Programming Exercise 3:
Multi-class Classification
&
Regularized Linear Regression and Bias v.s. Variance

Introduction

In this exercise, you will implement one-vs-all logistic regression. To get started with the exercise, you will need to download the starter code (available on Canvas, under Assignments - "LAB 3") and unzip its contents to the directory where you wish to complete the exercise. If needed, use the cd command in Octave/MATLAB to change to this directory before starting this exercise.

This exercise se two parts: Part 1 – Multi-class Classification and Part 2: Regularized Linear Regression and Bias v.s. Variance. Each part is saved in a separate folder: lab3-1 and lab3-2.

Where to get help

The exercises in this course use Octave[1] or MATLAB, a high-level programming language well-suited for numerical computations. If you do not have Octave or MATLAB installed, please refer to the installation instructions in the "Environment Setup Instructions" of the course website.
At the Octave/MATLAB command line, typing `help` followed by a function name displays documentation for a built-in function. For example, `help plot` will bring up help information for plotting. Further documentation for Octave functions can be found at the Octave documentation pages. MAT- LAB documentation can be found at the MATLAB documentation pages.

1 Multi-class Classification

Part 1 – Multi-class Classification is placed under lab3-1 folder and includes the following files.

`ex3.m` - Octave/MATLAB script that steps you through part 1

`ex3data1.mat` - Training set of hand-written digits
`displayData.m` - Function to help visualize the dataset
`fmincg.m` - Function minimization routine (similar to fminunc)

[1] Octave is a free alternative to MATLAB. For the programming exercises, you are free to use either Octave or MATLAB.

`sigmoid.m` - Sigmoid function
[*] `lrCostFunction.m` - Logistic regression cost function
[*] `oneVsAll.m` - Train a one-vs-all multi-class classifier
[*] `predictOneVsAll.m` - Predict using a one-vs-all multi-class classifier

* indicates files you will need to complete

Throughout the exercise, you will be using the main script `ex3.m`. This script set up the dataset for the problems and make calls to functions that you will write. You do not need to modify these scripts. You are only required to modify functions in other files, by following the instructions in this assignment.

For this exercise, you will use logistic regression and neural networks to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how the methods you've learned can be used for this classification task.

In the first part of the exercise, you will extend your previous implementation of logistic regression and apply it to one-vs-all classification.

1.1 Dataset

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits.[2] The `.mat` format means that that the data has been saved in a native Octave/MATLAB matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the `load` command. After loading, matrices of the correct dimensions and values will appear in your program's memory. The matrix will already be named, so you do not need to assign names to them.

```
% Load saved matrices from file
load('ex3data1.mat');
% The matrices X and y will now be in your Octave environment
```

There are 5000 training examples in `ex3data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is "unrolled" into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix X. This gives us a 5000 by 400 matrix `X` where every row is a training example for a handwritten digit image.

[2] This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/).

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector `y` that contains labels for the training set. To make things more compatible with Octave/MATLAB indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a "0" digit is labeled as "10", while the digits "1" to "9" are labeled as "1" to "9" in their natural order.

1.2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of ex3.m, the code randomly selects 100 rows from X and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. We have provided the `displayData` function, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 1.

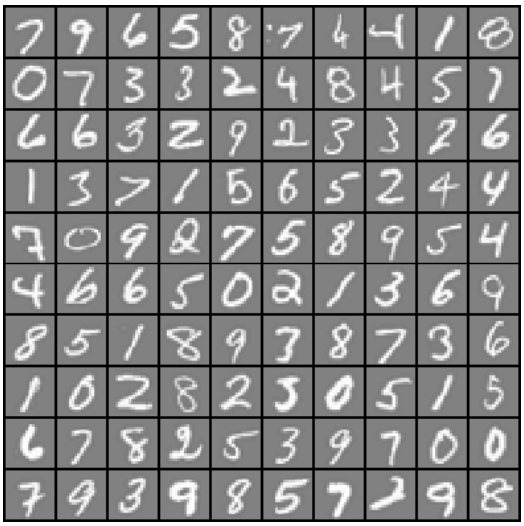

Figure 1: Examples from the dataset

1.3 Vectorizing Logistic Regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized. In this section, you will implement a vectorized version of logistic regression that does not employ any `for` loops. You can use your code in the last exercise as a starting point for this exercise.

1.3.1 Vectorizing the cost function

We will begin by writing a vectorized version of the cost function. Recall that in (unregularized) logistic regression, the cost function is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right].$$

To compute each element in the summation, we have to compute every example *i*, where the hypothesis function and sigmoid functions are

$$h_\theta(x^{(i)}) = g(\theta^T x^{(i)})$$

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

It turns out that we can compute this quickly for all our examples by using matrix multiplication. Let us define X and θ as

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix} \quad \text{and} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}.$$

Then, by computing the matrix product $X\theta$, we have

$$X\theta = \begin{bmatrix} — (x^{(1)})^T\theta — \\ — (x^{(2)})^T\theta — \\ \vdots \\ — (x^{(m)})^T\theta — \end{bmatrix} = \begin{bmatrix} — \theta^T(x^{(1)}) — \\ — \theta^T(x^{(2)}) — \\ \vdots \\ — \theta^T(x^{(m)}) — \end{bmatrix}$$

In the last equality, we used the fact that $a^T b = b^T a$ if a and b are vectors. This allows us to compute the products $\theta^T x^{(i)}$ for all our examples i in one line of code.

Your job is to write the unregularized cost function in the file `lrCostFunction.m` Your implementation should use the strategy we presented above to calculate $\theta^T x^{(i)}$. You should also use a vectorized approach for the rest of the cost function. A fully vectorized version of `lrCostFunction.m` should not contain any loops.

(Hint: You might want to use the element-wise multiplication operation (`.*`) and the sum operation `sum` when writing this function).

Implement J in `lrCostFunction` *and run* `ex3` *to test this implementation.*

1.3.2 Vectorizing the gradient

Recall that the gradient of the (unregularized) logistic regression cost is a vector where the j^{th} element is defined as

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right).$$

To vectorize this operation over the dataset, we start by writing out all the partial derivatives explicitly for all θ_j.

$$\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right). \\
\begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} &= \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}\right) \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}\right) \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}\right) \\ \vdots \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)}\right) \end{bmatrix} \\
&= \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x^{(i)}\right) \\
&= \frac{1}{m}X^T(h_\theta(x) - y).
\end{aligned} \tag{1}$$

where

$$h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(1)}) - y^{(m)} \end{bmatrix}$$

Note that $x^{(i)}$ is a vector, while $(h_\theta(x^{(i)}) - y^{(i)})$ is a scalar (single number). To understand the last step of the derivation, let $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$ and observe that

$$\sum_i \beta_i x^{(i)} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} = X^T \beta,$$

where the values $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$.

The expression above allows us to compute all the partial derivatives without any loops. If you are comfortable with linear algebra, we encourage you to work through the matrix multiplications above to convince yourself that the vectorized version does the same computations. You should now implement Equation 1 to compute the correct vectorized gradient. Once you are done, complete the function `lrCostFunction.m` by implementing the gradient.

Debugging Tip: Vectorizing code can sometimes be tricky. One common strategy for debugging is to print out the sizes of the matrices you are working with using the size function. For example, given a data matrix X of size 100 × 20 (100 examples, 20 features) and θ, a vector with dimensions 20×1, you can observe that $X\theta$ is a valid multiplication operation, while θX is not. Furthermore, if you have a non-vectorized version of your code, you can compare the output of your vectorized code and non-vectorized code to make sure that they produce the same outputs.

Calculate gradient in `lrCostFunction.m` and run ex3 to test the implementation.

1.3.3 Vectorizing regularized logistic regression

After you have implemented vectorization for logistic regression, you will now add regularization to the cost function. Recall that for regularized logistic regression, the cost function is defined as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2.$$

Note that you should *not* be regularizing θ_0 which is used for the bias term. Correspondingly, the partial derivative of regularized logistic regression cost for θ_j is defined as

θ

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for } j = 0$$

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

Now modify your code in `lrCostFunction` to account for regularization. Once again, you should not put any loops into your code.

Octave/MATLAB Tip: When implementing the vectorization for regularized logistic regression, you might often want to only sum and update certain elements of θ. In Octave/MATLAB, you can index into the matrices to access and update only certain elements. For example, `A(:, 3:5) = B(:,1:3)` will replaces the columns 3 to 5 of A with the columns 1 to 3 from B. One special keyword you can use in indexing is the end keyword in indexing. This allows us to select columns (or rows) until the end of the matrix. For example, `A(:,2:end)` will only return elements from the 2nd to last column of A. Thus, you could use this together with the sum and .^ operations to compute the sum of only the elements you are interested in (e.g., `sum(z(2:end).^2))`. In the starter code, `lrCostFunction.m`, we have also provided hints on yet another possible method computing the regularized gradient.

Implement J in `lrCostFunction` *and run* `ex3` *to test this implementation.*

1.4 One-vs-all Classification

In this part of the exercise, you will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset (Figure 1). In the handwritten digits dataset, $K = 10$, but your code should work for any value of K.

You should now complete the code in `oneVsAll.m` to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix Θ of dimension $Kx(N+1)$, where each row of Θ corresponds to the learned logistic regression parameters for one class. You can do this with a "for"-loop from 1 to K, training each classifier independently.

Note that the `y` argument to this function is a vector of labels from 1 to 10, where we have mapped the digit "0" to the label 10 (to avoid confusions with indexing).

When training the classifier for class $k \in \{1, ..., K\}$, you will want a m-dimensional vector of labels y, where yj ∈ 0, 1 indicates whether the j-th training instance belongs to class k ($y_j = 1$), or if it belongs to a different class ($y_j = 0$). You may find logical arrays helpful for this task.

Octave/MATLAB Tip: Logical arrays in Octave/MATLAB are arrays which contain binary (0 or 1) elements. In Octave/MATLAB, evaluating the expression `a == b` for a

vector a (of size m×1) and scalar b will return a vector of the same size as a with ones at positions where the elements of a are equal to b and zeroes where they are different. To see how this works for yourself, try the following code in Octave/MATLAB:

```
a = 1:10; % Create a and b
b = 3;
a == b % You should try different values of b here
```

Furthermore, you will be using `fmincg` for this exercise (instead of `fminunc`). `fmincg` works similarly to `fminunc`, but is more efficient for dealing with a large number of parameters.
After you have correctly completed the code for `oneVsAll.m`, the script `ex3.m` will continue to use your `oneVsAll` function to train a multi-class classifier.

You should now implement ad test the results running `ex3.m`.

1.4.1 One-vs-all Prediction

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the "probability" that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label (1, 2,..., or *K*) as the prediction for the input example.
You should now complete the code in `predictOneVsAll.m` to use the one-vs-all classifier to make predictions.

Once you are done, `ex3.m` will call your `predictOneVsAll` function using the learned value of Θ. You should see that the training set accuracy is about 94.9% (i.e., it classifies 94.9% of the examples in the training set correctly).

You should now implement ad test the results running `ex3.m`.

Submit the following files on Canvas, under Assignments and "LAB 3":

[*] `lrCostFunction.m` - Logistic regression cost function
[*] `oneVsAll.m` - Train a one-vs-all multi-class classifier
[*] `predictOneVsAll.m` - Predict using a one-vs-all multi-class classifier

Programming Exercise 3.2: Regularized Linear Regression and Bias v.s. Variance

Introduction

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

Files included in this exercise

`ex5.m` - Octave/MATLAB script that steps you through the exercise
`ex5data1.mat` - Dataset
`submit.m` - Submission script that sends your solutions to our servers
`featureNormalize.m` - Feature normalization function
`fmincg.m` - Function minimization routine (similar to `fminunc`)
`plotFit.m` - Plot a polynomial fit
`trainLinearReg.m` - Trains linear regression using your cost function
`[*] linearRegCostFunction.m` - Regularized linear regression cost function
`[*] learningCurve.m` - Generates a learning curve
`[*] polyFeatures.m` - Maps data into polynomial feature space
`[*] validationCurve.m` - Generates a cross validation curve

* indicates files you will need to complete

Throughout the exercise, you will be using the script `ex5.m`. These scripts set 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.

Where to get help

At the Octave/MATLAB command line, typing `help` followed by a function name displays documentation for a built-in function. For example, `help plot` will bring up help information for plotting. Regularized Linear Regression

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.
The provided script, `ex5.m`, will help you step through this exercise.

1.1 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: `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 of `ex5.m` 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.

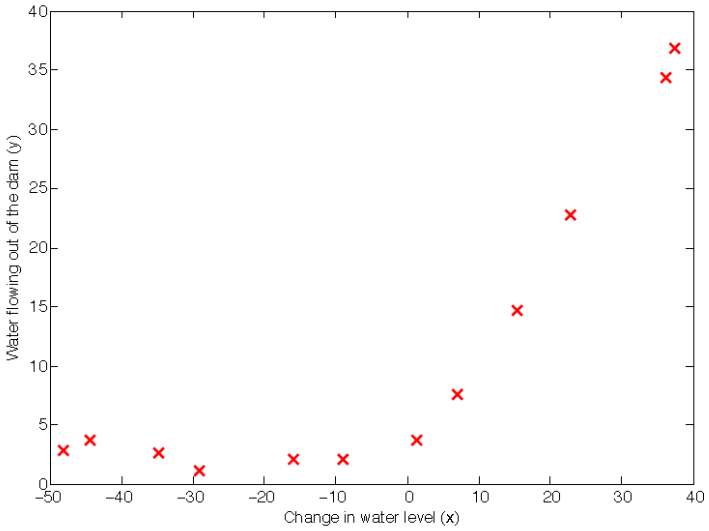

Figure 1: Data

1.2 Regularized linear regression cost function

Recall that regularized linear regression has the following cost function:

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

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. (In Octave/MATLAB, the θ_0 term is represented as `theta(1)` since indexing in Octave/MATLAB starts from 1).

You should now complete the code in the file `linearRegCostFunction.m`. 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 are finished, the next part of

`ex5.m` will run your cost function using `theta` initialized at `[1; 1]`. You should expect to see an output of 303.993.

You should now implement and test the assignment.

1.3 Regularized linear regression gradient

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

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for } j = 0$$

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

In `RegCostFunction.m`, add code to calculate the gradient, re- turning it in the variable `grad`. When you are finished, the next part of `ex5.m` will run your gradient function using `theta` initialized at `[1; 1]`. You should expect to see a gradient of `[-15.30; 598.250]`.

You should now implement and test the assignment.

1.4 Fitting linear regression

Once your cost function and gradient are working correctly, the next part of `ex5.m` 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 λ to zero. 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.

Finally, the `ex5.m` 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.

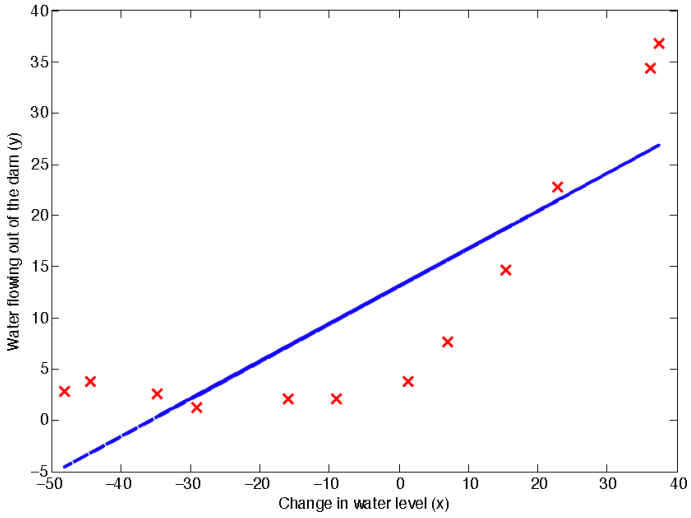

Figure 2: Linear Fit

2 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.

2.1 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 `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 train- ing 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, `ex5.m` will print the learning curves and produce a plot similar to Figure 3.

In Figure 3, 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.

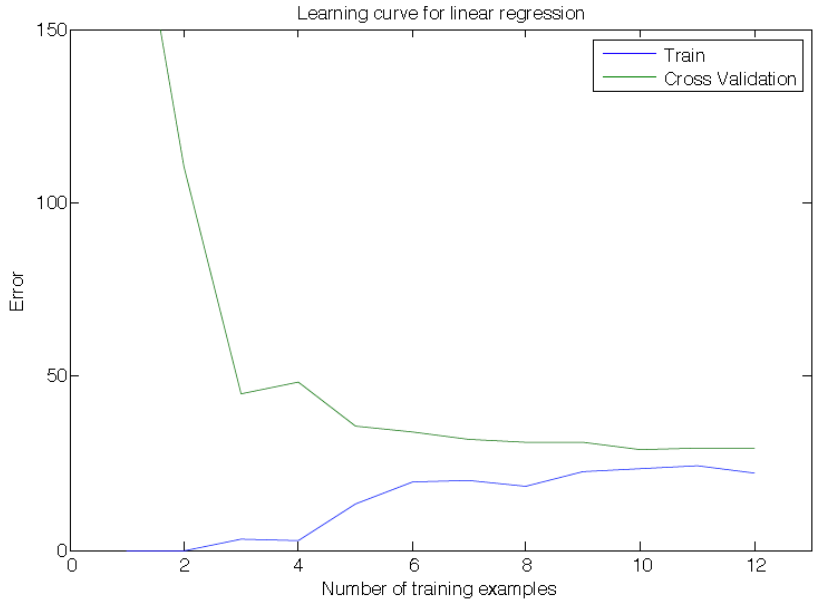

Figure 3: Linear regression learning curve

3 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:

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

Notice that by defining

x_1 = (waterLevel), x_2 = (waterLevel)^2, . . . , 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×1 into its higher powers. Specifically, when a training set X of size m×1 is passed into the function, the function should return a m×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 ex5.m will apply it to the training set, the test set, and the cross validation set (which you haven't used yet).

You should implement and test your implementation.

3.1 Learning Polynomial Regression

After you have completed polyFeatures.m, the ex5.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, will not work well as the features would be badly scaled (e.g., an example with $x = 40$ will now have a feature $x_8 = 40^8 = 6.5\ 10^{12}$).

Therefore, you will need to use feature normalization.

Before learning the parameters θ for the polynomial regression, ex5.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 (Figure 4,5) generated for polynomial regression with $\lambda = 0$.

From Figure 4, 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.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve (Figure 5) 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.

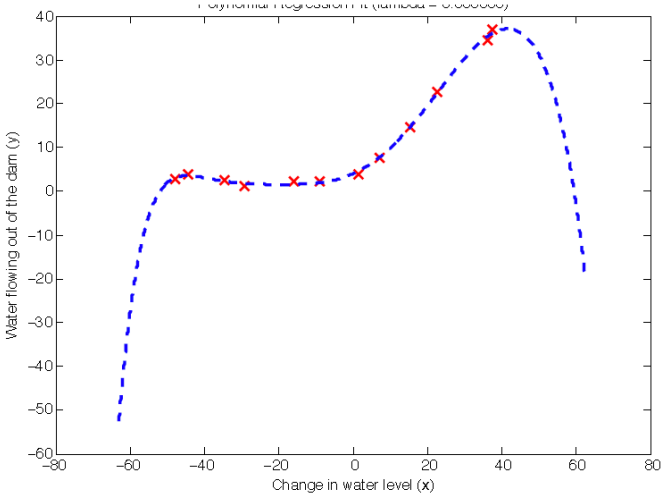

Figure 4: Polynomial fit, $\lambda = 0$

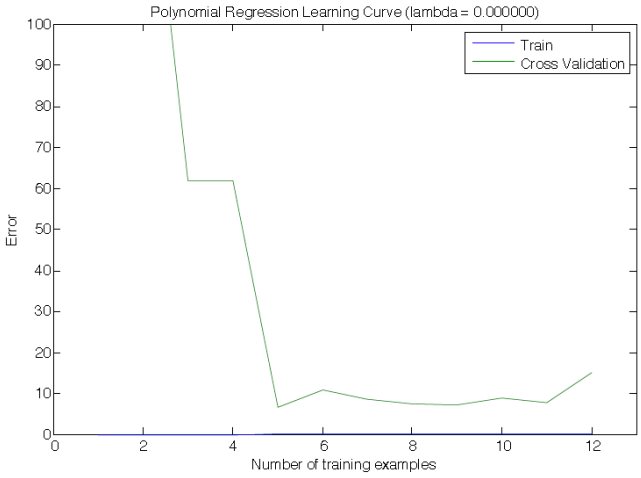

Figure 5: Polynomial learning curve, $\lambda = 0$

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

3.2 Adjusting the regularization parameter

In this section, you will get to observe how the regularization parameter affects the bias-variance of regularized polynomial regression. You should now modify the `lambda` parameter in the `ex5.m` and try $\lambda = 1, 100$. 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 that follows the data trend well (Figure 6) and a learning curve (Figure 7) showing that both the cross validation and training error 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 8) 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.

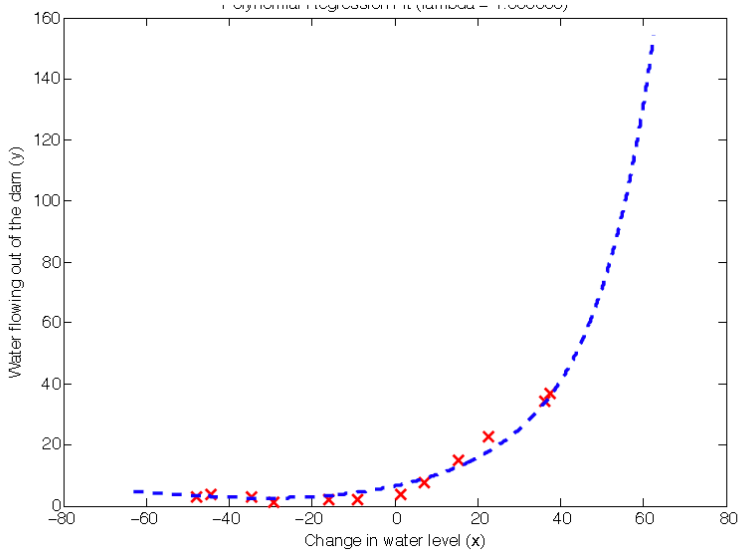

Figure 6: Polynomial fit, $\lambda = 1$

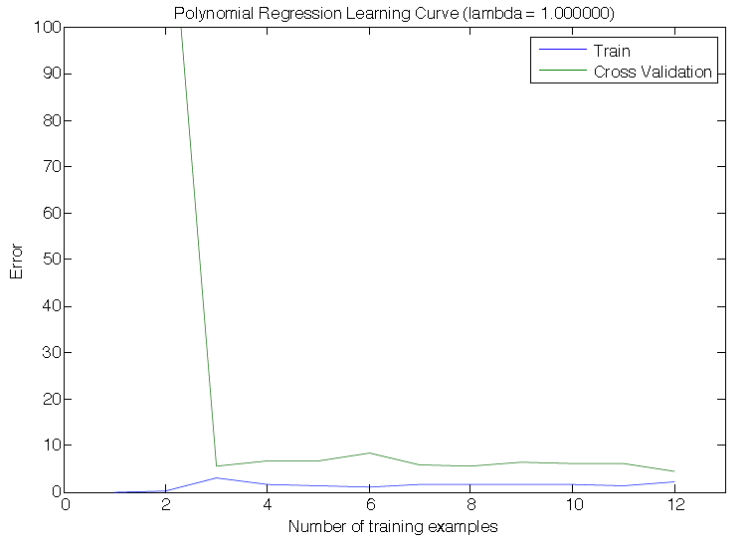

Figure 7: Polynomial learning curve, $\lambda = 1$

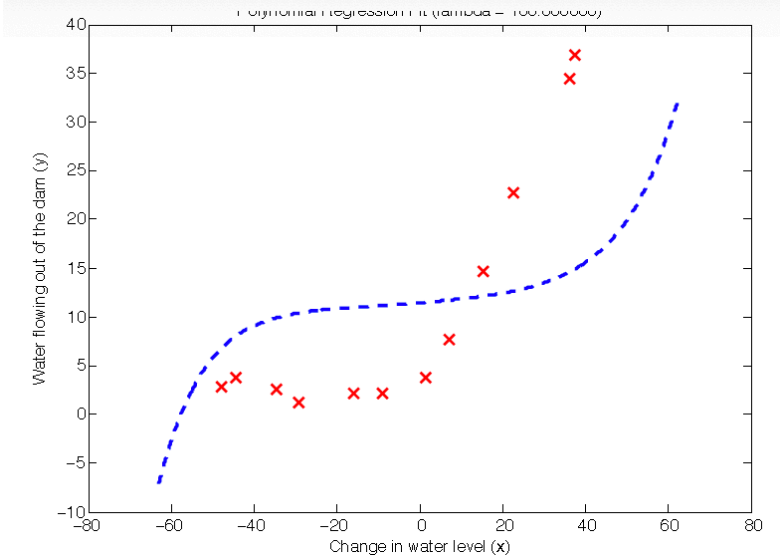

Figure 8: Polynomial fit, $\lambda = 100$

3.3 Selecting λ using a cross 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 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:

$\{0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10\}$.

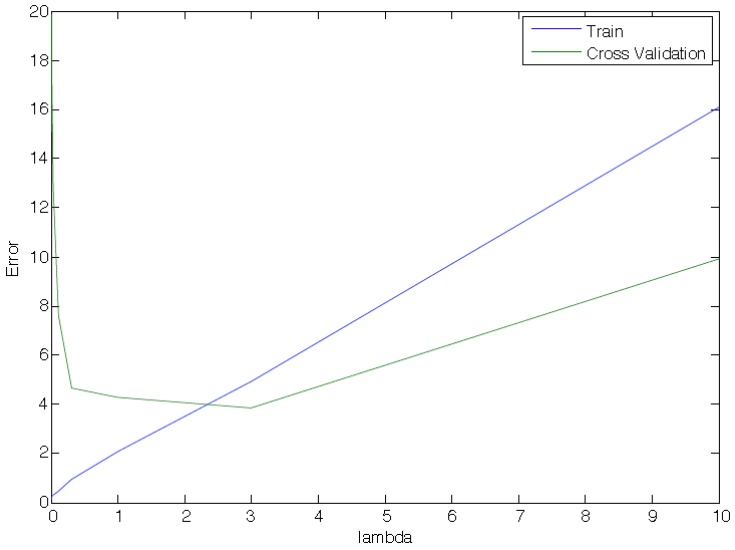

Figure 9: Selecting λ using a cross validation set

After you have completed the code, the next part of `ex5.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 Figure 9. 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.

3.4 Computing test set error

In the previous part of the exercise, you implemented code to compute the cross validation error for various values of the regularization parameter λ. However, 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 parameters, nor to learn the model parameters θ).
For this optional (ungraded) exercise, you should compute the test error using the best value of λ you found. In our cross validation, we obtained a test error of 3.8599 for $\lambda = 3$

3.5 Plotting learning curves with randomly selected examples

In practice, especially for small training sets, when you plot learning curves to debug your algorithms, it is often helpful to average across multiple sets of randomly selected examples to determine the training error and cross validation error.

Concretely, to determine the training error and cross validation error for i examples, you should first randomly select i examples from the training set and i examples from the cross validation set. You will then learn the parameters θ using the randomly chosen training set and evaluate the parameters θ on the randomly chosen training set and cross validation set. The above steps should then be repeated multiple times (say 50) and the averaged error should be used to determine the training error and cross validation error for i examples.

For this optional (ungraded) exercise, you should implement the above strategy for computing the learning curves. For reference, figure 10 shows the learning curve we obtained for polynomial regression with $\lambda = 0.01$. Your figure may differ slightly due to the random selection of examples.

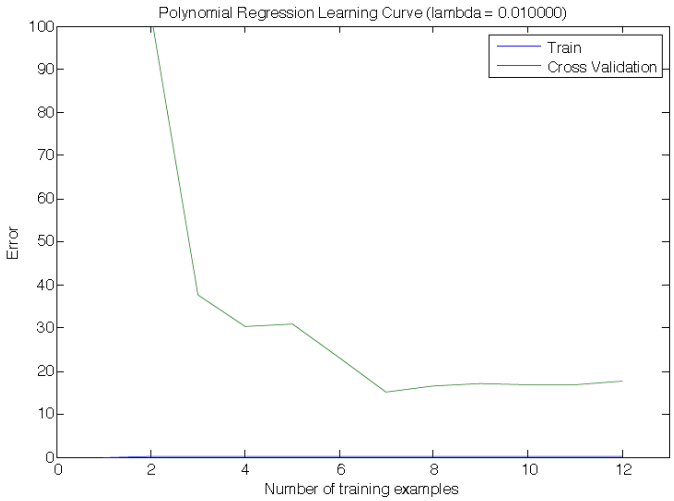

Figure 10: Optional (ungraded) exercise: Learning curve with randomly selected examples

Submission

Submit the following files on Canvas, under Assignments and "LAB 3".

From Lab3-1 folder:

[*] `lrCostFunction.m` - Logistic regression cost function
[*] `oneVsAll.m` - Train a one-vs-all multi-class classifier
[*] `predictOneVsAll.m` - Predict using a one-vs-all multi-class classifier

From Lab3-2 folder:

[*] `linearRegCostFunction.m` - Regularized linear regression cost function
[*] `learningCurve.m` - Generates a learning curve
[*] `polyFeatures.m` - Maps data into polynomial feature space
[*] `validationCurve.m` - Generates a cross validation curve