Machine Learning: Programming Exercise 2

Logistic Regression

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

Files needed for this exercise

- `ex2.mlx` - MATLAB Live Script that steps you through the exercise
- ex2data1.txt - Training set for the first half of the exercise
- ex2data2.txt - Training set for the second half of the exercise
- `submit.m` - Submission script that sends your solutions to our servers
- `mapFeature.m` - Function to generate polynomial features
- `plotDecisionBoundary.m` - Function to plot classifier's decision boundary
- `*plotData.m` - Function to plot 2D classification data
- `*sigmoid.m` - Sigmoid function
- `*costFunction.m` - Logistic regression cost function
- `*predict.m` - Logistic regression prediction function
- `*costFunctionReg.m` - Regularized logistic regression cost function

indicates files you will need to complete

Clear existing variables and confirm that your Current Folder is set correctly

Click into this section, then click the 'Run Section' button above. This will execute the `clear` command to clear existing variables and the `dir` command to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex2' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
clear
dir
```

```
.                   costFunctionReg.m   ex2data1.txt        lib                 plotDecisionBoundar
..                  ex2.mlx             ex2data2.txt        mapFeature.m        predict.m
costFunction.m      ex2_companion.mlx   lectureNotes        plotData.m          sigmoid.m
```

Before you begin

The workflow for completing and submitting the programming exercises in MATLAB Online differs from the original course instructions. Before beginning this exercise, make sure you have read through the instructions in `README.mlx` which is included with the programming exercise files. `README` also contains solutions to the many common issues you may encounter while completing and submitting the exercises in MATLAB Online. Make sure you are following instructions in `README` and have checked for an existing solution before seeking help on the discussion forums.

Table of Contents

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. `ex2.mlx` will guide you through the exercise. To begin, run the code below to load the data into MATLAB.

```
% Load Data
% The first two columns contain the exam scores and the third column contains the label.
data = load('ex2data1.txt');
X = data(:, [1, 2]); y = data(:, 3);
```

1.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. The code below will load the data and display it on a 2-dimensional plot by calling the function `plotData`. You will now 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.

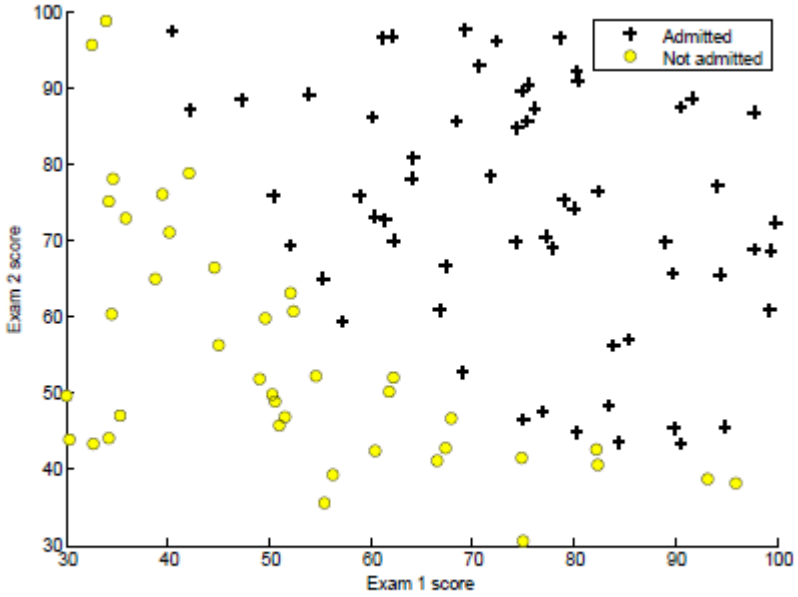


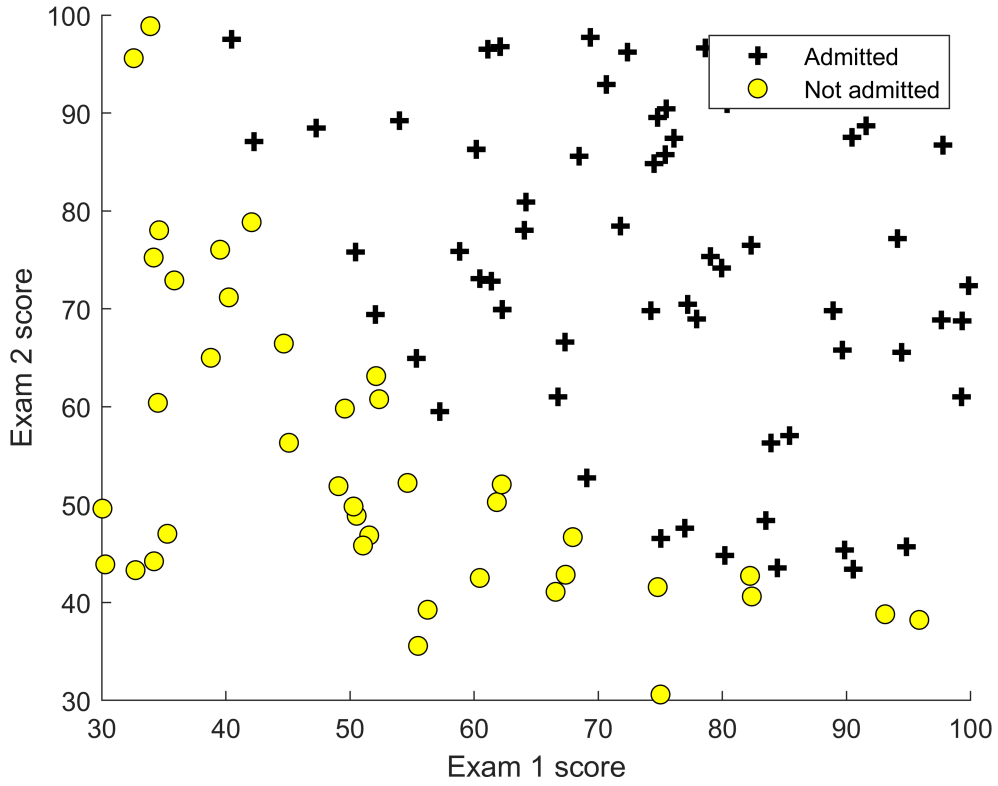
Figure 1: Scatter plot of training data

To help you get more familiar with plotting, we have left `plotData.m` empty so you can try to implement it yourself. However, this is an optional (ungraded) exercise. We also provide our implementation below so you can copy it or refer to it. If you choose to copy our example, make sure you learn what each of its commands is doing by consulting the MATLAB documentation.

```
% Find Indices of Positive and Negative Examples
pos = find(y==1); neg = find(y == 0); % get the index using find
% Plot Examples
plot(X(pos, 1), X(pos, 2), 'k+','LineWidth', 2, 'MarkerSize', 7);
plot(X(neg, 1), X(neg, 2), 'ko', 'MarkerFaceColor', 'y','MarkerSize', 7);  % MarkerFaceColor - y , k - b
```

Once you have added your own or the above code to `plotData.m`, run the code in this section to call the `plotData` function.

```
% Plot the data with + indicating (y = 1) examples and o indicating (y = 0) examples.
plotData(X, y);

% Labels and Legend
xlabel('Exam 1 score')
ylabel('Exam 2 score')

% Specified in plot order
legend('Admitted', 'Not admitted')
```



1.2 Implementation

1.2.1 Warmup exercise: 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)` in the code section below. 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.

```
% Provide input values to the sigmoid function below and run to check your implementation
sigmoid(0)
```

```
g = 0.5000
ans = 0.5000
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

1.2.2 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

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

and the gradient of the cost is a vector of the same length as θ where the jth element (for $j = 0, 1, \ldots, n$) is defined as follows:

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

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, run the code sections below to set up your data and call your `costFunction` using the initial parameters of θ. You should see that the cost is about 0.693 and gradients of about -0.1000, -12.0092, and -11.2628

```
%  Setup the data matrix appropriately
[m, n] = size(X);

% Add intercept term to X
X = [ones(m, 1) X];   % [1 23 43]

% Initialize the fitting parameters
initial_theta = zeros(n + 1, 1);

% Compute and display the initial cost and gradient
[cost, grad] = costFunction(initial_theta, X, y);
```

```
g = 100×1
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
     ⋮
```

```
fprintf('Cost at initial theta (zeros): %f\n', cost);
```

```
Cost at initial theta (zeros): 0.693147
```

```
disp('Gradient at initial theta (zeros):'); disp(grad);
```

```
Gradient at initial theta (zeros):
   -0.1000  -12.0092  -11.2628
```

1.2.3 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 a MATLAB built-in function called `fminunc`.

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. Concretely, 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)

**Constraints in optimization often refer to constraints on the parameters, for example, constraints that bound the possible values θ can take (e.g. $\theta < 1$). Logistic regression does not have such constraints since θ is allowed to take any real value.*

We already have code written below to call `fminunc` with the correct arguments:

- We first define the options to be used with `fminunc`. Specically, 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: `@(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, the remaining code will call your `costFunction` function using the optimal parameters of θ. You should see that the cost is about 0.203. 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. We also encourage you to look at the code in `plotDecisionBoundary.m` to see how to plot such a boundary using the θ values.

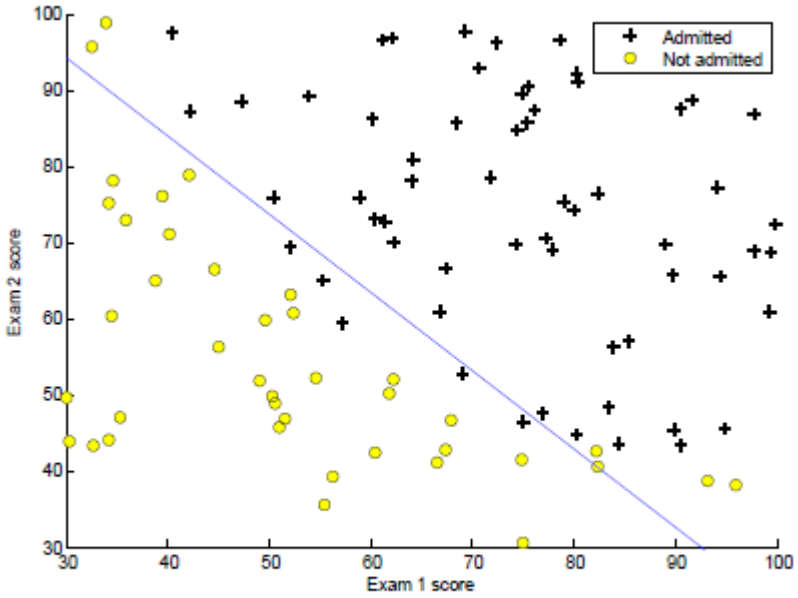


Figure 2: Training data with decision boundary

Run the code below and verify the results.

```
%  Set options for fminunc
options = optimoptions(@fminunc,'Algorithm','Quasi-Newton','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);
```

```
g = 100×1
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    ⋮
g = 100×1
     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
```

```
      ⋮
      ⋮
g = 100×1
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
      ⋮
      ⋮
g = 100×1
    0.9667
    0.9032
    0.9629
    0.9880
    0.9908
    0.9552
    0.9913
    0.9761
    0.9929
    0.9804
      ⋮
      ⋮
g = 100×1
    0.6546
    0.6044
    0.6498
    0.6978
    0.7084
    0.6412
    0.7108
    0.6690
    0.7186
    0.6776
      ⋮
      ⋮
g = 100×1
    0.6451
    0.5984
    0.6407
    0.6874
    0.6988
    0.6336
    0.6995
    0.6619
    0.7080
    0.6706
      ⋮
      ⋮
g = 100×1
    0.6383
    0.5949
    0.6345
    0.6811
    0.6944
    0.6295
    0.6925
    0.6600
    0.7025
```

```
    0.6694
      ⋮
      ⋮
g = 100×1
    0.6225
    0.5879
    0.6204
    0.6688
    0.6889
    0.6221
    0.6778
    0.6616
    0.6933
    0.6736
      ⋮
      ⋮
g = 100×1
    0.6040
    0.5814
    0.6044
    0.6575
    0.6885
    0.6164
    0.6630
    0.6707
    0.6875
    0.6870
      ⋮
      ⋮
g = 100×1
    0.5934
    0.5796
    0.5959
    0.6544
    0.6951
    0.6165
    0.6574
    0.6840
    0.6898
    0.7039
      ⋮
      ⋮
g = 100×1
    0.5944
    0.5815
    0.5972
    0.6580
    0.7009
    0.6197
    0.6607
    0.6906
    0.6949
    0.7114
      ⋮
      ⋮
g = 100×1
    0.5964
    0.5826
    0.5991
    0.6600
    0.7024
    0.6210
    0.6630
    0.6915
```

```
    0.6968
    0.7121
      ⋮
      ⋮
g = 100×1
    0.5973
    0.5830
    0.5999
    0.6608
    0.7028
    0.6215
    0.6639
    0.6915
    0.6974
    0.7120
      ⋮
      ⋮
g = 100×1
    0.5995
    0.5840
    0.6019
    0.6626
    0.7038
    0.6226
    0.6660
    0.6915
    0.6988
    0.7117
      ⋮
      ⋮
g = 100×1
    0.6026
    0.5853
    0.6046
    0.6650
    0.7050
    0.6240
    0.6689
    0.6915
    0.7007
    0.7113
      ⋮
      ⋮
g = 100×1
    0.6077
    0.5874
    0.6092
    0.6693
    0.7072
    0.6265
    0.6740
    0.6915
    0.7040
    0.7105
      ⋮
      ⋮
g = 100×1
    0.6158
    0.5906
    0.6163
    0.6760
    0.7106
    0.6302
    0.6821
```

```
    0.6913
    0.7093
    0.7093
      ⋮
      ⋮
g = 100×1
    0.6286
    0.5950
    0.6276
    0.6869
    0.7164
    0.6359
    0.6952
    0.6909
    0.7182
    0.7072
      ⋮
      ⋮
g = 100×1
    0.6483
    0.6006
    0.6449
    0.7044
    0.7260
    0.6443
    0.7163
    0.6899
    0.7327
    0.7036
      ⋮
      ⋮
g = 100×1
    0.6778
    0.6053
    0.6704
    0.7321
    0.7420
    0.6553
    0.7497
    0.6873
    0.7567
    0.6971
      ⋮
      ⋮
g = 100×1
    0.7186
    0.6019
    0.7051
    0.7742
    0.7689
    0.6670
    0.8000
    0.6808
    0.7953
    0.6852
      ⋮
      ⋮
g = 100×1
    0.7679
    0.5695
    0.7452
    0.8326
    0.8119
    0.6711
```

```
    0.8671
    0.6652
    0.8523
    0.6636
      ⋮
      ⋮
g = 100×1
    0.8121
    0.4566
    0.7760
    0.8989
    0.8729
    0.6449
    0.9352
    0.6319
    0.9203
    0.6283
      ⋮
      ⋮
g = 100×1
    0.8231
    0.2225
    0.7643
    0.9509
    0.9363
    0.5420
    0.9778
    0.5805
    0.9717
    0.5933
      ⋮
      ⋮
g = 100×1
    0.7580
    0.0480
    0.6552
    0.9761
    0.9753
    0.3454
    0.9928
    0.5432
    0.9923
    0.6068
      ⋮
      ⋮
g = 100×1
    0.5599
    0.0054
    0.4082
    0.9863
    0.9916
    0.1456
    0.9972
    0.5301
    0.9980
    0.6729
      ⋮
      ⋮
g = 100×1
    0.2971
    0.0005
    0.1731
    0.9906
    0.9968
```

```
    0.0474
    0.9986
    0.5143
    0.9994
    0.7362
      ⋮
      ⋮
g = 100×1
    0.1584
    0.0001
    0.0806
    0.9922
    0.9982
    0.0200
    0.9991
    0.4881
    0.9997
    0.7631
      ⋮
      ⋮
g = 100×1
    0.1046
    0.0000
    0.0499
    0.9928
    0.9987
    0.0118
    0.9993
    0.4604
    0.9998
    0.7671
      ⋮
      ⋮
g = 100×1
    0.0889
    0.0000
    0.0415
    0.9928
    0.9987
    0.0096
    0.9993
    0.4410
    0.9998
    0.7605
      ⋮
      ⋮
g = 100×1
    0.0855
    0.0000
    0.0397
    0.9926
    0.9987
    0.0091
    0.9993
    0.4291
    0.9998
    0.7519
      ⋮
      ⋮
g = 100×1
    0.0860
    0.0000
    0.0401
    0.9921
```

```
    0.9986
    0.0092
    0.9992
    0.4183
    0.9998
    0.7402
     ⋮
     ⋮
g = 100×1
    0.0889
    0.0000
    0.0419
    0.9915
    0.9984
    0.0097
    0.9991
    0.4141
    0.9998
    0.7322
     ⋮
     ⋮
g = 100×1
    0.0918
    0.0000
    0.0438
    0.9909
    0.9983
    0.0104
    0.9991
    0.4158
    0.9997
    0.7298
     ⋮
     ⋮
g = 100×1
    0.0926
    0.0000
    0.0446
    0.9905
    0.9982
    0.0108
    0.9990
    0.4202
    0.9997
    0.7319
     ⋮
     ⋮
g = 100×1
    0.0918
    0.0000
    0.0443
    0.9904
    0.9982
    0.0109
    0.9990
    0.4230
    0.9997
    0.7346
     ⋮
     ⋮
g = 100×1
    0.0912
    0.0000
    0.0440
```

```
    0.9904
    0.9982
    0.0108
    0.9990
    0.4234
    0.9997
    0.7354
     ⋮

g = 100×1
    0.0910
    0.0000
    0.0439
    0.9904
    0.9982
    0.0108
    0.9990
    0.4233
    0.9997
    0.7354
     ⋮

g = 100×1
    0.0910
    0.0000
    0.0439
    0.9904
    0.9982
    0.0108
    0.9990
    0.4232
    0.9997
    0.7354
     ⋮

g = 100×1
    0.0910
    0.0000
    0.0439
    0.9904
    0.9982
    0.0108
    0.9990
    0.4232
    0.9997
    0.7354
     ⋮

Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the optimality tolerance.

<stopping criteria details>
```

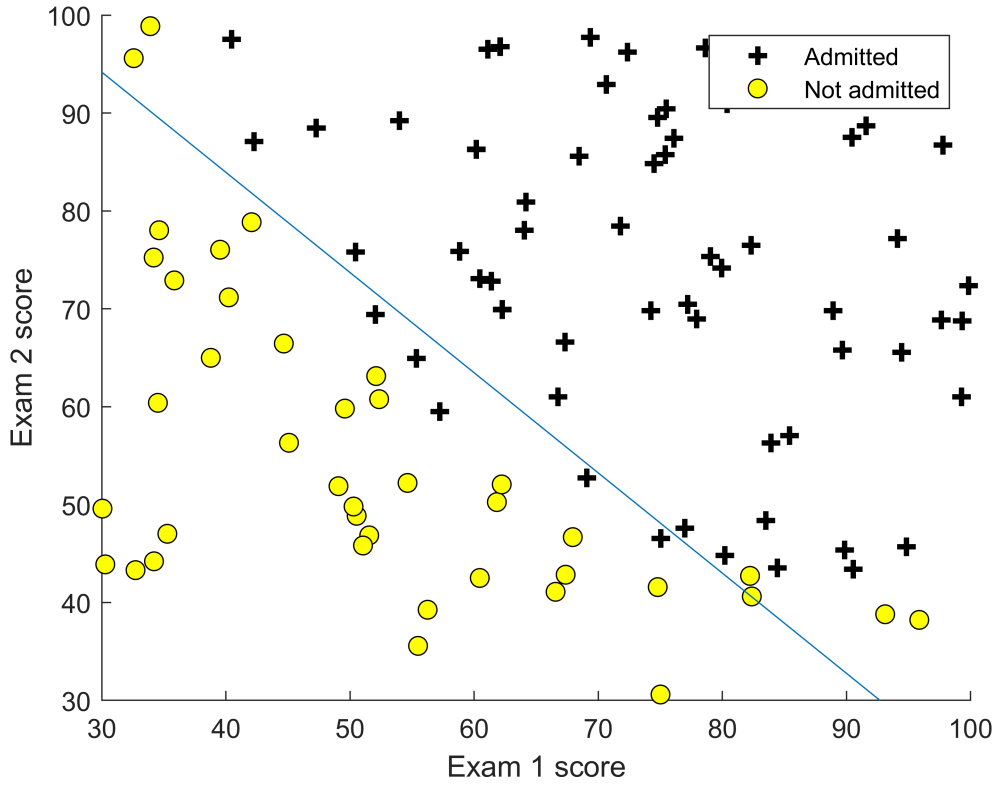
```
% Print theta
fprintf('Cost at theta found by fminunc: %f\n', cost);
```

```
Cost at theta found by fminunc: 0.203498
```

```
disp('theta:');disp(theta);
```

```
theta:
  -25.1613
    0.2062
    0.2015
```

```
% Plot Boundary
plotDecisionBoundary(theta, X, y);
% Add some labels
hold on;
% Labels and Legend
xlabel('Exam 1 score')
ylabel('Exam 2 score')
% Specified in plot order
legend('Admitted', 'Not admitted')
hold off;
```



1.2.4 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 code below will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

```
%  Predict probability for a student with score 45 on exam 1  and score 85 on exam 2
prob = sigmoid([1 45 85] * theta);
```

```
g = 0.7763
```

```
fprintf('For a student with scores 45 and 85, we predict an admission probability of %f\n\n',
```

```
For a student with scores 45 and 85, we predict an admission probability of 0.776291
```

```
% Compute accuracy on our training set
p = predict(theta, X);
```

```
g = 100×1
    0.0910
    0.0000
    0.0439
    0.9904
    0.9982
    0.0108
    0.9990
    0.4232
    0.9997
    0.7354
     ⋮
```

```
fprintf('Train Accuracy: %f\n', mean(double(p == y)) * 100);
```

```
Train Accuracy: 89.000000
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

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, `plotData` is used in the code below 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.

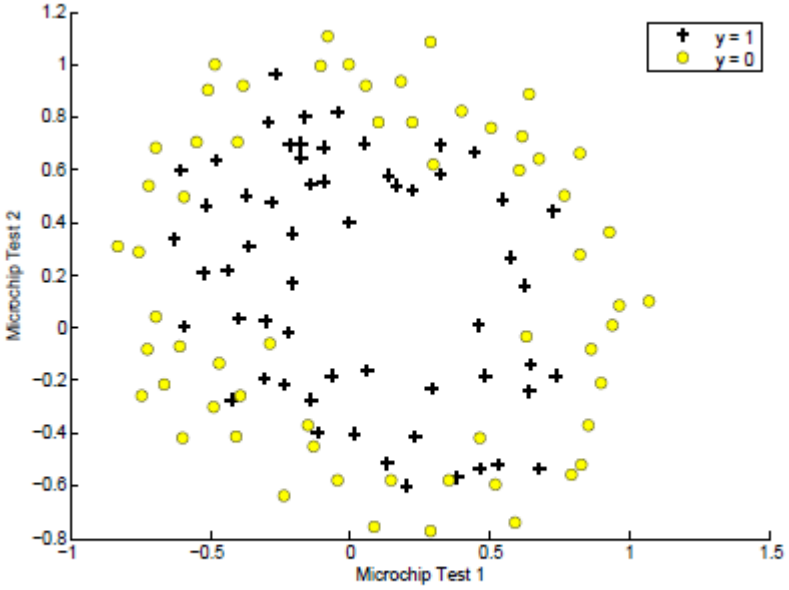


Figure 3: Plot of training data

Run the code below and confirm you plot matches Figure 3.

```
%  The first two columns contains the X values and the third column
%  contains the label (y).
data = load('ex2data2.txt');
X = data(:, [1, 2]); y = data(:, 3);

plotData(X, y);
% Put some labels
hold on;
% Labels and Legend
xlabel('Microchip Test 1')
ylabel('Microchip Test 2')
% Specified in plot order
legend('y = 1', 'y = 0')
hold off;
```

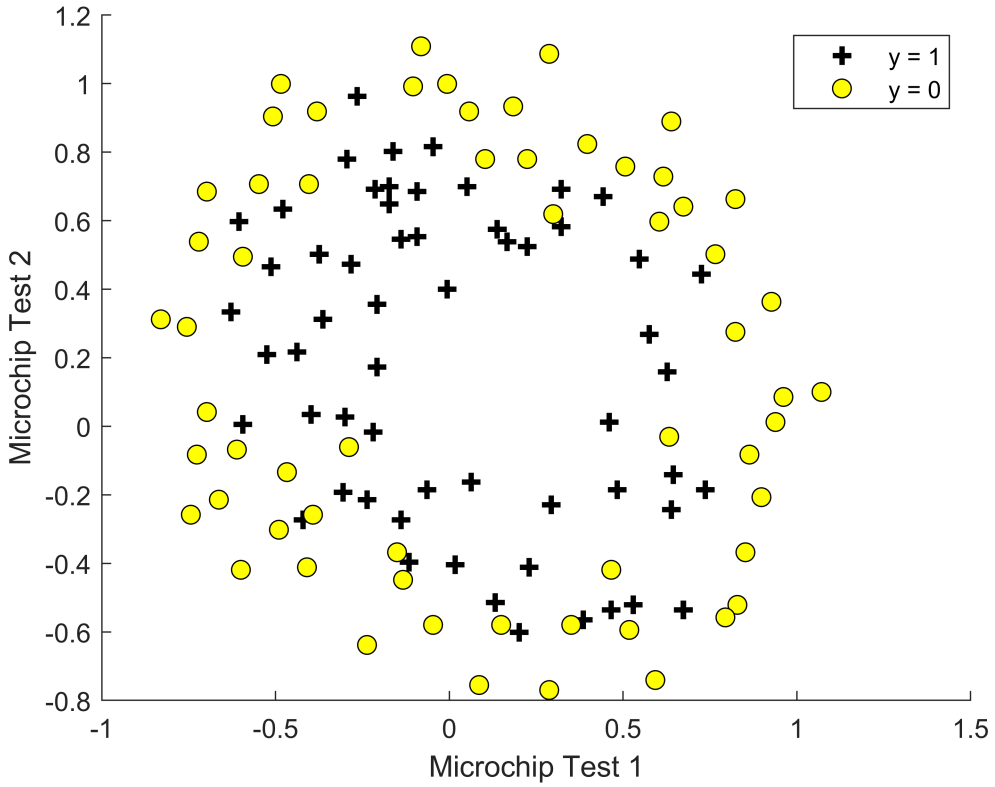


Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

2.2 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.

Run the code below to map the features.

```
% Add Polynomial Features
% Note that mapFeature also adds a column of ones for us, so the intercept term is handled
X = mapFeature(X(:,1), X(:,2));
```

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

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

Note that you should not regularize the parameter θ_0. In 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 jth element is defined as follows:

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

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

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

```
% Initialize fitting parameters
initial_theta = zeros(size(X, 2), 1);

% Set regularization parameter lambda to 1
lambda = 1;

% Compute and display initial cost and gradient for regularized logistic regression
[cost, grad] = costFunctionReg(initial_theta, X, y, lambda);
```

```
g = 118×1
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
```

```
0.5000
0.5000
0.5000
0.5000
  ⋮
```

```
fprintf('Cost at initial theta (zeros): %f\n', cost);
```

```
Cost at initial theta (zeros): 0.693147
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.3.1 Learning parameters using `fminunc`

Similar to the previous parts, the next step is to use `fminunc` to learn the optimal parameters. If you have completed the cost and gradient for regularized logistic regression (`costFunctionReg.m`) correctly, you should be able to run the code in the following sections to learn the parameters using `fminunc` for multiple values of λ.

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 (nonlinear) decision boundary that separates the positive and negative examples. In `plotDecisionBoundary.m`, we plot the nonlinear decision boundary by computing the classifier's predictions on an evenly spaced grid and then drew a contour plot of where the predictions change from $y = 0$ to $y = 1$. After learning the parameters, the code in the next section will plot a decision boundary similar to Figure 4.

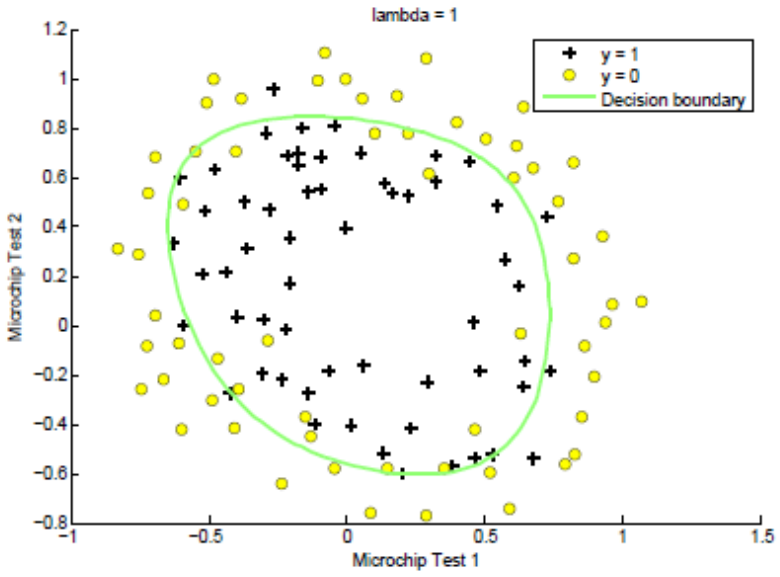


Figure 4: Training data with decision boundary ($\lambda = 1$)

2.5 Optional (ungraded) exercises

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 5).

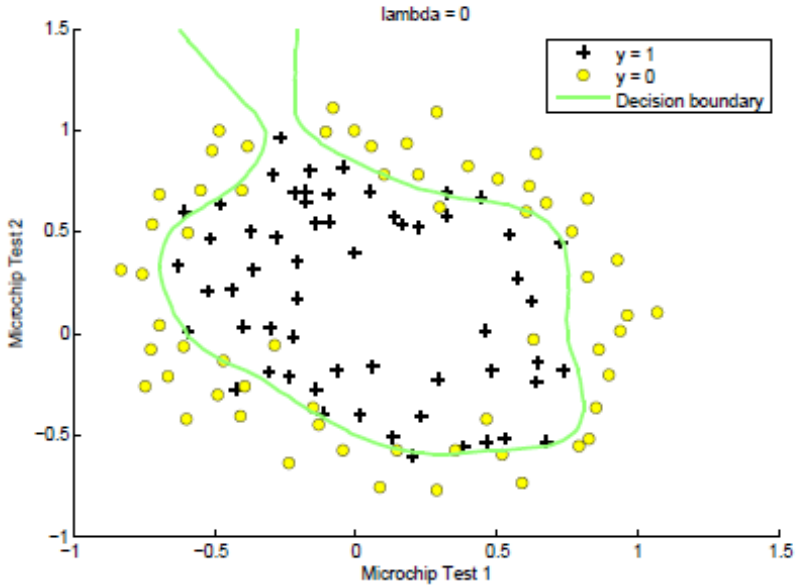


Figure 5: No regularization (Overfitting) ($\lambda = 0$)

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 an 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 6).

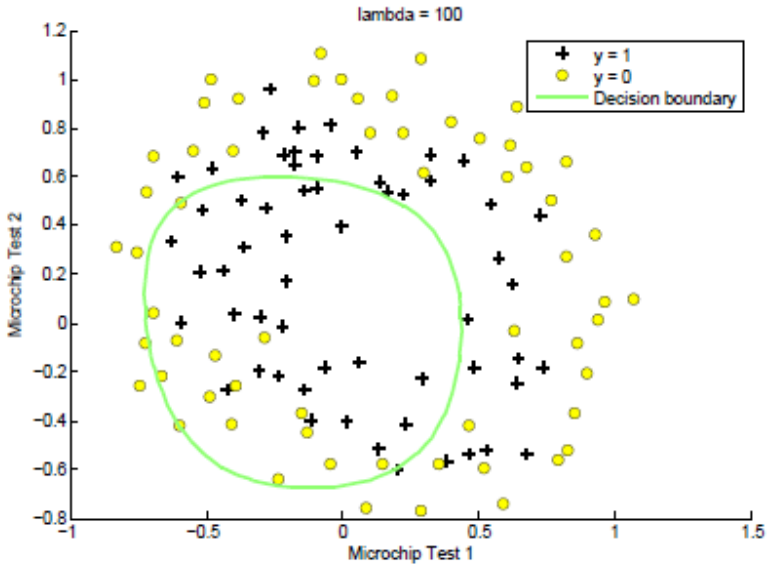


Figure 6: Too much regularization (Underfitting) ($\lambda = 100$)

Use the control below to try the following values of λ: 0, 1, 10, 100.

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

```
% Initialize fitting parameters
initial_theta = zeros(size(X, 2), 1);

lambda = 1;
% Set Options
options = optimoptions(@fminunc,'Algorithm','Quasi-Newton','GradObj', 'on', 'MaxIter', 1000);

% Optimize
[theta, J, exit_flag] = fminunc(@(t)(costFunctionReg(t, X, y, lambda)), initial_theta, options)
```

```
g = 118×1
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
    0.5000
      ⋮
g = 118×1
    0.4860
    0.4879
    0.4879
    0.4943
    0.4941
    0.4966
```

```
    0.4978
    0.4978
    0.4964
    0.4949
      ⋮
      ⋮
g = 118×1
    0.4771
    0.4947
    0.4951
    0.5450
    0.5378
    0.5483
    0.5566
    0.5488
    0.5331
    0.5148
      ⋮
      ⋮
g = 118×1
    0.5355
    0.5616
    0.5596
    0.6281
    0.6090
    0.6163
    0.6295
    0.6135
    0.5905
    0.5590
      ⋮
      ⋮
g = 118×1
    0.6186
    0.6468
    0.6387
    0.7091
    0.6731
    0.6715
    0.6918
    0.6680
    0.6452
    0.6024
      ⋮
      ⋮
g = 118×1
    0.6707
    0.6947
    0.6807
    0.7410
    0.6909
    0.6804
    0.7076
    0.6797
    0.6633
    0.6157
      ⋮
      ⋮
g = 118×1
    0.7115
    0.7278
    0.7071
    0.7502
    0.6827
```

```
    0.6606
    0.6971
    0.6656
    0.6610
    0.6111
      ⋮
g = 118×1
    0.7215
    0.7324
    0.7080
    0.7376
    0.6585
    0.6285
    0.6712
    0.6384
    0.6439
    0.5950
      ⋮
g = 118×1
    0.7156
    0.7252
    0.7002
    0.7244
    0.6423
    0.6099
    0.6541
    0.6215
    0.6308
    0.5834
      ⋮
g = 118×1
    0.7084
    0.7187
    0.6946
    0.7175
    0.6354
    0.6025
    0.6467
    0.6142
    0.6242
    0.5775
      ⋮
g = 118×1
    0.6980
    0.7115
    0.6900
    0.7128
    0.6320
    0.5988
    0.6425
    0.6094
    0.6190
    0.5720
      ⋮
g = 118×1
    0.6933
    0.7111
    0.6927
    0.7174
```

```
    0.6381
    0.6046
    0.6478
    0.6134
    0.6211
    0.5724
      ⋮
g = 118×1
    0.6962
    0.7167
    0.6999
    0.7267
    0.6481
    0.6147
    0.6579
    0.6223
    0.6280
    0.5773
      ⋮
g = 118×1
    0.6996
    0.7204
    0.7037
    0.7313
    0.6527
    0.6197
    0.6631
    0.6272
    0.6320
    0.5804
      ⋮
g = 118×1
    0.7014
    0.7220
    0.7051
    0.7331
    0.6544
    0.6220
    0.6657
    0.6297
    0.6340
    0.5818
      ⋮
g = 118×1
    0.7032
    0.7234
    0.7059
    0.7346
    0.6557
    0.6250
    0.6693
    0.6333
    0.6365
    0.5833
      ⋮
g = 118×1
    0.7031
    0.7228
    0.7047
```

```
    0.7342
    0.6551
    0.6268
    0.6719
    0.6361
    0.6378
    0.5833
      ⋮
g = 118×1
    0.7009
    0.7204
    0.7020
    0.7320
    0.6528
    0.6263
    0.6719
    0.6363
    0.6370
    0.5818
      ⋮
g = 118×1
    0.6992
    0.7186
    0.7002
    0.7303
    0.6510
    0.6250
    0.6707
    0.6352
    0.6358
    0.5805
      ⋮
g = 118×1
    0.6986
    0.7181
    0.6998
    0.7297
    0.6504
    0.6243
    0.6700
    0.6345
    0.6352
    0.5801
      ⋮
g = 118×1
    0.6983
    0.7179
    0.6995
    0.7294
    0.6500
    0.6237
    0.6695
    0.6340
    0.6350
    0.5800
      ⋮
g = 118×1
    0.6983
    0.7178
```

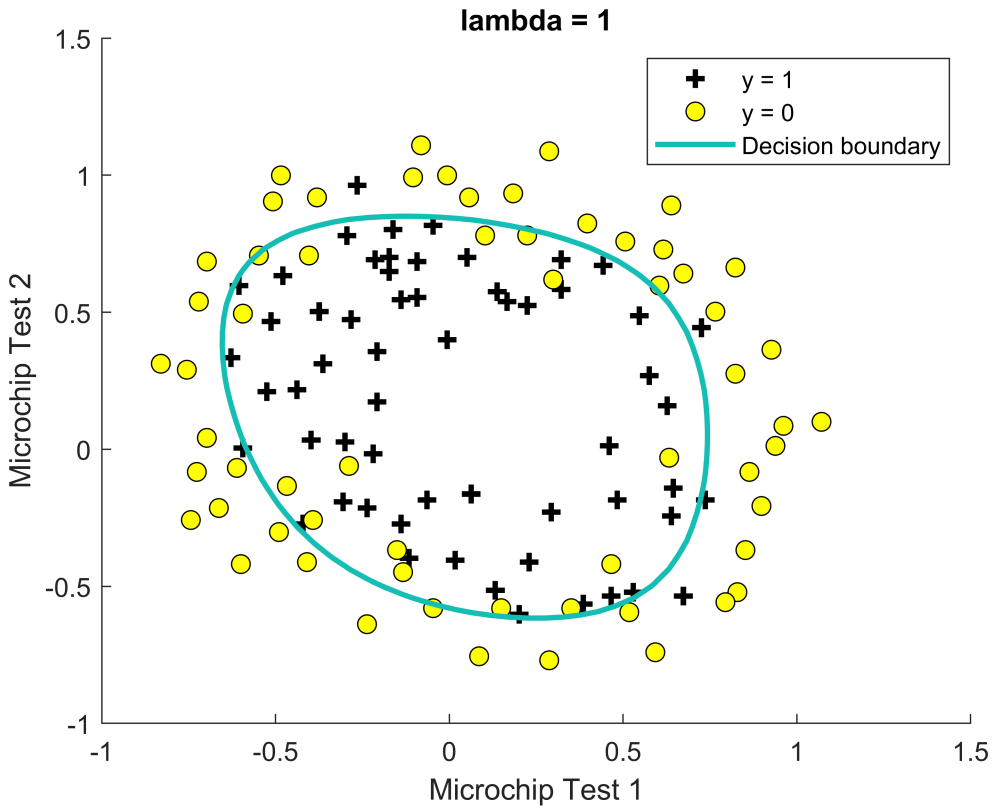
```
    0.6995
    0.7291
    0.6497
    0.6233
    0.6691
    0.6337
    0.6351
    0.5802
      ⋮
      ⋮
g = 118×1
    0.6986
    0.7180
    0.6996
    0.7292
    0.6496
    0.6233
    0.6693
    0.6339
    0.6355
    0.5809
      ⋮
      ⋮
g = 118×1
    0.6989
    0.7183
    0.6998
    0.7294
    0.6498
    0.6236
    0.6696
    0.6343
    0.6361
    0.5815
      ⋮
      ⋮
g = 118×1
    0.6990
    0.7184
    0.6999
    0.7295
    0.6500
    0.6238
    0.6698
    0.6345
    0.6363
    0.5818
      ⋮
      ⋮
g = 118×1
    0.6990
    0.7183
    0.6999
    0.7296
    0.6501
    0.6240
    0.6700
    0.6346
    0.6364
    0.5819
      ⋮
      ⋮
g = 118×1
    0.6988
```

```
    0.7182
    0.6998
    0.7297
    0.6502
    0.6241
    0.6700
    0.6346
    0.6364
    0.5819
     ⋮
g = 118×1
    0.6987
    0.7181
    0.6997
    0.7297
    0.6503
    0.6241
    0.6699
    0.6344
    0.6362
    0.5817
     ⋮
g = 118×1
    0.6986
    0.7180
    0.6996
    0.7297
    0.6503
    0.6241
    0.6698
    0.6342
    0.6359
    0.5815
     ⋮
g = 118×1
    0.6986
    0.7181
    0.6997
    0.7297
    0.6504
    0.6241
    0.6697
    0.6341
    0.6358
    0.5813
     ⋮
g = 118×1
    0.6987
    0.7181
    0.6997
    0.7297
    0.6504
    0.6241
    0.6697
    0.6341
    0.6358
    0.5813
     ⋮
g = 118×1
```

```
    0.6987
    0.7182
    0.6998
    0.7297
    0.6504
    0.6241
    0.6697
    0.6341
    0.6358
    0.5813
      ⋮
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
```

```
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
      ⋮
Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the optimality tolerance.

<stopping criteria details>
```

```
% Plot Boundary
plotDecisionBoundary(theta, X, y);
hold on;
title(sprintf('lambda = %g', lambda))

% Labels and Legend
xlabel('Microchip Test 1')
ylabel('Microchip Test 2')

legend('y = 1', 'y = 0', 'Decision boundary')
hold off;
```




```
% Compute accuracy on our training set
p = predict(theta, X);
```

```
g = 118×1
    0.6987
    0.7182
    0.6998
    0.7298
    0.6505
    0.6241
    0.6698
    0.6341
    0.6358
    0.5813
     ⋮
```

```
fprintf('Train Accuracy: %f\n', mean(double(p == y)) * 100);
```

```
Train Accuracy: 83.050847
```

You do not need to submit any solutions for these optional (ungraded) exercises.

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Sigmoid Function	`sigmoid.m`	5 points
Compute cost for logistic regression	`costFunction.m`	30 points
Gradient for logistic regression	`costFunction.m`	30 points
Predict Function	`predict.m`	5 points
Compute cost for regularized LR	`costFunctionReg.m`	15 points
Gradient for regularized LR	`costFunctionReg.m`	15 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration.