

# EECS 391 Programming Project 2

*Anna Sedlackova*

*07 December, 2018*

## Contents

0.1	Exercise 1: Linear Decision Boundaries . . . . .	1
0.1.1	Part A . . . . .	1
0.1.2	Part B . . . . .	3
0.1.3	Part C . . . . .	3
0.1.4	Part D . . . . .	5
0.1.5	Part E . . . . .	5
0.2	Exercise 2: Neural Networks . . . . .	8
0.2.1	Part A . . . . .	8
0.2.2	Part B . . . . .	8
0.2.3	Part C . . . . .	10
0.2.4	Part D . . . . .	11
0.2.5	Part E . . . . .	12
0.3	Exercise 3: Learning a Decision Boundary Through Optimization . . . . .	14
0.3.1	Part A and Part B . . . . .	14
0.3.2	Part C . . . . .	15
0.3.3	Part D . . . . .	21
0.3.4	Part E . . . . .	21
0.4	Extra Credit . . . . .	21

## 0.1 Exercise 1: Linear Decision Boundaries

### 0.1.1 Part A

Lets load and import the iris data set

```
iris_data <- read.csv("./irisdata.csv")
summary(iris_data)
```

```
##      sepal_length      sepal_width      petal_length      petal_width
## Min.      :4.300    Min.      :2.000    Min.      :1.000    Min.      :0.100
## 1st Qu.:5.100    1st Qu.:2.800    1st Qu.:1.600    1st Qu.:0.300
## Median :5.800    Median :3.000    Median :4.350    Median :1.300
## Mean   :5.843    Mean   :3.054    Mean   :3.759    Mean   :1.199
## 3rd Qu.:6.400    3rd Qu.:3.300    3rd Qu.:5.100    3rd Qu.:1.800
## Max.   :7.900    Max.   :4.400    Max.   :6.900    Max.   :2.500
##      species
## setosa      :50
## versicolor:50
## virginica   :50
##
##
##
```

```
# Lets look at the head of the iris data set
head(iris_data)
```

```
##   sepal_length sepal_width petal_length petal_width species
## 1         5.1         3.5         1.4         0.2   setosa
## 2         4.9         3.0         1.4         0.2   setosa
## 3         4.7         3.2         1.3         0.2   setosa
## 4         4.6         3.1         1.5         0.2   setosa
## 5         5.0         3.6         1.4         0.2   setosa
## 6         5.4         3.9         1.7         0.4   setosa
```

```
# Number of rows of the iris dataset (number of individual observations)
nrow(iris_data)
```

```
## [1] 150
```

```
# All of the columns are numeric except for the species column
# They are continuous because they correspond to length and width (except for species)
# The species are discrete
sapply(iris_data, class)
```

```
## sepal_length sepal_width petal_length petal_width species
##   "numeric"   "numeric"   "numeric"   "numeric"   "factor"
```

```
# The only categorical variable is species in this iris dataset
sapply(iris_data, class)
```

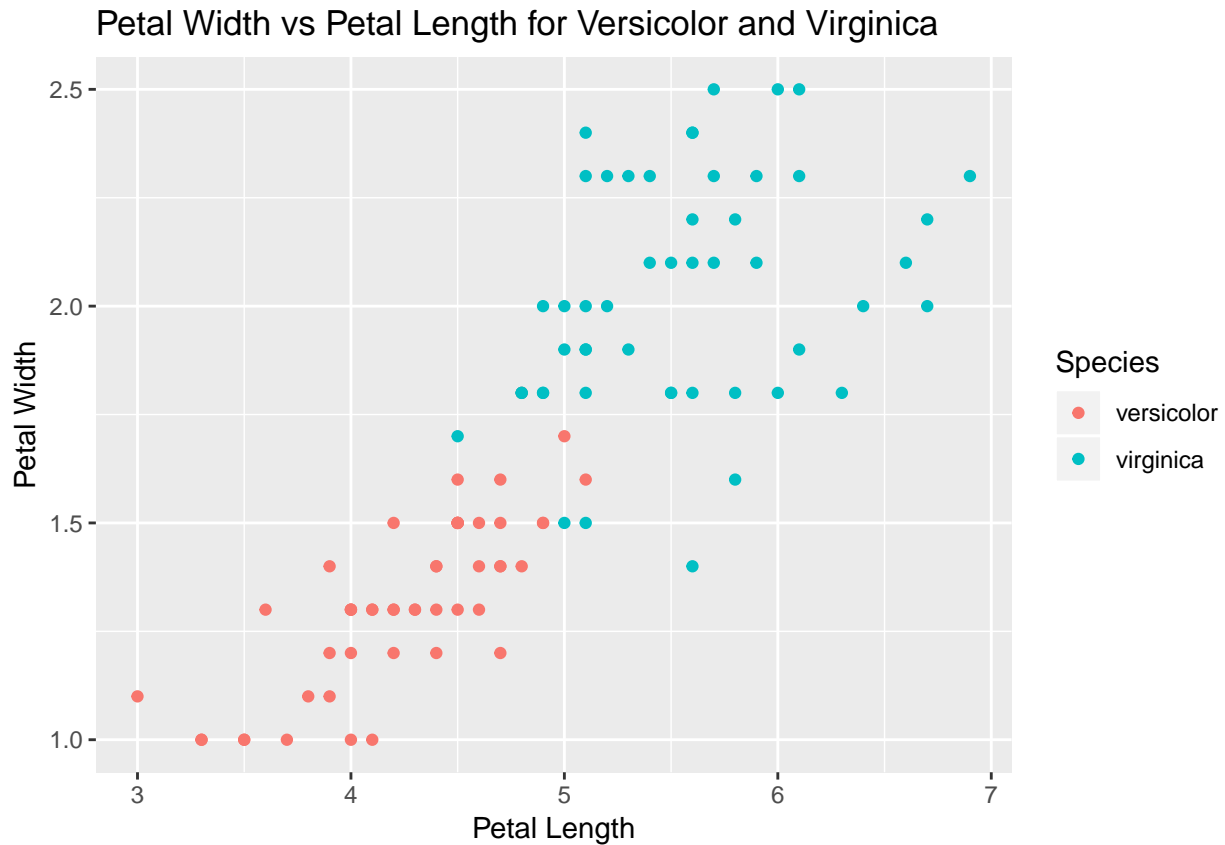
```
## sepal_length sepal_width petal_length petal_width species
##   "numeric"   "numeric"   "numeric"   "numeric"   "factor"
```

From our observations it looks like there are 50 samples of three species: setosa, versicolor and virginica. Each of the samples has measurements of sepal length, sepal width and petal length.

Let's plot the second (versicolor) and third (virginica) iris classes:

```
# Lets pick the 2nd and 3rd classes: versicolor and virginica and let's plot the subset
plot1 <- iris_data %>% filter(species %in% c('versicolor', 'virginica')) %>%
ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
ggtitle("Petal Width vs Petal Length for Versicolor and Virginica")
```

```
plot1
```



### 0.1.2 Part B

The following function will take in five inputs ( $w_0$ ,  $w_1$ ,  $w_2$ ) which are constants that will be estimated later in this exercise and ( $x_1$ ,  $x_2$ ) which correspond to petal width and petal length. The output will return a value between a probability value between 0 and 1 where all values greater than or equal to 0.5 will correspond to the 3rd iris class (this class will be denoted by 1), and values below 0.5 to the 2nd iris class (this class will be denoted by 0).

```
# inputs: c0, c1, c2, petal_width, petal_length
# outputs: value between 0 and 1 representing the probability

one_layer_neural_network <- function(w0, w1, w2, petal_length, petal_width) {

  # Linear function
  z <- w0 + w1*petal_length + w2*petal_width

  #result
  result <- 1/(1 + exp(-z))

  return (result)
}
```

### 0.1.3 Part C

By playing around with the parameters, we select  $w_0 = -2.8$ ,  $w_1 = 0.25$  and  $w_2 = 1$ , which gives a decision boundary that separates the data decently (see below). The boundary that separates the data arises by

noting the following:

Starting with the condition that a given iris flower is in class 1 (virginica):

$$\sigma(\mathbf{w}^T \mathbf{x}) \geq 0.5$$

where  $\sigma$  is the logistic function,  $\mathbf{w} = (w_0, w_1, w_2)$ , and  $\mathbf{x} = (1, x_1, x_2)$  ( $x_1$  corresponds to petal length and  $x_2$  corresponds to petal width). After a bit of manipulation, the above inequality implies the following:

$$\mathbf{w}^T \mathbf{x} \geq 0$$

Writing this out with our chosen values for the coefficients above gives us:

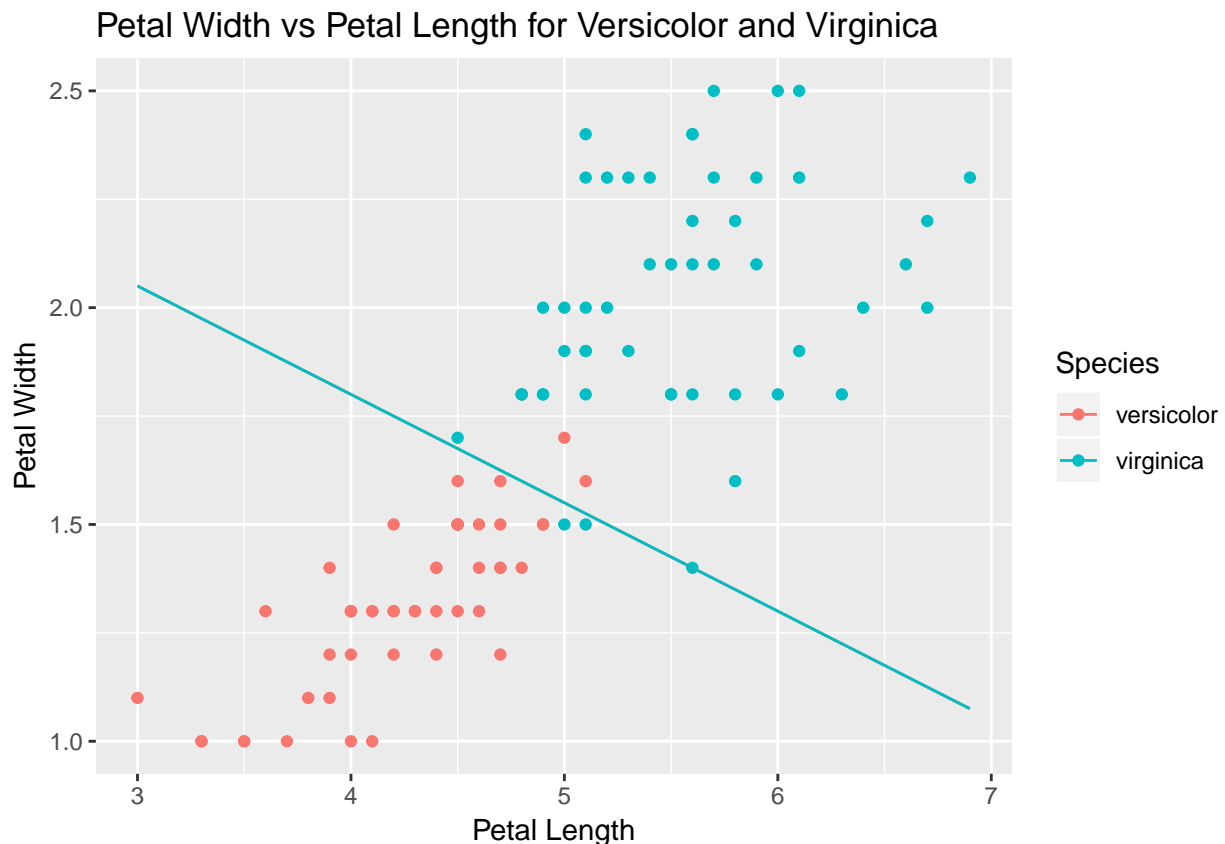
$$-2.8 + 0.25x_1 + x_2 \geq 0$$

Consequently, the decision boundary is given by the line  $x_2 = 2.8 - 0.25x_1$  (above this line will be class 1 (virginica), and below this line will be class 0 (versicolor)). This result is plotted below:

```
z_function <- function(x) (2.8 + (-1/4)*x)

# Lets pick the 2nd and 3rd classes: versicolor and virginica and let's plot the subset
plot2 <- iris_data %>% filter(species %in% c('versicolor', 'virginica')) %>%
  ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
  labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
  ggtitle("Petal Width vs Petal Length for Versicolor and Virginica") +
  stat_function(fun = z_function)
```

plot2



#### 0.1.4 Part D

```
# Create vectors of the input dimensions discretized
x_dim <- seq(0, 10, by = 0.1)
y_dim <- seq(0, 10, by = 0.1)

# Output of neural network at all the discretized points
z_dim <- matrix(data = NA, nrow = length(x_dim), ncol = length(y_dim), byrow = FALSE, dimnames = NULL)
for (i in 1:length(x_dim)){
  for (j in 1:length(y_dim)){
    z_dim[i,j] = one_layer_neural_network(-2.8, 0.25, 1, x_dim[i], y_dim[j])
  }
}

# Plot the surface
plot_ly(x = x_dim, y = y_dim, z = z_dim) %>%
  add_surface() %>%
  layout(
    title = "Neural Network Output vs. Input Dimensions",
    scene = list(
      xaxis = list(title = "Petal Length"),
      yaxis = list(title = "Petal Width"),
      zaxis = list(title = "Output")
    )
  )
```

## PhantomJS not found. You can install it with `webshot::install_phantomjs()`. If it is installed, please

Because the 3D plotting package has issues the 3D plot is attached separately as an image. You can still view the 3D file when running the rmd file.

#### 0.1.5 Part E

Now let's see how are function from Part B performs:

```
# Versicolor
data1 <- iris_data %>% filter (species == 'versicolor')

for (i in 1:50){

  if (one_layer_neural_network(-2.8, 0.25, 1, data1$petal_length[i],
                                data1$petal_width[i]) < 0.5){
    resulting_flower <- "versicolor"

  } else {

    resulting_flower <- "virginica"
  }
  print(paste0("actual: versicolor, predicted: ", resulting_flower))
}

## [1] "actual: versicolor, predicted: versicolor"
## [1] "actual: versicolor, predicted: versicolor"
## [1] "actual: versicolor, predicted: versicolor"
## [1] "actual: versicolor, predicted: versicolor"
## [1] "actual: versicolor, predicted: versicolor"
```



```

} else {
  resulting_flower <- "virginica"
}
print(paste0("actual: virginica, predicted: ", resulting_flower))
}

```

[illegible]

```
## [1] "actual: virginica, predicted: virginica"
## [1] "actual: virginica, predicted: virginica"
```

Results match to what we have seen in the graph above.

## 0.2 Exercise 2: Neural Networks

### 0.2.1 Part A

Mean squared error calculations. The inputs are in order: the data vectors (as the iris dataset that the petal length and petal width information is taken from), the parameters defining the neural network ( $w_0$ ,  $w_1$ ,  $w_2$ ).

```
# Input feature vectors
X_data <- iris_data %>% filter(species %in% c("versicolor", "virginica")) %>%
  select(petal_length, petal_width)
# Pattern classes for each feature vector above
y_data <- iris_data %>%
  filter(species %in% c("versicolor", "virginica")) %>%
  select(species) %>%
  mutate(label = ifelse(species == "versicolor", 0, 1)) %>%
  select(-species)

mean_squared_error <- function(X, y, w0, w1, w2) {

  # Outputs of neural network for each example
  NN_outputs <- rep(0, nrow(X))

  for (i in 1:nrow(X)){
    NN_outputs[i] = one_layer_neural_network(w0, w1, w2, X$petal_length[i], X$petal_width[i])
  }

  # Compute the differences between the NN outputs and the labels
  result = mean((NN_outputs - y$label)^2)

  # Return result
  return(result)
}
```

### 0.2.2 Part B

I have decided to look at three different values. First, I computed the mean square error of the original and got a mean square error of approximately 0.1489:

```
# Mean squared error for decision boundary we guessed above
paste0("Previous Mean Square Error: ", mean_squared_error(X_data, y_data, -2.8, 0.25, 1))
```

```
## [1] "Previous Mean Square Error: 0.148888121336409"
```

Then I made a completely incorrect estimate of the coefficients such that the boundary is not even touching any of the data points and got a much worse mean square error of 0.4401.

```
# High error
paste0("High Mean Square Error: ", mean_squared_error(X_data, y_data, 1, -0.25, -1))
```

```
## [1] "High Mean Square Error: 0.440092737113904"
```



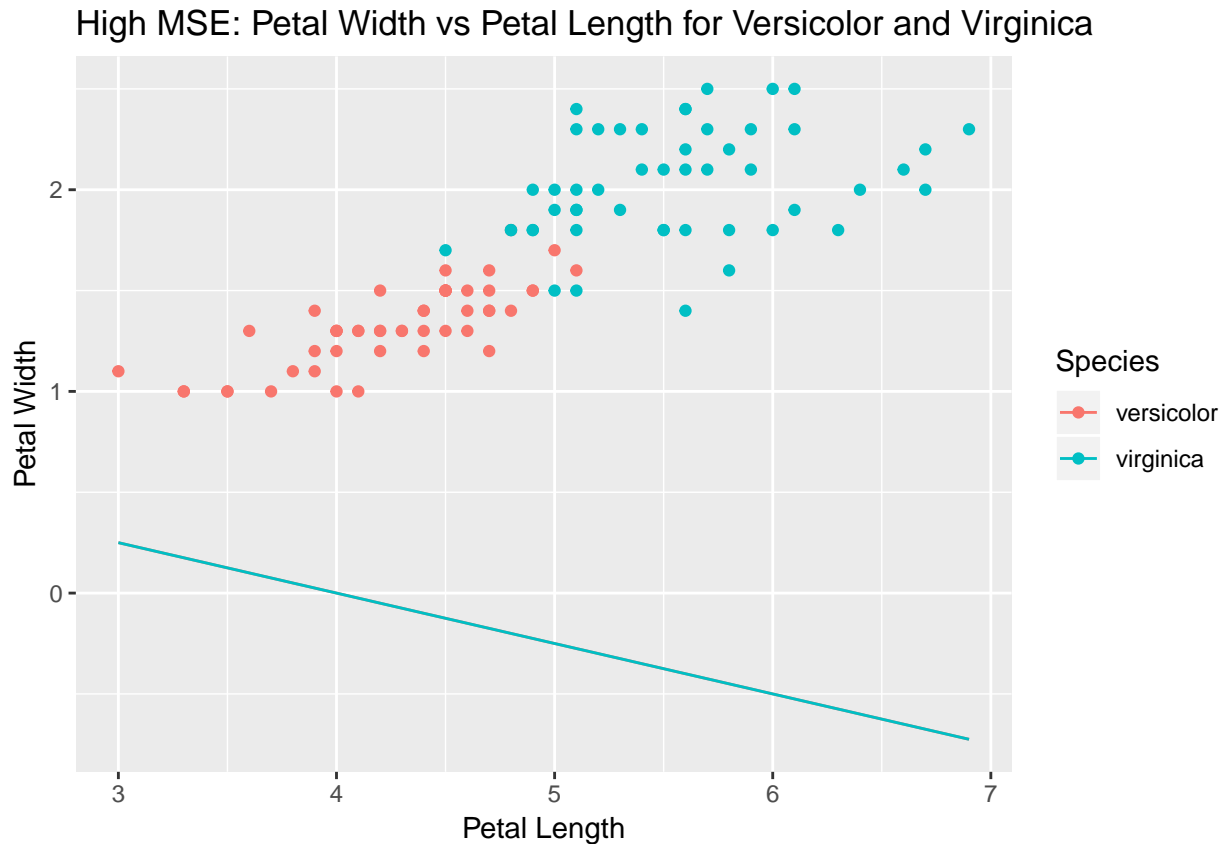
```

z_function2 <- function(x) (1 - (0.25)*x)

plot3 <- iris_data %>% filter(species %in% c('versicolor','virginica')) %>%
ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
ggtitle("High MSE: Petal Width vs Petal Length for Versicolor and Virginica") +
stat_function(fun = z_function2)

```

plot3



Finally, I found a slightly better boundary line that made only three points dislocated and decreased the mean square error slightly to 0.1423.

```

# Low error
paste0("Low Mean Square Error: ", mean_squared_error(X_data, y_data, -3.15, 1/3, 1))

```

```
## [1] "Low Mean Square Error: 0.142297154674927"
```

```

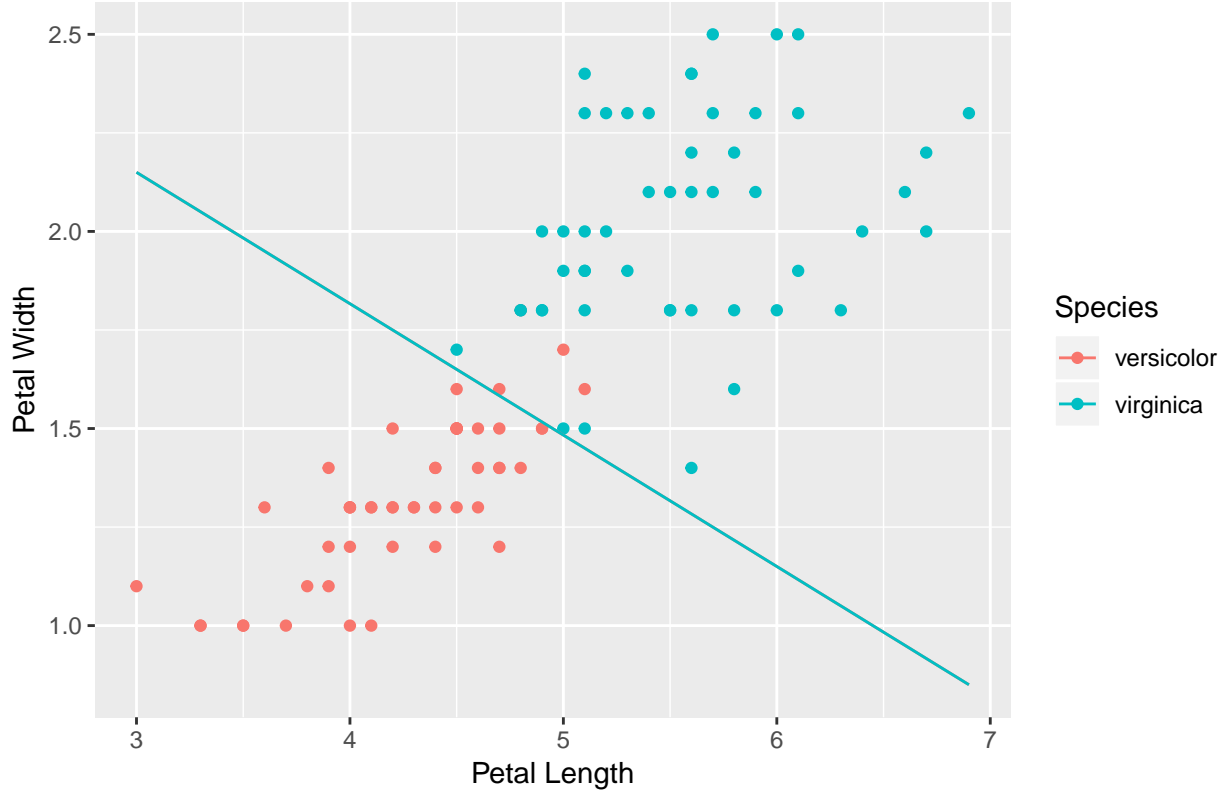
z_function3 <- function(x) (3.15 + (-1/3)*x)

plot4 <- iris_data %>% filter(species == 'versicolor' | species == 'virginica') %>%
ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
ggtitle("Low MSE: Petal Width vs Petal Length for Versicolor and Virginica") +
stat_function(fun = z_function3)

```

plot4

Low MSE: Petal Width vs Petal Length for Versicolor and Virginica



### 0.2.3 Part C

Before we begin with the derivation, we will set up some notation for convenience. We will denote the number of examples in our data as  $m$ , which in this case is 100. Our data will be collected in the following matrix:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} \\ \vdots & \vdots & \vdots \\ 1 & x_{m1} & x_{m2} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_m^T \end{bmatrix}$$

where  $\mathbf{x}_i^T = [1 \ x_{i1} \ x_{i2}]$  is a single example with petal length  $x_{i1}$  and petal width  $x_{i2}$ . We will also collect all the corresponding species/classes of these examples in a single vector  $\mathbf{y}$  below:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

where  $y_i$  is a value (0 or 1) corresponding to the respective species of example  $i$ . We will also write the coefficients in a single vector  $\mathbf{w}^T = [w_0 \ w_1 \ w_2]$ . Also, once again, the logistic function is given by  $\sigma(z) = (1 + \exp(-z))^{-1}$ . With this notation set up, we write the mean squared error (MSE) as a function of the parameter vector as follows:

$$MSE(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^m [\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i]^2$$

Noting that the derivative of the logistic function is  $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ , we can compute the partial derivatives of the MSE with respect to each parameter ( $w_0, w_1, w_2$ ) as follows:

$$\frac{\partial MSE}{\partial w_0} = \frac{2}{m} \sum_{i=1}^m [\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i] \frac{\partial \sigma(\mathbf{w}^T \mathbf{x}_i)}{\partial w_0} = \frac{2}{m} \sum_{i=1}^m [\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i] [\sigma(\mathbf{w}^T \mathbf{x}_i)] [1 - \sigma(\mathbf{w}^T \mathbf{x}_i)]$$

Similarly, we can write out the partials with respect to the other two parameters:

$$\begin{aligned} \frac{\partial MSE}{\partial w_1} &= \frac{2}{m} \sum_{i=1}^m [\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i] [\sigma(\mathbf{w}^T \mathbf{x}_i)] [1 - \sigma(\mathbf{w}^T \mathbf{x}_i)] x_{i1} \\ \frac{\partial MSE}{\partial w_2} &= \frac{2}{m} \sum_{i=1}^m [\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i] [\sigma(\mathbf{w}^T \mathbf{x}_i)] [1 - \sigma(\mathbf{w}^T \mathbf{x}_i)] x_{i2} \end{aligned}$$

The  $x_{i1}$  and  $x_{i2}$  arises in the two results above because, for example,  $\frac{\partial (\mathbf{w}^T \mathbf{x}_i)}{\partial w_1} = \frac{\partial}{\partial w_1} (w_0 + w_1 x_{i1} + w_2 x_{i2}) = x_{i1}$ .

The combination of these three partial derivatives then gives the gradient of the MSE:

$$\nabla_{\mathbf{w}} MSE = \begin{bmatrix} \frac{\partial MSE}{\partial w_0} \\ \frac{\partial MSE}{\partial w_1} \\ \frac{\partial MSE}{\partial w_2} \end{bmatrix}$$

## 0.2.4 Part D

Now, having derived the scalar form of the result above, we will attempt to vectorize the result in a cleaner, convenient form. First, if we look at each of the partial derivatives above, there is a common factor in all of them. Let's denote  $z_i$  as the following:

$$z_i = [\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i] [\sigma(\mathbf{w}^T \mathbf{x}_i)] [1 - \sigma(\mathbf{w}^T \mathbf{x}_i)]$$

Then the gradient becomes the following:

$$\nabla_{\mathbf{w}} MSE = \frac{2}{m} \begin{bmatrix} \sum_{i=1}^m z_i \\ \sum_{i=1}^m z_i x_{i1} \\ \sum_{i=1}^m z_i x_{i2} \end{bmatrix} = \frac{2}{m} \begin{bmatrix} 1 & \dots & 1 \\ x_{11} & \dots & x_{m1} \\ x_{12} & \dots & x_{m2} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} = \frac{2}{m} X^T \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix}$$

Now, let's focus on the vector of all  $z_i$ . Notice the following:

$$\begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} [\sigma(\mathbf{w}^T \mathbf{x}_1) - y_1] [\sigma(\mathbf{w}^T \mathbf{x}_1)] [1 - \sigma(\mathbf{w}^T \mathbf{x}_1)] \\ \vdots \\ [\sigma(\mathbf{w}^T \mathbf{x}_m) - y_m] [\sigma(\mathbf{w}^T \mathbf{x}_m)] [1 - \sigma(\mathbf{w}^T \mathbf{x}_m)] \end{bmatrix} = \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}_1) - y_1 \\ \vdots \\ \sigma(\mathbf{w}^T \mathbf{x}_m) - y_m \end{bmatrix} * \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}_1) \\ \vdots \\ \sigma(\mathbf{w}^T \mathbf{x}_m) \end{bmatrix} * \begin{bmatrix} 1 - \sigma(\mathbf{w}^T \mathbf{x}_1) \\ \vdots \\ 1 - \sigma(\mathbf{w}^T \mathbf{x}_m) \end{bmatrix}$$

where in the above the  $*$  denotes component-wise multiplication of vectors. Defining the following convenient notation,

$$\sigma(X\mathbf{w}) = \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}_1) \\ \vdots \\ \sigma(\mathbf{w}^T \mathbf{x}_m) \end{bmatrix}$$

we can then express our final vector expression conveniently as the following result:

$$\nabla_{\mathbf{w}} MSE = \frac{2}{m} X^T [(\sigma(X\mathbf{w}) - \mathbf{y}) * \sigma(X\mathbf{w}) * (\mathbf{1} - \sigma(X\mathbf{w}))]$$

where  $\mathbf{1}$  is the  $m$ -dimensional vector with 1 in all components.

### 0.2.5 Part E

Let us write the code to computer the summed gradient for an ensemble of patterns. The algorithm goes as follows:

```
gradMSE <- function(X, y, w0, w1, w2) {
  dMSE_dw0 <- 0
  dMSE_dw1 <- 0
  dMSE_dw2 <- 0
  m = nrow(X)

  # Loop through all the examples in the input data
  for (i in 1:m){
    x_i1 = X$petal_length[i]
    x_i2 = X$petal_width[i]
    y_i = y$label[i]
    # Output of neural network for example i
    sigma = one_layer_neural_network(w0, w1, w2, x_i1, x_i2)

    # Append the contributions to the sums for each gradient component
    dMSE_dw0 <- dMSE_dw0 + (2/m)*(sigma - y_i)*(sigma)*(1 - sigma)*(1)
    dMSE_dw1 <- dMSE_dw1 + (2/m)*(sigma - y_i)*(sigma)*(1 - sigma)*(x_i1)
    dMSE_dw2 <- dMSE_dw2 + (2/m)*(sigma - y_i)*(sigma)*(1 - sigma)*(x_i2)
  }

  # Compile all components into a 3-dimensional vector
  gradient_result = c(dMSE_dw0, dMSE_dw1, dMSE_dw2)

  # Return result as a vector
  return(gradient_result)
}
```

In order to demonstrate how the gradient changes, we are going to run through the algorithm 10000 times and create a before and after plot.

```
# Initially constants are set to one
W <- c(1, 1, 1)
print(W)

## [1] 1 1 1

print(paste("MSE Before Update:", mean_squared_error(X_data, y_data, W[1], W[2], W[3])))

## [1] "MSE Before Update: 0.498299605634765"

# We are going to iterate 10000 times and print the respective plots
for (i in 1:10000){
  # Step size
  W <- W - (0.01)*gradMSE(X_data, y_data, W[1], W[2], W[3])
}
```

```
print(W)

## [1] -1.2396779 -0.3644774  1.9081652
print(paste("MSE After Update:", mean_squared_error(X_data, y_data, W[1], W[2], W[3])))

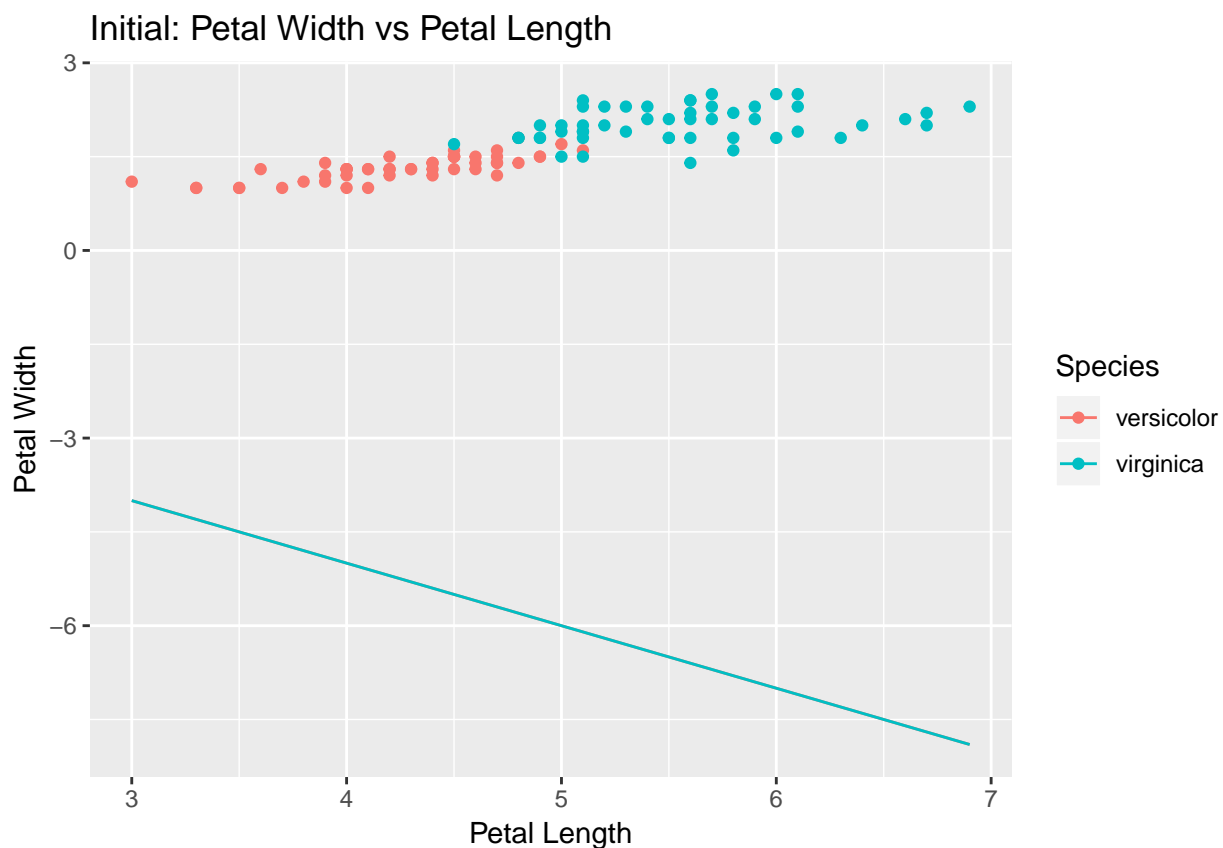
## [1] "MSE After Update: 0.168168545747278"

The initial plot started with the value of the constants equal to  $w_1 = 1$ ,  $w_2 = 1$ ,  $w_3 = 1$ .

z_function4 <- function(x) ((-1) + (-1)*x)

plot5 <- iris_data %>% filter(species == 'versicolor' | species == 'virginica') %>%
  ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
  labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
  ggtitle("Initial: Petal Width vs Petal Length") +
  stat_function(fun = z_function4)

plot5
```

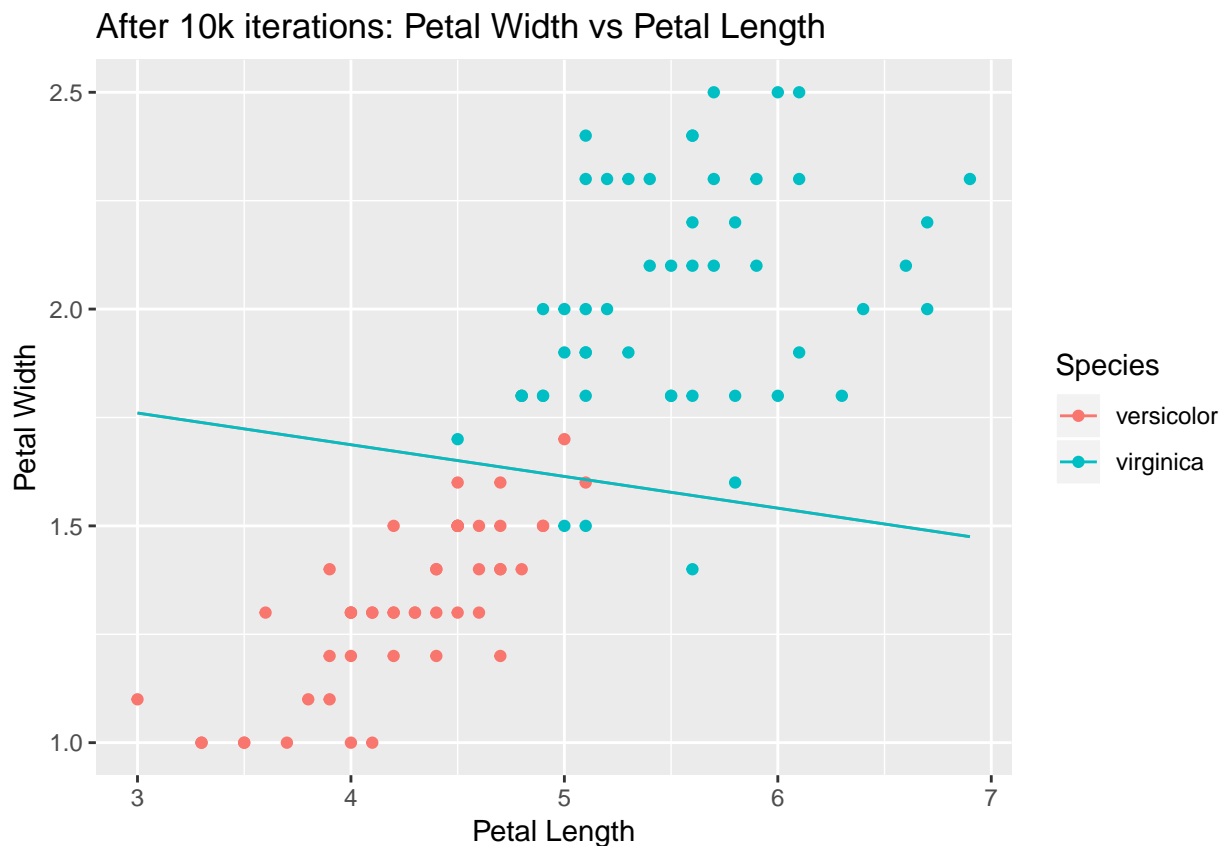


After 10,000 iterations with the learning step size equal to 0.01 we get the following graph:

```
z_function5 <- function(x) ((4.3889725/2.2170461) + (-0.1621191/2.2170461)*x)

plot6 <- iris_data %>% filter(species == 'versicolor' | species == 'virginica') %>%
  ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
  labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
  ggtitle("After 10k iterations: Petal Width vs Petal Length") +
  stat_function(fun = z_function5)
```

plot6



### 0.3 Exercise 3: Learning a Decision Boundary Through Optimization

#### 0.3.1 Part A and Part B

Let us put everything together so we only have one method with the initial constants as an input as well as the gradient step size and the number of iterations. We are also going to make both of our plotting functions inside our method directly.

```
optimizing <- function(w0, w1, w2, step_size, number_of_iterations, display_learning_curve) {  
  W <- c(w0,w1,w2)  
  
  mse <- c(0,number_of_iterations)  
  iteration <- c(0, number_of_iterations)  
  
  for (i in 1:number_of_iterations){  
    if (i == 1){  
      previousW <- c(0,0,0)  
    }  
    mse[i] <- mean_squared_error(X_data, y_data, W[1], W[2], W[3])  
    iteration[i] <- i  
  
    # Break out of the loop using stopping criterion if necessary  
    if (abs(W[1] - previousW[1]) < 0.00005 & abs(W[2] - previousW[2]) < 0.00005 &
```

```

    abs(W[3] - previousW[3]) < 0.00005){
      print("Weights differ by less than 0.00005.")
      break
    }

    previousW <- W
    W <- W - (step_size)*gradMSE(X_data,y_data,W[1],W[2],W[3])
  }

  # Make a new data frame to store results for the learning curve
  learning_curve_data <- cbind(mse, iteration)
  learning_curve_data <- as.data.frame(learning_curve_data)

  # Lets print the constants
  print(W)

  # Lets print the MSE at this point as well
  print(paste("MSE: ",mean_squared_error(X_data,y_data,W[1],W[2],W[3])))

  optimizing_function <- function(x) ((-W[1]/W[3]) + (-W[2]/W[3])*x)

  # Learning curve
  optimizing_curve <- ggplot(learning_curve_data, aes(x = iteration, y = mse)) +
    geom_point(size = 1.2, col = 'darkgreen', shape = 2) +
    geom_line(lwd = 1.2, col = 'darkgreen') + ylab('Mean Squared Error') +
    ggtitle(sprintf('Learning Curve with step: %0.4f', step_size)) +
    theme_classic(12)

  # Current decision boundary location overlayed on the data
  optimizing_plot <- iris_data %>% filter(species == 'versicolor' | species == 'virginica') %>%
    ggplot(aes(x = petal_length, y = petal_width, color = species)) + geom_point() +
    labs(color = "Species") + xlab("Petal Length") + ylab("Petal Width") +
    ggtitle("Versicolor and Virginica with Decision Boundary") +
    stat_function(fun = optimizing_function)

  if (display_learning_curve == FALSE){
    optimizing_plot
  } else {
    optimizing_curve
  }
}

```

### 0.3.2 Part C

We are going to test our function above with different (randomized) starting points. The points were adjusted so the initial line would fit in the plot. We can see that at each iteration the line moves closer to the “correct” dividing area and the mean square error gets smaller. Interestingly, at the final stage we just break out of the loop because the weights stay about the same at the different iterations of the loop.

```

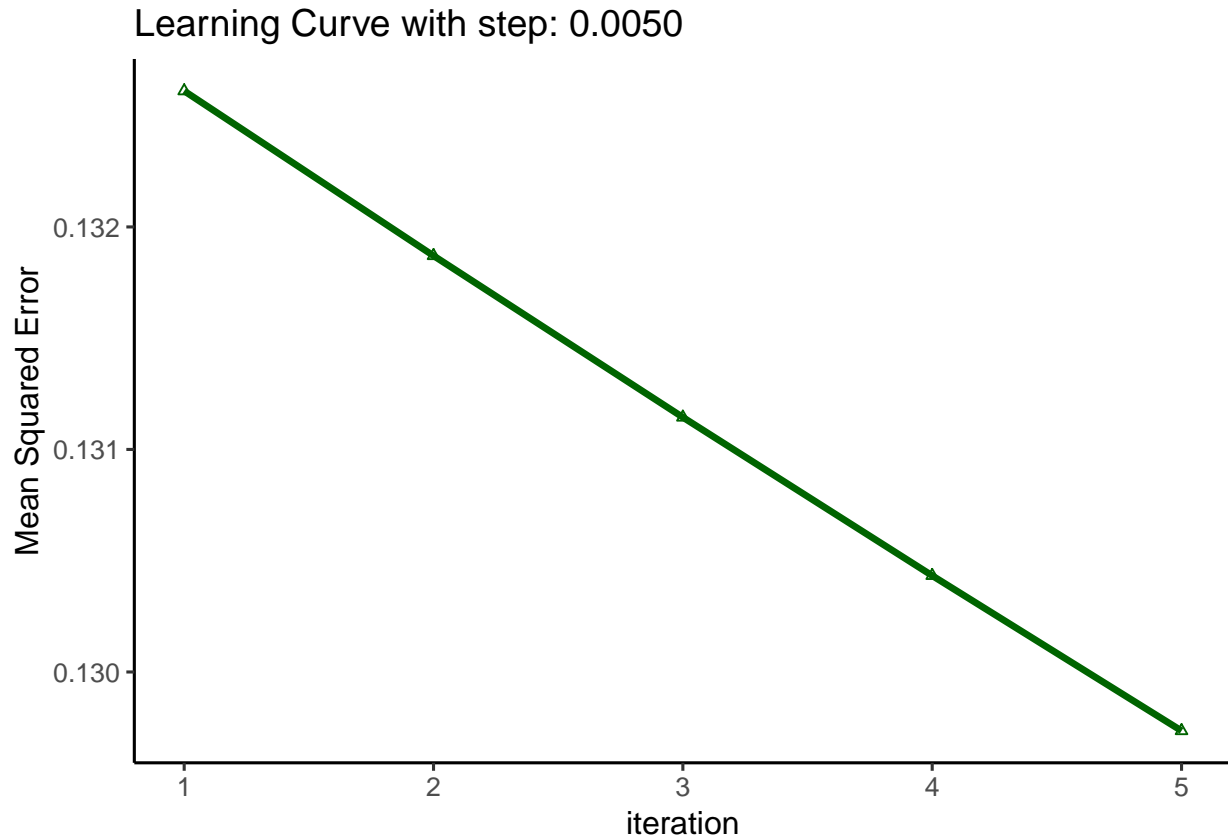
# Let's test this two times
c1 <- runif(1, min=-10, max=0)
c2 <- runif(1, min=0, max=1)
c3 <- runif(1, min=0, max=10)

```

```
# Initial  
optimizing(c1,c2,c3, 0.005, 5, TRUE)
```

```
## [1] -4.7018604 0.3111536 2.3988592
```

```
## [1] "MSE: 0.129052339894594"
```

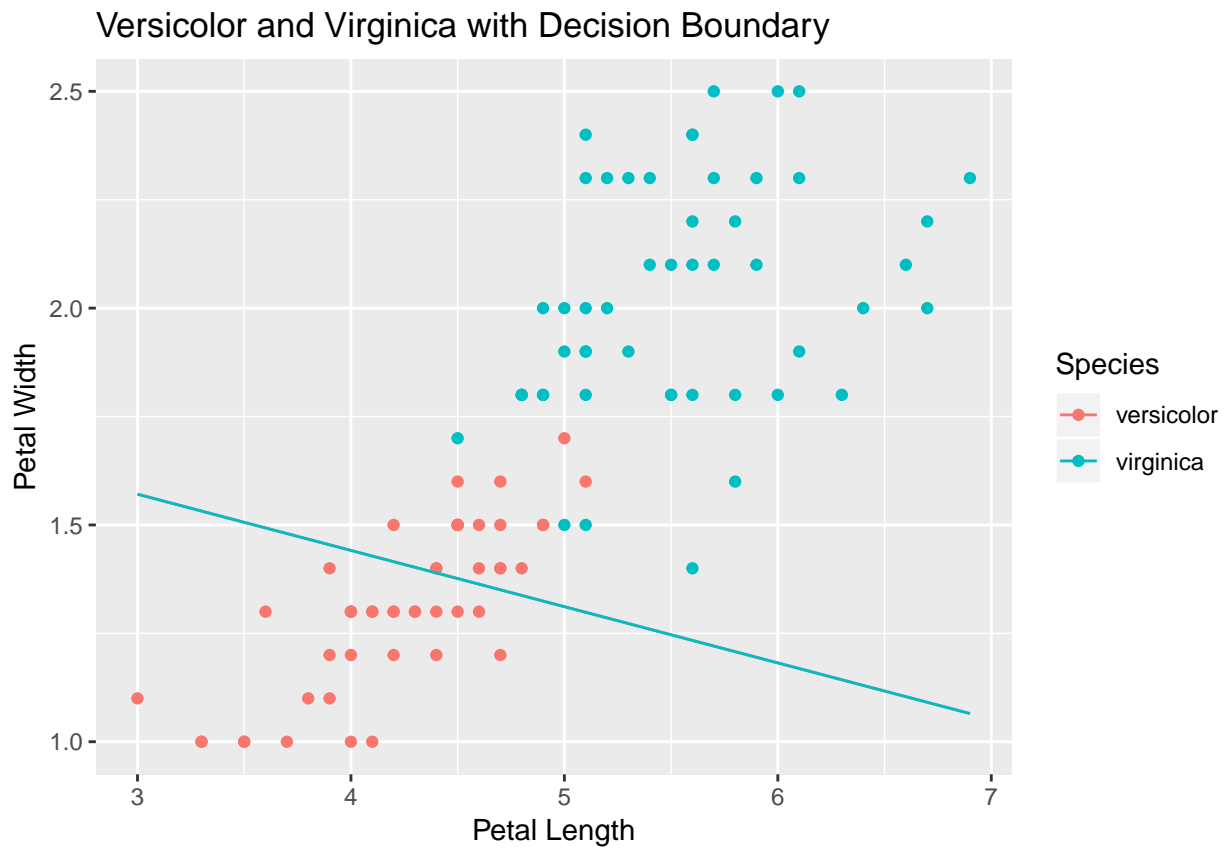


```
optimizing(c1,c2,c3, 0.005, 5, FALSE)
```

```
## [1] -4.7018604 0.3111536 2.3988592
```

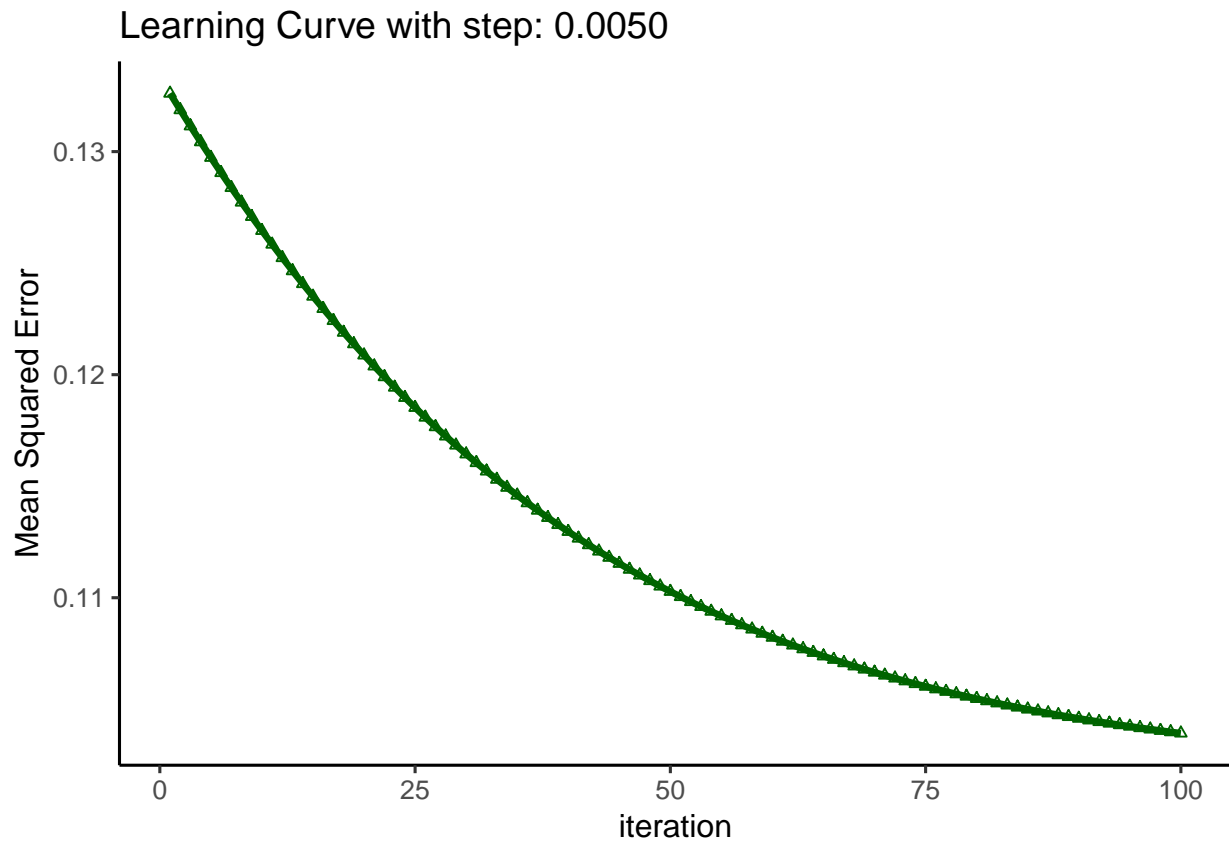
```
## [1] "MSE: 0.129052339894594"
```





```
# Middle
optimizing(c1,c2,c3, 0.005, 100, TRUE)

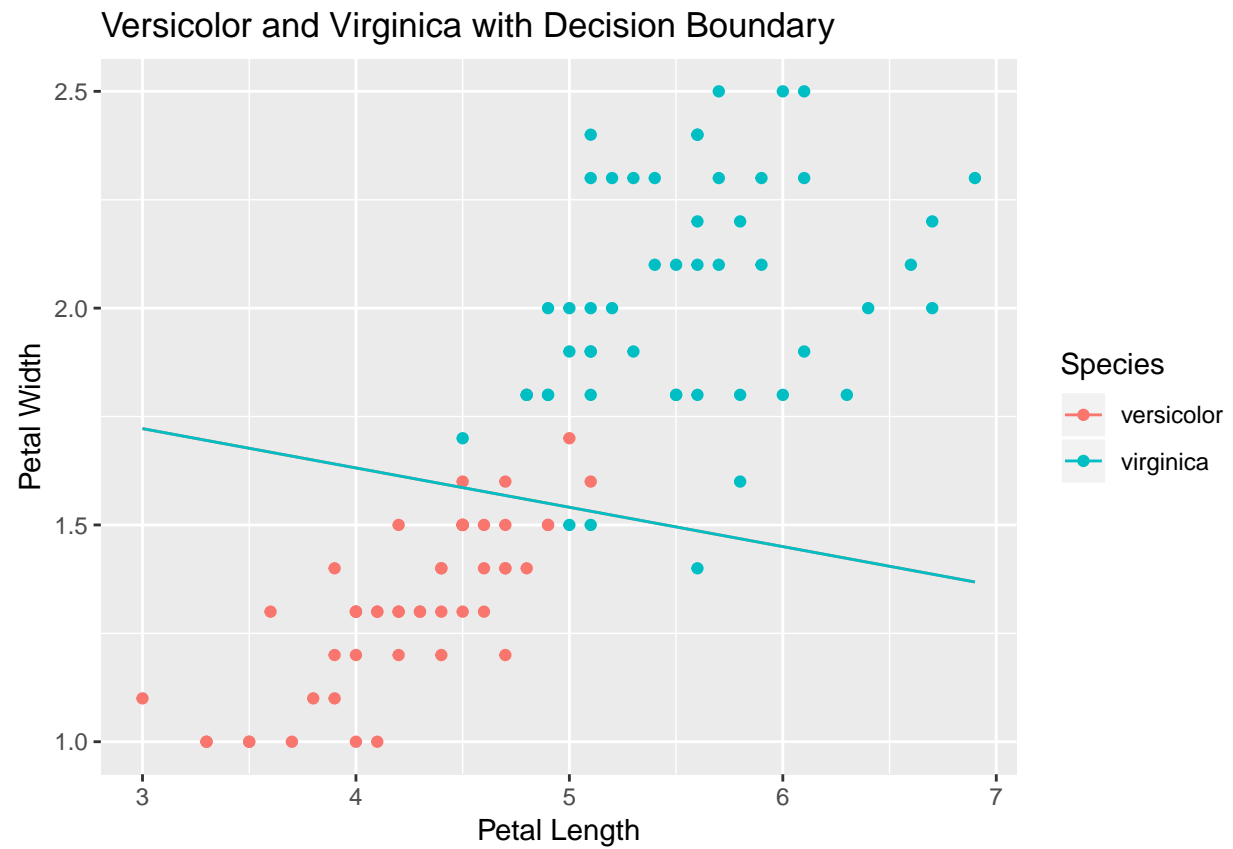
## [1] -4.7272377  0.2150118  2.3704102
## [1] "MSE:  0.103869005623496"
```



```
optimizing(c1,c2,c3, 0.005, 100, FALSE)
```

```
## [1] -4.7272377 0.2150118 2.3704102
```

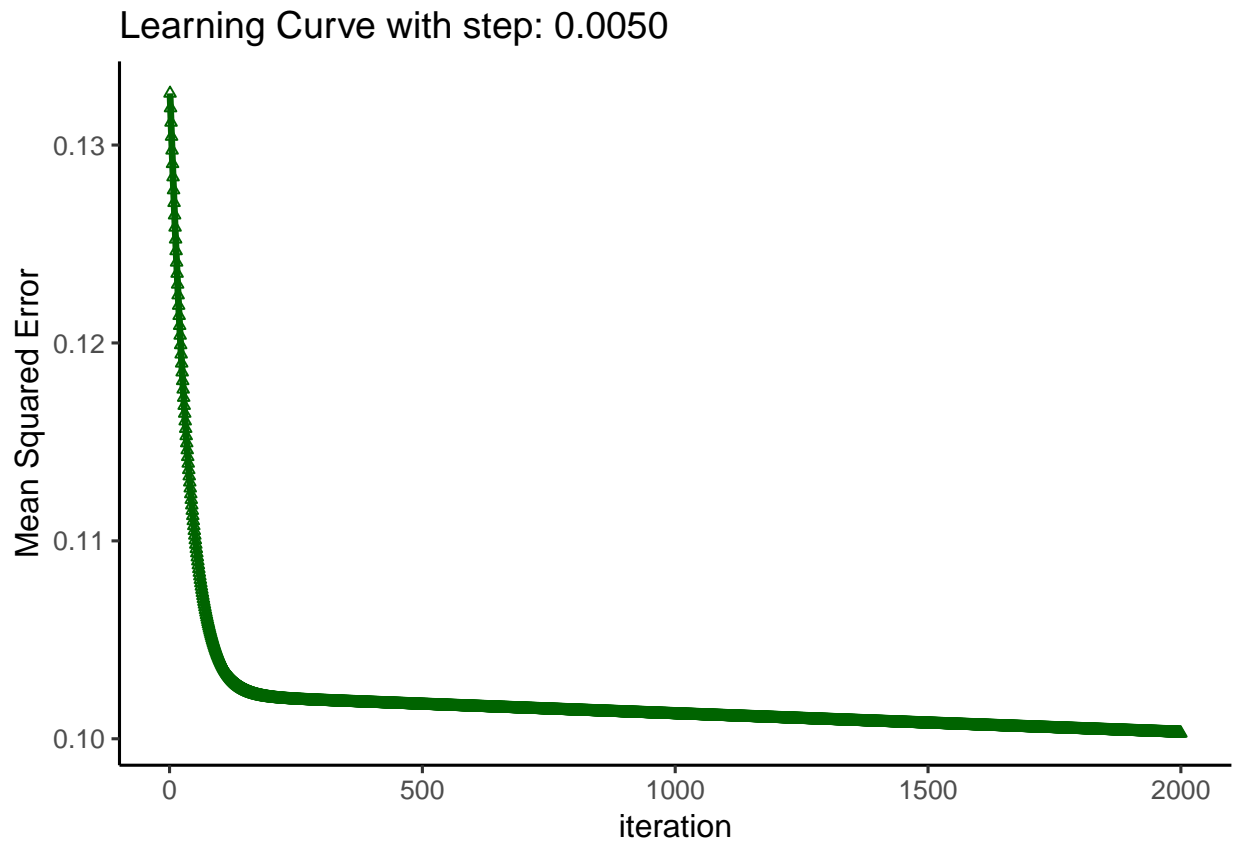
```
## [1] "MSE: 0.103869005623496"
```



```
# Final  
optimizing(c1,c2,c3, 0.005, 2000, TRUE)
```

```
## [1] -4.8456750  0.1831125  2.4276679
```

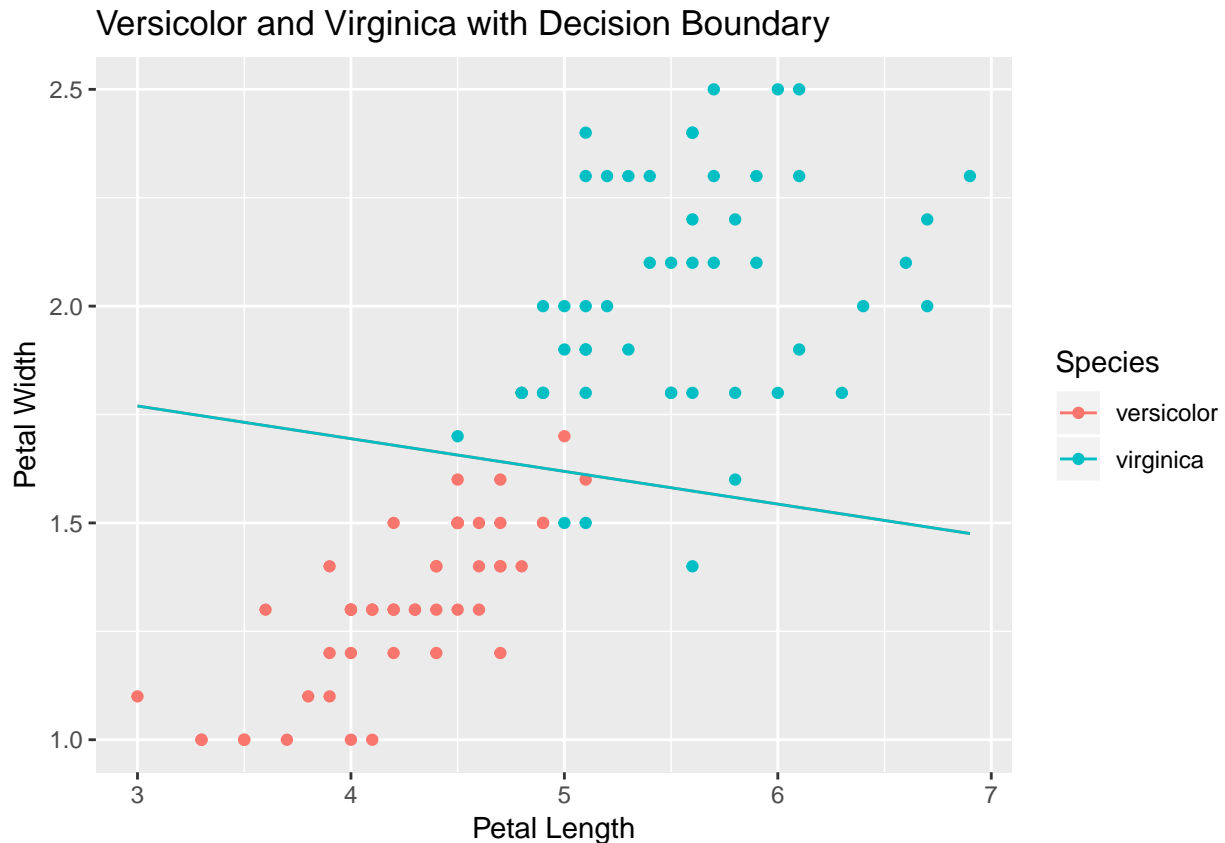
```
## [1] "MSE:  0.100279096567"
```



```
optimizing(c1,c2,c3, 0.005, 2000, FALSE)
```

```
## [1] -4.8456750 0.1831125 2.4276679
```

```
## [1] "MSE: 0.100279096567"
```



### 0.3.3 Part D

The gradient step size was chosen by trial and error. If the gradient step size is too large, the algorithm will never converge and instead it will jump around the minimum. If the gradient step size is too small, the algorithm will converge after a very long time. After experimenting with different types of learning rates, I decided on 0.005. This learning rate seemed to converge after a short time and at the same time did not get stuck on a local minimum.

### 0.3.4 Part E

I have used two different checks for the stopping criteria. The first was based on a tolerance threshold for the minimum change in the gradients. If the gradients were not changing above a threshold set to 0.00005 that the algorithm would stop. This stopping criterion is effective because the magnitude of the gradients is directly proportional to the error.

The second stopping measure was the number of iterations. If the step size is too large and the gradient keeps jumping around a minimum, this measure will force it to stop running.

## 0.4 Extra Credit

For part a, the result said that we have "Correct classifications: 93/100". For part b, the result said "Correct classifications: 141 out of 150". Code attached as a python file separately.