also use *length(vocabList)* to get the number of words in the vocabulary.

2.3 Extracting Features from Emails

You will now implement the feature extraction that converts each email into a vector. The binary feature x_i for an email corresponds to whether the *i*-th word in the dictionary occurs in the email. That is, $x_i = 1$ if the *i*-th word is in the email and $x_i = 0$ if the *i*-th word is not present in the email. Thus, for a typical email, this feature would look like, *n* is the number of words in the vocabulary list:

$$x = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^n$$

You should now complete function *emailFeatures.m* to generate a feature vector for an email, given the word indices. Once you have implemented *emailFeatures.m*, the main script will run your code on the email sample. You should see that the feature vector has length 1899 and 45 non-zero entries.

2.4 Training SVM for Spam Classification

After you have completed the feature extraction, the next step will load a preprocessed training dataset that will be used to train a SVM classifier. *spamTrain.mat* contains 4000 training examples of spam/non-spam emails, *spamTest.mat* contains 1000 test examples. Each original email was processed using the *processEmail* and *emailFeatures* functions and converted into a vector $x^{(i)} \in R^{1899}$. After loading the train dataset, the main script will proceed to train a SVM to classify between spam (y = 1) and non-spam (y = 0) emails. Once the training completes, you should see that the classifier gets a training accuracy of about 99.8% and a test accuracy of about 98.5%.

2.5. Top Predictors for Spam

```
our click remov guarante visit basenumb dollar will price pleas nbsp
most lo ga dollarnumb
```

Fig. 15: Top predictors for spam email

To better understand how the spam classifier works, we can inspect the parameters to see which words the classifier thinks are the most predictive of spam. The next step finds the parameters with the largest positive values in the classifier and displays the corresponding words (Fig. 15). Thus, if an email contains words such as “guarantee”, “remove”, “dollar”, and “price” (the top predictors), it is likely to be classified as spam.

2.6. Try your own emails

Now that you have trained a spam classifier, you can try it out on your own emails. In the starter code, we have included two email examples (*emailSample1.txt*, *emailSample2.txt*) and two spam examples (*spamSample1.txt*, *spamSample2.txt*). Apply the learned SVM spam classifier to see if the classifier gets them right. Try your own emails by replacing the examples (plain text files) with your own emails.

2.7. Build your own dataset

In this project, we provided a preprocessed training set and test set. These datasets were created using the same functions (*processEmail.m* and *emailFeatures.m*) that you have completed. Download the original files from the public corpus, run the *processEmail* and *emailFeatures* functions on each email to extract a feature vector from each email. This will allow you to build a dataset *X, y* of examples. You should then randomly divide the dataset into training, cross validation and test sets.
While you are building your own dataset, you may also build your own vocabulary list (by selecting the high frequency words that occur in the dataset) and adding any additional features that you find useful.