Homework 2: Logistic Regression

Machine Learning

In this exercise, you will implement logistic regression and apply it to two different datasets.

1 Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university. Suppose that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams. You have historical data from previous applicants that you can use as a training set for logistic regression. For each training example, you have the applicant's scores on two exams and the admissions decision. Your task is to build a classification model that estimates an applicant's probability of admission based the scores from those two exams.

1.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of your code, it will load the data and display it on a 2-dimensional plot by calling the function plotData. You will now create the code so that it displays a Figure like Figure 1 (`ex2data1`), where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

Helper:

```
import numpy as np
import pandas as pd
import matplotlib as mpl
from matplotlib import rc, cm
from scipy import optimize

with open('ex2data1.txt') as f1, open('ex2data2.txt') as f2:
    dataset1 = np.loadtxt(f1, delimiter = ',',
        dtype = 'float', usecols = None)
    dataset2 = np.loadtxt(f2, delimiter = ',',
        dtype = 'float', usecols = None)

X = dataset1[:, :-1]
Y = dataset1[:, 2]
KO = np.where(Y == 0)[0]
OK = np.where(Y)[0]

Your Code here (drawing)
```

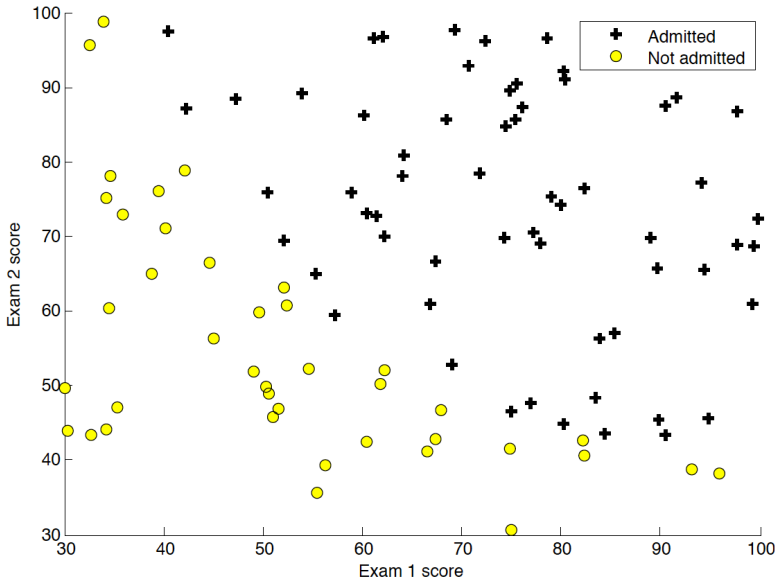

Figure 1: Scatter plot of training data

1.2 Implementation

Before you start with the actual cost function, recall that the logistic regression hypothesis is defined as:

$$h(\mathbf{\Theta}) = g(\mathbf{\Theta}^t \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{\Theta}^t \mathbf{x}}}$$

where function g is the sigmoid function:

$$g(z) = \frac{1}{1 + e^{-\mathbf{z}}}$$

Your first step is to implement this function in sigmoid so it can be called by the rest of your program. When you are finished, try testing a few values by calling `sigmoid(x)`. For large positive values of x, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0. Evaluating `sigmoid(0)` should give you exactly 0.5. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid function on every element.

Helper:

```
def sigmoid (z):
    return 1.0 / (1.0 + np.exp(-z))
```

1.3 Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Recall that the cost function in logistic regression is

$$J(\mathbf{\Theta}) = -\frac{1}{m}\left(\sum_{i=1}^{m}(y)_i \log h((\mathbf{x})_i) + (1 - (y)_i) \log(1 - h((\mathbf{x})_i)\right)$$

and the gradient of the cost is a vector of the same length as $\mathbf{\Theta}$ where the j^{th} element (for $j = 0, 1, \ldots, n$) is defined as follows:

$$\frac{\partial J(\mathbf{\Theta})}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h((\mathbf{x})_i) - y_i)(x_j)_i$$

Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h(\mathbf{x})$ (used for the logistic regression) or $f(\mathbf{x})$ (used for the linear regression).

Helper:

```
def costFunction (theta, x, y):
    m = len(x)
    h = sigmoid(np.inner(x, theta))
    return Your Code here
```

1.4 Learning parameters using `scipy.optimize.minimize`

In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradient descent. You wrote a cost function calculated its gradient, then took a gradient descent step accordingly. This time, instead of taking gradient descent steps, you will use a built-in function called `scipy.optimize.minimize` which is an optimization solver that finds the minimum of an unconstrained function. For logistic regression, you want to optimize the cost function $J(\Theta)$ with parameters Θ. Concretely, you are going to use `scipy.optimize.minimize` to find the best parameters Θ for the logistic regression cost function, given a fixed dataset (of x and y values). If you have completed your cost function correctly, `scipy.optimize.minimize` will converge on the right optimization parameters and return the final values of the cost and Θ. This final value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 2

Helper:

```
newX = np.concatenate((np.ones((100, 1)), X), axis=1)
    theta = (0, 0, 0)
    theta = Your Code here...

plt.figure()

Your Code here...
```

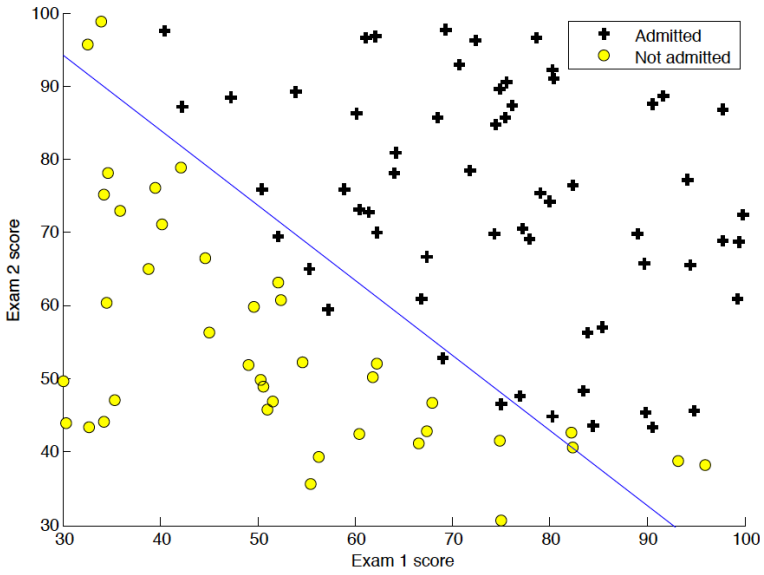

Figure 2: Training data with decision boundary

1.5 Evaluating logistic regression

After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.776.

2 Regularized logistic regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly. Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.

2.1 Visualizing the data

Similar to the previous parts of this exercise, the figure (`ex2data2`) shows the two test scores, the positive ($y = 1$, accepted) and negative ($y = 0$, rejected) examples are shown with different markers. Figure shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

2.2 Freature mapping

One way to fit the data better is to create more features from each data point. You can create a"mapFeature" function, to map the features into all polynomial terms of x_1 and x_2 up to the sixth power:

$$\begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{pmatrix}$$

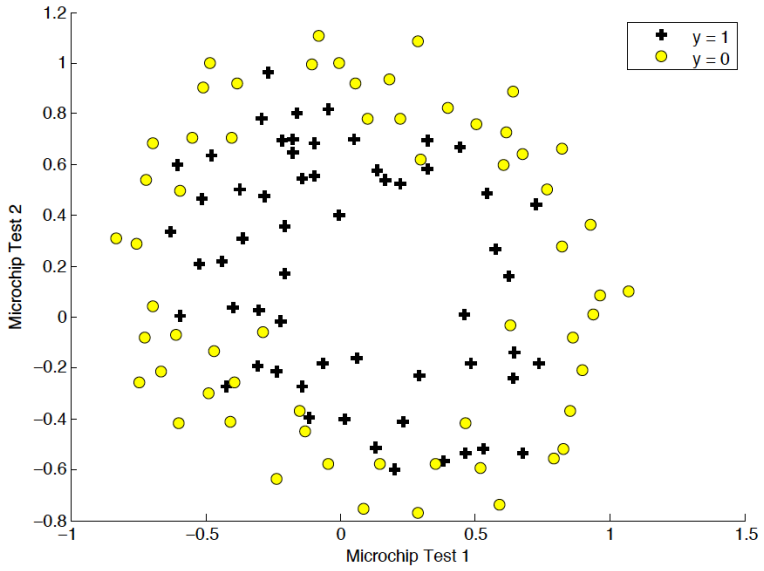

Figure 3: Plot of training data

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-dimensional vector. A logistic regression classifer trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear when drawn in our 2-dimensional plot. While the feature mapping allows us to build a more expressive classifier, it also more susceptible to overftting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

2.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Recall that the regularized cost function in logistic regression is

$$J(\mathbf{\Theta}) = -\frac{1}{m}\left(\sum_{i=1}^{m}(y)_i \log h((\mathbf{x})_i) + (1-(y)_i)\log(1-h((\mathbf{x})_i)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Note that you should not regularize the parameter θ_0. The gradient of the cost function is a vector where the j^{th} element is defined as follows:

$$\frac{\partial J(\mathbf{\Theta})}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h(\mathbf{x}_i) - y_i)(x_0)_i$$

for $j = 0$ and for $j \geq 1$,

$$\frac{\partial J(\mathbf{\Theta})}{\partial \theta_j} = \left[\frac{1}{m} \sum_{i=1}^{m} \left(h(\mathbf{x}_i) - y_i \right) (x_j)_i + \frac{\lambda}{m} \theta_j \right]$$

The initial values of $\mathbf{\Theta}$ are all zero. Once you have implemented the function you should find a cost of approximatively 0.693.

2.4 Learning paramters using `scipy.optimize.minimize`

Similar to the previous parts, you will use `scipy.optimize.minimize` to learn the optimal parameters.

2.5 Plotting the decision boundary

To help you visualize the model learned by this classifier, you can plot the (non-linear) decision boundary that separates the positive and negative examples. You can plot the non-linear decision boundary by computing the classifier's predictions on an evenly spaced grid and then and drew a contour plot of where the predictions change from $y = 0$ to $y = 1$. After learning the parameters, the next step is to plot a decision boundary similar to the Figure.

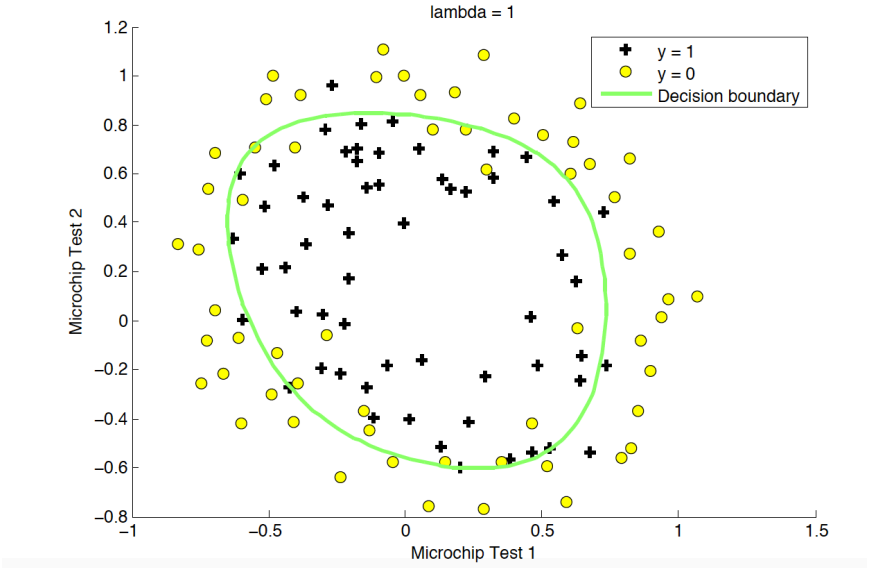

Figure 4: Training data with decision boundary ($\lambda = 1$)

3 Regularized Linear Regression and Biais-Variance

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

3.1 Visualizing the dataset

We will begin by visualizing the dataset containing historical records (`ex5data1`) 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: x, y
- A **cross validation** set for determining the regularization parameter: $xval$, $yval$
- A **test** set for evaluating performance. These are "unseen" examples which your model did not see during training: $xtest$, $ytest$

The next step will be to plot the training data (Figure). 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.

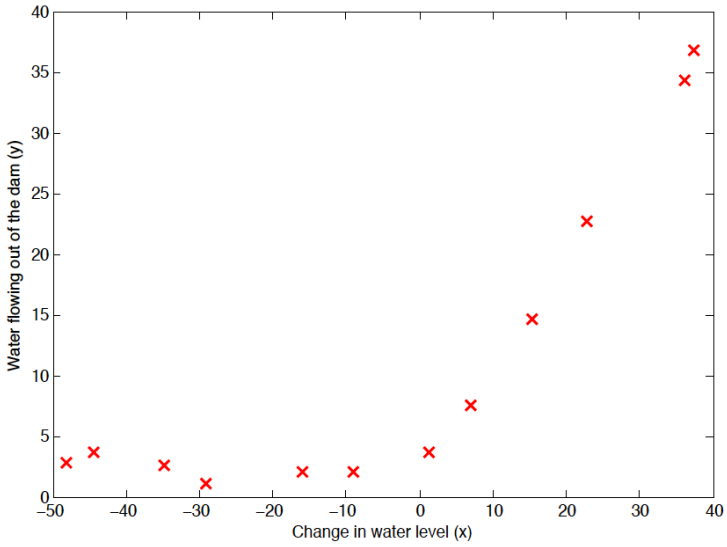

Figure 5: Data

3.2 Regularized linear regression cost function

Recall that regularized linear regression has the following cost function:

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

where λ is a regularization parameter which controls the degree of regularization (thus, help preventing overfitting). The regularization term puts a penalty on the overal cost J. As the magnitudes of the model parameters j increase, the penalty increases as well. Note that you should not regularize the 0 term. You should now complete your code. 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.

3.3 Regularized linear regression gradient

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

$$\frac{\partial J(\mathbf{\Theta})}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(f(\mathbf{x}_i) - y_i)(x_0)_i$$

for $j = 0$ and for $j \geq 1$,

$$\frac{\partial J(\mathbf{\Theta})}{\partial \theta_j} = \left[\frac{1}{m}\sum_{i=1}^{m}\left(f(\mathbf{x}_i) - y_i\right)(x_1)_i + \frac{\lambda}{m}\theta_j\right]$$

Complete your code to calculate the gradient.

3.4 Fitting linear regression

Once your cost function and gradient are working correctly, the next part consists to compute the optimal values of θ (you can use an optimization function). In this part, we set regularization parameter λ to zero. Because our current implementation of linear regression is trying to 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.

Finally, your script should also plot the best fit line, resulting in an image similar to the Figure below. 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.

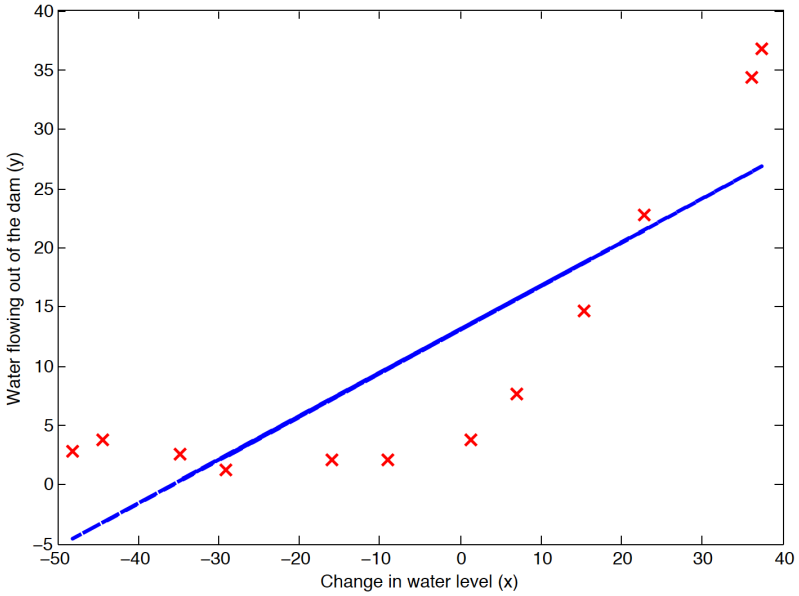

Figure 6: Linear Fit

3.5 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 test errors on a learning curve to diagnose bias-variance problems.

3.6 Learning curves

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 your code ('learningcurve' or the name you gave to it) 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_i and y_i). 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

$$J_{\text{train}}(\mathbf{\Theta}) = \frac{1}{2m} \sum_{i=1}^{m} (h((\mathbf{x})_i) - (y)_i)^2$$

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_i and y_i) (instead of the entire training set). However, for the cross validation error, you should compute it over the entire cross validation set. When you are finished, you should print a figure such as

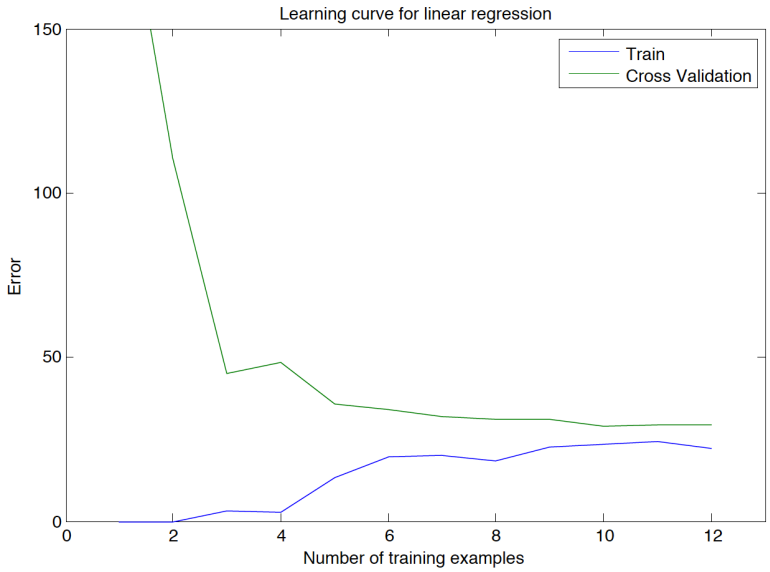

Figure 7: Linear regression learning curve

In the 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 fit a better model for this dataset.

3.7 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. For use polynomial regression, our hypothesis has the form:

$$\begin{aligned} f(\mathbf{x}) &= \theta_0 + \theta_1 \times (\text{WaterLevel}) + \theta_2 \times (\text{WaterLevel})^2 + \ldots + \theta_p \times (\text{WaterLevel})^p \\ &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_p x_p \end{aligned}$$

with x_1 =(WaterLevel), x_2 =(WaterLevel)^2. ..., 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. You should create a function (polyfeatures). Specifically, when a training set x of size $m \times 1$ is passed the function should return a matrix $m \times p$ where column 1 holds the original values of x, colomn 2 the values of x^2, and so on. Now you have a function that will map features to a higher dimension, you can apply it to the training set, the test set and the cross validation set.

3.8 Learning Polynomial Regression

After you have completed your function (polyfeatures), your code should 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, will not work well as the features would be badly scaled (e.g., an example with $x = 40$ will have a feature $x_8 = 40^8 = 6.5^{12}$). Therefore, you will need to use feature normalization. You should write a function (FeatureNormalization for instance) which stores the parameters μ and σ separately. After learning the parameters θ you should be able to plot figures generated for the polynomial regression with $\lambda = 0$:

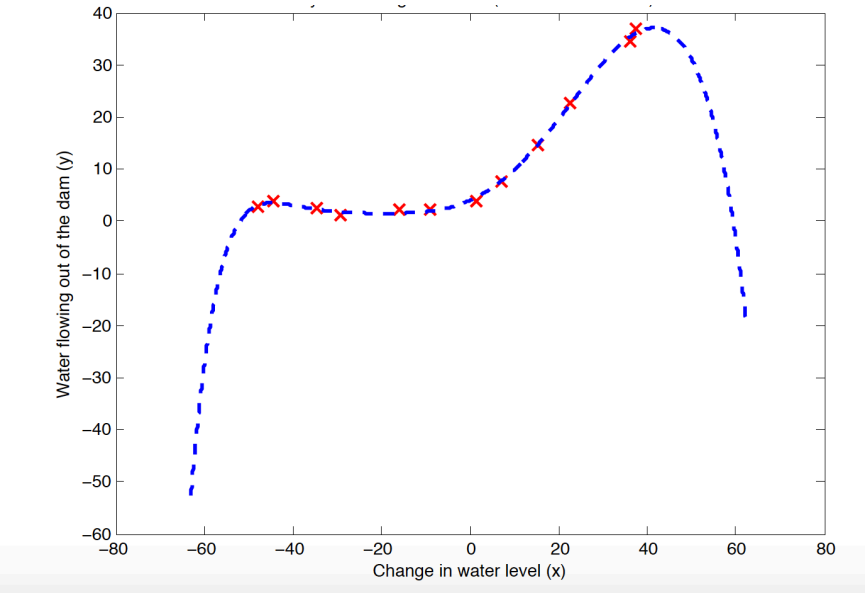

Figure 8: Polynomial fit, $\lambda = 0$

From this figure, 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.

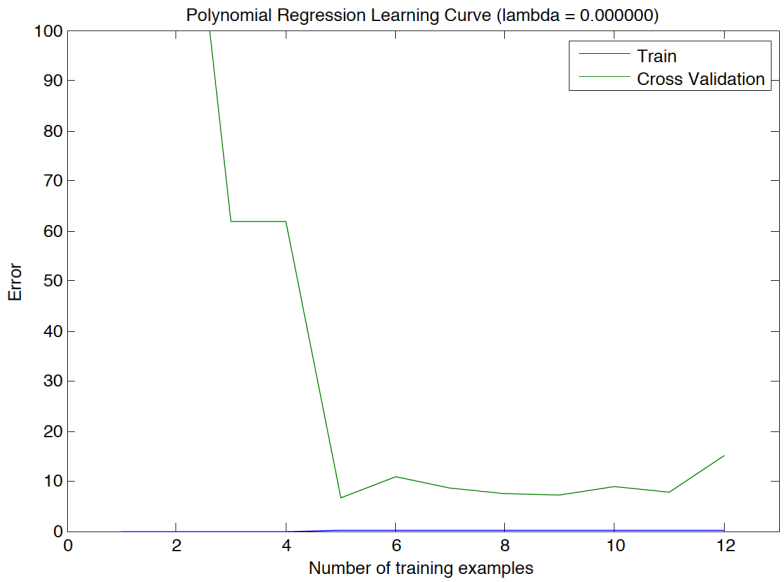

Figure 9: Polynomial learning curve, $\lambda = 0$

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve shows (last figure) 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. In the next section, you will get to try different parameters to see how regularization can lead to a better model.