**1.2.1**

*Sigmoid.m*

The first task was to create a sigmoid function since this is the function in a logistic regression. We created the “sigmoid” function by calling the ‘function’ command and specifying the output variable after that: function g

We set this equal to the name of the function and the input values of the function, which is just z: function g = sigmoid(z)

We initialized the output variable g by defining it as a matrix of all 0s with the rows and column number equal to the input value. For example, if we say sigmoid(3), it will return a 3x3 matrix of 0s. Still unsure why this is necessary.

Next, we write the equation for what g is equal to. We need to do this because our hypothesis for logistic regression is: h(x) = g(z), where z = (theta)’X, and g(z) is 1 / (1 + e-z).

So we make g = 1 ./ (1 + exp(-z)). The “./” indicates that we are taking 1 over each element of z, as a vectorized method. This way, if we input multiple z, we get back multiple g (if we input a matrix for z). If we put in 0, we get back 0.5. This is because 1 / (1 + e-0) = 1 / (1+1) = ½.

**1.2.2**

*CostFunction.m*

Remember that the cost of a fitted formula is essentially the mean squared error of a formula. As such, we need the h(x) value for each x and y from the data. The parameters theta will determine what the line (or other formula) looks like. Thus our cost function will need to know x and y, and also the h(x). And we get the h(x) from our x and theta.

First we will define the function. This function will have output J, which is our cost, and grad, which is the gradient. It will take the inputs of x, y, and theta.

Y is the output variable. It is just a vector that we took from our data. This has to be subsetted for. X is a matrix of the predictor variables. Again, this is subsetted for. This data has a length of ‘m’, so we first define that with the length() command: m = length(y)

Then we initialize our outputs. J is a single number, so we can initially define it as J = 0. The next output is the gradient, which is specific for each variable. As we take partial derivatives, each variable has its own gradient as we perform the gradient descent. We initialize this by creating a matrix that is the same size as the theta matrix: grad = zeros(size(theta))

Now we write the equation for z. we know that z = theta’ X to make a vector of h(x)s for each observation. This is what we are doing here. Though the math is the same, we are going to write the equation a little differently because of the way the matrices are set up; we are going to make z = X \* theta.

This is because X is an [m x n] matrix with m observations and n variables. Theta is a [n x 1] matrix, a single column of n slopes (one for each *variable*). So when we do X \* theta, we do a [m x n] [n x 1] matrix multiplication giving us an [m x 1] column of Zs.

But this is not the h(x). Remember that our variable, y, is binary, 1 & 0. And this is logistic regression, so our fitted outcome variable must be between 0 and 1. So we fit the z on the sigmoid curve. We do this by saying: hx = sigmoid(z) which creates an [m x 1] matrix of predicted h(x) values for each observation.

Now we can write the equation for the cost function. Recall that for logistic regression, the cost function is:

J = (1/m) \* sum( -y \* log(h(x)) – (1-y) \* log(1 – h(x)) )

But h(x) and y are matrixes – [m x 1] each. So they can’t be multiplied together normally, it will give an error. We want each observation’s h(x) and y for the arithmetic. So in the Octive formula, we use .\* for the multiplication. And then the sum() function takes the sum for each observation.

Then recall that we get the new thetas from the gradient where:

Theta = theta – alpha \* partial derivative of cost for variable j

The partial derivative for thetaj is (1/m) \* sum(h(x) – y)\*xj

So in the code, X is a [m x n] matrix while both h(x) and y are [m x 1] matrices. And we expect to get a theta for each variable (n) right? So gradient will be a vector with n elements. In order to do this, we transpose X. X’ is now a [n x m] matrix. A few things to notice about the code: grad = (1/m) \* X' \* (hx - y);

We do X’ \* (hx-y). This combines the multiplication part and the sum part in one step. This is a [n x m] [m x 1] matrix multiplication. What this does is multiply the difference in y and h(x) *for each observation* by the xj variable *for each observation*. And sums the that value from each observation. So each variable gets a gradient. Thus, grad returns an [n x 1] matrix.

*Ex2.m*

We will go over what ex2.m does when we run it.

After we load the data, we subset X and Y as the predictor and outcome variables. Remember that X is a set of 2 exam scores per student, and Y is a binary of whether or not the student was admitted to the college or whatever. Here is how we did it (remember that the data is put into an object called “data”

X = data( : , [1,2]) *this takes all the observations from the first 2 columns*

Y = data( : , 3) *this takes all the observations from the 3rd column*

Part 1 plots. We’ll skip this a go to part 2.

First we define “m” and “n” from the object X

When we say: [m , n] = size(X)

We are saying, take the size, columns and rows, and put them into m and n respectively. So we have “m” observations and “n” independent variables

Ok, recall that in any equation, there is an intercept. The intercept is theta0, and each observation should have an intercept. We will create a placeholder value of 1 in each observation so when it is multiplied by the theta vector, we have a theta0.

The code: X = [ones(m, 1) X] does this. It creates a m x 1 matrix of 1’s. And then concatenates it to X, saving into a new X. Now, X is a [m x 3] matrix.

Now we make a vector of the thetas. We create the initial\_theta variable = zeros(n + 1, 1) which makes an [n+1 x 1] matrix of 0s. Why n+1? Well recall that n is the number of variables right? That’s n variables – 2 in this example. But we also have an intercept – we made the [m x 3] matrix in the last example with the placeholder of 1. We need a theta for that placeholder, the intercept, so our theta vector will be an n+1.

Now the program calls the costFunction function with initial\_theta as the theta and X and Y as the predictor and outcome variables. It then does this for test\_theta too.

Basically, in this program, we supply the thetas and find what the cost and gradient are. Look at the output. It gives the calculated cost and the expected cost (answer for the exercise). They should be the same. If they are not, you won’t get credit for the assignment. It also gives three gradients for the initial\_theta. It compares it to that is expected in this exercise in order to get credit.

There is also a test\_thea. Same concept. Note that is does **not** perform gradient descent. It simply gives the cost and the gradient at the supplied thetas.

2 Regularized logistic regression

We need to go to a little theory here. Remember that overfitting can be a problem with regression. That is, there are too many features in the learned hypothesis, so now the equation will not be generalizable. There are two ways to address overfitting:

1. Reduce the number of features: manually select with features to keep or use model selection algorithms.
2. Regularization: Keep all the features, but reduce the magnitude of theta. This works well with many features that all contribute to predicting Y.

The idea is, if we add a regularization parameter based on theta to the cost function, the thetas all have to be very small to keep the cost small.

Cost function for linear regression: J = 1/2m \* sum( hx – y )2 + lambda \* sumnj=1( thetaj2 )

The hypothesis and regularization term for logistic regression is different, but the effect is the same.

Notice that each feature’s theta is regularized, that that the intercept, theta0, is not regularized. We have to account for this in the assignment. Also remember that Octave indexes from 1. So in the vector of all the thetas, theta2 is the start of parameters for the features.

On to the assignment.

*PlotData.m*

The only thing that needs to be done here is making sure that the data is from ex2data2.txt

*mapFeature.m*

There is nothing to do in this assignment. But just understand that this takes the 2 columns of scores, and turns in into 28 features by going up to the 6th power for each column and different combinations of multiplying each column. Just understand that now there are 28 features now in 28 columns. Each column has 118 observations. Thus, the features are now a [118 x 28] matrix. Which is [m x n]

X = [m x n] = [observations x features] = [rows x columns] = [118 x 28]

*costFunctionReg.m*

The first thing this does is define the function and its outputs – cost and gradient

function [J, grad]

the function is called costFunctionReg, and takes the inputs theta, X, y, and lambda.

Thus we have **function [J, grad] = costFunctionReg(theta, X, y, lambda)**

First we need to make a few definitions. The ‘m’ is the number of observations, so we say **m = length(y)** to define that m is the number of observations, which is 118.

Then we initialize the cost variable: **J = 0**

And we initialize the gradient vector. Remember that each feature has a gradient, so we just initialize it as a matrix of zeros, the size of theta: **grad = zeros(size(theta))**

A few more definitions with regards to logistic regression. There should be more detail above. But recall that the hypothesis is: h(x) = 1 / (1 + e-theta’\*X). We set z = -theta’ \* X.

We use the code **z = X \* theta**

Why is this different from theta’ \* X? Because X is a [m x n] matrix of [118 x 28] and theta is a [n x 1] matrix of [28 x 1]. And theta’ \* X assumes that the theta is a row vector. But it is already a column vector, so we don’t need to transpose it.

We can just to X \* theta, [m x n] \* [n \* 1] to get z which is [m x 1], [118 x 1]

(Side note, technically its n+1, we’ll come back to this)

And **hx = sigmoid(z)** just uses the equation in the *sigmoid.m* function to put z on a sigmoid curve.

Now we define the cost function of a logistic regression. Look above for the detailed explanation of how its written. But with this cost function, we also add the regularization formula: lamba / 2m \* sumj=1 ( thetaj2)

The first theta, theta0, which is the intercept, is not regularized. So the code most indicate that all thetas after the first are included. Here is where recalling that Octave indexes from 1 is important.

Here is the code for the regularization term: **(lambda / (2\*m)) \* sum(theta(2:length(theta)) .^ 2);**

Two things to notice:

1. The indexing of the theta. We use theta(2:length(theta)) to indicate that we are only using the second theta to the last. The end of the index is ‘length(theta)’ but alternatively, ‘end’ could be used.
2. Each element in the theta vector is squared, so we need use ‘.^2’ to indicate as such

Now for the gradients. The ‘grad’ object is a vector, initialized as the size of ‘theta’. See earlier in the code. In regularized gradient descent, because the first theta is not regularized, the first gradient is not regularized.

As a review, here is regularized gradient descent:

Repeat{

thetaj := thetaj – alpha \* (1/m) sum( h(x) – y ) Xj + (lambda/m) \* thetaj

}

Green is the regularized portion

Grey highlight is the derivative of J(theta), the ***gradient***

So for the gradient of the first theta, put into the first element of grad is:

**grad(1) = (1/m) \* X(:,1)' \* (hx-y);**

And the gradient for the other thetas is:

**grad(2:end) = (1/m) \* (X(:,2:end)' \* (hx-y)) + (lambda/m) \* theta(2:end);**

Let’s discuss a few things here. First, Why do we have X(: , 1)’?

This does a few things for us:

* This isolates all the rows (observations) of the first column: ( : , 1). What is the first column? Just 1s, it’s a place holder so when we multiply by the thetas, the first theta becomes the intercept. We don’t see when the column of 1s is added because this is done in the mapFeature.m function.
* X is also then transposed. Normally, the first column of X would be a [m x 1] matrix, or [118 x 1]. When we transpose it, it becomes [1 x m], or [1 x 118]
  + This is because (hx-y) is a [m x 1] matrix, or [118 x 1]
  + Now when we do the [1 x m] \* [m x 1] matrix multiplication, we get a 1x1 vector, which is just a number. And in matrix multiplication, because we also do the addition portion, we don’t need ‘**sum()**’ in the code for the ∑ of the gradient.

A few things are similar for the second part of calculating the gradient.

We’ll discuss the **X(: , 2:end)'**

Here, we are isolating all the rows (observations) of X from the 2nd column to the end. That is a [m x n] matrix. Then we transpose it make a [n x m] matrix. Then we multiply that matrix by hx-y matrix, which is a [m x 1]

The [n x m] \* [m x 1] multiplication makes a [n x 1] vector of the gradients for each theta.