Ellis Saupe ems236

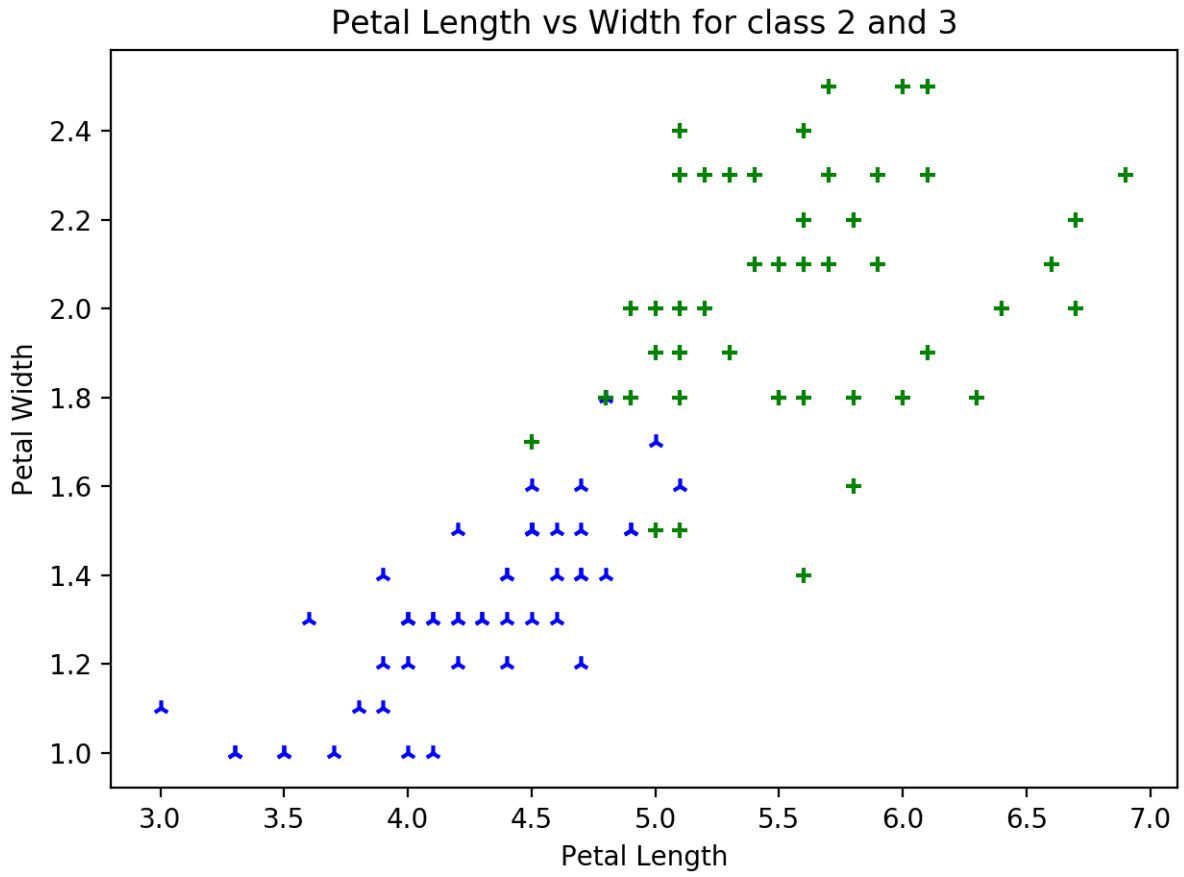
P2 Writeup

Code: irisNet.py

Part 1

a)

A graph of the imported data is shown below.



b)

Consider a logistic function σ(x). σ(x) is symmetric about 0 such that σ(0) = 0.5 and has a range of [0, 1]. Therefore, a decision boundary can be set at σ(x) = 0.5. A linear function of the data **x**

can be defined as a(**x**) = w_0 + $\mathbf{w}^T\mathbf{x}$, where **x** is a data vector, **w** is a weight vector and w_0 is a bias. Hence, a logistic regression of linear classification is given by σ(a(**x**)). σ(a(**x**)) can be used as a binary classifier given data vectors **x** such that **x** is belongs to c1 for σ(a(**x**)) > 0.5 and **x** does not belong to c1 for σ(a(**x**)) < 0.5. For the binary range of "class2" or "class3", **x** can be assigned to class 2 for σ(a(**x**)) < 0.5 and **x** can be assigned to class 3 for σ(a(**x**)) > 0.5. σ(a(**x**)) = 0.5 is completely ambiguous, meaning each class is equally likely given the pattern. On the space of reals with a finite dataset, this is an edge case so my classifier classifies 0.5 as class3 for convenience.

Code implementing σ(a(**x**)) and the resulting classification function is shown below.

```
#a logistic function that works with arrays
def logistic(x):
    return (1/(1+np.exp(-1 * x)))

#expects weight to include some w0, the bias.
def WeightByData(weight, data):
    return weight[0] + np.inner(weight[1:], data)

#classify a single data vector x
def classify(weight, x):
    if logistic(WeightByData(weight, x)) >= 0.5:
        return 1
    else:
        return 0
```

c)

The decision boundary of this classifier is set at 0.5. That is

$$0.5 = \sigma\big(\mathrm{a}(\mathbf{x})\big) = \frac{1}{1 + e^{-a(x)}}$$

$$0.5 + 0.5e^{a(x)} = 1$$

$e^{a(\boldsymbol{x})} = 1$

$a(\boldsymbol{x}) = 0$

$w_0 + \boldsymbol{w}^T \boldsymbol{x} = 0$

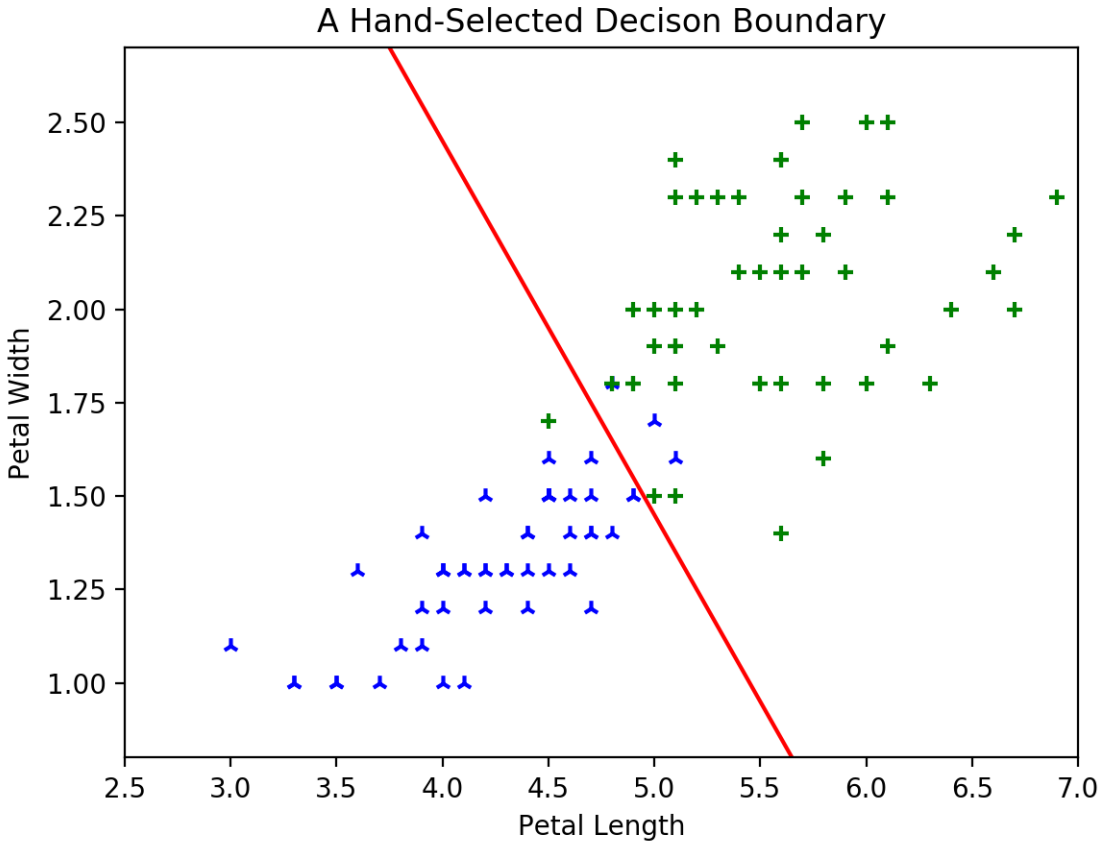
In this case **w, x** are 2 dimensional

$w_0 + w_1 x_1 + w_2 x_2 = 0$

To plot a line in 2d space

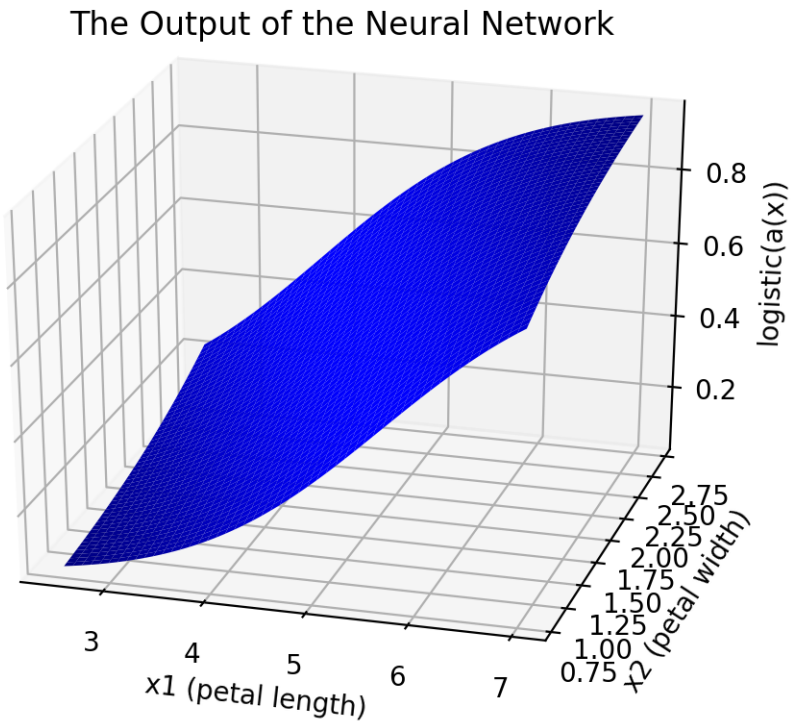
$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$

Using w_0 = -6.45 and **w** = < 1., 1.>, the decision boundary that reasonably divides class 1 and 2 is shown below.



d)

The output of the network appears as a sigmoid in 3d space, because **x** is a 2 dimensional vector. Graphing z = σ(a(**x**)) is shown below.



e)

For w_0 = -6.45 and **w** = < 1., 1.>,

Consider the class 2 iris with petal dimensions (3.5, 1).

σ(a(3.5, 1)) = σ(-6.45 + 3.5*1 + 1 * 1)

= σ(-1.95) = 0.1246. The classifier classifies this as unambiguously class2, as 0.1246 is relative far from the boundary of 0.5 and close to the class definition of 0.

Consider the class 2 iris with petal dimensions (4.9, 1.5).

σ(a(4.9, 1.5)) = σ(-6.45 + 4.9*1 + 1.5 * 1)

= σ(-0.05) = 0.4875. From the classifier, this is not very clearly a class2 iris. This is an ambiguous result as the output is close to the boundary of 0.5 and matches neither class definition of 1 or 0 very well.

Consider the class 3 iris with petal dimensions (4.5, 1.7)

σ(a(4.5, 1.7)) = σ(-6.45 + 4.5*1 + 1.7 * 1)

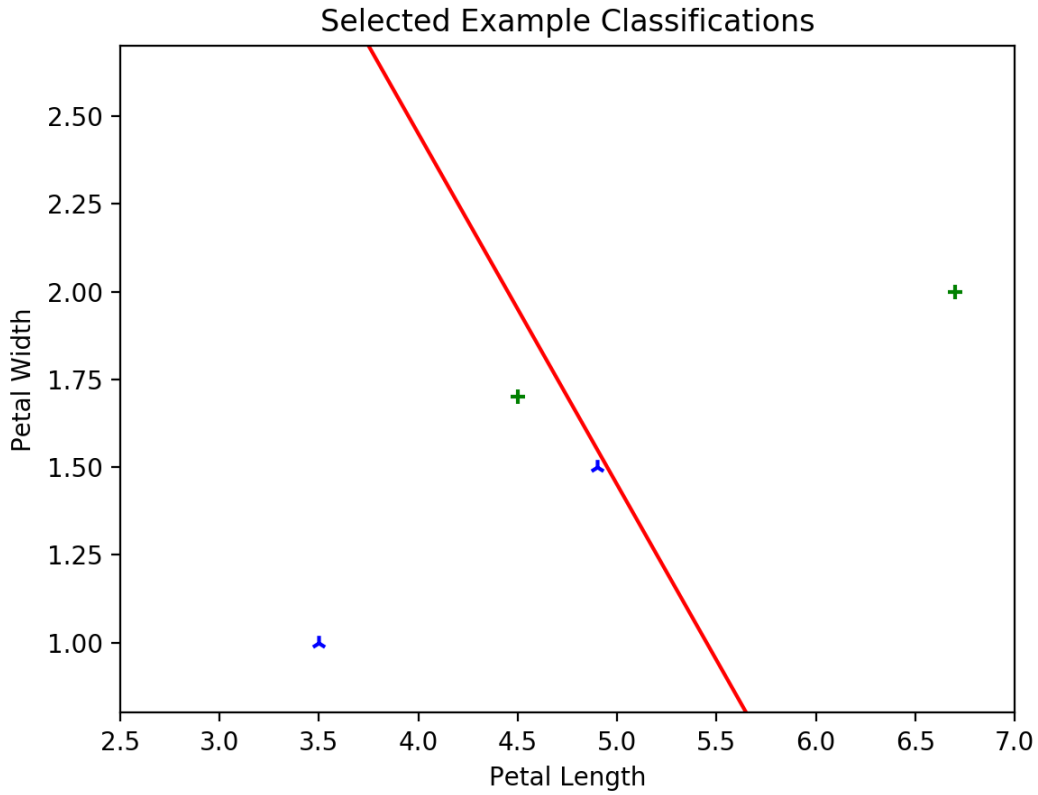
= σ(-0.25) = 0.4378. Although this is a class3 iris, it is classified as class 2. This is an ambiguous result as the output is close to the boundary of 0.5 and matches neither a 1 or 0 very well.

Consider the class 3 iris with petal dimensions (6.7, 2)

σ(a(6.7, 2)) = σ(-6.45 + 6.7*1 + 2 * 1)

= σ(2.25) = 0.9047. This is unambiguously a classified as class 3, as 0.9 is relatively close to the class definition of 1 and far from the boundary of 0.5.

When plotted against the decision boundary, the more ambiguous results are much closer than the non-ambiguous results as shown below.



Part 2

a)

The mean squared error is defined by the function

$$E = \frac{1}{2n}\sum_{i=0}^{n}\left(\sigma\left(\mathrm{a}(\mathbf{x}_i)\right) - \mathrm{c}_i\right)^2$$

Where c_i is binary: 0 for item i belongs to class2, 1 for item i belongs to class3.

and $a(\mathbf{x}_i)$ is the weighted linear function $a(\mathbf{x}) = w_0 + \mathbf{w}^T\mathbf{x}$.

A function to produce this error is shown below. Patterns is the array of **x** values, the patterns of the data. Weight is the parameters of the neural network, in this case w_0, w_1, and w_2. Data is an array of binary C values containing the actual class data of each data point **x**.

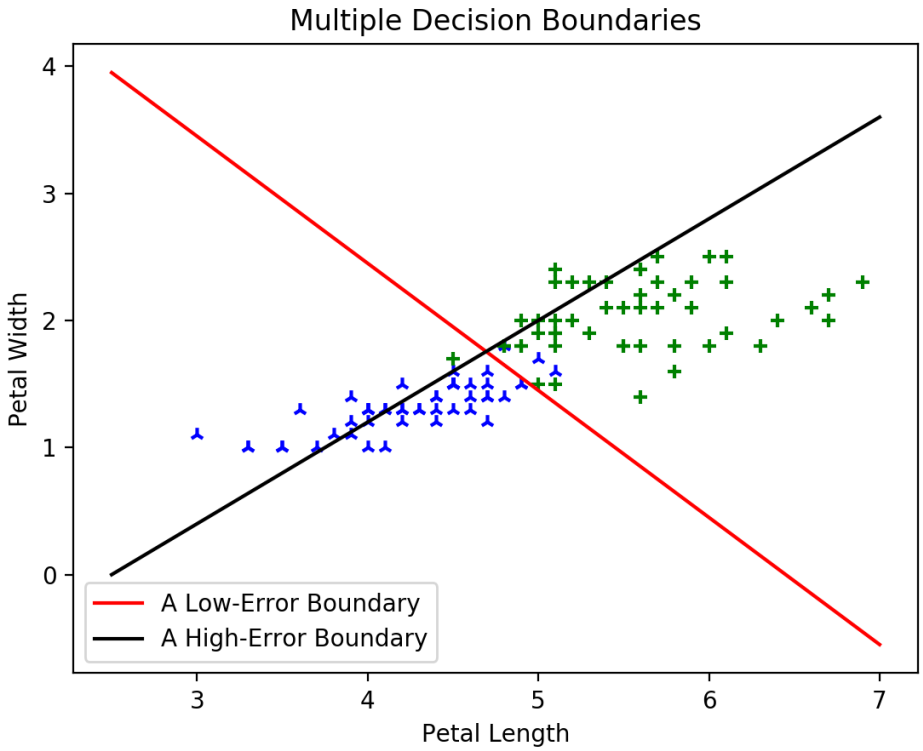
```
def meanSquareError(data, weight, patterns):
    # E = 1/2n(sum(y(xi, w) - pattern)^2)
    output = logistic(WeightByData(weight, patterns))
    #the error for each data point
    stepError = output - data
    return 0.5 * np.sum(stepError ** 2) / len(data)
```

logistic() and WeightByData() are shown in 1b). They correspond to σ(x) and a(**x**) respectively.

b)

Using weights $w_{1,0}$ = -6.45 and $\mathbf{w_1}$ = < 1., 1.>, $w_{2,0}$ = 2 and $\mathbf{w_2}$ = < -0.8, 1.>

The mean squared error using $\mathbf{w_1}$ = 0.0493 as calculated in my function from 2a). The mean squared error using $\mathbf{w_2}$ = 0.1501 using my function. A graph of each boundary is shown below with the boundary from $\mathbf{w_1}$ in red.



c and d)

Using the objective function as the total squared error:

$$E = \frac{1}{2}\sum_{i=1}^{n}\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right)^2 \; for \; \mathrm{a}(\mathbf{x_i}) = \; w_0 \; + \; \mathbf{w^T x}$$

$$Consider \frac{\partial E}{\partial \boldsymbol{W}} = \langle \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m} \rangle \; for \; m - dimensional \; data$$

$$Consider \; \frac{\partial E}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \sum_{i=1}^{n}\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right) * \frac{\partial E}{\partial w_j}\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right) \; by \; the \; chain \; rule$$

$$= \sum_{i=1}^{n}\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right) * \sigma\left(\mathrm{a}(\mathbf{x_i})\right) * \left(1 - \sigma\left(\mathrm{a}(\mathbf{x_i})\right)\right) * \frac{\partial E}{\partial w_j}\mathrm{a}(\mathbf{x_i}) \; by \; the \; chain \; rule$$

$$and \; \frac{d}{dx}\sigma(\mathrm{x}) = \sigma(\mathrm{x})(1 - \sigma(\mathrm{x}))$$

$$= \sum_{i=1}^{n}\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right) * \sigma\left(\mathrm{a}(\mathbf{x_i})\right) * \left(1 - \sigma\left(\mathrm{a}(\mathbf{x_i})\right)\right) * \frac{\partial E}{\partial w_j}(w_0 \; + \; w_1 x_{i1} + \cdots + w_0 x_{im})$$

$$\frac{\partial E}{\partial w_j} = \sum_{i=1}^{n}\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right) * \sigma\left(\mathrm{a}(\mathbf{x_i})\right) * \left(1 - \sigma\left(\mathrm{a}(\mathbf{x_i})\right)\right) * \begin{cases} 1 \; for \; j = 0 \\ x_{ij} for \; j \in [1, m] \end{cases}$$

This is the scalar form of the partial derivative of the error with respect to any dimension j or the bias, j=0. The vector form of the gradient with respect to weights and bias **w** is given by $\langle \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m} \rangle$.

Every term before the x_{ij} is independent of the dimension j, so the vector form can be expressed as

$$\frac{\partial E}{\partial \boldsymbol{W}} = \sum_{i=1}^{m}\left(\left(\sigma\left(\mathrm{a}(\mathbf{x_i})\right) - \mathrm{c}_i\right) * \sigma\left(\mathrm{a}(\mathbf{x_i})\right) * \left(1 - \sigma\left(\mathrm{a}(\mathbf{x_i})\right)\right)\right) \langle 1 \mid \mathbf{x_i} \rangle$$

$Where\ \langle 1 \mid \mathbf{x_i} \rangle\ is\ the\ augmentation\ of\ 1\ and\ \mathbf{x_i} = \langle 1, x_{i1}, x_{i2}, \ldots, x_{im} \rangle.$

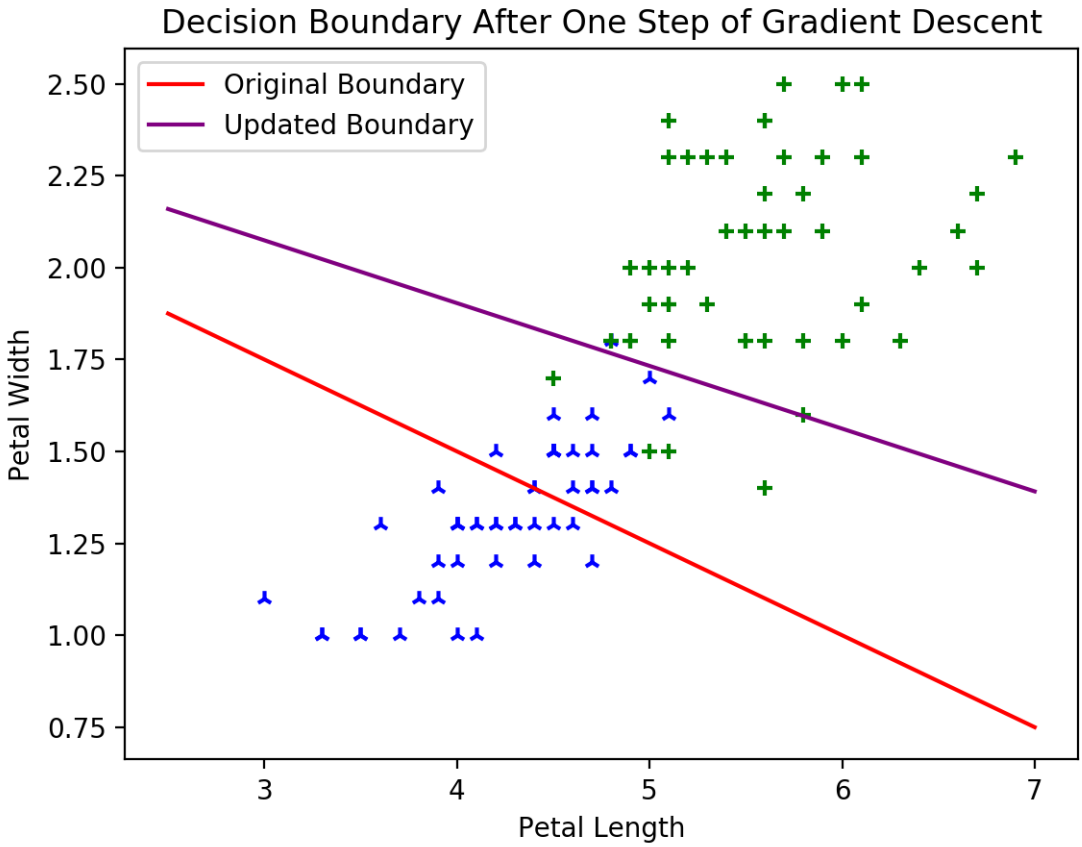
This could also be expressed in the same way as the lecture notes by adding a dimension x_{i0} to the data that is always equal to one and to avoid writing the augmentation <1| **x**>. Each is mathematically identical.

e)

The condensed code to compute this gradient function is shown below.

```
def gradientError(data, weight, patterns):
    #logistic(wT x + w0)
    sigmoidData = logistic(WeightByData(weight, patterns))
    #logistic(wT x + w0) - c
    error = sigmoidData - data
    coefficient = error * sigmoidData * (1 - sigmoidData)
    #make the augmented pattern vector
    augPatterns = np.ones((len(patterns), len(patterns[0]) + 1))
    augPatterns[:, 1:] = patterns
    sumTerm = np.zeros((len(patterns), len(patterns[0]) + 1))
    for i in range(len(coefficient)):
        sumTerm[i] = augPatterns[i] * coefficient[i]
    #sum values
    return np.sum(sumTerm, axis = 0)
```

The decision boundary after a single step using this gradient function and weight **w** = < -5, 0.5, 2.> and step size ε = 0.01 is shown below. The original boundary is in red, the new boundary is in purple. The original mean-squared error is 6.3614 and the error after one step of gradient descent is 5.28. As shown by both the error decreasing on the lower number of misclassified patterns, the new decision boundary is clearly an improvement.



Part 3

a)

The code for gradient descent an initial weight, class data, and pattern vectors is shown below. The descent has a limit on max iterations set to 10000, but the method usually reaches stopping conditions well before that. The stopping condition for this descent is when the 2-norm of the gradient is below a tolerance set to 0.005. This guarantees convergence to at least a local minimum if not a global minimum. The quality of this stopping condition will be discussed in part e).

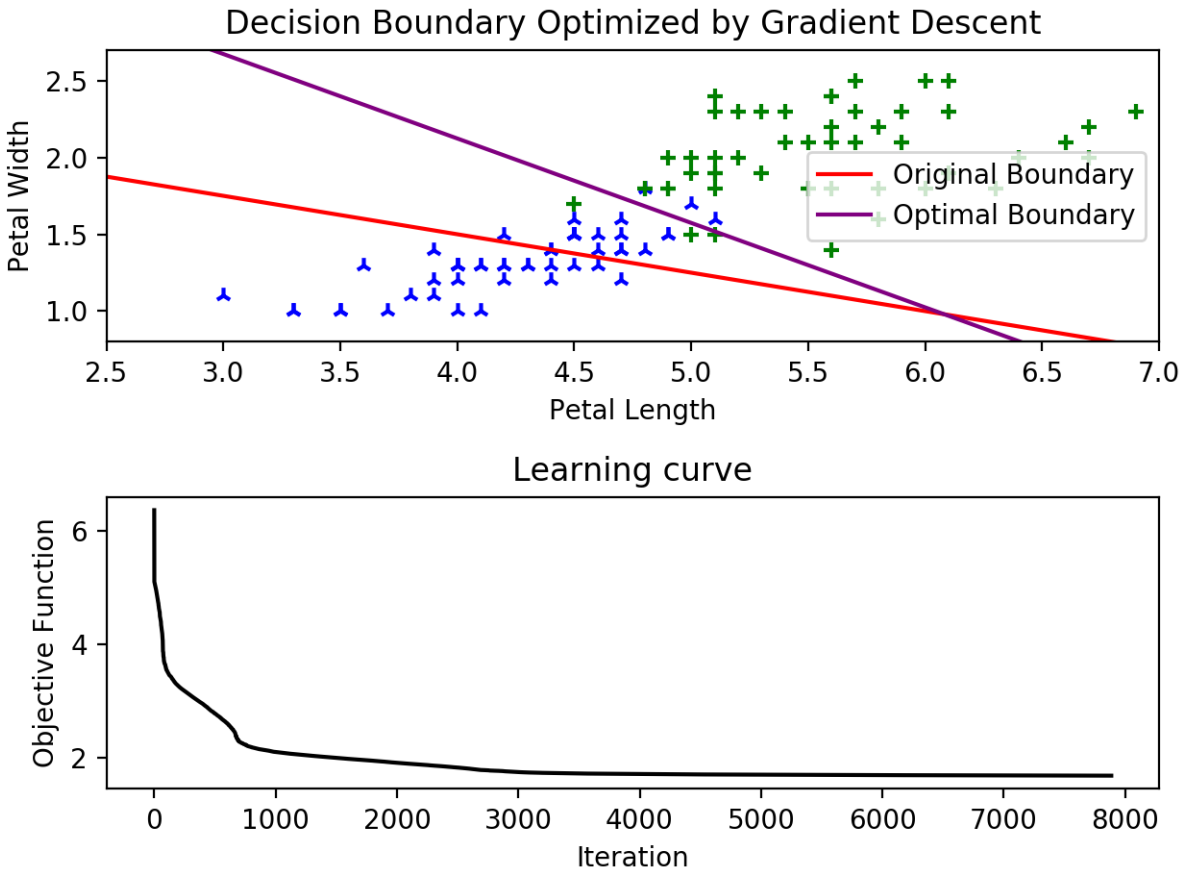
```
while la.norm(currentGrad) > tolerance and iteration < 100000:
        #finalWeight will be the most recent weight with an improvement.
        if improvement > 0:
            finalWeight = currentWeight
        #update weight using backtracking search
        currentWeight = nextWeight(alpha, beta, currentError, currentGrad, currentWeight, petalData, patterns)
        #find next error
        nextError = totSquareError(petalData, currentWeight, patterns)
        improvement = currentError - nextError
        currentError = nextError
        currentGrad = gradientError(petalData, currentWeight, patterns)
        iteration = iteration + 1

        #store arrays of values to be returned
        weights = np.append(weights, np.array([currentWeight]), axis=0)
        iterationList = np.append(iterationList, iteration)
        objectiveFunc = np.append(objectiveFunc, currentError)
```

nextWeight() returns the weight adjusted by a step size described in part d). Returns **w** - ε Δ**w** for a step size ε; it is the update function for the weight vector. There is additional setup and teardown to initialize and finalize return parameters, but this is not essential to the logic of the gradient descent

b)

Using the above and the weight used in 2e), **w** = < -5, 0.5, 2.>, the optimized decision boundary and learning curve are shown below. As shown in the graph, the descent over the first few iterations improves the error very quickly, then the decrease in error slowly levels off for a few

hundred iterations until the 2-norm of the gradient is below the tolerance.



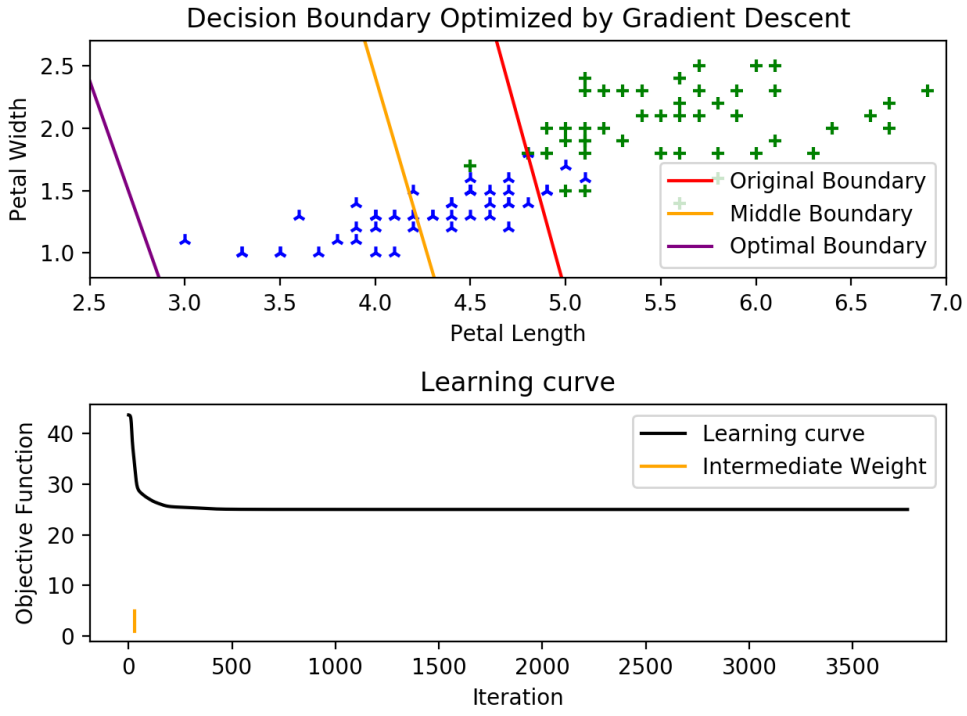
The original error is 6.3614 and the error after the gradient descent is 1.69, with a new weight of <-31.006, 3.943, 7.176>

c)

Using the method from a, I implemented gradient descent starting with a random weight. To select a random weight, I chose a random point that would be displayed on the graph above, generated a random slope, and derived the weights that would result a decision boundary matching that line. I selected a representative intermediate weight as the weight at which the objective function was 95% converged to the final value of the objective function,

that is for initial error e and final error f, middle is the weight at which e(middle) = e + 0.95(e-f). The solution converges to this middle value very quickly. 95% of the improvement occurs in the first few 100 or iterations and the next few thousand reduce the error very slightly.

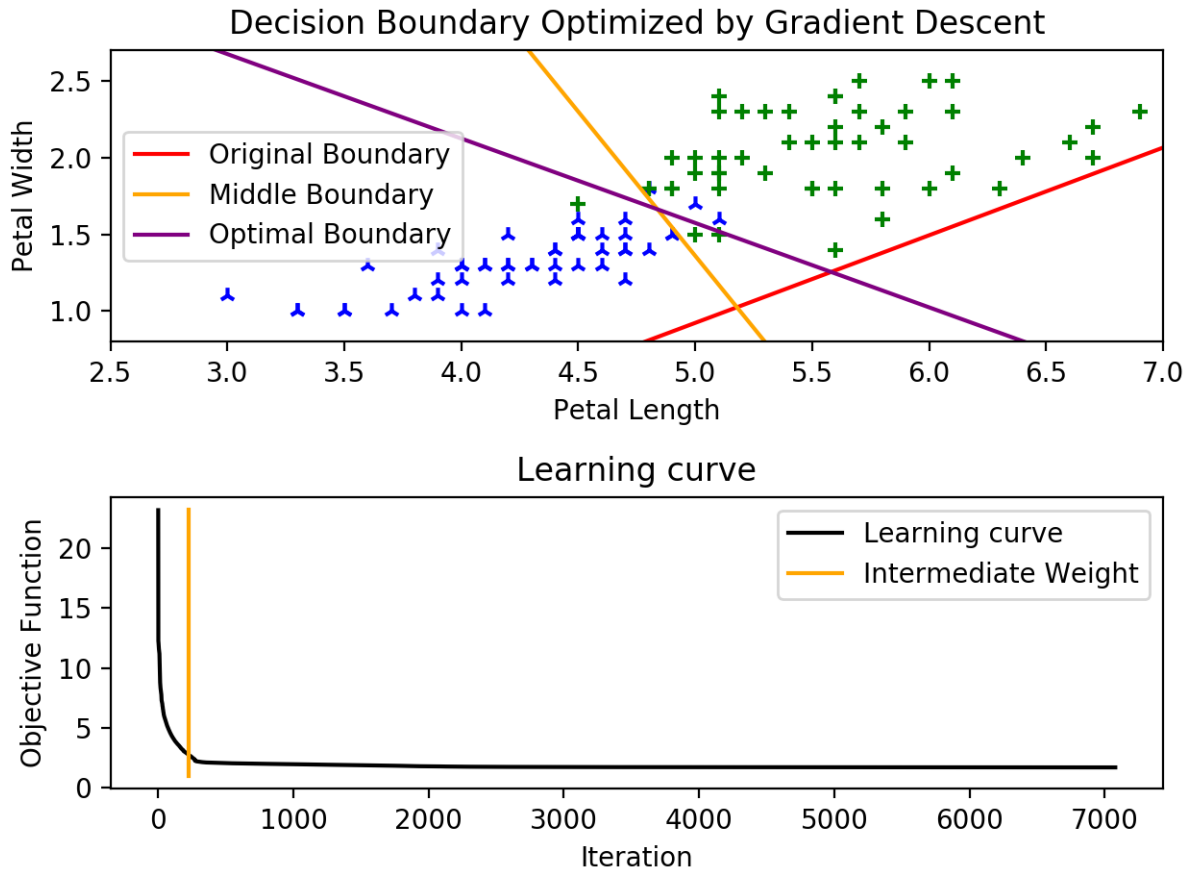
Before beginning the gradient descent process on a random starting weight, the method checks if the objective function is above 25 for the initial weight and if that would be improved by multiplying the weights by -1. The prevents the situation where the initial weights misclassify nearly every point, and a non-ideal local minimum is found by classifying every pattern as the same class. An example of such a case is shown here.



The random original decision boundary looks pretty nice, but it classifies everything completely wrong; everything on the left is classified as class3 (actual classes shown in green) while everything on the right is classified as class2 (actual classes shown in blue). A local minimum of the objective function exists where every data point is assigned the same class, and the weight

converges to a very poor classifier.

A functioning example of a random starting weight is shown below.



In this example, the initial **w** is < -13.036, 3.856, -6.765> with an objective function value of 23.17. In 223 iterations, the function reaches the intermediate value of **w** = < -15.510, 2.709, 1.444> and an objective function value of 2.76. The stopping point of a $||\text{gradient}||_2$ below 0.005 was reached in 7080 iterations, where **w** = < -30.998, 3.941, 7.175> and the objective function is 1.69.

d)

I chose the step size of my gradient descent using linear backtracking and the Armijo rule. I do not have a sufficient background in analysis and optimization to fully justify the Armijo rule.

The algorithm selects a relatively large step size α_0

$\alpha_{i+1} = \beta\alpha_i \; for \; some \; 0 < \beta < 1$

$Return \; the \; maximum \; \alpha_i \; that \; satisfies \; the \; Armijo \; rule. That \; is$

$$E\big(weight - \alpha_i \Delta E(weight)\big) \leq E(weight) - \frac{1}{2}\alpha_i \|E(weight)\|_2^2$$

The Armijo condition prevents overstepping and becoming further from a minimum, while the initially high, incrementally decreasing α produces a step size that is reasonably large. This makes my descent converge to the stopping condition much faster. Using a constant step ε = 0.01, the function always converged, but could take around 80000 iterations. That same descent using linear backtracking with α=1, β=0.5 converged in around 7500 iterations.

The code for determining the updated weight is shown below

```
def nextWeight(alpha, beta, error, gradient, weight, petalData, patterns):
    #backtracking line search:
    stepSize = alpha
    while totSquareError(petalData, weight - (stepSize * gradient), patterns) > error - (0.5 * stepSize * la.norm(gradient) ** 2):
        stepSize = stepSize * beta
    return weight - (stepSize * gradient)
```

e)

My gradient descent ends when $\|\Delta E(weight)\|_2 \leq tolerance$. In all of my functions, tolerance was set to 0.005. This value is relatively arbitrary. I tried a few different values and this one had good balance between time to convergence and overall error at convergence. The norm of the gradient being below some tolerance is a good stopping condition for gradient descent because the function is approaching a local/global minimum. At a minimum, the gradient and

its 2-norm are 0. Approaching some small tolerance of 0 signifies that the function is essentially minimized and further iteration will not result in a significant change. The gradient would also be zero at a local minimum or saddle point, but this did not significantly affect my data. A simple solution to this would be selecting the best converged point from a few random starts or some form of simulated annealing.

Part 4 Extra Credit

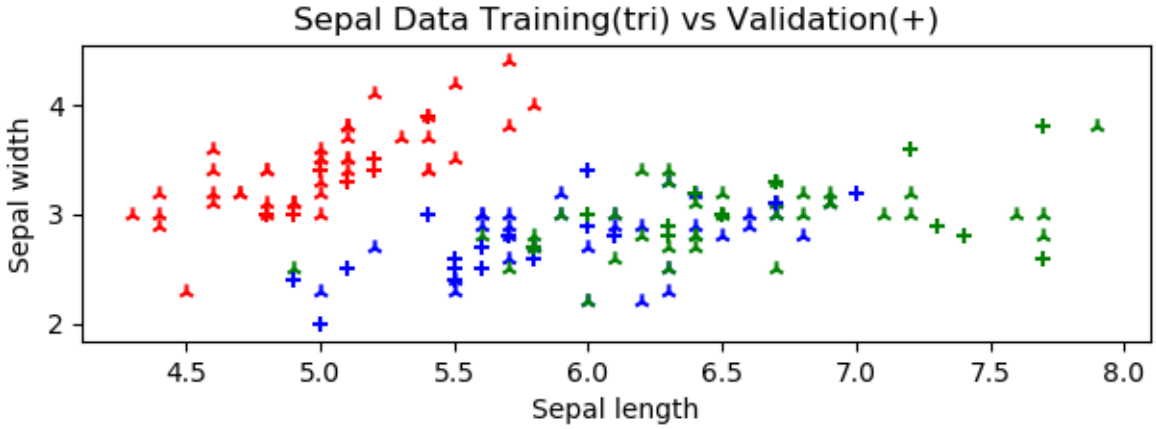
File: em236extraCred.py

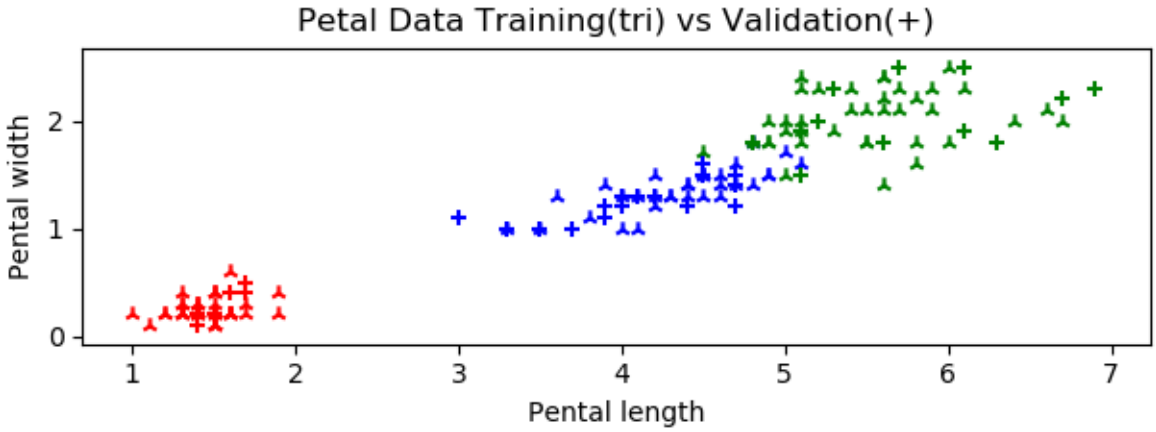
a)

Using Keras with a sequential model, sigmoid activation, rmsprop optimization, and mean squared error loss, the neural network generally had around 92-98% accuracy for both the random training and validation samples after 2000 epochs.

b)

The generalize the network for classifying all 3 sets of iris data, the output for a pattern **x** must be changed to a one hot vector with <1, 0, 0> for class1 etc. Little needs to change to account for 4-dimensional data because this simply adds 2 more dimensions to **w** and 2 more terms to the inner product <**w**, **x**>. Using the sequential model, sigmoid activation, rmsprop optimization, and mean squared error loss, this network was also around 95% accurate on both the random training a validation samples after 2000 epochs. The graph below shows the training and validation data. Training the network on this data resulted in a training accuracy of 94.6% and a validation accuracy of 97.4%.


Sepal Data Training(tri) vs Validation(+)
Sepal width
Sepal length
4
3
2
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Petal Data Training(tri) vs Validation(+)
Pental width
Pental length
2
1
0
1
2
3
4
5
6
7