Problem Set 2: Classification[1]

see website for due date

Submission

What to submit (staple in this order):

1. grade2.txt
2. **All the files** under starter code folder, with your completed code
3. Screenshots of the output
4. Solutions to the written exercises.

Grading. We will randomly choose a subset of submissions and verify that 1) the code produces the answers and 2) the points computation in “grade.txt” is correct. Only one of the written problems will be chosen at random for grading. Total: 130 points.

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. This outline and the framework code in ex2.m will guide you through the exercise.

1.1 Implementation

1.1.1 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 ex2.m, the code will load the data and display it on a 2-dimensional plot by calling the function plotData and it should display a figure like Figure 1, where the axes are the two exam scores, and the positive and negative examples are shown

[1] Note: parts of this assignment are adapted from programming assignments 2 and 3 of Andrew Ng’s online Machine Learning course(https://class.coursera.org/ml).

with different markers.

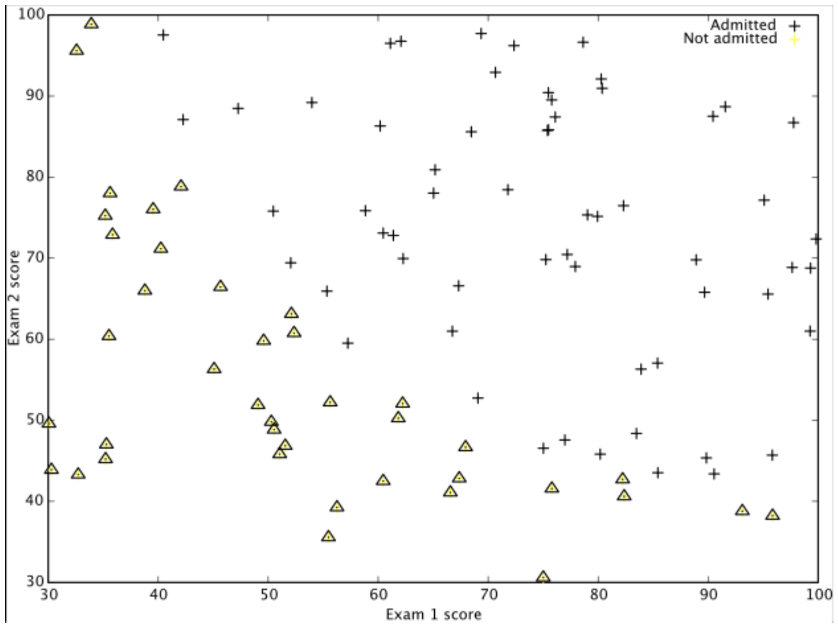

Figure 1: Scatter plot of training data

1.1.2 Sigmoid function

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

$$h_\theta(x) \;=\; g(\theta^T x)$$

where function g is the sigmoid function. The sigmoid function is defined as:

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

Your first step is to implement this function in sigmoid.m so it can be called by the rest of your program. When you are finished, try testing a few values by calling sigmoid(x) at the octave command line. 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. [5 points for this part]

1.1.3 Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Complete the code in costFunction.m to return the cost and gradient. Recall that the cost function in logistic regression is

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

and the gradient of the cost is a vector of the same length as θ where the j^{ith} element (for j = 0,1,...,n) is defined as follows:

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

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 $h_\theta(x)$. Once you are done, ex2.m will call your costFunction using the initial parameters of θ. You should see that the cost is about 0.693. [15 points for getting this answer]

1.1.4 Learning parameters using fminunc

In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradient descent. You wrote a cost function and calculated its gradient, then took a gradient descent step accordingly. This time, instead of taking gradient descent steps, you will use an Octave built-in function called $fminunc$.

For now, you do not need to worry about this function. In ex2.m, we already have code written to call $fminunc$ with the correct arguments. This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 2.

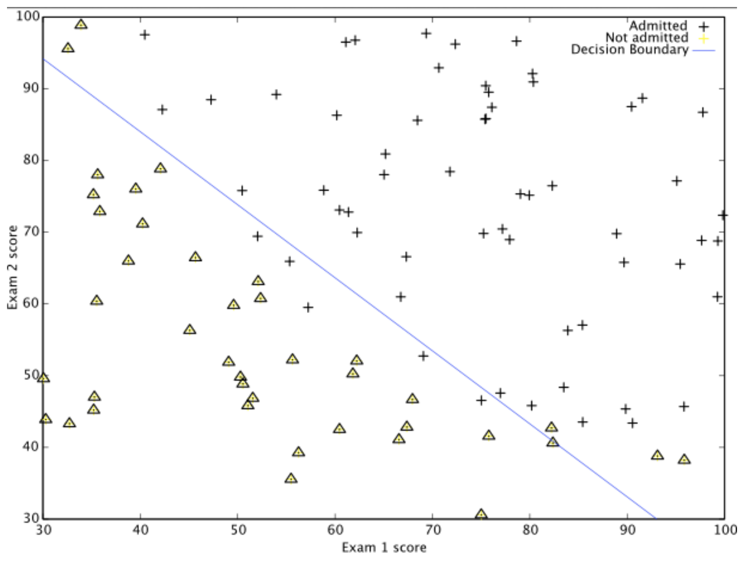

Figure 2: Training data with decision boundary

1.1.5 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.

Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set. In this part, your task is to complete the code in predict.m. The predict function will produce “1” or “0” predictions given a dataset and a learned parameter vector θ.

After you have completed the code in predict.m, the ex2.m script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. You should also see a Training Accuracy of 89.0. [20 points for getting this answer]

2. Regularized logistic regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant pass 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.

You will use another script, ex2_reg.m to complete this portion of the exercise.

2.1 Visualizing the data

Similar to the previous parts of this exercise, plotData is used to generate a figure like Figure 3, where the axes are the two test scores, and the positive (y = 1, accepted) and negative (y = 0, rejected) examples are shown with different markers.

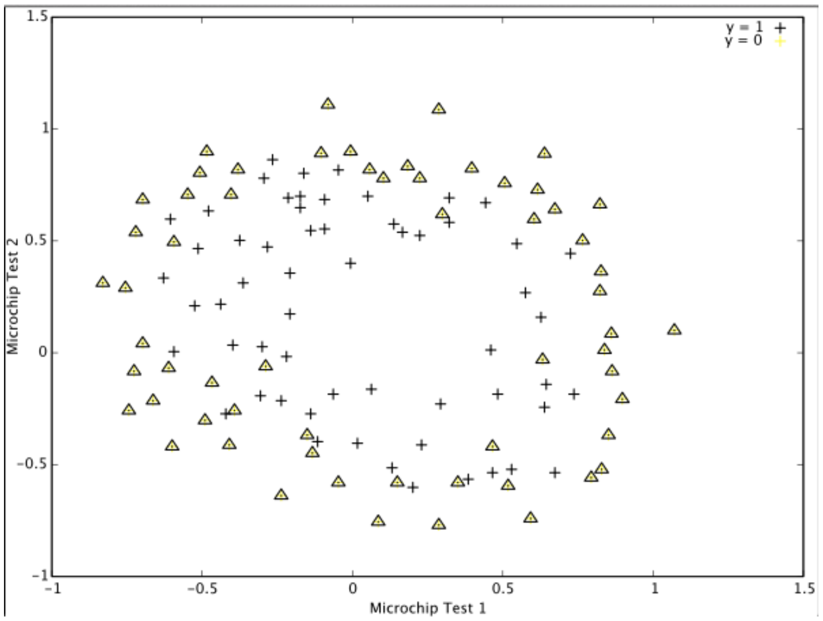

Figure 3: Plot of training data

Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight line. 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 Nonlinear feature mapping

One way to fit the data better is to create more features from each data point. In the provided function mapFeature.m, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power, as shown in the figure below. Familiarize yourself with this code.

$$\text{mapFeature}(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

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 classifier 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 overfitting. 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. Complete the code in costFunctionReg.m to return the cost and gradient.

Recall that the regularized cost function in logistic regression is

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

Note that you should not regularize the parameter θ_0. In Octave, recall that indexing starts from 1, hence, you should not be regularizing the theta(1) parameter (which corresponds to θ_0) in the code. The gradient of the cost function is a vector where the j^{th} element is defined as follows:

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

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

Once you are done, ex2_reg.m will call your costFunctionReg function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

2.3.1 Learning parameters using fminunc

Similar to the previous parts, you will use $fminunc$ to learn the optimal parameters θ. If you have completed the cost and gradient for regularized logistic regression (costFunctionReg.m) correctly, you should be able to step through the next part of ex2_reg.m to learn the parameters θ using $fminunc$. You should see a Training Accuracy of 83.05. [20 points for getting this answer]

2.4 Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided the function plotDecisionBoundary.m which plots the (non-linear) decision boundary that separates the positive and negative examples. In plotDecisionBoundary.m, we plot the non-linear decision boundary by computing the classifier's predictions on an evenly spaced grid and then and draw a contour plot of where the predictions change from y = 0 to y = 1.

After learning the parameters θ, the next step in ex2_reg.m will plot a decision boundary similar to Figure 4.

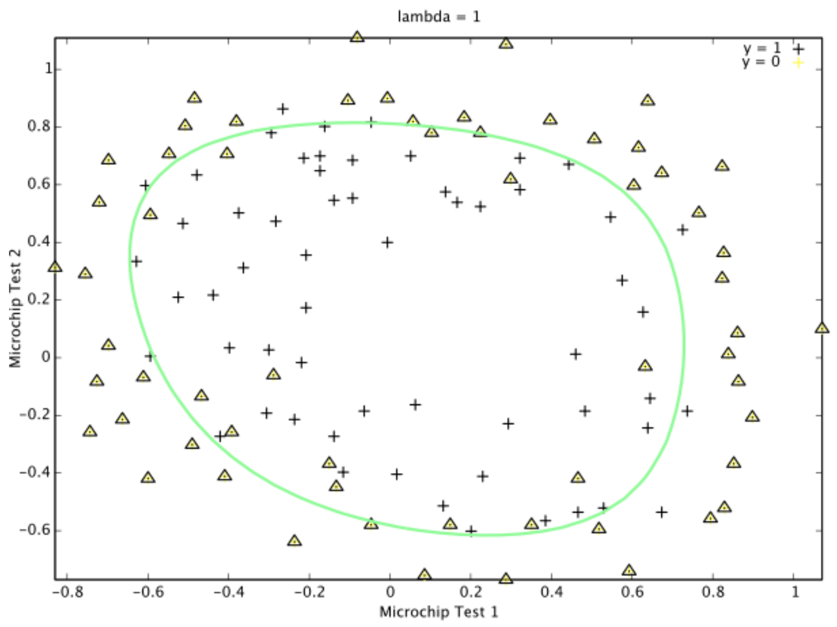

Figure 4: Training data with decision boundary

3. Multi-class Classification

In this part of the exercise, you will use logistic regression 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 and you will extend your previous implementation of logistic regression and apply it to one-vs-all classification. Throughout this part of the exercise, you will be using the scripts ex2_one_vs_all.m.

3.1 Dataset

You are given a data set in ex2data3.mat that contains 5000 training examples of handwritten digits.[2] The .mat format means 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('ex2data3.mat');
```

There are 5000 training examples in ex2data3.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.

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

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.

3.2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of ex2_one_vs_all.m the code randomly selects 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 5.

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

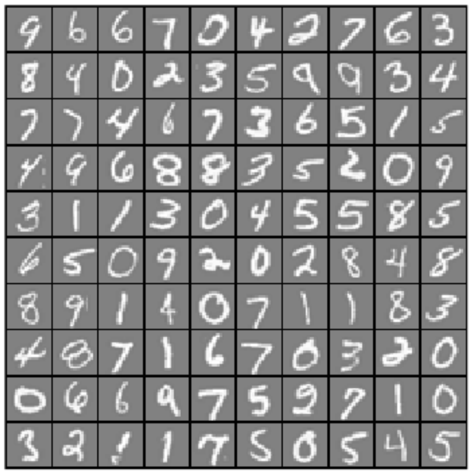

Figure 5: Examples from the dataset

3.3 Vectoring 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.

3.3.1 Vectoring the cost function

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

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

To compute each element in the summation, we have to compute $h_\theta(x^{(i)})$ for every example i, where $h_\theta(x^{(i)}) = g(\theta^T x^{(i)})$ and $g(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. 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.

3.3.2 Vectorizing the gradient

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

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

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

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

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:

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

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. 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.

3.3.3 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

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

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

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

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

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

Octave Tip: When implementing the vectorization for regularized logistic regression, you might often want to only sum and update certain elements of θ. In Octave, 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 2^{nd} 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., $sum(z(2:end).\hat{}2)$). In the starter code, lrCostFunction.m, we have also provided hints on yet another possible method computing the regularized gradient.

3.4 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 . 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 $\Theta \in R^{K\ x\ (N+1)}$, 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 $y_j \in 0,1$ 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 Tip: Logical arrays in Octave are arrays which contain binary (0 or 1) elements. In Octave, 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:

```
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 more efficient for dealing with
a large number of parameters.

After you have correctly completed the code for oneVsAll.m, the script ex2_one_vs_all.m will continue to use your oneVsAll function to train a multi-class classifier.

3.4.1 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, ex2_one_vs_all.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). [20 points for getting this answer]

4. Linear Discriminant Analysis (LDA)

In this part of the exercise, you will re-visit the problem of predicting whether a student gets admitted into a university. However, in this part, you will build a linear discriminant analysis (LDA) classifier for this problem. You will be using the script ex2_lda.m for this exercise.

4.1 Linear Discriminant Analysis

LDA is a generative model for classification that assumes the class covariances are equal. Given a training dataset of positive and negative features (x, y) with $y \in \{0, 1\}$, LDA models the data x as generated from class-conditional Gaussians:

$$P(x,y) \ = \ P(x|y)P(y) \ \ where \ \ P(y=1) = \pi \ \ and \ \ P(x|y) = N(x; \mu^y, \Sigma)$$

where means μ^y are class-dependent but the covariance matrix Σ is class-independent (the same for all classes). A novel feature x is classified as a positive if $P(y=1|x) > P(y=0|x)$, which is equivalent to *a(x)>0,* where the linear classifier *a(x) =* $w^T x + w_0$ has weights given by $w = \Sigma^{-1}(\mu^1 - \mu^0)$. (See lecture notes and problem 5.1 below for the derivation.) In practice, and in this assignment, we use a(x)>some threshold, or equivalently, $w^T x > T$ for some constant T.

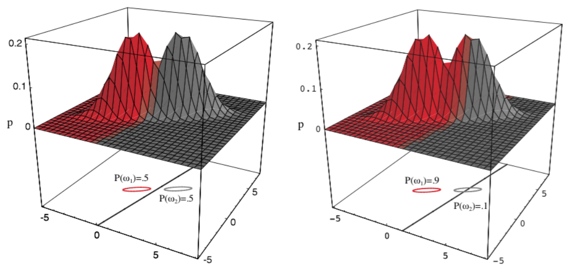

Figure 6: Illustration of LDA with same (left) or different (right) prior probabilities.

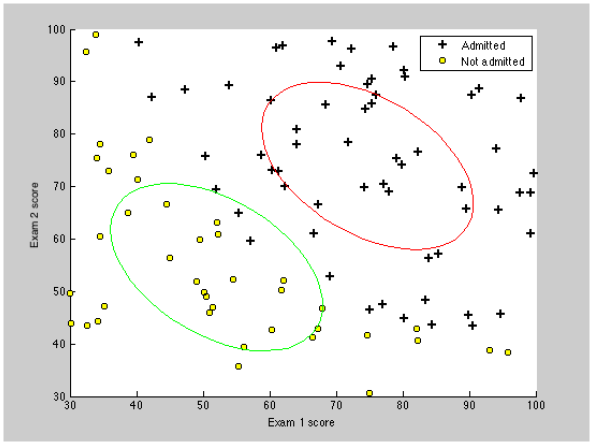

Figure 7: Visualization of the Gaussian ellipsoids of the two classes.

As we saw in lecture, LDA and logistic regression can be expressed in the same form

$$p(y = 1|x) = \frac{1}{1 + e^{-\theta^T x}}$$

however, they generally produce different solutions for the parameter theta.

4.2 Implementation

In this assignment, you can assume the prior probabilities for the two classes are the same (although the number of the positive and negative samples in the training data is not the same), and that the threshold T is zero. As a bonus, you are encouraged to explore how the different prior probabilities shift the decision boundary.

Implement the LDA classifier by completing the code in lda.m. As an implementation detail, you should first center the positive and negative data separately, so that each has a mean equal to 0, before computing the covariance, as this tends to give a more accurate estimate. You should center the whole training data set before applying the classifier. Namely, subtract the middle value of the two classes' means ($\frac{1}{2}(pos\ mean + neg\ mean)$), which is on the separating plane when their prior probabilities are the same and becomes the 'center' of the data.

The provided visualization code in lda.m will create a plot showing the estimated class-conditional distributions, similar to Figure 7. Completing the code in lda.m by implementing the LDA classifier and using it to compute the training set accuracy. You should get a training accuracy around 89%. [20 points for getting this answer and plot]