Homework 1: Linear Model*

Pattern Recognition and Machine Learning

September 21, 2016

Instructions

- **Submission:** 本次作业需提交的内容包括实验报告文档和代码, 实验报告文档主要描述原理和结果，请每位同学压缩打包, 压缩包的命名名字格式为“学号-班级-姓名-PRMLHW1.zip”. 请各班负责人（班长、学委等）统一将本班的所有作业收集并按班级和作业名的格式统一打包（如计算机1401班的命名为“计算机1401-PRMLHW1.zip”），在deadline 前发到yxliang@csu.edu.cn. 未按时提交的则在deadline后自行按指定的命名格式打包后发到yxliang@csu.edu.cn.

- **Late homework policy:** 本次作业提交的deadline为2016.10.31 24:00, 约占平时成绩的1/3, 迟交则按每天0.8的比例进行计算, 如本来作业能打10分, 迟交1天则算8分, 迟交2天则算6.4分, 以此类推.

- **Collaboration policy:** Homeworks must be done individually, except where otherwise noted in the assignments. “Individually” means each student must hand in their own answers, and each student must write and use their own code in the programming parts of the assignment. It is acceptable for students to collaborate in figuring out answers and to help each other solve the problems, though you must in the end write up your own solutions individually, and you must list the names of students you discussed this with. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such collaboration.

Help

The exercises in this course use Octave[1], a high-level programming language well-suited for numerical computations. At the Octave 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.

Part I

Linear Regression

Files included in this part

- `ex1.m` - Octave script that will help step you through the exercise
- `ex1_multi.m` - Octave script for the later parts of the exercise

*本次作业内容来自Andrew Ng’s mlClass `http://www.ml-class.org/course/class/index/`.

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

- `ex1data1.txt` - Dataset for linear regression with one variable
- `ex1data2.txt` - Dataset for linear regression with multiple variables
- `warmUpExercise.m` - Simple example function in Octave
- `plotData.m` - Function to display the dataset
- `computeCost.m` - Function to compute the cost of linear regression
- `gradientDescent.m` - Function to run gradient descent
- `computeCostMulti.m` - Cost function for multiple variables
- `gradientDescentMulti.m` - Gradient descent for multiple variables
- `featureNormalize.m` - Function to normalize features
- `normalEqn.m` - Function to compute the normal equations

Throughout the part, you will be using the scripts `ex1.m` and `ex1_multi.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do NOT need to modify either of them. You are ONLY required to modify functions in other files, by following the instructions in this assignment.

1 Simple octave function

The first part of ex1.m gives you practice with Octave syntax and the home work submission process. In the file warmUpExercise.m, you will find the outline of an Octave function. Modify it to return a 5×5 identity matrix by filling in the following code:

```
A = eye(5);
```

When you are finished, run ex1.m (assuming you are in the correct directory, type 'ex1' at the Octave prompt) and you should see output similar to the following:

```
ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

Now ex1.m will pause until you press any key, and then will run the code for the next part of the assignment. If you wish to quit, typing **ctrl+c** will stop the program in the middle of its run.

2 Linear regression with one variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities.

You would like to use this data to help you select which city to expand to next.

The file ex1data1.txt contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss. The ex1.m script has already been set up to load this data for you.

2.1 Plotting the Data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). (Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-D plot.)

In `ex1.m`, the dataset is loaded from the data file into the variables X and y:

```
data = load('ex1data1.txt'); % read comma separated data
X = data(:, 1); y = data(:, 2);
m = length(y); % number of training examples
```

Next, the script calls the plotData function to create a scatter plot of the data. Your job is to complete plotData.m to draw the plot; modify the file and fill in the following code:

```
plot(x, y, 'rx', 'MarkerSize', 10); % Plot the data
ylabel('Profit in $10,000s'); % Set the y-axis label
xlabel('Population of City in 10,000s'); % Set the x-axis label
```

Now, when you continue to run ex1.m, our end result should look like Figure 1, with the same red "x" markers and axis labels.

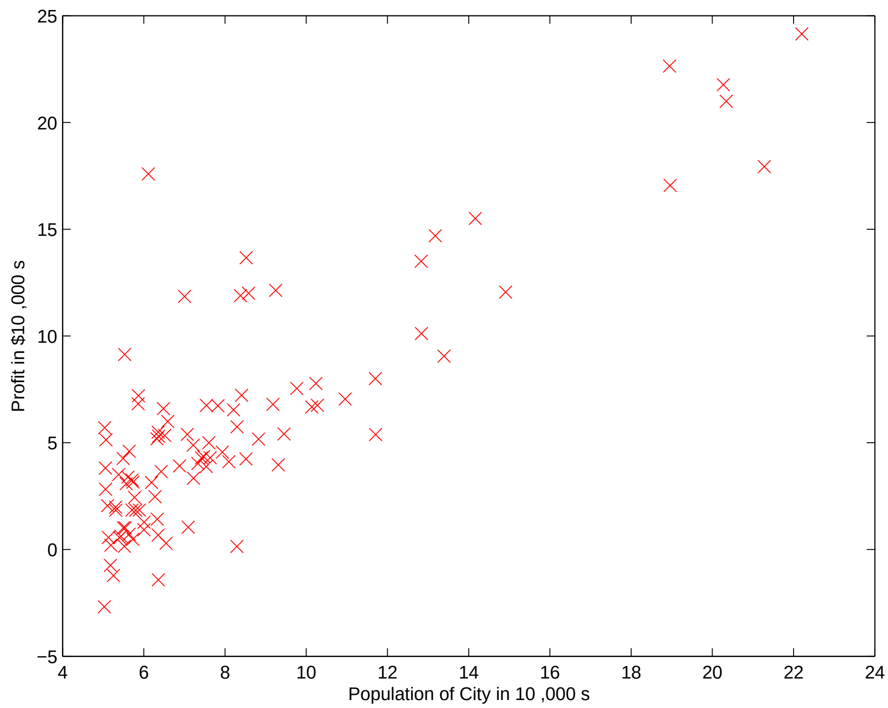


Figure 1: Scatter plot of training data

To learn more about the plot command, you can type **help plot** at the Octave command prompt or to search online for plotting documentation. (To change the markers to red 'x', we used the option 'rx' together with the plot command, i.e., plot(..,[your options here],.., 'rx');)

2.2 Gradient Descent

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent.

2.2.1 Update Equations

The objective of linear regression is to minimize the cost function

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

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

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall 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

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{).}$$

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

Implementation Note: We store each example as a row in the the X matrix in Octave. To take into account the intercept term (θ_0), we add an additional first column to X and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

2.2.2 Implementation

In `ex1.m`, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term. We also initialize the initial parameters to 0 and the learning rate alpha to 0.01.

```
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
theta = zeros(2, 1); % initialize fitting parameters

iterations = 1500;
alpha = 0.01;
```

2.2.3 Computing the cost $J(\theta)$

As you perform gradient descent to learn minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation. Your next task is to complete the code in the file `computeCost.m`, which is a function that computes $J(\theta)$. As you are doing this, remember that the variables X and y are not scalar values, but matrices whose rows represent the examples from the training set.

Once you have completed the function, the next step in `ex1.m` will run `computeCost` once using θ initialized to zeros, and you will see the cost printed to the screen.

You should expect to see a cost of `32.07`.

2.2.4 Gradient descent

Next, you will implement gradient descent in the file `gradientDescent.m`. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost $J(\theta)$ is parameterized by the vector θ, not X and y. That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing X or y. Refer to the equations in this handout and to the video lectures if you are uncertain.

A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. The starter code for `gradientDescent.m` calls `computeCost` on every iteration and prints the cost. Assuming you have implemented gradient descent and `computeCost` correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm.

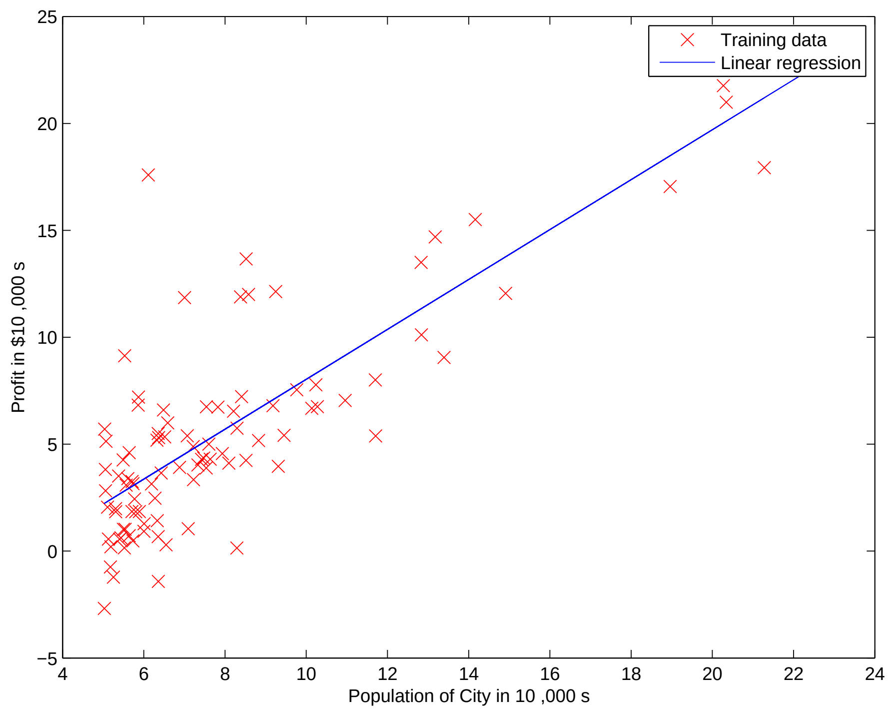


Figure 2: Training data with linear regression fit

After you are finished, `ex1.m` will use your final parameters to plot the linear fit. The result should look something like Figure 2. Your final values for $J(\theta)$ will also be used to make predictions on profits in areas of 35,000 and 70,000 people. Note the way that the following lines in `ex1.m` uses matrix multiplication, rather than explicit summation or looping, to calculate the predictions. This is an example of code vectorization in Octave.

2.3 Debugging

Here are some things to keep in mind as you implement gradient descent:

- Octave array indices start from one, not zero. If you're storing θ_0 and θ_1 in a vector called `theta`, the values will be `theta(1)` and `theta(2)`.

- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you're adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the size command will help you debug.

- By default, Octave interprets math operators to be matrix operators. This is a common source of size incompatibility errors. If you don't want matrix multiplication, you need to add the "dot" notation to specify this to Octave. For example, `A*B` does a matrix multiply, while `A.*B` does an element-wise multiplication.

2.4 Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

In the next step of `ex1.m`, there is code set up to calculate $J(\theta)$ over a grid of values using the `computeCost` function that you wrote.

```
% initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1:length(theta0_vals)
    for j = 1:length(theta1_vals)
    t = [theta0_vals(i); theta1_vals(j)];
    J_vals(i,j) = computeCost(X, y, t);
    end
end
```

After these lines are executed, you will have a 2-D array of $J(\theta)$ values. The script `ex1.m` will then use these values to produce surface and contour plots of $J(\theta)$ using the surf and contour commands. The plots should look something like Figure 4.

The purpose of these graphs is to show you that how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function $J(\theta)$ is bowl-shaped and has a global minimum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

3 Linear regression with multiple variables

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to first collect information on recent houses sold and make a model of housing prices.

The file `ex1data2.txt` contains a training set of housing prices in Port- land, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house.

The `ex1_multi.m` script has been set up to help you step through this exercise.

3.1 Feature Normalization

The `ex1_multi.m` script will start by loading and displaying some values from this dataset. By looking at the values, note that house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

Your task here is to complete the code in `featureNormalize.m` to [2]

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations".

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within ±2 standard deviations of the mean); this is an alternative

[2] 另外也可以实现课堂上讲的(0,1)-normalization.

to taking the range of values (max-min). In Octave, you can use the "`std`" function to compute the standard deviation. For example, inside featureNormalize.m, the quantity `X(:,1)` contains all the values of x_1 (house sizes) in the training set, so `std(X(:,1))` computes the standard deviation of the house sizes. At the time that featureNormalize.m is called, the extra column of 1's corresponding to $x_0 = 1$ has not yet been added to `X` (see `ex1_multi.m` for details).

You will do this for all the features and your code should work with dataset of all sizes (any number of features / examples). Note that each column of the matrix `X` corresponds to one feature.

Implementation Note: When normalizing the features, it is important to store the values used for normalization - the *mean value* and the *standard deviation* used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new `x` value (living room area and number of bedrooms), we must first normalize `x` using the mean and standard deviation that we had previously computed from the training set.

3.2 Gradient Descent

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there is one more feature in the matrix `X`. The hypothesis function and the batch gradient descent update rule remain unchanged.

You should complete the code in `computeCostMulti.m` and `gradientDescentMulti.m` to implement the cost function and gradient descent for linear regression with multiple variables. If your code in the previous part (single variable) already supports multiple variables, you can use it here too.

Make sure your code supports any number of features and is well-vectorized. You can use '`size(X, 2)`' to find out how many features are present in the dataset.

Implementation Note: In the multivariate case, the cost function can also be written in the following vectorized form:

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y)$$

where

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}, \quad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

The vectorized version is efficient when you're working with numerical computing tools like Octave. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

3.2.1 Selecting learning rates

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying `ex1_multi.m` and changing the part of the code that sets the learning rate.

The next phase in `ex1_multi.m` will call your gradientDescent.m function and run gradient descent for about 50 iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector J. After the last iteration, the `ex1_multi.m` script plots the J values against the number of the iterations.

If you picked a learning rate within a good range, your plot look similar Figure **??**. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on).

You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and 'blow up', resulting in values which are too large for computer calculations. In these situations, Octave will tend to return NaNs. NaN stands for 'not a number' and is often caused by undefined operations that involve $-\infty$ and $+\infty$.

Octave Tip: To compare how different learning learning rates affect convergence, it's helpful to plot J for several learning rates on the same figure. In Octave, this can be done by performing gradient descent multiple times with a '`hold on`' command between plots. Concretely, if you've tried three different values of alpha (you should probably try more values than this) and stored the costs in `J1`, `J2` and `J3`, you can use the following commands to plot them on the same figure:

```
plot(1:50, J1(1:50), 'b');
hold on;
plot(1:50, J2(1:50), 'r');
plot(1:50, J3(1:50), 'k');
```

The final arguments 'b', 'r', and 'k' specify different colors for the plots.

Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Using the best learning rate that you found, run the `ex1_multi.m` script to run gradient descent until convergence to find the final values of θ. Next, use this value of θ to predict the price of a house with 1650 square feet and 3 bedrooms. You will use value later to check your implementation of the normal equations. Don't forget to normalize your features when you make this prediction!

3.3 Normal Equations

In the lecture videos, you learned that the closed-form solution to linear regression is

$$\theta = (X^T X)^{-1} X^T y.$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.

Complete the code in `normalEqn.m` to use the formula above to calculate θ. Remember that while you don't need to scale your features, we still need to add a column of 1's to the `X` matrix to have an intercept term (θ_0). The code in `ex1.m` will add the column of 1's to X for you.

Now, once you have found θ using this method, use it to make a price prediction for a 1650-square-foot house with 3 bedrooms. You should find that gives the same predicted price as the value you obtained using the model fit with gradient descent.

Part II

Logistic Regression for Classification

Files included in this exercise

- `ex2.m` - Octave script that will help step you through the exercise
- `ex2_reg.m` - Octave script for the later parts of the exercise
- `ex2data1.txt` - Training set for the first half of the exercise
- `ex2data2.txt` - Training set for the second half of the exercise
- `mapFeature.m` - Function to generate polynomial features

- `plotDecisionBounday.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

Throughout the exercise, you will be using the scripts `ex2.m` and `ex2_reg.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions in other files, by following the instructions in this assignment.

4 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.

4.1 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`.

You will now complete the code in `plotData` so that it displays a figure like Figure 5, where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

To help you get more familiar with plotting, we have left `plotData.m` empty so you can try to implement it yourself. 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 Octave documentation.

```
% Find Indices of Positive and Negative Examples
pos = find(y==1); neg = find(y == 0);
% 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);
```

4.2 Implementation

4.2.1 Warmup exercise: sigmoid function

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

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

where

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

For now, let's take the choice of g as given. Other functions that smoothly increase from 0 to 1 can also be used, but the choice of the logistic function is a fairly natural one. Before moving on, here's a useful property of the derivative of the sigmoid function, which we write as g':

$$\begin{aligned} g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2}\left(e^{-z}\right) \\ &= \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{(1+e^{-z})}\right) \\ &= g(z)(1-g(z)). \end{aligned}$$

Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, g(z), and hence also $h(x)$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^{n}\theta_j x_j$.

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.**

4.3 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

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

and

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

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.

4.4 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`.

Octave's `fminunc` is an optimization solver that finds the minimum of an unconstrained[3] function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ.

Concretely, you are going to use `fminunc` to

nd 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.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.m 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 6. We also encourage you to look at the code in plotDecisionBoundary.m to see how to plot such a boundary using the θ values.

4.5 Evaluating logistic regression

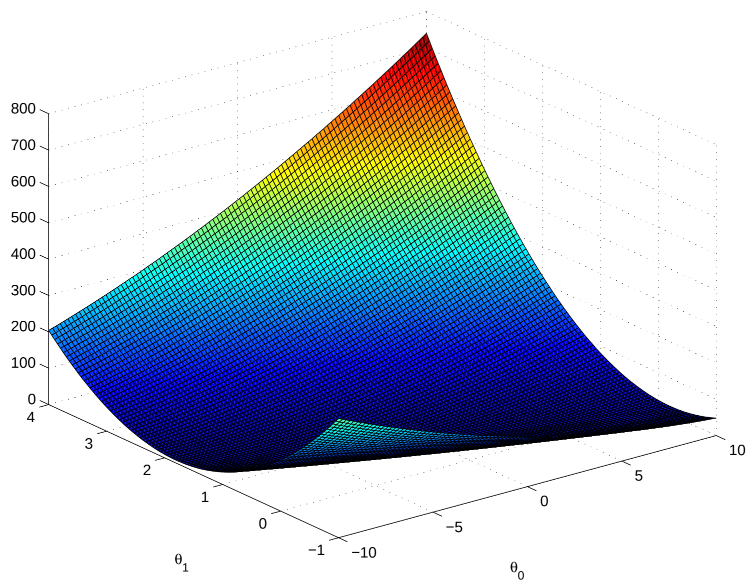
After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.776.

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.

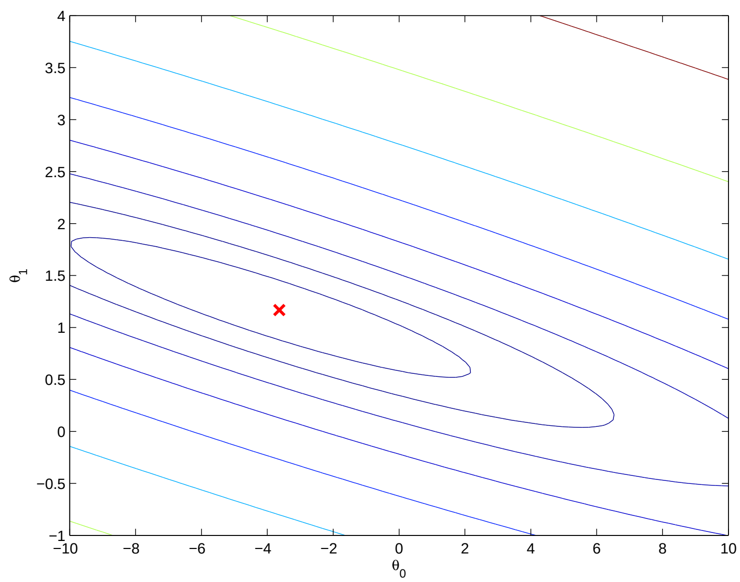
---

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

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.



(a) Surface



(b) Contour, showing minimum

Figure 3: Cost function $J(\theta)$.

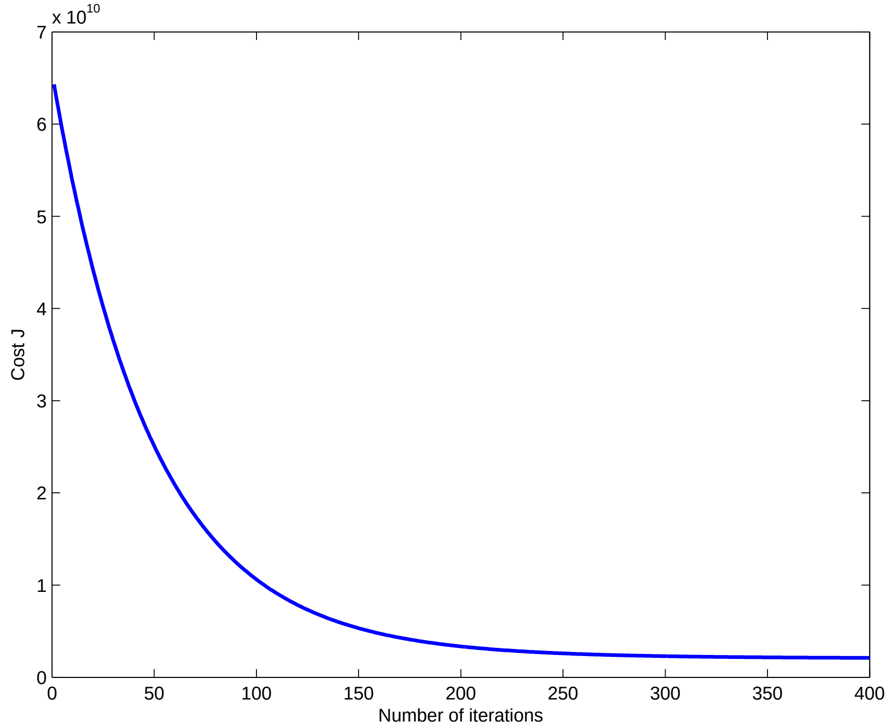


Figure 4: Convergence of gradient descent with an appropriate learning rate.

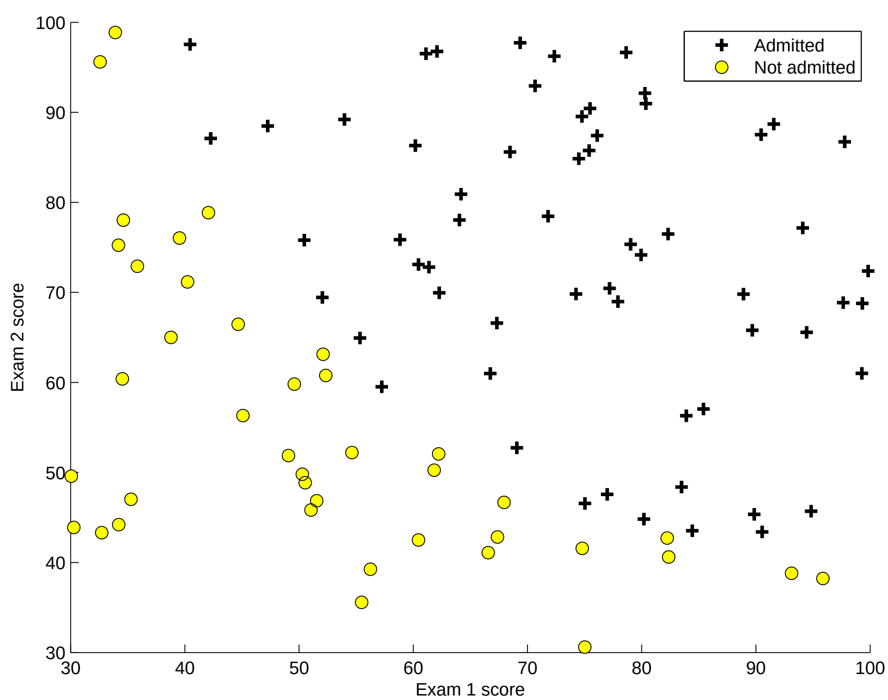


Figure 5: Scatter plot of training data.

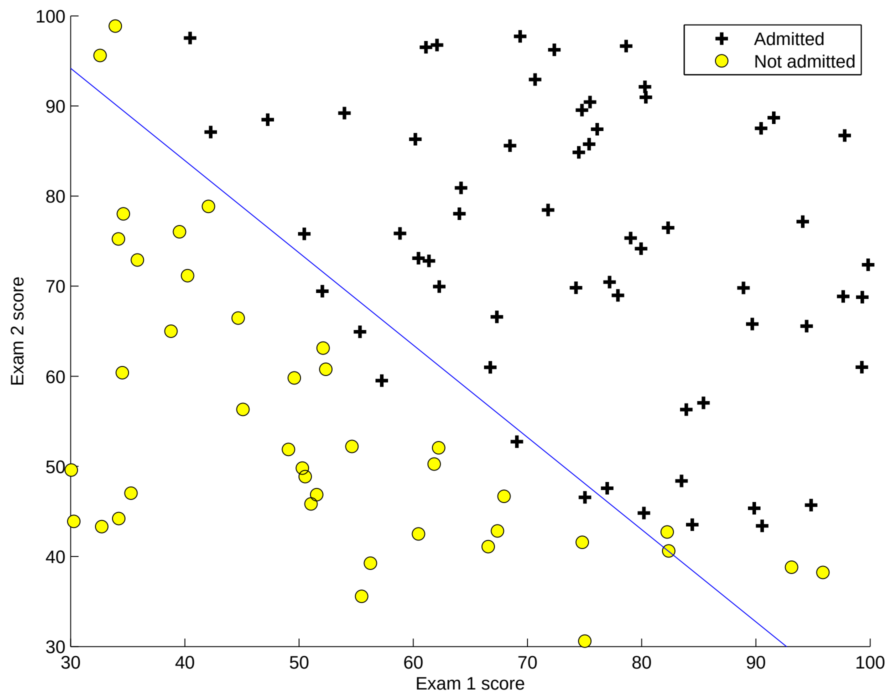


Figure 6: Training data with decision boundary.