Machine Learning Programming Exercise 4: Regularized Linear Regression and Bias vs Variance

adapted to Python language from Coursera/Andrew Ng

October 2, 2024

1 Introduction

In this exercise, you will implement regularized linear regression and use it to study models with different bias-variance properties.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise.

2 Files included in this exercise

- **ex4.py** - Python script that will help step you through the exercise
- **ex4data1.mat** - Dataset
- **featureNormalize.py** - Feature normalization function
- **plotFit.py** - Plot a polynomial fit
- **trainLinearReg.py** - Trains linear regression using your cost function

⋆ **linearRegCostFunction.py** - Regularized linear regression cost function

⋆ **linearRegGradientFunction.py** - Regularized linear regression gradient function

⋆ **polyFeatures.py** - Maps data into polynomial feature space

[⋆] indicates files you will need to complete.

Throughout the exercise, you will be using the script **ex4.py**. This script sets up the dataset for the problems and make calls to functions that you will write. You are only required to modify functions in other files, by following the instructions in this assignment.

3 Regularized Linear Regression

In the first section of the exercise, 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. Then, you will go through some diagnostics of debugging learning algorithms and examine the effects of bias v.s. variance.

The provided script, **ex4.py**, will help you step through this exercise.

3.1 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 flowing out of the dam, y.

This dataset is divided into three parts:

- A training set that your model will learn on: $\mathbf{X}_{train}$, $\mathbf{y}_{train}$
- A validation set for determining the regularization hyper-parameter: $\mathbf{X}_{val}$, $\mathbf{y}_{val}$

- A test set for evaluating the final performance. These are "unseen" examples which your model did not see during training: $\mathbf{X}_{test}$, $\mathbf{y}_{test}$

The next step of **ex4.py** will plot the training data (figure 1). In the following parts, you will implement linear regression and use that 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.

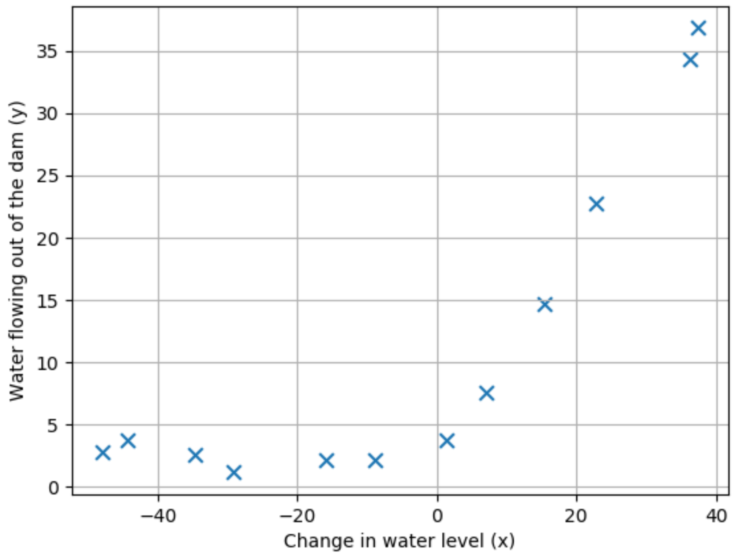

Figure 1: Data used in this assignment.

3.2 Regularized linear regression cost function

Recall that regularized linear regression has the following cost function:

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=0}^{m-1} \left(h_{\boldsymbol{\theta}}(\mathbf{x}^i) - y^i\right)^2 + \frac{\lambda}{2m} \left(\sum_{j=1}^{n} \theta_j^2 \right)$$

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. 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.py**. Your task is to write a function to calculate the regularized linear regression cost function. If possible, try to vectorize your code and avoid writing loops. When you have finished, the next part of **ex4.py** will run your cost function using theta initialized at $[1, 1]^T$. You should expect to see an output of 303.993.

3.3 Regularized linear regression gradient

Correspondingly, the partial derivative of regularized linear regression's cost for θ_j is defined as

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = \frac{1}{m} \sum_{i=0}^{m-1} \left(h_{\boldsymbol{\theta}}\left(\mathbf{x}^i\right) - y^i\right) x_j^i \qquad \text{pour } j = 0$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = \frac{1}{m} \left(\sum_{i=0}^{m-1} \left(h_{\boldsymbol{\theta}}\left(x^i\right) - y^i\right) x_j^i + \lambda \theta_j \right) \qquad \text{pour } j \geq 1$$

In **linearRegGradientFunction.py**, add code to calculate the gradient, returning it in the variable grad. When you are finished, the next part of **ex4.py** will run your gradient function using theta initialized at $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\lambda = 1$. You should expect to see a gradient of [-15.30 598.250]^T.

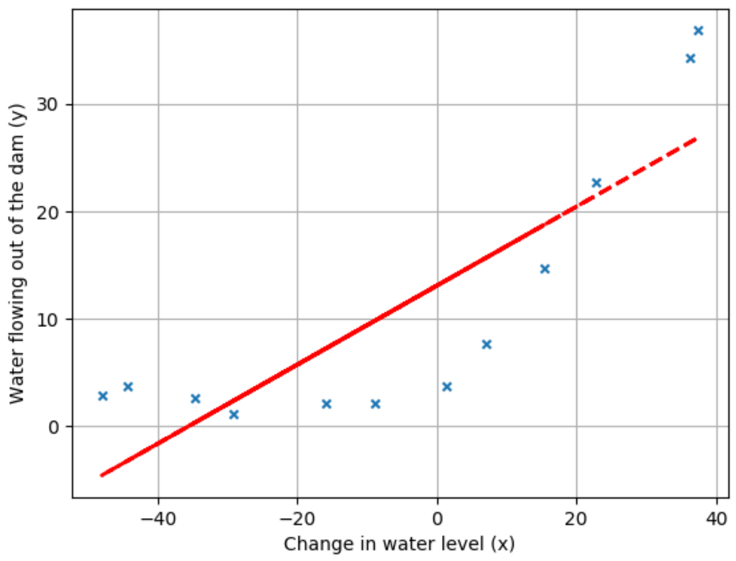

Figure 2: Linear fit on the data.

3.4 Fitting linear regression

Once your cost function and gradient are working correctly, the next part of **ex4.py** will call the function **trainLinearReg.py**. You should complete this function to compute the optimal values of $\boldsymbol{\theta}$ and the training and validation costs according to iteration number.

Recall that the training/validation cost for a dataset is defined as

$$J_{train/validation}(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=0}^{m-1}\left(h_{\boldsymbol{\theta}}(x^i) - y^i\right)^2$$

In particular, note that the training/validation cost does not include the regularization term. One way to compute the training/validation error is to use your existing cost function and set λ to 0 only when using it to compute the training/validation error.

In this specific part of **ex4.py**, we set regularization parameter λ to zero. Because our current implementation of linear regression is trying to fit a 2-dimensional $\boldsymbol{\theta}$, regularization will not be incredibly helpful for a $\boldsymbol{\theta}$ of such low dimension. In the later parts of the exercise, you will be using polynomial regression with regularization.

Finally, the **ex4.py** script should also plot the best fit line, resulting in an image similar to Figure 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. 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 section, 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.

4 Bias-variance

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 validation/test costs on a learning curve to diagnose bias-variance problems.

You may wish to supplement the lesson by watching :this video!

4.1 Learning curves

You will now implement code to generate the classical learning curves that will be useful in debugging learning algorithms. Recall that a learning curve plots training and validation/test costs as a function of the iteration number. Your job is to fill in **learningCurve.py** plotting a vector of costs for the training set and validation set.

To plot the learning curve, we need a training and validation set cost for the different iterations. You can use the **trainLinearReg** function to find the θ parameters and the costs according the iteration number.

When you are finished, **ex4.py** will print the learning curves and produce a plot similar to Figure 3.

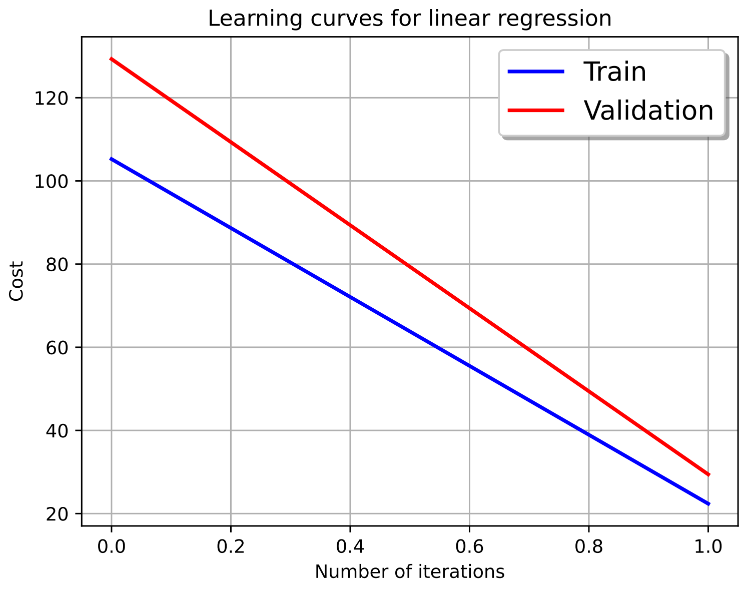

Figure 3: Linear regression learning curve

In Figure 3, you can observe that both the training and validation costs are high. This reflects a high bias problem in the model, the linear regression model is too simple and is unable to fit our dataset well as depicted in Figure 2. In the next section, you will implement polynomial regression to fit a better model for this dataset.

You may wish to supplement the lesson by watching :this video.

5 Polynomial regression

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.

To use polynomial regression, our hypothesis has the form:

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

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.py** in which the function maps the original training set $\mathbf{X}_{train}$ 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 $Xpoly$, 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-th power in this function.

Now you have a function that will map features to a higher dimension,and Part 6 of **ex4.py** will apply it to the training set, the validation set, and the test set (which you haven't used yet).

5.1 Learning Polynomial Regression

After you have completed **polyFeatures.py**, the **ex4.py** 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 6. 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 $x^6 = 40^6 = 4096000000$). Therefore, you will need to use feature normalization.

Before learning the parameters $\boldsymbol{\theta}$ for the polynomial regression, **ex4.py** 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 assignment.

After learning the parameters $\boldsymbol{\theta}$, you should see two plots (Figure 4) generated for polynomial regression with $\lambda = 0$.

From Figure 4 (left), you should see that the polynomial fit is able to follow the datapoints very well - thus, obtaining a low training cost. 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 (λ= 0) model, you can see that the learning curve (Figure 4 right) shows the same effect where the training error is low, but the validation error 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. In the next section, you will get to try different λ parameters to see how regularization can lead to a better model.

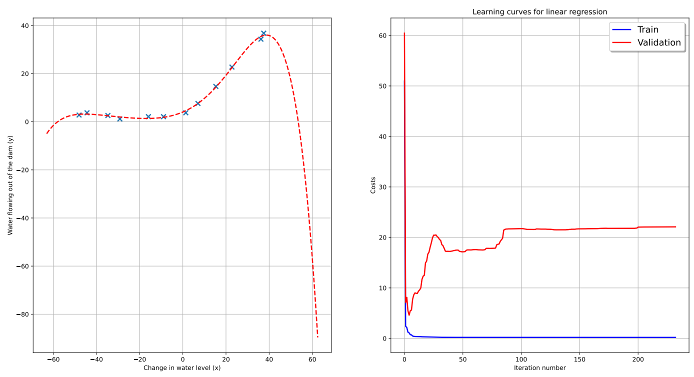

Figure 4: Polynomial (a) fit for $\lambda = 0$, (b) learning curve.

5.2 Adjusting the regularization hyper-parameter

In this section, you will get to observe how the regularization parameter affects the bias-variance of regularized polynomial regression. The script **ex4.py** proposes a loop over $\lambda \in \{0, \dots, 100\}$. For each of these values, the script should generate a polynomial fit to the data and also a learning curve.

For $\lambda = 1$, you should see a polynomial fit (Figure 5) that follows the data trend well and a learning curve showing that both the validation and training costs converge to a relatively low value. This shows the $\lambda = 1$ regularized polynomial regression model does not have the high-bias or high-variance problems. In effect, it achieves a good trade-off between bias and variance.

For $\lambda = 100$, you should see a polynomial fit (Figure 6) that does not follow the data well. In this case, there is too much regularization and the model is unable to fit the training data.

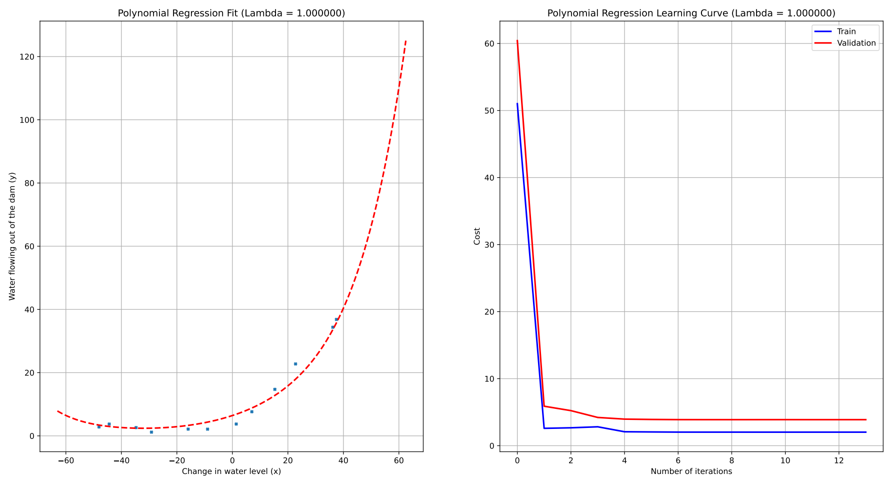

Figure 5: Polynomial (a) fit for $\lambda = 1$, (b) learning curve.

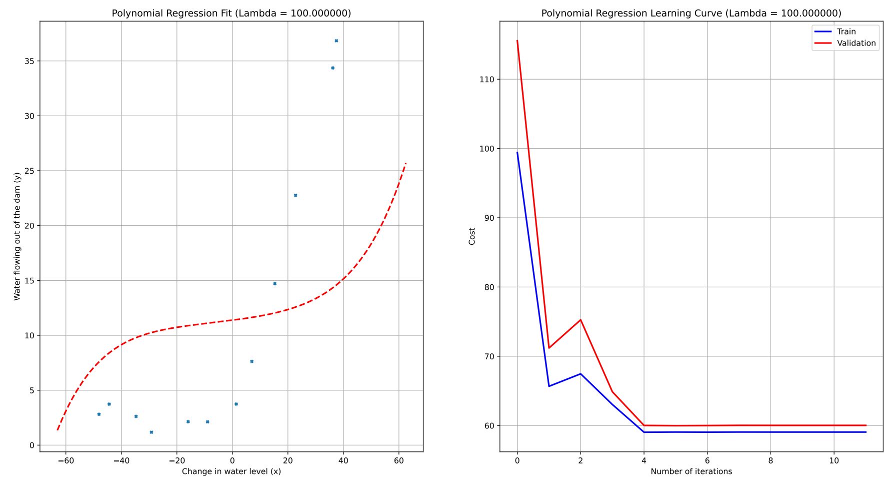

Figure 6: Polynomial (a) fit for $\lambda = 100$, (b) learning curve.

You may wish to supplement the lesson by watching :this video.

5.3 Selecting λ using a validation set

From the previous parts of the exercise, you 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 (λ= 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.

The **ex4.py** proposes a last plot of the training and validation costs versus the various lambda values. Due to randomness in the training and validation splits of the dataset, note that the validation error can sometimes be lower than the training error.

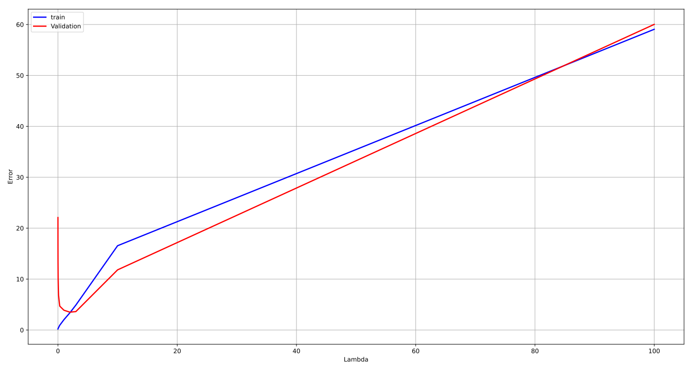

Figure 7: Selection λ using a validation set.

5.4 Computing test set error

After selecting the best λ hyper-parameter 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. To get a better indication of the 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 λ **hyper-parameter**, nor to learn the model **parameters** $\boldsymbol{\theta}$).

Note that the final training set used for the final evaluation is the concatenation of the initial training and validation set.

5.5 Some questions, you have to answer...

Le codage n'est qu'un prétexte pour comprendre les différents concepts du machine learning. **Ici aucune équation...Exprimez avec vos mots les concepts pour les comprendre.**

L'évaluation des performances est extrêmement important; il s'agit de bien comprendre les concepts du machine learning.

- Comment découpe-t-on un jeu de données pour entraîner et évaluer un modèle de prédiction?
- Définissez ce qu'est la capacité d'un modèle de prédiction (i.e. sa complexité). Sur quel ensemble l'évalue-t-on?
- Définissez le concept d'erreur de généralisation? Sur quel ensemble l'évalue-t-on?
- A quoi sert l'ensemble de validation?
- Comment trouver les paramètres et fixer les hyperparamètres ?
- A quoi sert la "learning curve"?