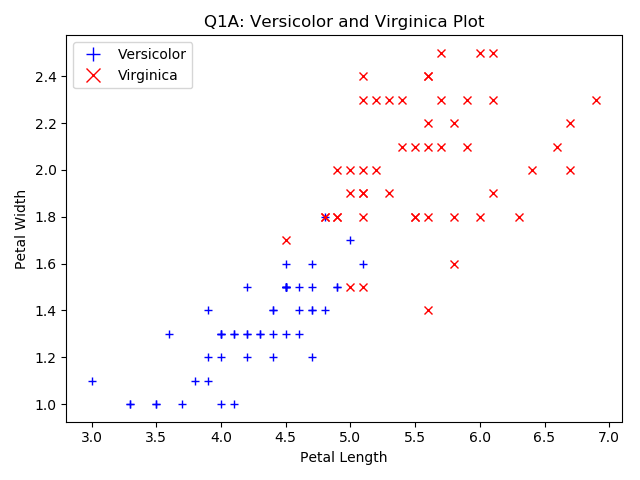
TungHo Lin

EECS391

PA2 WriteUp

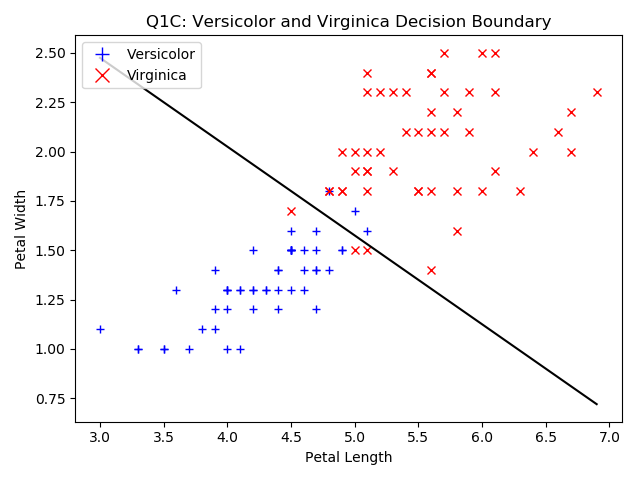
1  
The program is written in Python 3.5.7.

a)  
   
The function importData will load the irisdata.csv and return the data in a list and the legend, which is [sepal\_length, sepal\_width, petal\_length, petal\_width, species]  
The function plotData will use the given set of data and plot them according to their species. Different species is represented differently on the plot, as labelled in the legend.

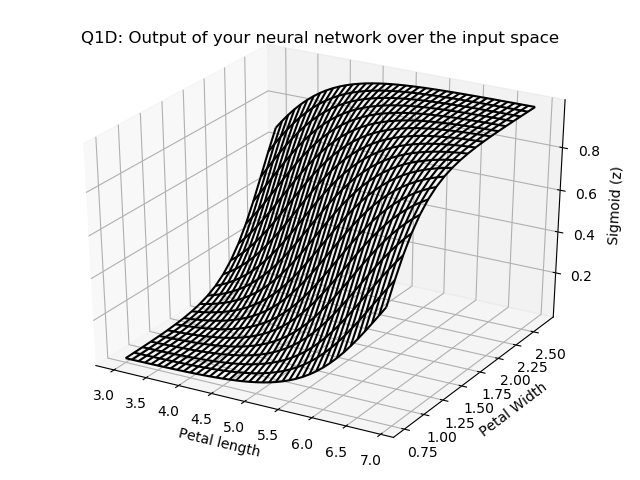
b)   
z = w1x1 + w2x2 + b  
where z is the output  
 w1 and w2 are the weight of x1 and x2  
 x1 is the petal length  
 x2 is the petal width  
 b is the bias

The output functions is:  
Φ (z) = 1/(1+e^-z), which is the Logistics function/Sigmoid function.  
This function gives the probability of an input belongs to Virginica (1).

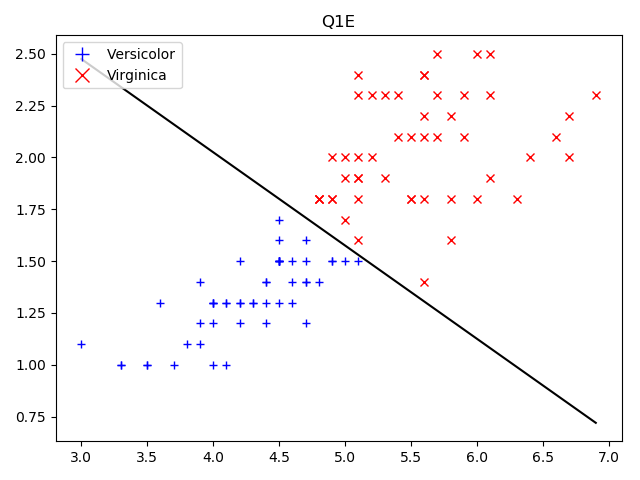
c)   
  
The decision boundary can be found by setting z to 0. So we get:  
w1x1 + w2x2 + b = 0  
To rearrange it into a line formula:  
x2 = -(w1x1 + b)/w2  
x2 = -(w1/w2) x1 + (-b/w2)  
We then do trial and error substituting weights into w1, w2 and b by hand in order to get the best line to roughly separate the two groups of data.  
I used w1 = 1.8, w2 = 4, b = -15.3 for the weight and bias values:  
x2 = -1.8/4 \* x1 + (-15.3/4)  
x2 = (-1.8/4) \* x1 - 3.825



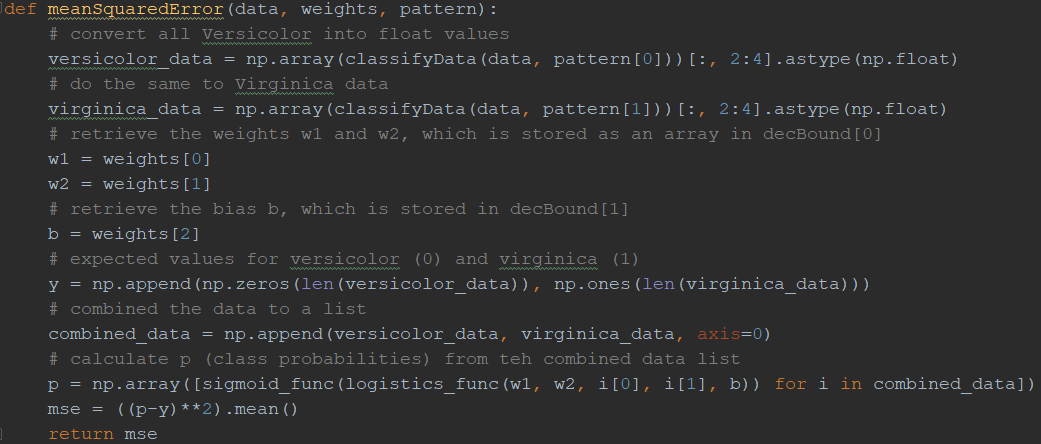
d)  
  
I originally used the ax.plot\_surface for the surface plot but it comes with a colormap. I have inspected the surface plot of Fig 18.17c in the book and it looks more like a wireframe. Therefore, I used ax.plot\_wireframe for the graph.   
As for the X and Y axes, I used the approximate range from the graph in c; so x ranges from 3 to 7 with an interval of 0.1, y ranges from 0.75 to 2.75 with an interval of 0.1.   
Z axis is defined as the sigmoid function of the output of the logistics function of each value of x and y. In other words:  
Z = 1/(1+e^-z), where z = w1x1 + w2x2 + b



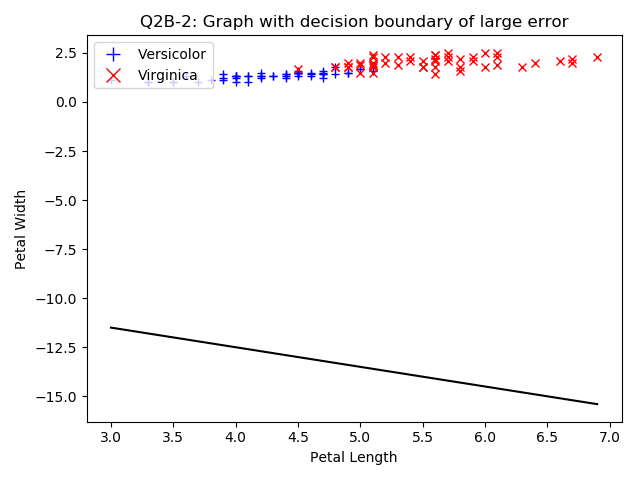
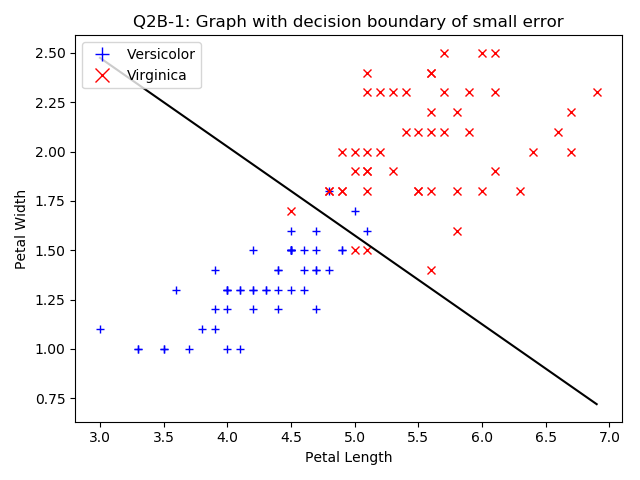
e)  
  
I used the decision boundary defined in part c to determine which species will each input be classified into. Therefore, as you can see in the graph produced, all the input above the decision boundary is classified as Virginica and those below the boundary is classified as Versicolor.



2)

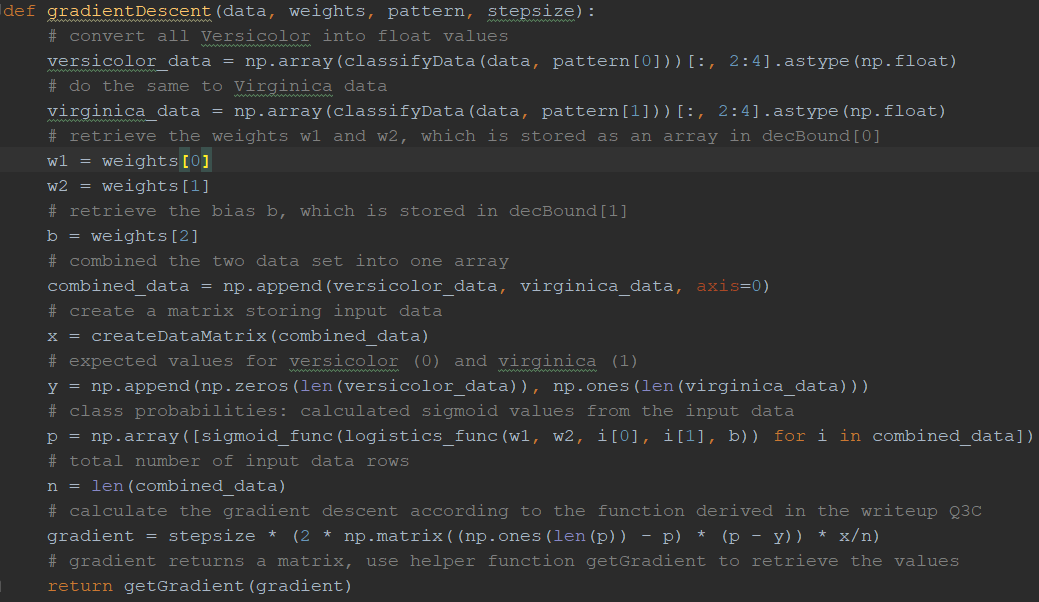
a)  
   
The function meanSquaredError() takes 3 input parameters, the data vectors, the weights and the pattern class.   
It first converts the data into float for calculations. I have passed in the weights (w1, w2) and the bias as an array of length 3 so I retrieved them for later sigmoid function calculation.  
I set the expected value for Versicolor and Virginica to be 0 and 1 respectively because Versicolor should produce a sigmoid output of 0 and Virginica should output a 1. I combined the expected values into y.   
Then I combined the versicolor\_data and virginica\_data into a list “combined\_data”.  
Then I computed the sigmoid outputs of the combined\_data and produced list p (the class probabilities)   
Then I subtracted the y from the p, squared it and calculate its mean (a.k.a MSE).  
I returned the MSE.

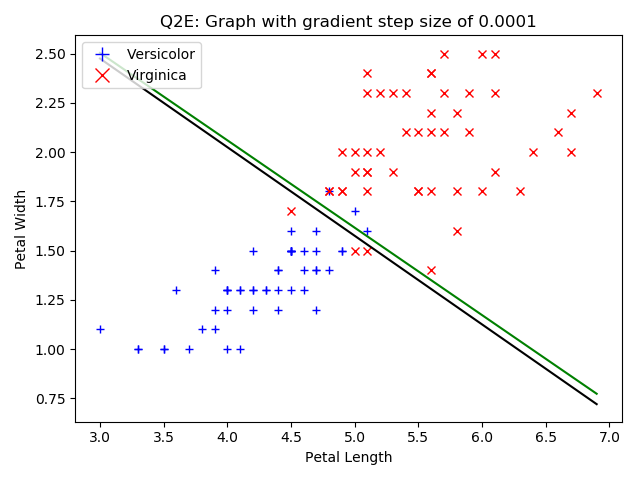
b)   
  
For a decision boundary with small errors, I use the same weights and bias from Question 1c. I set:  
m1 = 1.8, m2 = 4 and b = -15.3.   
Only a few (~3) Virginica is classified as Versicolor and vice versa. It gives a MSE of :  
|  
Graph:  
  
  
As for a decision boundary with large errors, I set:  
m1 = 1.8, m2 = 1.8 and b = +15.3.  
Now all Versicolor entries have been classified as Virginica and there it gives a relatively much larger MSE:  
  
Graph:



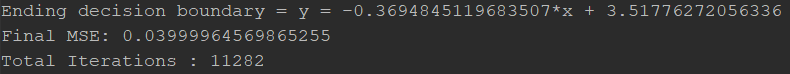
c)  
The class probabilities of an equation is defined earlier in MSE as p:  
p =   
where x1 and x2 are the petal length and x2 is the petal width   
 w0 is the bias, w1 and w2 are the weights of x1 and x2 respectively   
We defined the mean squared error to be:  
mse =   
mse =   
where y is the expected values, N is the total number of data rows  
The activation function is:  
To find the gradient with respects to the weights, we take the partial derivative of the mse wrt the w:  
where N is the total number of data rows, x is the input matrix, w is the weights and y is the expected values  
Since we have already calculated to be p, our final equation is:

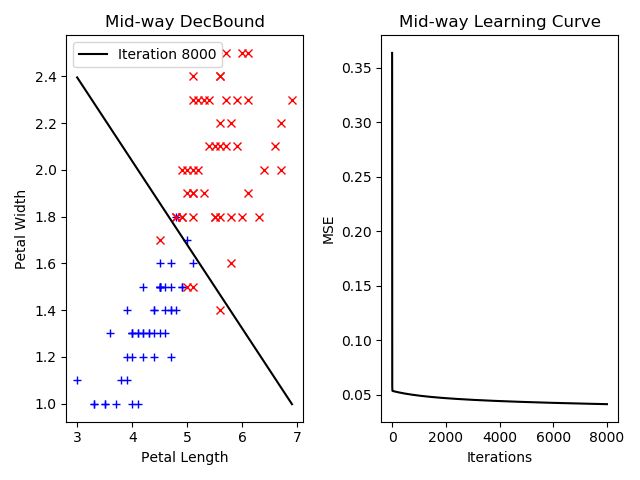
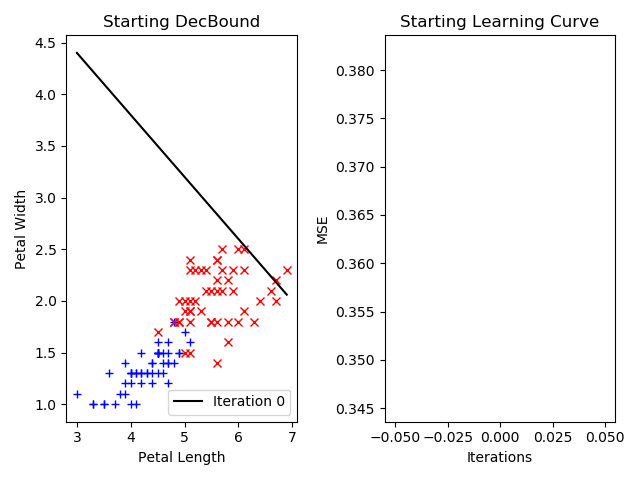
d)  
The gradient in scalar form:  
   
The gradient in vector form:

e)  
  
   
The function gradientDescent() takes four inputs (I added the stepsize in order to test different stepsize) and returns the steps for the weights and the bias.   
It first converts all the data into float values for easier calculations. Then it retrieves the weights and bias from the weights parameters, similar to the mse function. It then combined the two data list into one.  
I wrote another helper function that creates a matrix from the input data by appending 1 to each row of data (e.g. Each row of data looks like: [1, x1 value, x2 value] with different x1 and x2 values for each row). I also calculated y and p the same way I did in mse function. n is the length of the data.   
To calculate the gradient, I used the formula derived in part c) and multiplied it by the stepsize.   
Lastly, since the calculated gradient is still in matrix form, I wrote a helper method to retrieve the weight steps and the bias steps and return them.  
I picked the step size to be 0.1 because I have tried both 1 and 0.01 and the first one will not converge and the second one will takes about 30000 steps to converge, which is too much. 0.1 stepsize keeps the total number of iterations at around 10000 to 15000 when the Starting MSE is about 0.4 to 0.5

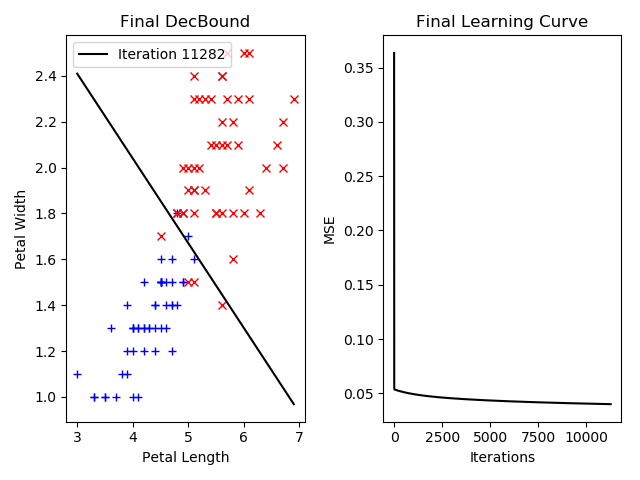


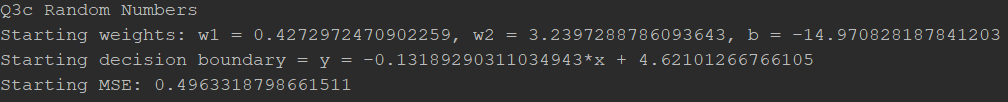
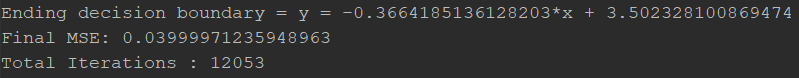
3)

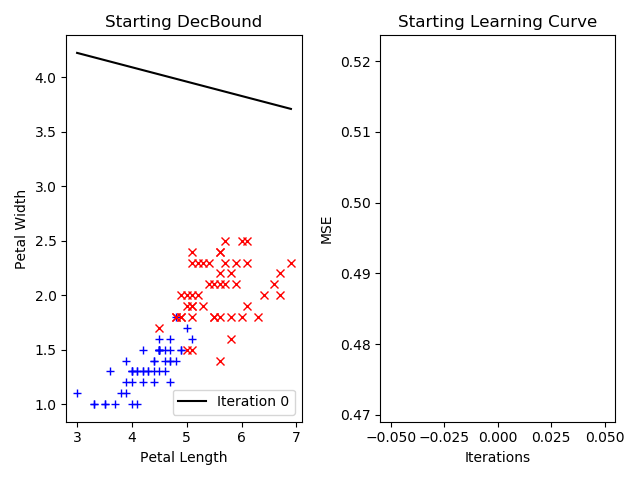
a,b)  
Both 3a and 3b can be achieved by running the program once. Since it shows learning and converging to a correct decision boundary and at the same time, plotting its learning process.  
Since the learning function is relatively long and has to be run multiple times. I wrote a helper function (plotGradientDescent()) for it. As for the explanation, I will separate it into three parts: setup, stepping and ending. They are labelled accordingly in the code.  
  
Setup:  
I first setup the data used for calculating gradient descent. I classified the input data into Versicolor and Virginica. I then set the range of x, assign the weights and bias input to w and b. I then calculate the y equation for the starting decision boundary. I also created a list for mse. All mses during stepping will be appended to the list to create the learning curve. I set the current iteration to be 0. The program will print out the starting decision boundary and the starting mse. I append the first mse to the list and start to plot the starting graphs.  
Plot Setup:  
I assigned the plotnumber and plot the two sets of data on the graph, then I add the starting decision boundary. As for the learning curve, I simply plot all of the mse in the mse list on the graph over the number of iterations. The starting decision boundary should be far from a “clear” boundary for the two classes. The starting learning curve should be empty/a dot (if you can see it) since there is only one mse stored.  
  
Stepping:  
I set the while loop condition (stopping criterion) to be if the last (most recent) mse is less than 0.04 because the decision boundary created after the most recent mse is less than 0.04 is accurate enough to distinguish between the two classes but also, it keeps the step size under 20000, which optimally should take around 10 seconds to compute.   
In the while loop, I get the weight and bias steps from calling the gradient descent function and update the weight and bias. I append the current mse onto the mse list and increment the number of iterations.  
Since we are required to plot a “middle” point to show the learning process, I will plot a graph of the current process when it reached 8000 iterations. I did the same plotting process as the plot setup.  
  
Ending:  
The system should end when the mse is less than 0.4. It will print out the ending decision boundary and mse, also the total iterations. It will utilize the plot setup again to plot the final decision boundary and the learning curve.   
  
I used w1 = 1.5, w2 = 2.5 and b = -15.5  
This produces a starting MSE and decision boundary of:  
  
  
The final results:  
  
Starting Plot:  
  
  
Mid-way Plot:



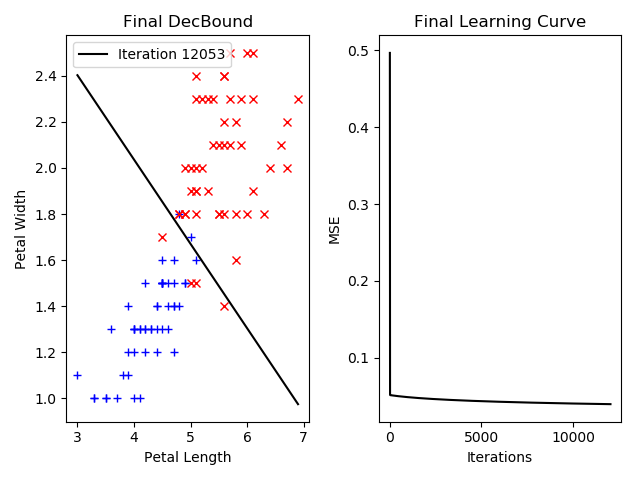
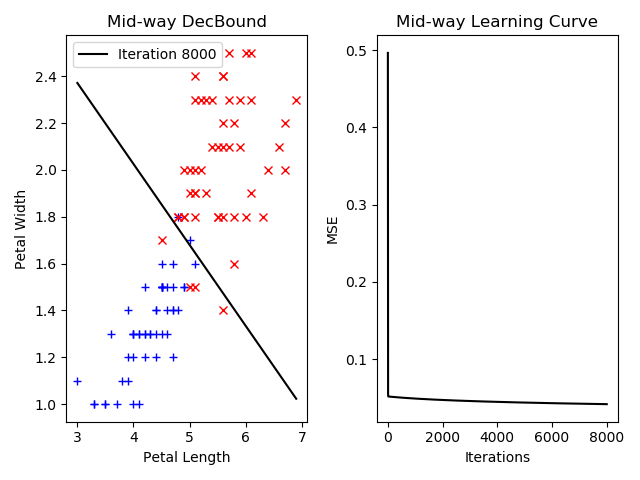
Final Plot:  
  
Note: the decision boundary of the mid-way and the final plots are similar since the mse difference between them are a lot smaller than the difference between the starting and the mid-way mse.

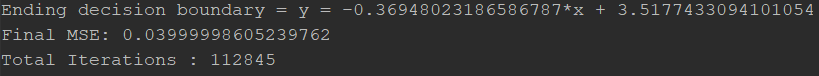


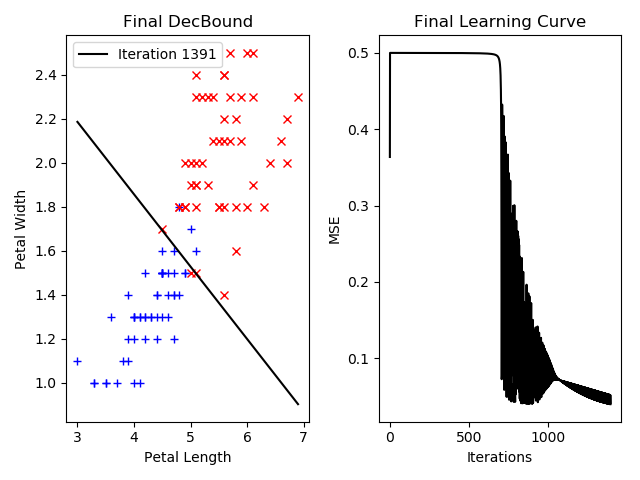
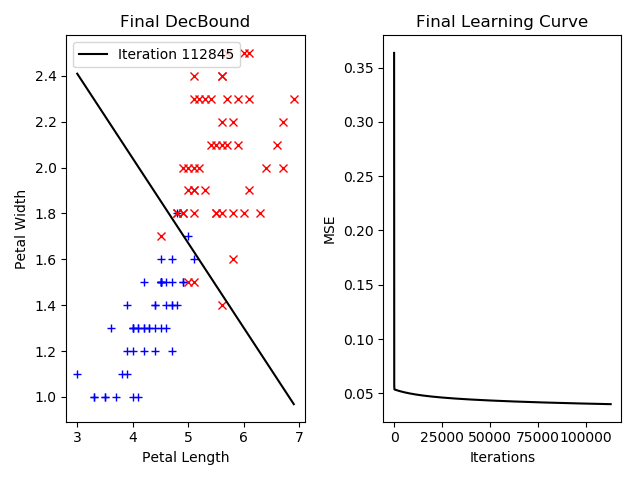
c)  
I randomize w1 to be a float number between 0 and 2.  
 w2 to be a float number between 2 and 4.  
 b to be a float number between -10 and -20.  
This will make sure the decision boundary is not too far away from the two sets of data.   
I pass in the value in the function plotGradientDescent(), and its process is the same as Q3a.  
The random values used in Q3c and the starting MSE and decision boundary:  
  
The results:  
  
Starting Plots:



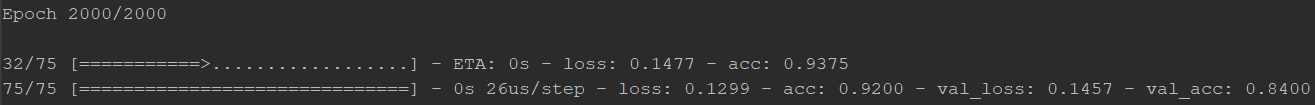
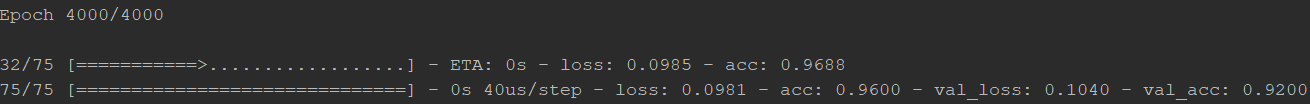
Mid-way Plots:  
  
Final Plots:

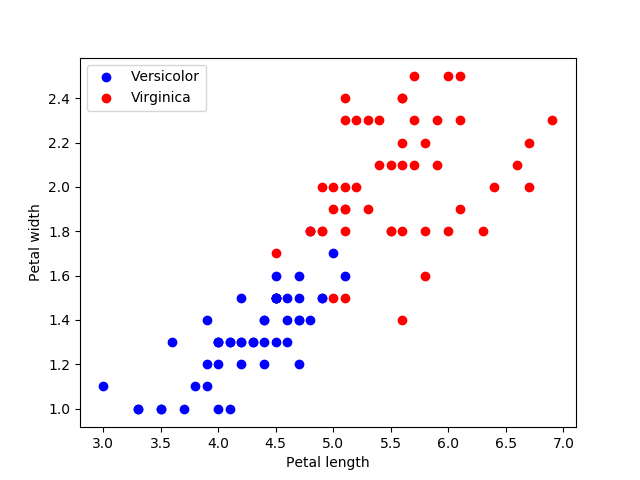


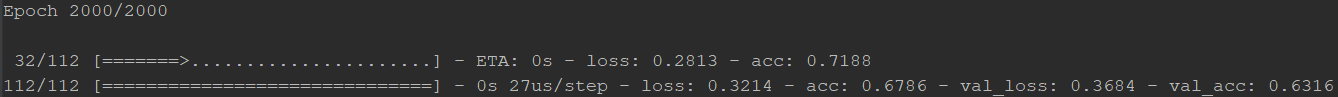
Q3d)  
I used a step size of 0.1 because it converges relatively faster than using 0.01.   
I tried using 0.01 at first for Q3A and the results took about 2 mins to show up running on a 5.0Ghz 8 core i9 processor. Here are the results:  
  
  
I tried using 1 as the stepsize and it oscillates between different sides of the minimum for some time before it finally converges. I have run Q3A with stepsize of 1. The graph shows oscillation:



e)  
I set the stopping criterion to be when the most recent (last in the mse list that I keep all the mse at all steps) mse is less than 0.04. First of all, when the mse hits 0.04, it gives a relatively accurate decision boundary that separates the two classes. Second of all, I have tried using mse < 0.03 in Q3A but it does not converge.  
Using 0.05 will not produce a decision boundary as accurate as using 0.04 as the stopping criterion.  
  
Extra Credit  
4

a)  
I completed the tutorial on Keras. Here are the plot and the output:  
  
  
From what I learned, each epoch consists of one full training cycle on the training set, a full pass over all of the training data (100 for part a). Loss is the MSE, a value we try to minimize during the training of the model. Accuracy is the current accuracy of each epoch. During training, the accuracy of a model is maximized. The difference in accuracy and validation accuracy means the model is underfitting, which means the variance is not being explained enough.   
For part a (2000 epochs), the accuracy is at 92% and the validation accuracy is at 84%. The loss is 0.1299. This is relatively accurate for classifying between two classes.  
I modified the function to accept an input of the number of epochs and tried doing 4000 epochs; and it gave even better results:  
  
The loss is even smaller and the accuracy is higher. I also noticed the difference between accuracy and validation accuracy is smaller, meaning the variance between the two classes are better defined.



b)   
Part b requires me to use all 3 classes and all 4 data dimensions (including sepal data). I modified the code a bit to add in all the data from irisdata.csv, label Setosa 0, Versicolor 1 and Virginica 2. I plotted them on the graph:  
  
Then I changed the model’s input\_dim to 4 because we are evaluating over 4 data dimensions. The results are:  
  
As you can see the accuracy is around 67.86% and that is not very accurate. Its loss is 0.3214, which is also relatively higher than part a). After examining the dataset, I can deduce that the reason for lower accuracy is that the sepal lengths and widths of the three classes do not variate that much; and that affected differentiation between the 3 classes when these data are included. Petal lengths and widths are more accurate in determining the 3 classes as we can see in the plot above. If we plot sepal lengths and widths of the three classes, we don’t get a plot that has 3 clearly separated sets of data (maybe Setosa has a clear variance in sepal data then Versicolor and Virginica but that is not enough to differentiate between Versicolor and Virginica). Here is the scatter plot:  
  
Therefore, we used only the petal data to differentiate between Versicolor and Virginica for the entire PA2 to get better results.  
  
Thank you so much and have a great and warm winter break!!!!!

