Assignment 2 Writeup

December 9, 2017

Contents

1 Exercise 1. Linear decision boundaries

1. Inspect irisdata.csv which is provided with the assignment file on Canvas. Write a program that loads the iris data set and plots the 2nd and 3rd iris classes, similar to the plots shown in lecture on neural networks (i.e. using the petal width and length dimensions).

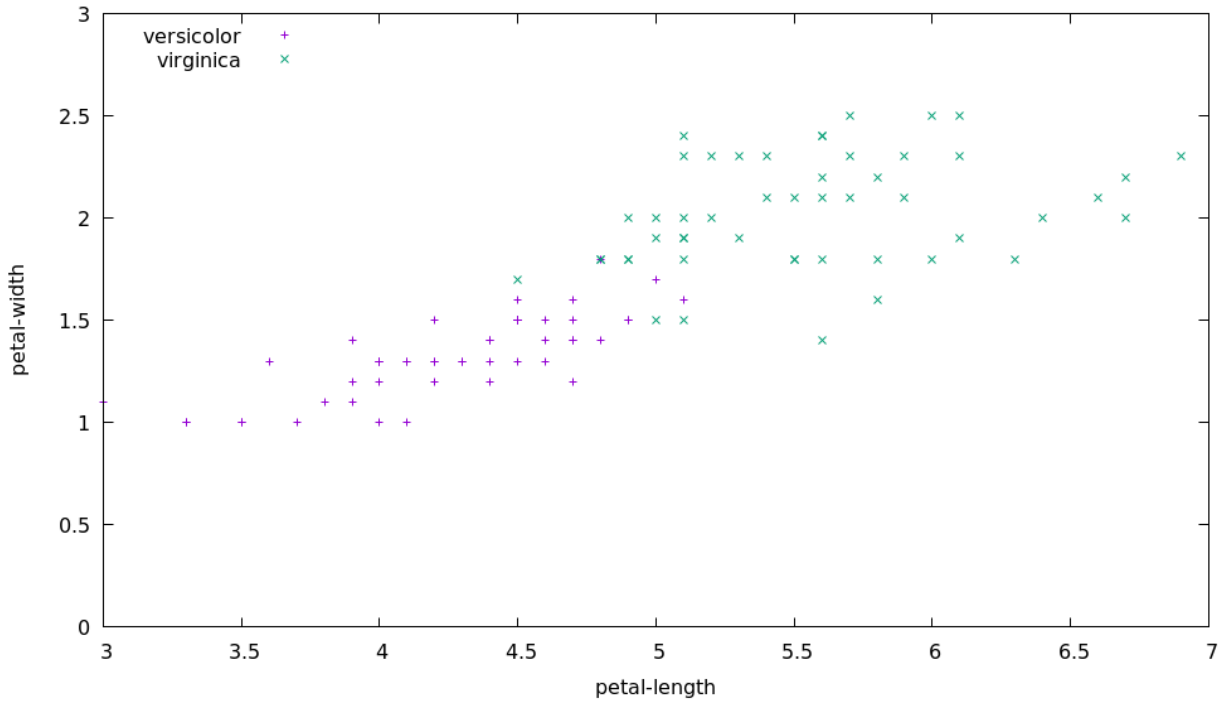
   - Loading the program

     ```
     (defparameter *csv* (cl-csv:read-csv #P"irisdata.csv"))
     ```

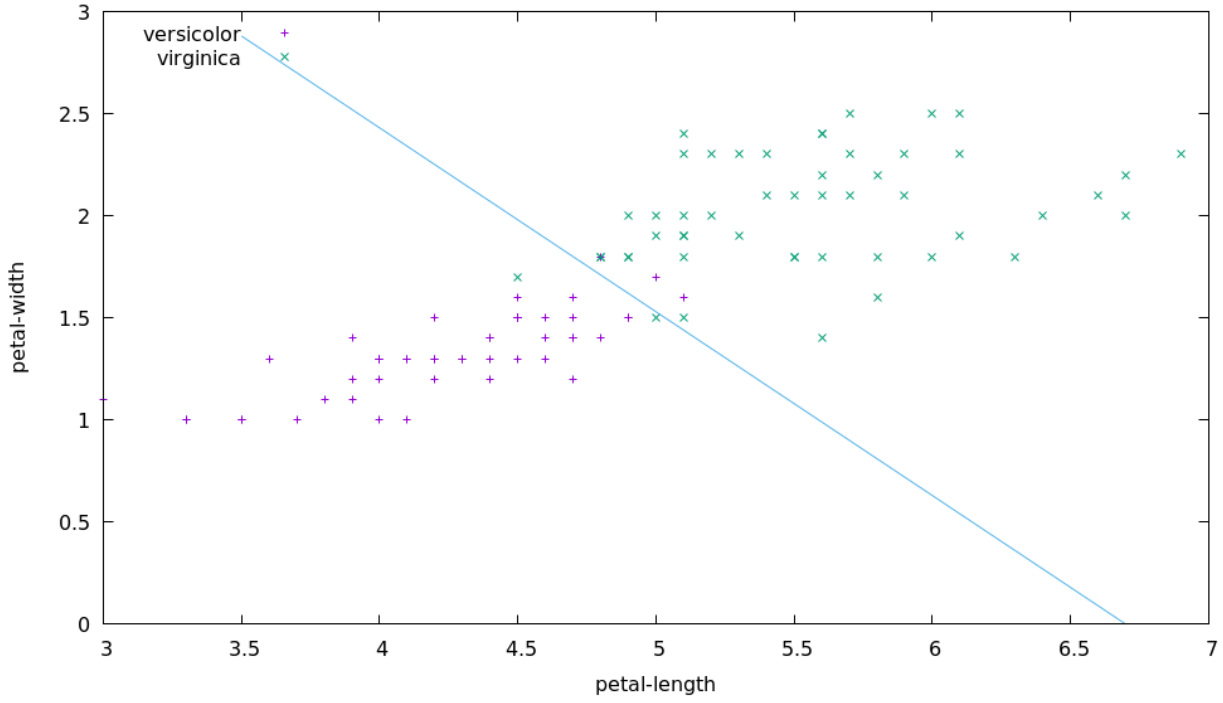
   - Plotting the petal width and length
     - Note that I won't be including graph-1-clos in here, as it's a function that calls a macro that generates the proper graph.
     - Really the magic in this code is corrected, that sorts and types the data so it can conveniently be graphed

       ```
       (flet ((graph-1-clos (str &optional x)
                (graph-1 str (cdr (corrected *csv*)) x)))
         (defun answer-1.a ()
           (graph-1-clos "images/1-a.png")))
       ```

   - The Plot



2. Write a function that defines and plots a linear decision boundary (i.e. an arbitrary line) overlaid on the iris data. Choose a boundary (by hand) that roughly separates the two classes.
   - The arbitrary line here is the `(draw-boundary (x) (* .9 (+ -.3 (abs (- 7 x)))))`
   - This is mapped over the range from 3.5 to 7
   - And as one can see it roughly separates the two classes

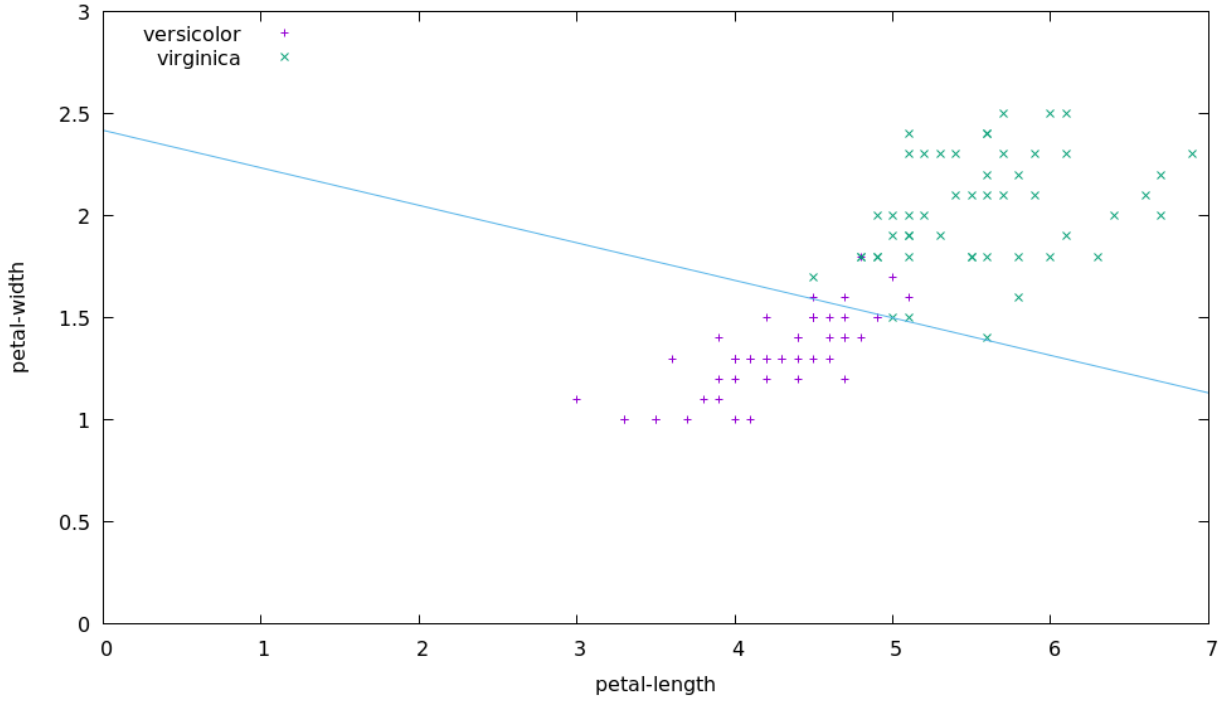


3. Define a simple threshold classifier using the above decision boundary. Illustrate the output of the classifier using examples from each of the two classes.

- The threshold is defined in the code below, it has an optional max argument, since the data might not be linearly separable, and thus would take too long to compute
- note that I make a second function `linear-decision`, that turns the corrected iris data into a list of data-vectors and class-vectors

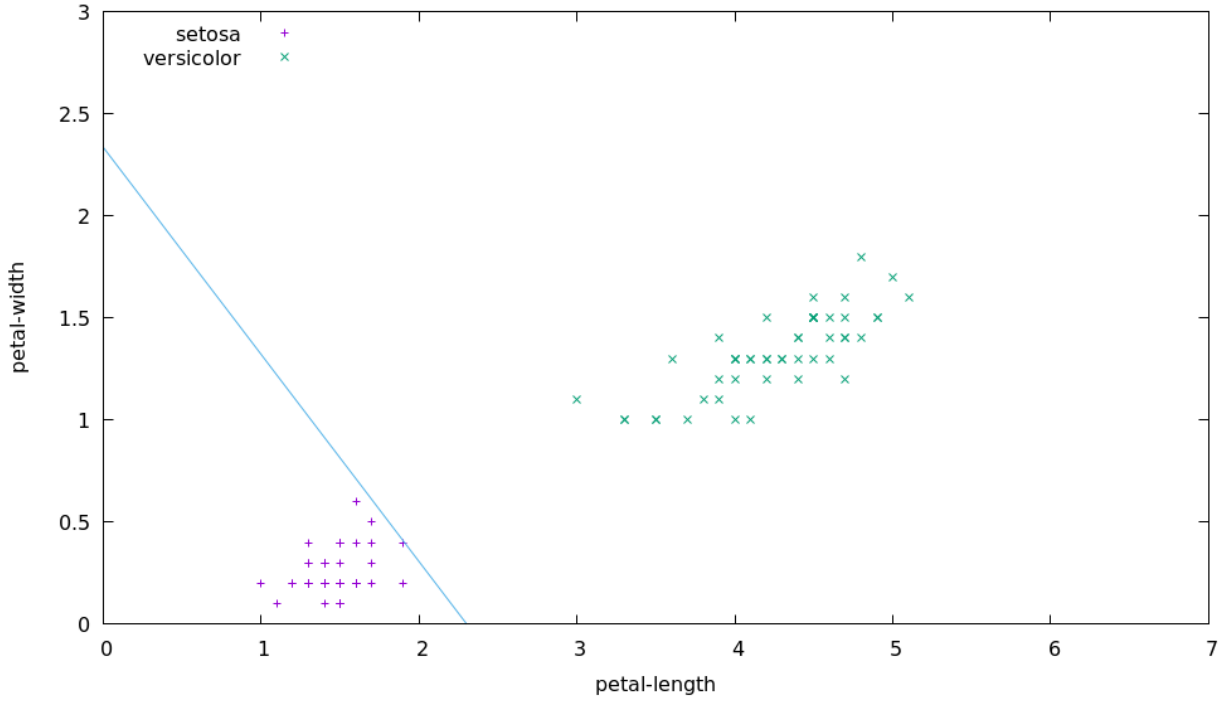
```
(defun decision-boundary (data-vec class-vec &optional (max 20000))
  (let* ((num-samples (length data-vec))
         (zipped      (zip data-vec class-vec))
         (δ           1e-10))
    (labels ((rec (iter vector)
               (if (> iter max)
                   vector
                   (let ((new-vec
                          (reduce (lambda (ys x)
                                    (let* ((class  (cadr x))
                                           (vector (car x))
                                           (dotted (dot-prduct ys vector)))
                                      (if (= class (signum dotted))
                                          ys ; don't update
                                          (let* ((normed (expt (norm vector) 2))
                                                 (η      (/ (- class dotted) normed)))
                                            (mapcar (lambda (a y) (+ y (* a η))) vector ys)))))
                                  zipped :initial-value vector)))
                     (if (< (* (/ 1 num-samples)
                               (abs (apply #'+ (mapcar #'- vector new-vec)))) δ)
                         vector
                         (rec (1+ iter) new-vec))))))
      (rec 0 (list 0 0 0)))))
```

- the function `answer-1.c` in the source file graphs the classifier on all permutations of the two classes
- versicolor and virginica

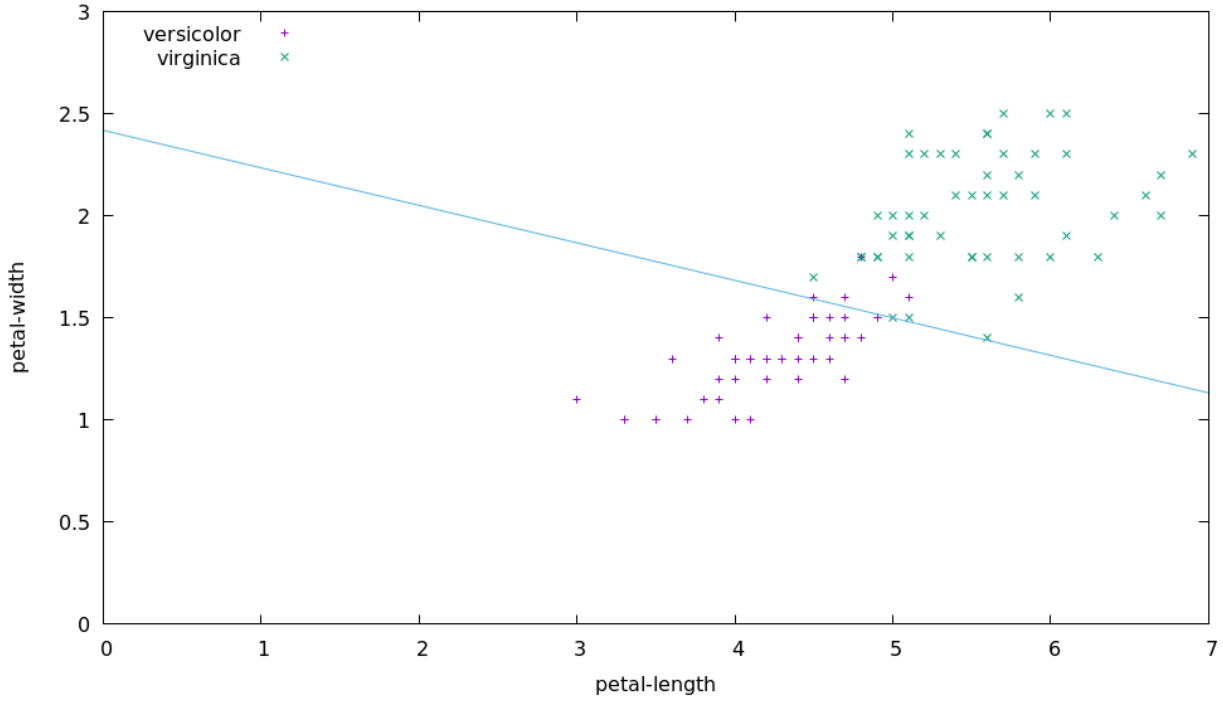


-

- setsona and versicolor



- setsona and virginica



2 Exercise 2. Objective functions

1. Write a program that calculates the mean-squared error for the iris data given a decision boundary. The function should take three arguments: the data vectors, the decision boundary parameters, and the pattern classes.

- Means squared error code

```
(flet ((new-class-val (class)
         (if (= -1 class) 0 1)))
  (defun mean-square-error (boundary-vec data-vec class-vec)
    (/ (apply #'+
              (mapcar (lambda (location class)
                        (expt (- (dot-prduct location boundary-vec)
                                 (new-class-val class))
                              2))
                      data-vec class-vec))
       2))
```

2. Compute the mean squared error for two different decision boundaries that give large and small errors respectively. Plot both boundaries on the dataset as above.

   - Here are two decision boundaries that I found

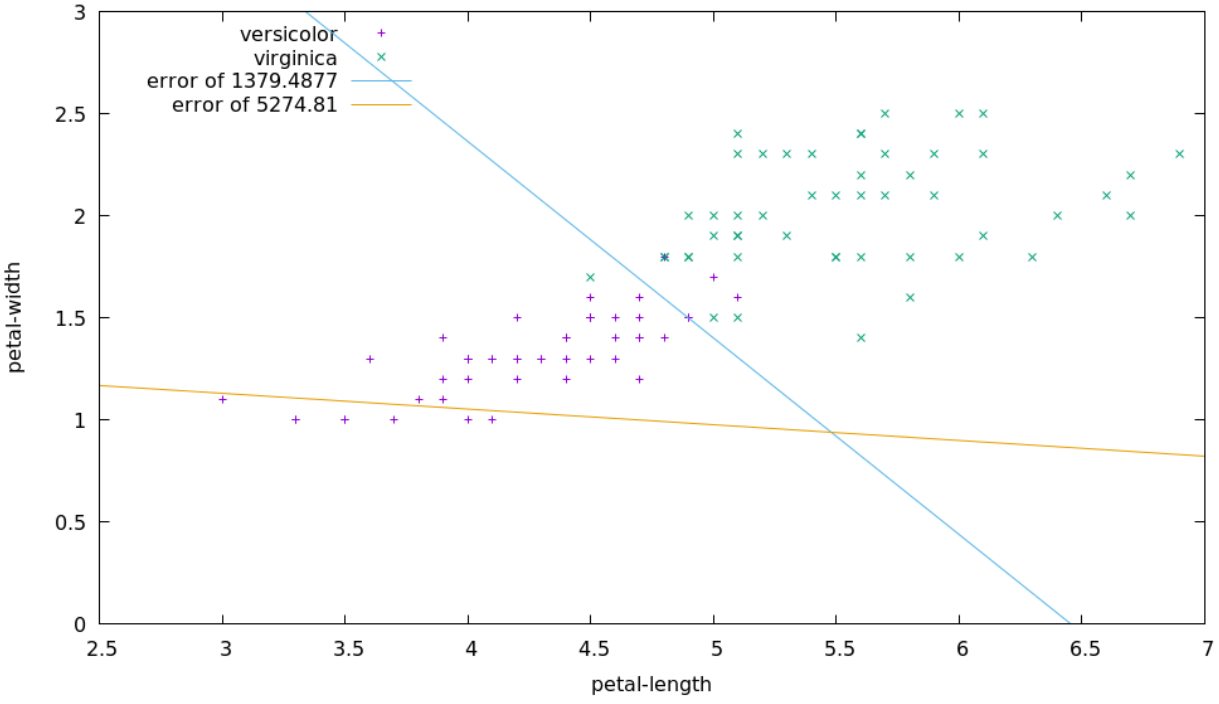
     ```
     (defparameter *small-error* '(-30.16284 4.6719003 4.854288))
     (defparameter *large-error* '(-17.7 1 13))
     ```

     – so the formatting is as such, the equation
       * $\theta + \alpha x + \beta y = 0$
     – is modeled by the first second and third values respectively I.E.
       * $-17.7 + x + 13y = 0$

   - Computing the error values and plotting is shown below

     ```
     (defun answer-2.b ()
       (flet ((calced (error-set)
                (let ((val (data-class-vectors *csv*)))
                  (mean-square-error error-set (car val) (cadr val)))))
         (graph-2 "images/2-b.png" (cdr (corrected *csv*))
                  *small-error* (calced *small-error*)
                  *large-error* (calced *large-error*))))
     ```

   -



3. Give a mathematical derivation the gradient of the objective function above with respect to the decision boundary weights. Use w_0 to represent the bias term. You should show and explain each step.

$$E = \frac{1}{2}\sum_{i=1}^{M}\left(w^T x_i - c_i\right)^2 \qquad \text{MSE function described above} \qquad (1)$$

$$= \frac{1}{2}\sum_{i=1}^{M}(\sum_{n=0}^{N} w_n x_{i,n} - c_i)^2 \qquad \text{definition of the dot product} \qquad (2)$$

$$\implies \frac{\partial E}{\partial w_j} = \frac{1}{2}\sum_{i=1}^{M}\frac{\partial}{\partial w_j}(\sum_{n=0}^{N} w_n x_{i,n} - c_i)^2 \qquad \text{Linearity of the deriative} \qquad (3)$$

$$= \frac{1}{2}\sum_{i=1}^{M} 2(\sum_{n=0}^{N} x_{i,n} - c_i)(\sum_{n=0}^{N}\frac{\partial}{\partial w_j} w_n x_{i,n} - c_i) \qquad \text{chain rule of the derivative} \qquad (4)$$

$$= \sum_{i=1}^{M}(\sum_{n=0}^{N} w_N x_{i,n} - c_i)x_{j,i} \qquad \text{Every other term goes to 0} \qquad (5)$$

$$= \sum_{i=1}^{M}(w^T x_i - c_i)x_{j,i} \qquad \text{Definition of dot product} \qquad (6)$$

$$\implies \frac{\partial E}{\partial w} = \sum_{i=1}^{M}(w^T x_i - c_i)x_i \qquad \text{we know where all of the J's go} \qquad (7)$$

4. Show how the gradient can be written in both scalar and vector form
   - In my derivation above, (5) shows the gradient function written in scalar form
   - And likewise, the final implies (line (7)) shows the gradient in vector form
     - note that we are able to derive this line from (6) since if we know where all J's go, then we know where the entire vector goes

5. Write code that computes the summed gradient for an ensemble of patterns. Illustrate the gradient by showing (i.e. plotting) how the decision boundary changes for a small step.
   - The code for gradient

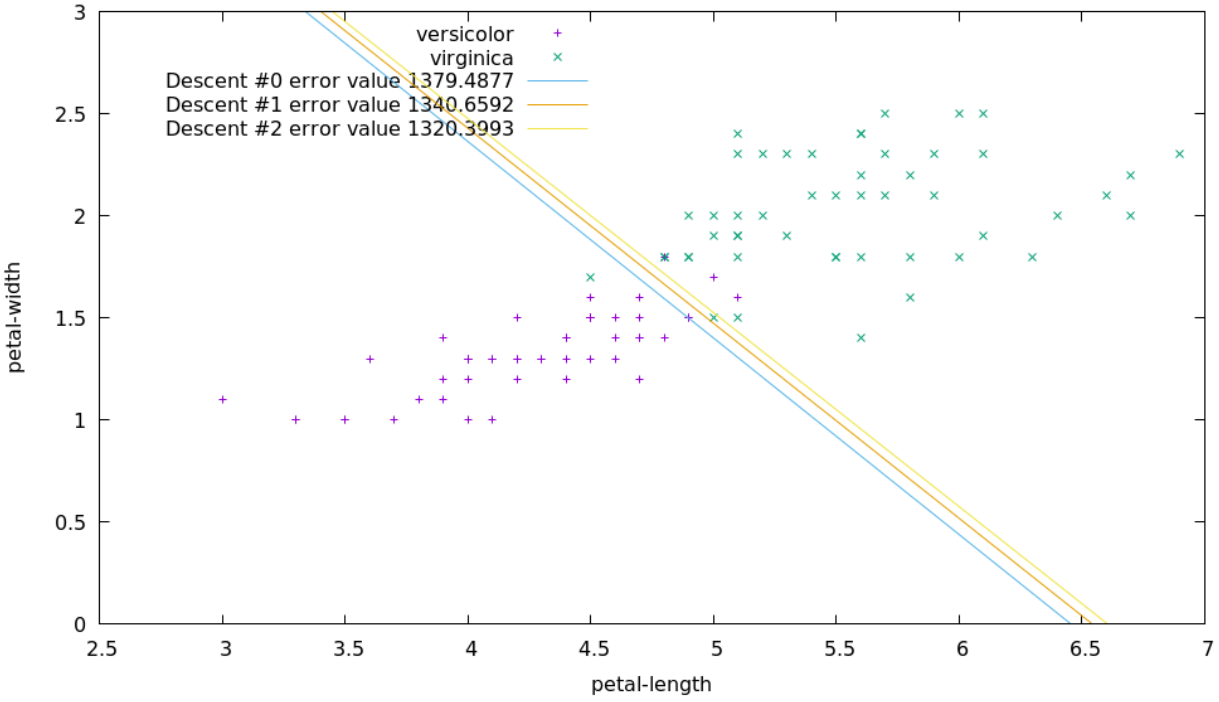
```
(flet ((new-class-val (class)
         (if (= -1 class) 0 1)))
(defun gradient-square (boundary-vec data-vec class-vec)
  (apply (curry mapcar #'+)
         (mapcar (lambda (location class)
                   (mapcar (curry *  (- (dot-prduct location boundary-vec)
                                        (new-class-val class)))
                           location))
                 data-vec class-vec))))
```

     - Here we just apply the formula from the above question
   - The code for the small step of the gradient

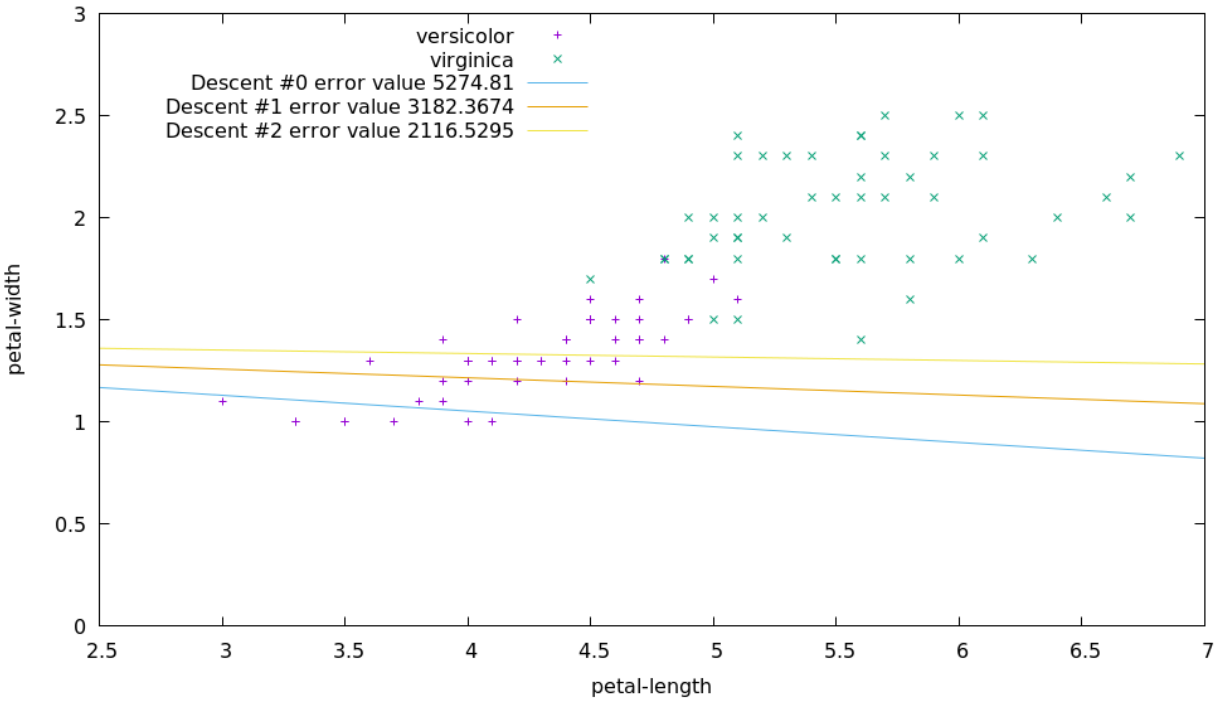
```
(defun gradient-step (boundary-vec data-vec class-vec &optional (ε 1e-4))
  (mapcar #'- boundary-vec
             (mapcar (curry * ε)
                     (gradient-square boundary-vec data-vec class-vec))))
```

     - here we simply apply gradient-square and then subtract the boundary vector by a small factor of this

- The graphs below are generated by the function `answer-2.e`
  - step on the boundary with small error



  - step on the boundary with large error



3 Exercise 3. Learning a decision boundary through optimization

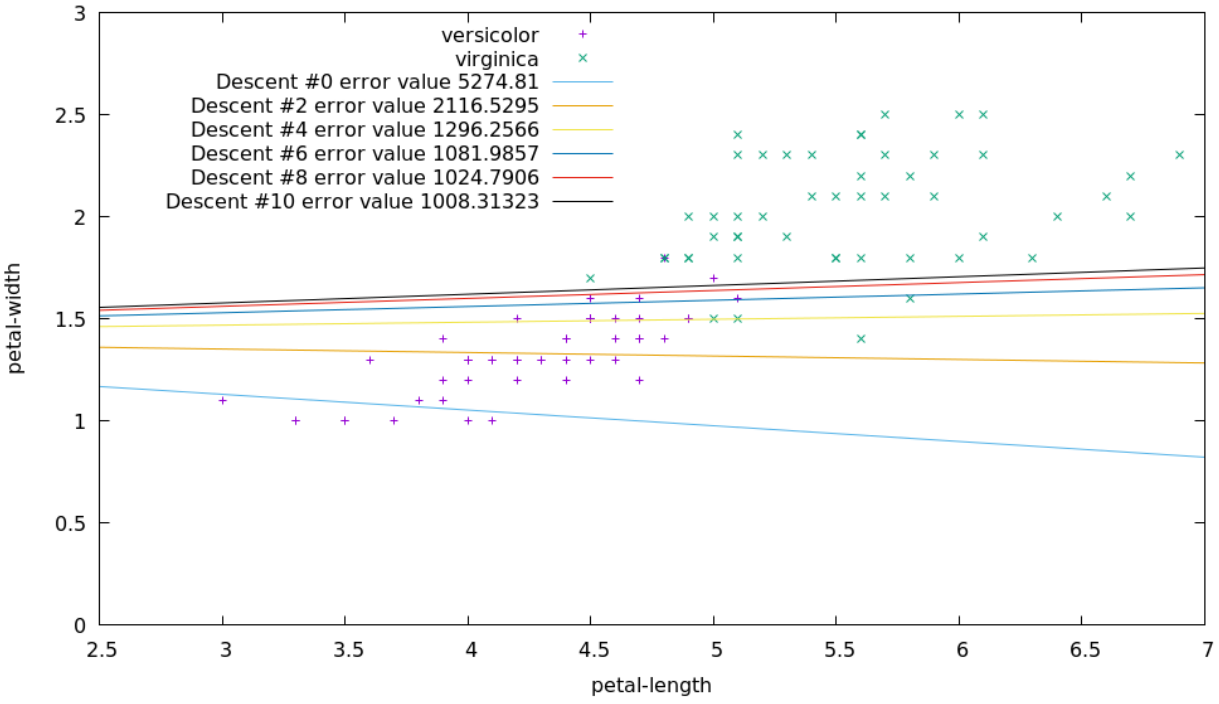
1. Using your code above, write a program that implements gradient descent to optimize the decision boundary for the iris dataset.

```
(defun gradient-descent (boundary-vec data-vec class-vec &optional (max-iter 5000) (δ 1e-2))
  (labels ((rec (boundary iter)
             (let ((stepped (gradient-step boundary data-vec class-vec)))
               (if (or (< (norm-2 stepped boundary) δ) (>= iter max-iter))
                   boundary
                   (rec stepped (1+ iter))))))
    (rec boundary-vec 0)))
```

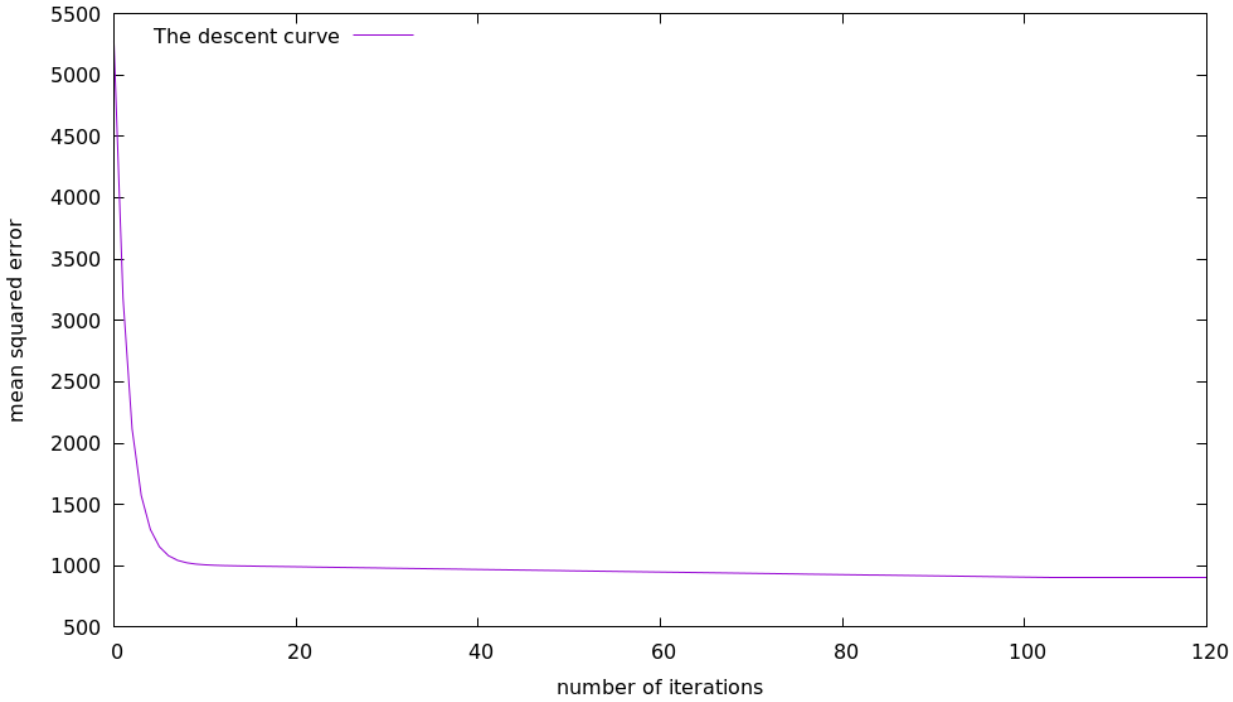
   - so in the code above I simply just call gradient-step on our current boundary point, and check whether we at our limit of calls, or if the new vector doesn't really move. If either of these two are met, we return the new vector, else we keep recursing

2. In your program, include code that shows the progress in two plots: the first should show the current decision boundary location overlaid on the data; the second should show the learning curve, i.e. a plot of the objective function as a function of the iteration.

   - I start with the boundary with the large error
   - I also only chose to overlay the even iterations to 10 as to not make the graph cluttered
   - the results can be seen below



   - As for the function error graph, I graphed all points to 120
   - For the graph below I choose a stop value of 1e-2, which as can see converges at 100, however if we choose a value of 1e-3, it does not converge even after 1000 steps (it gets the value to around 348!).

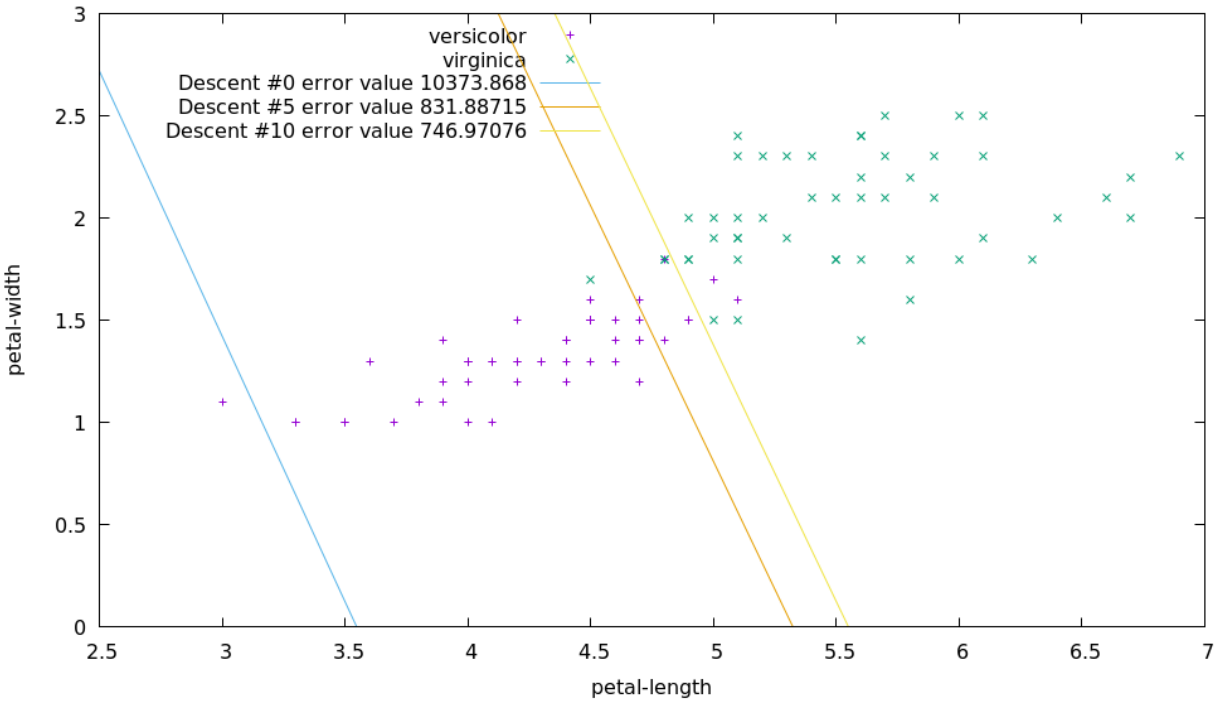


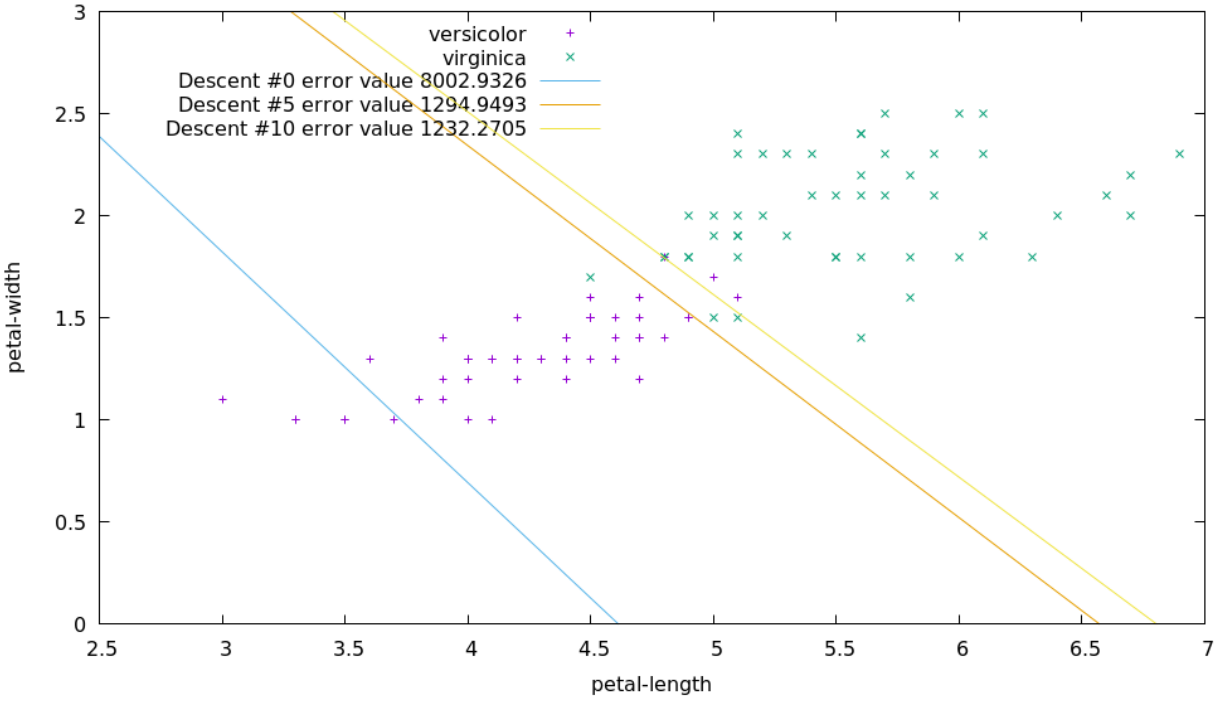
3. Run your code on the iris data set starting from a random setting of the weights. Note: you might need to restrict the degree of randomness so that the initial decision boundary is visible somewhere in the plot. In your writeup, show the two output plots at the initial, middle, and final locations of the decision boundary.

   - I ended up playing with random numbers until I was able to somewhat reliably have the original value show up, below is such a vector

     ```
     (defun make-random-boundary ()
       (list (- (random-from-range 24 35))
             (- (random 8.0) 1)
             (- (random 10.0) 2)))
     ```

   - The two graphs below are taken at steps zero, five, and ten as they illustrate the movement of gradient descent (higher numbers don't really make much of an impression on the graph)





4. Explain how you chose the gradient step size.
   - I ended up choosing a step size of 1e-4.
   - I chose this step size, as it didn't move the data too much (like 1e-2 would have), nor did it move the data too little (like 1e-7 did)
5. Explain how you chose a stopping criterion.
   - I ended up choosing 1e-2, as it ends up converging after a decent number of steps (think around a few hundred), note that 1e-3 also seems like a decent stopping criteria as it stops around 8500

iterations. So in summery, 1e-2 doesn't converge too quickly nor to slowly, but if one wants a better measurement that converges at around 9000, 1e-3 is preferred.