Programming Exercise 4: Regularization in logistic regression

In this 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.

The file `ex2data2.txt` contains a training set of two test results from some microchips. The first column is the score of test 1 and the second column is the score for test 2. Third column is the decision about the microchip test results (1 means passed, 0 implies failure).

For working on this exercise, you will be given a starter code. You may need to use `cd` command to change to the right directory.

Following files are either complete or partially complete. If it is partially complete, you will have to complete it.

`ex2_reg.m`	`%%complete`
`ex2data2.txt`	`Dataset`
`plotData.m`	`Given`
`Sigmoid.m`	`% given`
`predict.m`	`%given`
`costFunctionReg`	`%incomplete`

You will be running `ex2_reg.m`. The code in `ex2_reg.m` will set up the dataset for the problem and make calls to functions that you will have to write. This file is complete. However, you will have to modify functions in other files according to the instructions that will be provided later in this document.

1. Logistic Regression

In this part of the exercise, you will build a regularized logistic regression model to predict whether a microchip should be accepted or rejected.

You have historical data from previous testing of microchips that you can use as a training set for logistic regression. For each training example, you have the microchip's test scores on two tests and the accept/reject decision.

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_reg.m`, the code will load the data and display it on a 2-dimensional plot by calling the function `plotData`.

Complete the code in `plotData` so that it displays a figure like **Figure 1**, where the axes are the two exam scores, and the positive and negative examples are shown with different markers. (This code is provided, actually)

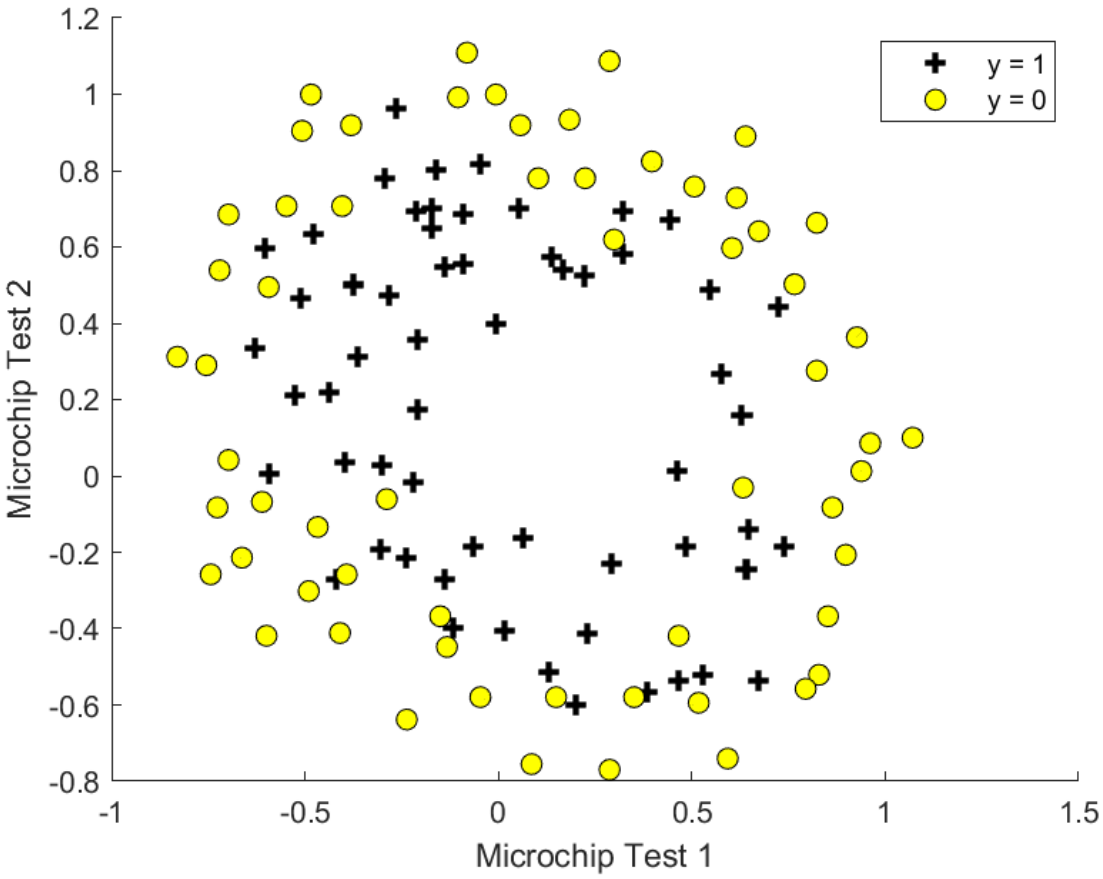

Figure 1

```
pos = find(y==1); neg = find(y == 0);
plot(X(pos, 1), X(pos, 2), 'k+','LineWidth', 2,'MarkerSize', 7);
plot(X(neg, 1), X(neg, 2), 'ko', 'MarkerFaceColor', 'y', 'MarkerSize', 7);
```

Hint: See code above

Figure 1 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight-forward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

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.

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

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. Implementation

2.1 Sigmoid 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:

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

You now have 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/MATLAB 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.

You should be able to use the same sigmoid function you wrote for previous lab problem without modification.

2.2 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

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

Recall, for linear regression, it was

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta\left(x^{(i)}\right) - y^{(i)})^2$$

For logistic regression without regularization, it was

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

Note that you should not regularize the parameter θ_0. In Octave/MATLAB, 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:

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

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

It is important to remember that even though this gradient looks identical to the linear regression gradient, the formula is different as linear and logistic regression has different definition of $h_\theta(x)$.

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.

Gradient Descent Theory

In this part, you will fit the logistic regression parameters θ to our dataset using gradient descent. Your goal is to find θ through gradient descent.

Update equations:

The objective of logistic regression is to minimize the cost function

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

where the hypothesis $h_\theta(x)$ is given by the model

$h_\theta(x) = g(\theta^T x) = g(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6 + \ldots.)$

Remember that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update as follows:

```
repeat until convergence: {
```

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

```
}
```

However, for n features, we have repeat 'n' times.

```
repeat until convergence: {
```

$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta\big(x^{(i)}\big) - y^{(i)}\big).x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta\big(x^{(i)}\big) - y^{(i)}\big).x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta\big(x^{(i)}\big) - y^{(i)}\big)x_2^{(i)}$$

. . .

}

(You must update θ_j simultaneously for all j)

With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost: $J(\theta)$.

2.3 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/MATLAB built-in function called `fminunc`. You already have used it in the last exercise involving pure logistic regression without regularization.
Octave/MATLAB's `fminunc` 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.

So, you are going to use `fminunc` to find the best parameters θ for the logistic regression cost function, given a fixed dataset (of X and y values). You will pass to `fminunc` the following inputs:

- The initial values of the parameters we are trying to optimize.
- A function that, when given the training set and a particular θ, computes the logistic regression cost and gradient with respect to θ for the dataset (X, y)

In `ex2_reg.m`, we already have code written to call `fminunc` with the correct arguments.

```
Set options for fminunc
options = optimset('GradObj', 'on', 'MaxIter', 400);
% Run fminunc to obtain the optimal theta
% This function will return theta and the cost
[theta, cost] =  fminunc(@(t)(costFunction(t, X, y)), initial theta,
options);
```

In this code snippet, we first defined the options to be used with `fminunc`. Specifically, we set the `GradObj` option to `on`, which tells `fminunc` that our function returns both the cost and the gradient. This allows `fminunc` to use the gradient when minimizing the function. Furthermore, we set the `MaxIter` option to 400, so that `fminunc` will run for at most 400 steps before it terminates.

To specify the actual function we are minimizing, we use a "short-hand" for specifying functions with the `@(t) (costFunction(t, X, y) )` . This creates a function, with argument `t`, which calls your `costFunction`. This allows us to wrap the `costFunction` for use with `fminunc`.

If you have completed the `costFunction` correctly, `fminunc` will converge on the right optimization parameters and return the final values of the cost and θ. Notice that by using `fminunc`, you did not have to write any loops yourself, or set a learning rate like you did for gradient descent. This is all done by `fminunc`: you only needed to provide a function calculating the cost and the gradient.

Once `fminunc` completes, `ex2_reg.m` will call your `costFunction` function using the optimal parameters of θ. You should see that the cost is about 3.16.

This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure like **Figure 2, 3, 4**. We have provided the code for drawing the boundary line between the two classes in `plotDecisionBoundary` .m. You should look at the code in `plotDecisionBoundary.m` to see how to plot such a boundary using the θ values.

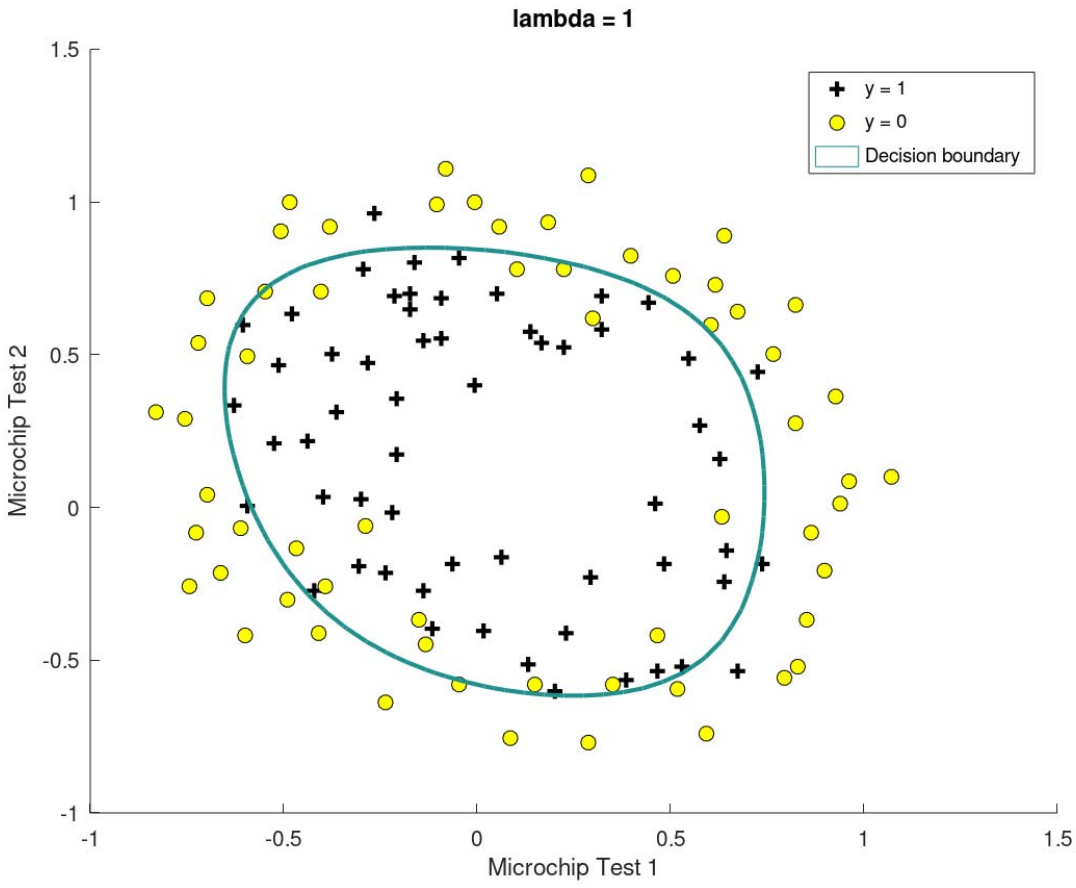

Figure 2

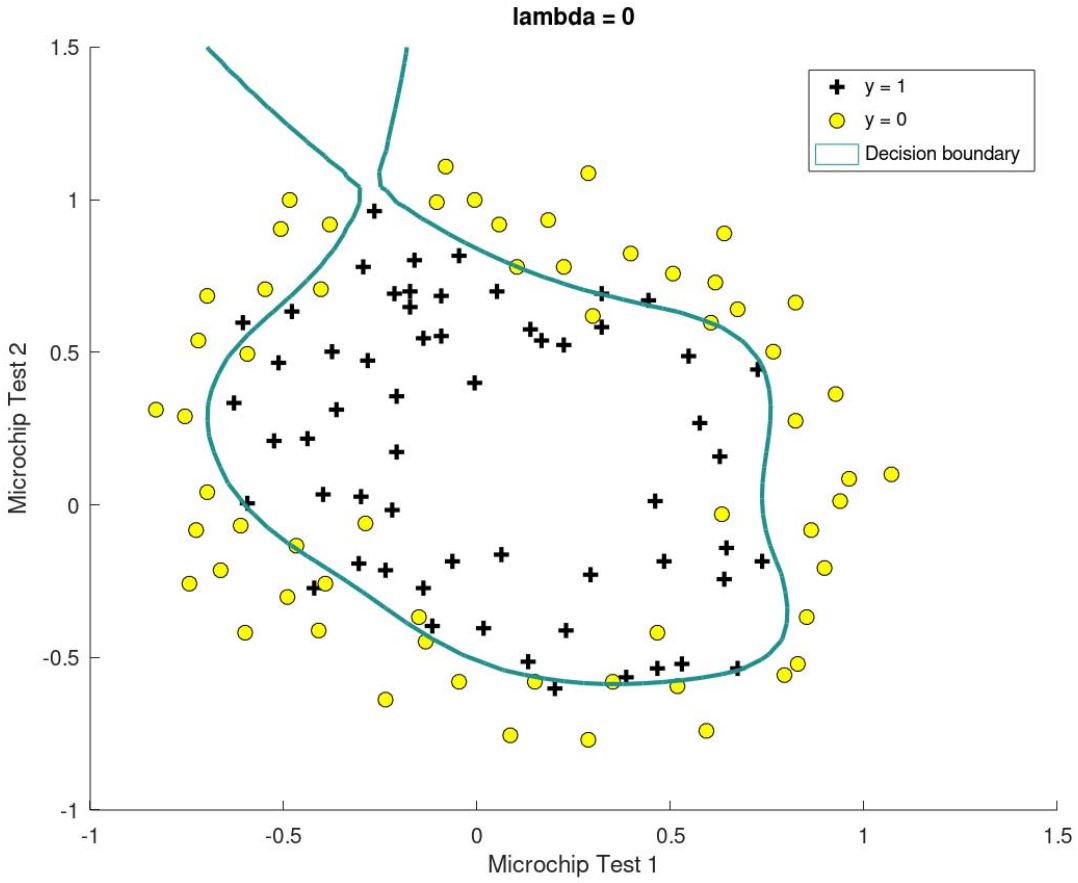

Figure 3

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 drew a contour plot of where the predictions change from $y = 0$ to $y = 1$. After learning the parameters θ, the next step in `ex_reg.m` will plot a decision boundary similar to Fig 2.

3. Evaluating logistic Regression

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.

You should be able to use the same code for `predict.m` as in the `ex2.m` script. This code will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

Playing with Lambda

In this part of the exercise, you will get to try out different regularization parameters for the dataset to understand how regularization prevents overfitting. Notice the changes in the decision boundary as you vary λ. With a small λ, you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data (Figure 3). This is not a good decision boundary: for example, it predicts that a point at x = (-0.25, 1.5) is accepted (y = 1), which seems to be an incorrect decision given the training set.

With a larger λ, you should see a plot that shows a simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to too high a value, you will not get a good fit and the decision boundary will not follow the data so well, thus underfitting the data. (Figure 4).

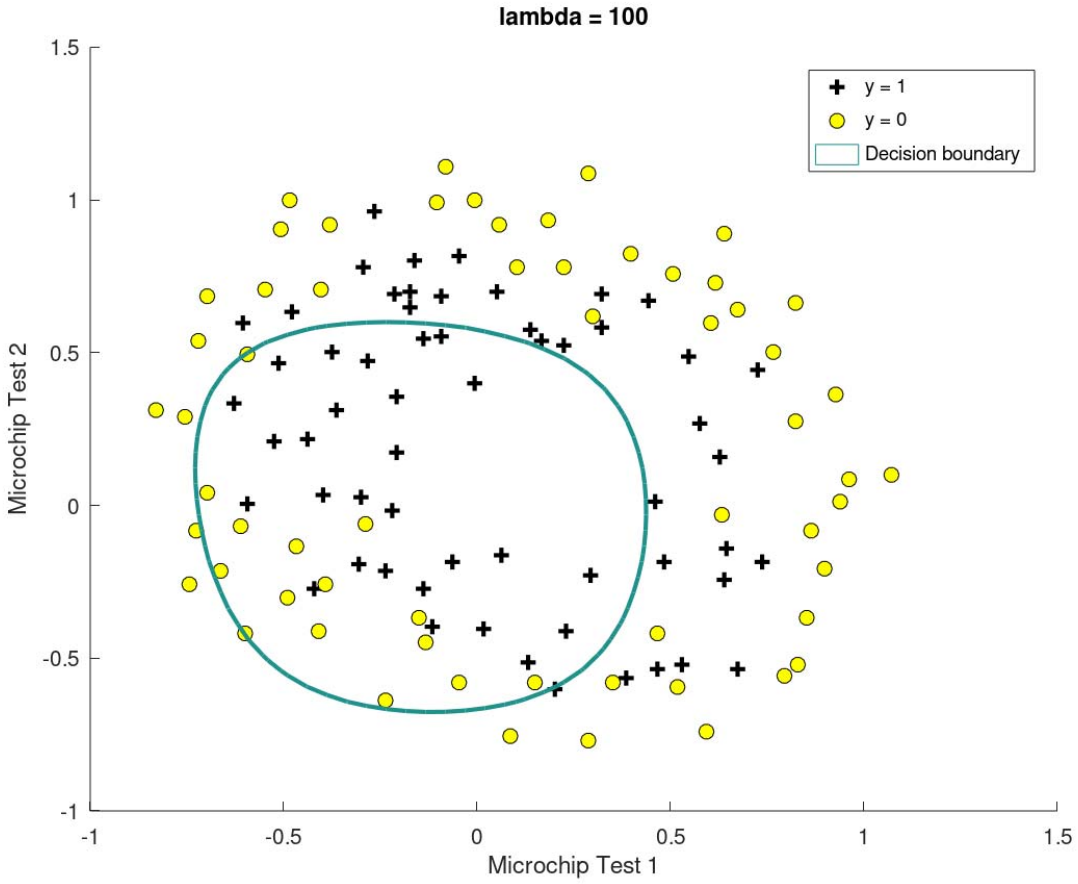

Figure 4

Distribution of points

Finish in the class, for 100% points.
Submit by midnight today, for 92% of points.
Submit by midnight tomorrow, for 88% of points.
Submit by midnight day after tomorrow, for 84% of points.

Problem	**Points**
Sigmoid	
plotData	
a) Compute cost gradient for logistic Regression	30
b) Predict function	10
c) Comparison with different lambda and explanation	15

Part c) elaboration.
In this part, you will get to try different values of lambda and see how regularization affects the decision boundary.

Try the following values of lambda (0, 1, 10, 100).

How does the decision boundary change when you vary lambda? How does the training set accuracy vary?