Programming Assignment 2 - CSDS 391

Minh Pham

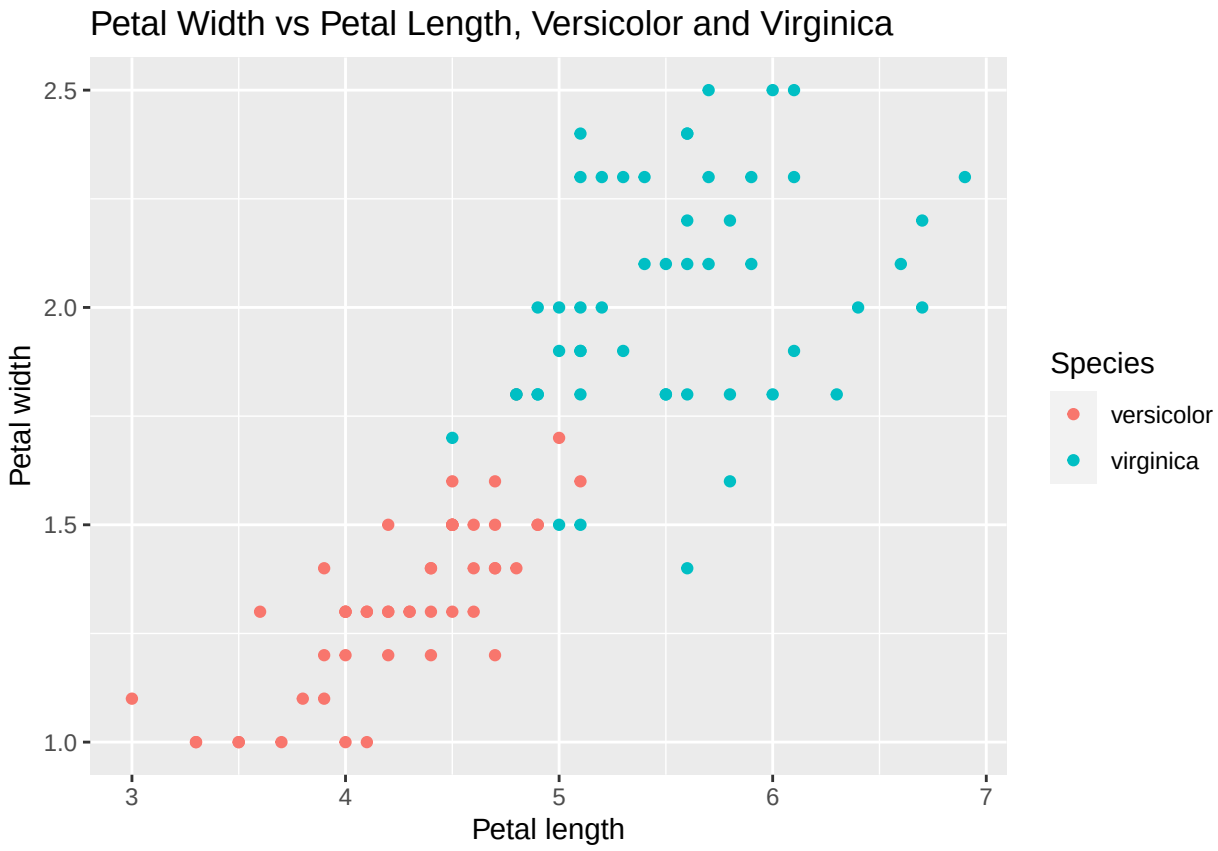
11/29/2020

Problem 1

(a)

I first load the iris data set, filter the needed data and store it in an object for later use, and make the plot.

```
irisdata <- read.csv(file = 'irisdata.csv')
plotdata <- irisdata %>% filter(species == "versicolor" | species == "virginica")
plot1 <- ggplot(plotdata, aes(petal_length, petal_width, color = species)) + geom_point() + labs(x = "P
plot1
```



(b)

To classify the data points linearly using logistic non-linearity, there are two options: using a threshold and using the sigmoid function. Since there is no clear threshold at first, I decided to use the sigmoid function, then clasify the data points using my chosen threshold, 0.5. If the computed value is smaller than 0.5, it is classified as versicolor, else it is classified as virginica.

The first function, computer_logistics, outputs a value between 0 and 1. This value is computed from the inputed weight, petal width and length, then transformed through the sigmoid function. The second function, classifier, classifies each data point based on its corresponding value from compute_logistics. 0 means versicolor, and 1 means virginica.

```
compute_logistics <- function(w0, w1, w2, petal_length, petal_width) {
  y <- w0 + w1*petal_length + w2*petal_width
  z <- 1/(1 + exp(-y))
  return (z)
}

classifier <- function(w0, w1, w2, x1, x2) {
  x <- compute_logistics(w0, w1, w2, x1, x2)
  if (x < 0.5)
    return (0)
  else
    return(1)
}
```

(c)

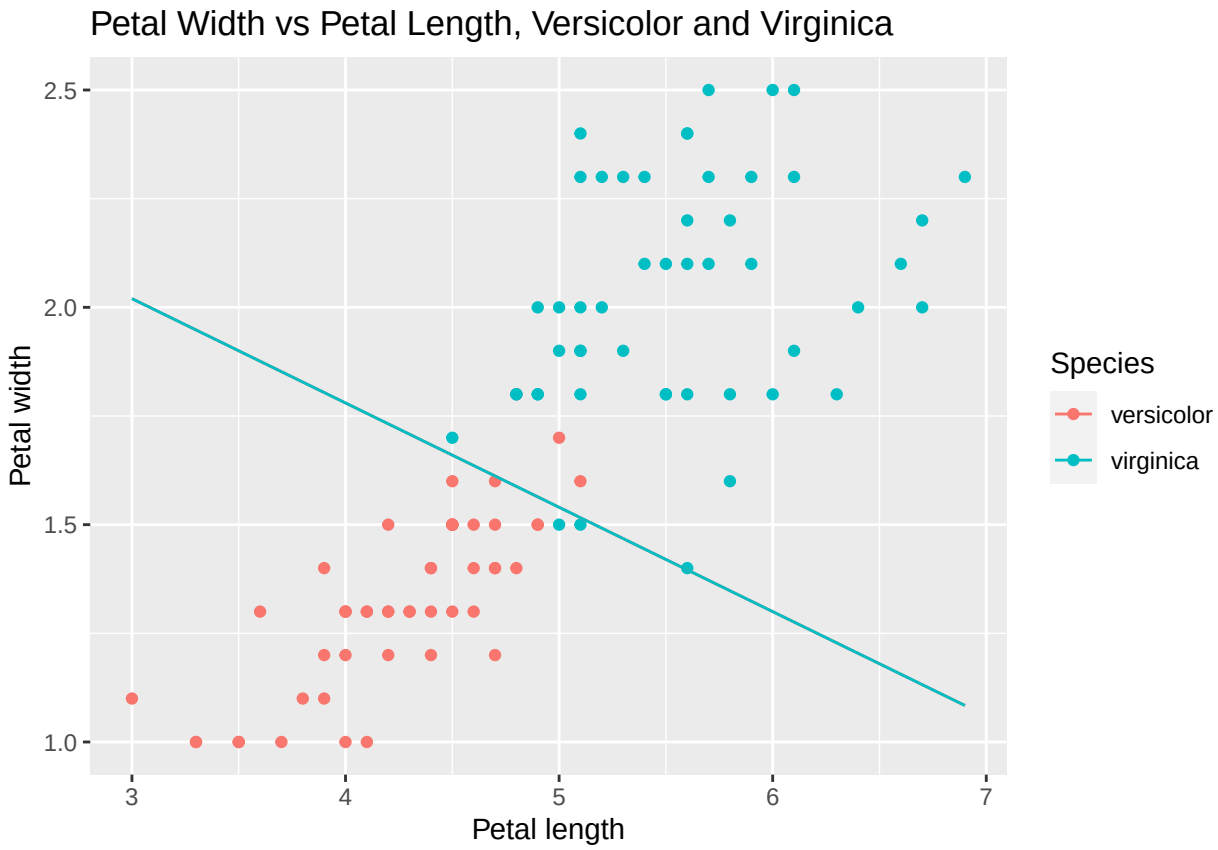
The boundary foundation computes x_2 based on inputed weights (w_0, w_1, w_2) and x_1. Creating such a function separately proved to be a right decision, as I later realized I would have to reuse this function multiple times for different inputs.

The chosen weights for my boundaries was w_0 = -2.74, w_1 = 0.24 and w_2 = 1, and as displayed in the plot, they make a relatively accurate decision boundary. These values were chosen after multiple experiments and tryouts.

```
boundary <- function(w0, w1, w2, x1) {
  x2 <- (w0 + w1*x1)/(-w2)
  return (x2)
}

plot2 <- plot1 + stat_function(fun = boundary, args = list(w0 = -2.74, w1 = 0.24, w2 = 1))
plot2
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```



(d)

I created a matrix that stores the results of values computed by the compute_logistics function, with our chosen weights and values of petal length and width running from 0 to 10. Then, I made a 3D plot of that matrix with respect to x_1 and x_2

Generating a pdf from RMarkdown does not support the display of this plot, so I will attach a picture of it in the zip file.

```
w0_f = -2.74
w1_f = 0.24
w2_f = 1

x <- seq(0, 10, by = 0.1)
y <- seq(0, 10, by = 0.1)
z <- matrix(data = NA, nrow = length(x), ncol = length(y), byrow = FALSE, dimnames = NULL)

for (i in 1:length(x)) {
  for (j in 1:length(y)) {
    z[i,j] = compute_logistics(w0_f, w1_f, w2_f, x[i], y[j])
  }
}

plot3 <- plot_ly(x = x, y = y, z = z, type = 'mesh3d') %>% add_surface() %>% layout(scene = list(xaxis =
plot3
```

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

(e)

For each data points in both classes, I printed whether if the decision made from our chosen boundary matches with the actual class. There are 48/50 data points predicted correctly in each class.

```
versicolor <- plotdata %>% filter(species == 'versicolor')
for (i in 1:50) {
  if (classifier(w0_f, w1_f, w2_f, versicolor$petal_length[i], versicolor$petal_width[i]) == 0)
      print('versicolor: matched')
  else
      print('versicolor: unmatched')
}
```

```
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: unmatched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: unmatched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: unmatched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
```

```
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
## [1] "versicolor: matched"
```

```
virginica <- plotdata %>% filter(species == 'virginica')
for (i in 1:50) {
  if (classifier(w0_f, w1_f, w2_f, virginica$petal_length[i], virginica$petal_width[i]) == 1)
      print('virginica: matched')
  else
      print('virginica: unmatched')
}
```

```
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: unmatched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
```

```
## [1] "virginica: unmatched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
## [1] "virginica: matched"
```

Problem 2

(a)

For this method, which computes the mean-squared error, the inputs are assumed to be a 2-column table called "data", which contains data about the petal length and width of each data point, a 1-column table called "class", which contains data about the class of each data point, with 0 denoting versicolor and 1 denoting virginica, and the weights of the decision boundary.

```
mean_square_error <- function(data, class, w0, w1, w2) {
  result <- rep(0, nrow(data))
  for (i in 1:nrow(data)) {
    result[i] = compute_logistics(w0, w1, w2, data$petal_length[i], data$petal_width[i])
    result[i] = (result[i] - class[i])^2
  }
  mse = mean(result)
  return (mse)
}
```

(b)

I first created the data and class tables so that they match with the desired inputs for the mean_square_error function. Then I made 2 plots, one with my chosen weight values, which I knew would produce low error, and another with random weight values. The difference is very obvious and can be seen from the plots.

```
d <- plotdata %>% select(petal_length, petal_width)
c <- rep(0, nrow(d))

for (i in 1:nrow(d)) {
  if (plotdata$species[i] == 'virginica')
    c[i] = 1
}
mean_square_error(d, c, w0_f, w1_f, w2_f)
```

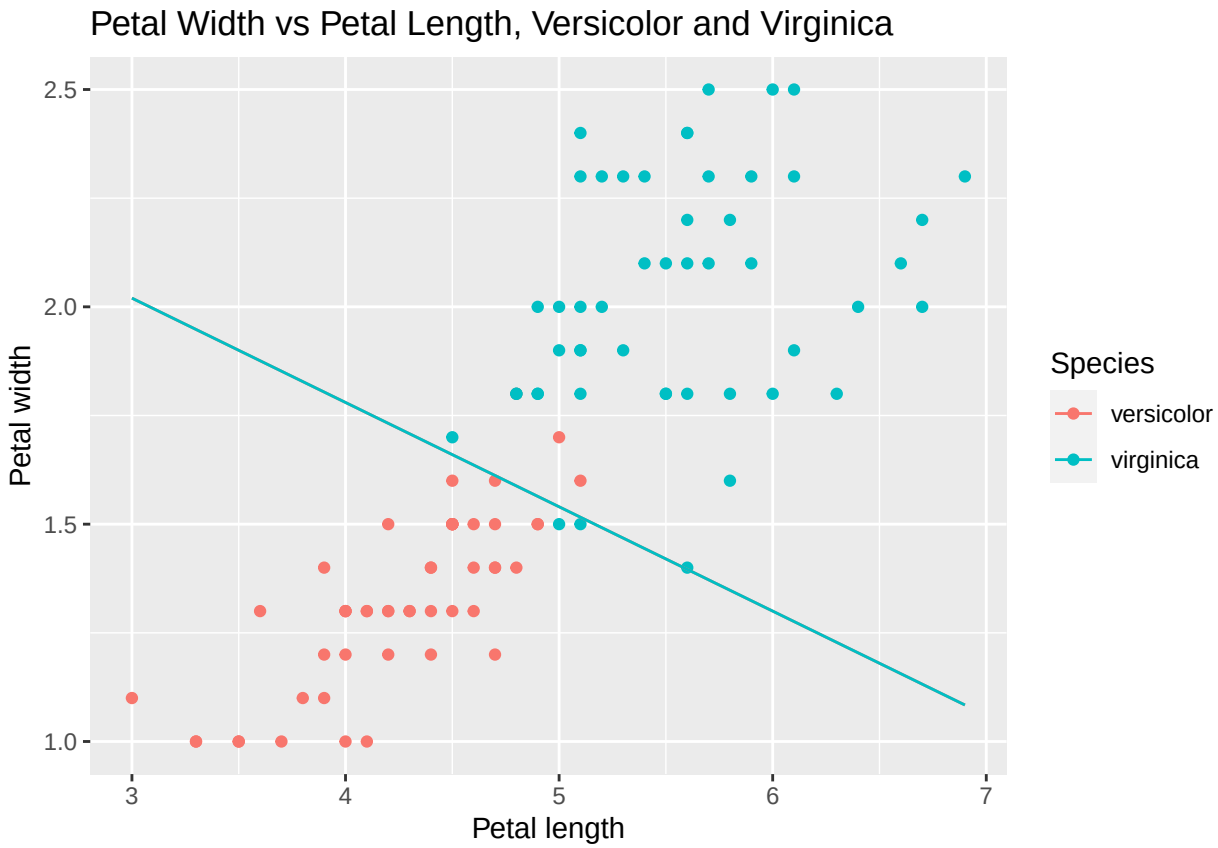
```
## [1] 0.1500215
```

```
mean_square_error(d, c, 1, 2, 3)
```

```
## [1] 0.499996
```

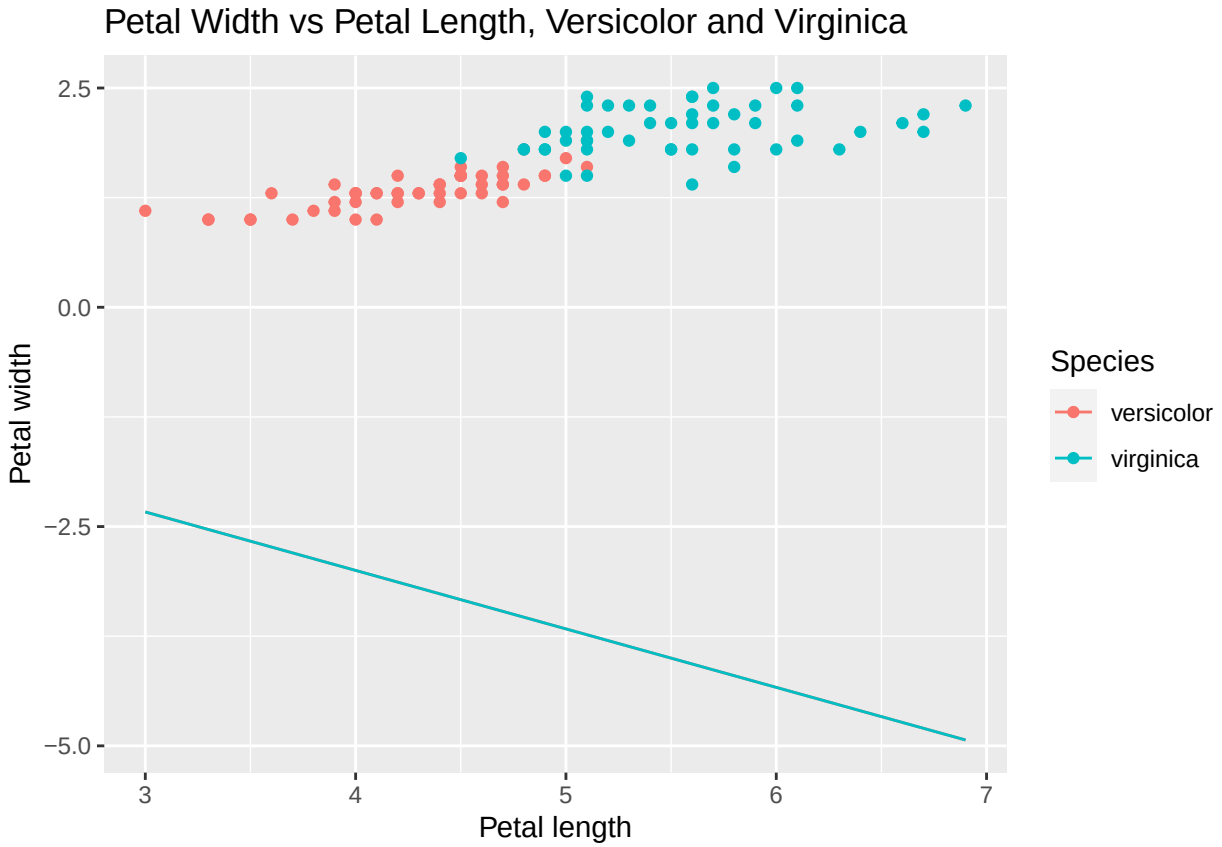
```
plot4 <- plot1 + stat_function(fun = boundary, args = list(w0 = w0_f, w1 = w1_f, w2 = w2_f))
plot5 <- plot1 + stat_function(fun = boundary, args = list(w0 = 1, w1 = 2, w2 = 3))
plot4
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```




```
plot5
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```



(c)

The mean-squared error expression is defined to be as follow:

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

where $\mathbf{w}^T = [w_0, w_1, w_2]$, and $\mathbf{x}_i = [1, x_{i1}, x_{i2}]$.

Taking the derivative with respect to w_0 gives us:

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

From the slides, we have $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. In addition, $\frac{\partial(\mathbf{w}^T\mathbf{x}_i)}{\partial w_0} = \frac{\partial}{\partial w_0}(w_0 + w_1x_{i1} + w_2x_{i2}) = 1$. Therefore, our expression can be simplified into the following:

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

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

$$\frac{\partial MSE}{\partial w_1} = \frac{2}{n}\sum_{i=1}^{n}[\sigma(\mathbf{w}^T\mathbf{x}_i) - c_i][\sigma(\mathbf{w}^T\mathbf{x}_i)][1 - \sigma(\mathbf{w}^T\mathbf{x}_i)]x_{i1}$$

$$\frac{\partial MSE}{\partial w_2} = \frac{2}{n}\sum_{i=1}^{n}[\sigma(\mathbf{w}^T\mathbf{x}_i) - c_i][\sigma(\mathbf{w}^T\mathbf{x}_i)][1 - \sigma(\mathbf{w}^T\mathbf{x}_i)]x_{i2}$$

The x_{i1} and x_{i2} appears in the expressions 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:

$$\nabla MSE = \begin{bmatrix} \frac{\partial MSE}{\partial w_0} \\ \frac{\partial MSE}{\partial w_1} \\ \frac{\partial MSE}{\partial w_2} \end{bmatrix}$$

(d)

(I wasn't entirely sure what this question was asking, so I asked an alum who took this class before, she explained the concept to me and I tried adapting her idea into mine)

We have shown in part (c) that the gradient can be written in the scalar form. In this part, we will combine the derivatives of the objective function to create the vector form.

Notice that there is a common factor in each componet of the gradient vector. We will denote z_i as that, and factor it out so it is easier to work with:

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

Then the gradient becomes the following:

$$\nabla MSE = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{n} z_i \\ \sum_{i=1}^{n} z_i x_{i1} \\ \sum_{i=1}^{n} z_i x_{i2} \end{bmatrix} = \frac{2}{n}\begin{bmatrix} 1 & \dots & 1 \\ x_{11} & \dots & x_{n1} \\ x_{12} & \dots & x_{n2} \end{bmatrix}\begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = \frac{2}{n}X^T\begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$$

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

$$\begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} [\sigma(\mathbf{w}^T\mathbf{x}_1) - c_1][\sigma(\mathbf{w}^T\mathbf{x}_1)][1 - \sigma(\mathbf{w}^T\mathbf{x}_1)] \\ \vdots \\ [\sigma(\mathbf{w}^T\mathbf{x}_n) - c_n][\sigma(\mathbf{w}^T\mathbf{x}_n)][1 - \sigma(\mathbf{w}^T\mathbf{x}_n)] \end{bmatrix} = \begin{bmatrix} \sigma(\mathbf{w}^T\mathbf{x}_1) - c_1 \\ \vdots \\ \sigma(\mathbf{w}^T\mathbf{x}_n) - c_n \end{bmatrix} * \begin{bmatrix} \sigma(\mathbf{w}^T\mathbf{x}_1) \\ \vdots \\ \sigma(\mathbf{w}^T\mathbf{x}_n) \end{bmatrix} * \begin{bmatrix} 1 - \sigma(\mathbf{w}^T\mathbf{x}_1) \\ \vdots \\ 1 - \sigma(\mathbf{w}^T\mathbf{x}_n) \end{bmatrix}$$

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

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

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

$$\nabla MSE = \frac{2}{n}X^T\left[(\sigma(X\mathbf{w}) - \mathbf{c}) * \sigma(X\mathbf{w}) * (\mathbf{1} - \sigma(X\mathbf{w}))\right]$$

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

(e)

This function computes the gradient vector given the data, class and weights of boundary function. It uses the expression derived in part (c)

```
gradient_mse <- function(data, class, w0, w1, w2) {
  dMSE_dw <- c(0, 0, 0)
  for (i in 1:nrow(data)) {
    x_i1 = data$petal_length[i]
    x_i2 = data$petal_width[i]
    c_i = class[i]

    y_i = compute_logistics(w0, w1, w2, x_i1, x_i2)

    dMSE_dw[1] = dMSE_dw[1] + (2/nrow(data)) * (y_i-c_i) * y_i * (1-y_i)
    dMSE_dw[2] = dMSE_dw[2] + (2/nrow(data)) * (y_i-c_i) * y_i * (1-y_i) * x_i1
    dMSE_dw[3] = dMSE_dw[3] + (2/nrow(data)) * (y_i-c_i) * y_i * (1-y_i) * x_i2
  }
  return (dMSE_dw)
}
```

I wanted to be consistent and continue with the example of w = [1, 2, 3] as above. Only after conducting multiple tryouts of different values did I realize this wasn't a good example. At first, since I knew this choice of decision boundary is very far from being accurate, I had to take big steps to move it closer to the peak (if the level of accuracy of the decision boundary was a bell-like curve, w = [1, 2, 3] would be the point at the very beginning, where there's rarely any change unless you take big steps towards the peak). This is what the first for loop with step size 500 does.

After seeing that it is now closer to the peak, where you have to be more careful about the step size if you don't want to pass the peak, I reduced the step size and increase the number of iterations. After the second for loop, we are now very close to the peak, so I reduced the step size by a significant amount and increase the number of iterations to very large so we can get as close to the peak as possible. The improvements in the level of accuracy can also be observed through printed errors and the plots.

```
w <- c(1, 2, 3)
print(paste("MSE: ", mean_square_error(d, c, w[1],w[2],w[3])))
```

```
## [1] "MSE:  0.499995978537485"
```

```
plot6 <- plot1 + stat_function(fun = boundary, args = list(w0 = w[1], w1 = w[2], w2 = w[3]))
plot6
```

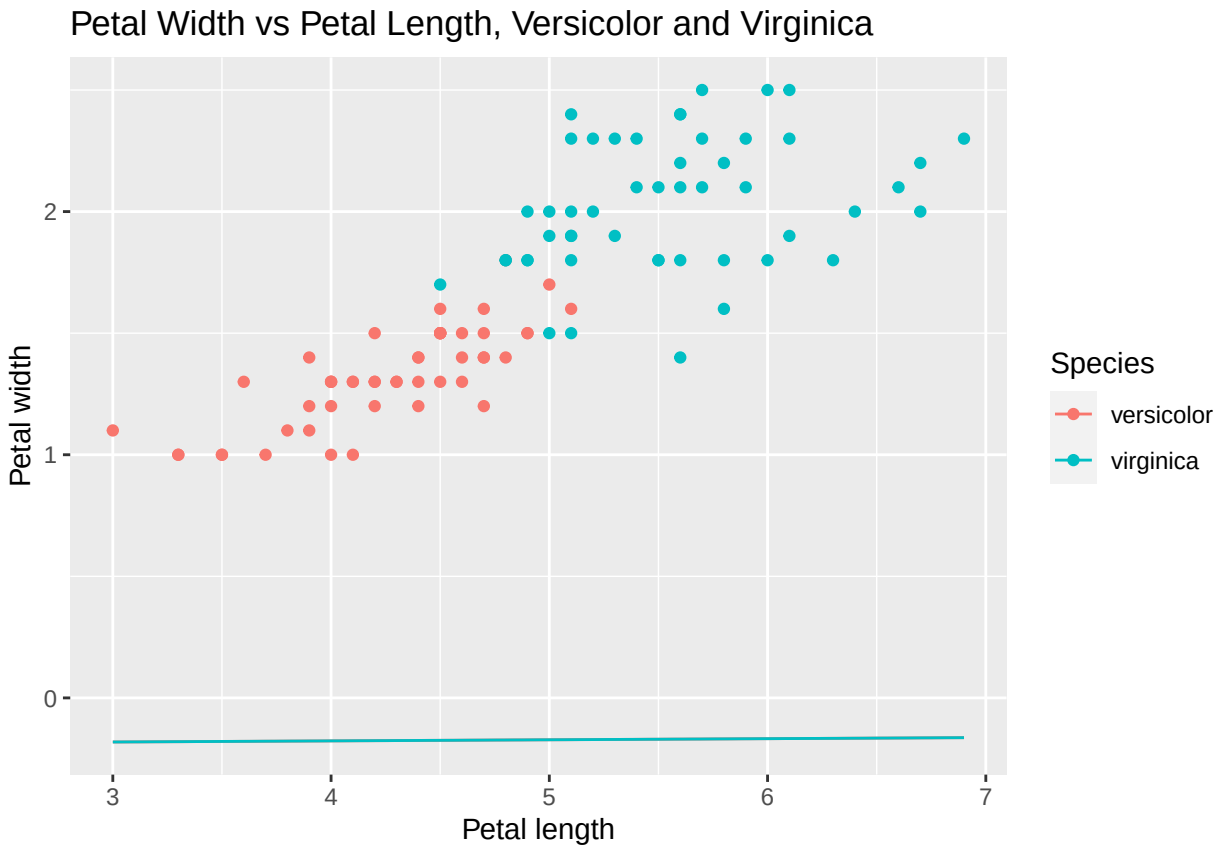
```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```


```
for (i in 1:35) {
  w <- w - 500 * gradient_mse(d, c, w[1], w[2], w[3])
}
print(paste("MSE: ", mean_square_error(d, c, w[1],w[2],w[3])))
```

```
## [1] "MSE:  0.470905375909084"
```

```
plot6 <- plot1 + stat_function(fun = boundary, args = list(w0 = w[1], w1 = w[2], w2 = w[3]))
plot6
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```

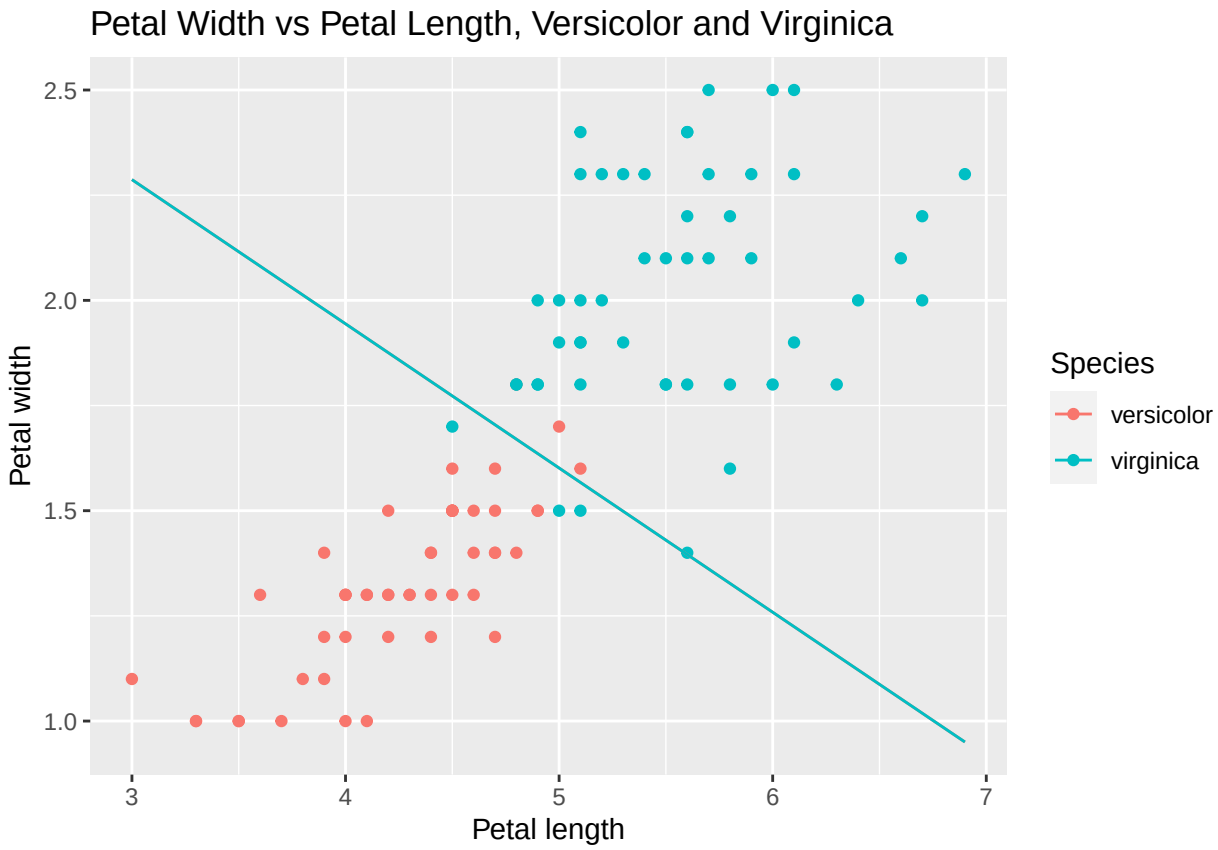



```
for (i in 1:5000) {
  w <- w - 1 * gradient_mse(d, c, w[1], w[2], w[3])
}
print(paste("MSE: ", mean_square_error(d, c, w[1],w[2],w[3])))
```

```
## [1] "MSE:  0.0404719926440933"
```

```
plot6 <- plot1 + stat_function(fun = boundary, args = list(w0 = w[1], w1 = w[2], w2 = w[3]))
plot6
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```

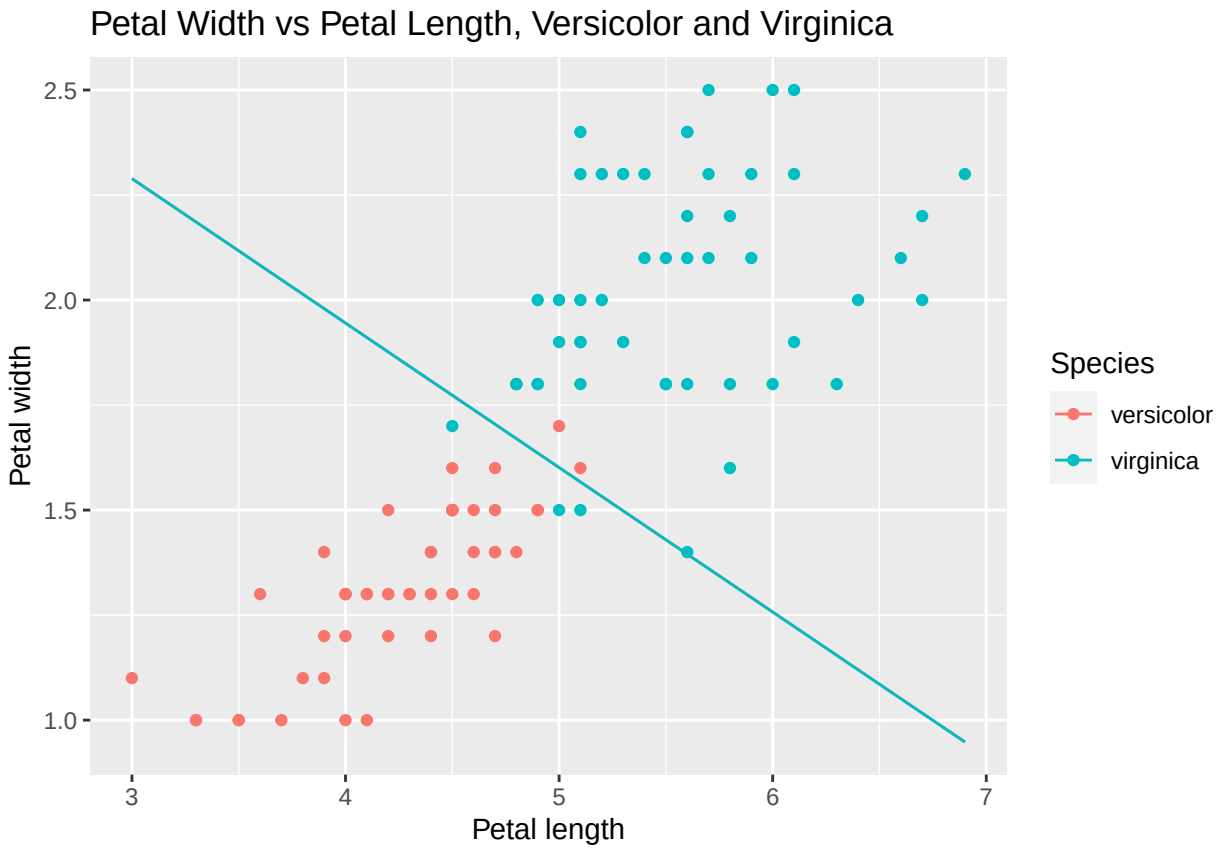



```
for (i in 1:20000) {
  w <- w - 0.002 * gradient_mse(d, c, w[1], w[2], w[3])
}
print(paste("MSE: ", mean_square_error(d, c, w[1],w[2],w[3])))
```

```
## [1] "MSE:  0.0404166177576363"
```

```
plot6 <- plot1 + stat_function(fun = boundary, args = list(w0 = w[1], w1 = w[2], w2 = w[3]))
plot6
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```



Problem 3

(a) and (b)

This function takes the weights, number of steps and number of iterations as inputs, then continuously updates the weights based on the calculated gradient. In lecture slides, gradient descent was defined to be w_i <- w_i + step $*$ gradient (with a plus), however, in the textbook, gradient descent is defined to be w_i <- w_i - step $*$ gradient (with a minus). The textbook definition made more sense to me intuitively so I went with that.

This function also makes two plots, one of the plotted data with the decision boundary, and one with the learning curve.

```
optimizer <- function(w0, w1, w2, step, numIteration) {
  w <- c(w0, w1, w2)
  mse <- rep(0, numIteration)
  ite <- seq(1, numIteration, by = 1)

  for (i in 1:numIteration) {
    if (i == 1)
      prev <- c(0, 0, 0)

    if (abs(sum(w)- sum(prev)) < 0.0001)
      break;
```

```
    mse[i] = mean_square_error(d, c, w[1], w[2], w[3])
    prev <- w
    w <- w - step * gradient_mse(d, c, w[1], w[2], w[3])
  }

  print(w)
  print(paste("MSE: ", mean_square_error(d, c, w[1],w[2],w[3])))

  plot_a <- plot1 + stat_function(fun = boundary, args = list(w0 = w[1], w1 = w[2], w2 = w[3]))
  plot_b <- ggplot(data.frame(ite, mse), aes(x = ite, y = mse)) + geom_point()
  ggarrange(plot_a, plot_b, ncol = 1, nrow = 2, heights = c(3,2), align = "hv")
}
```

(c)

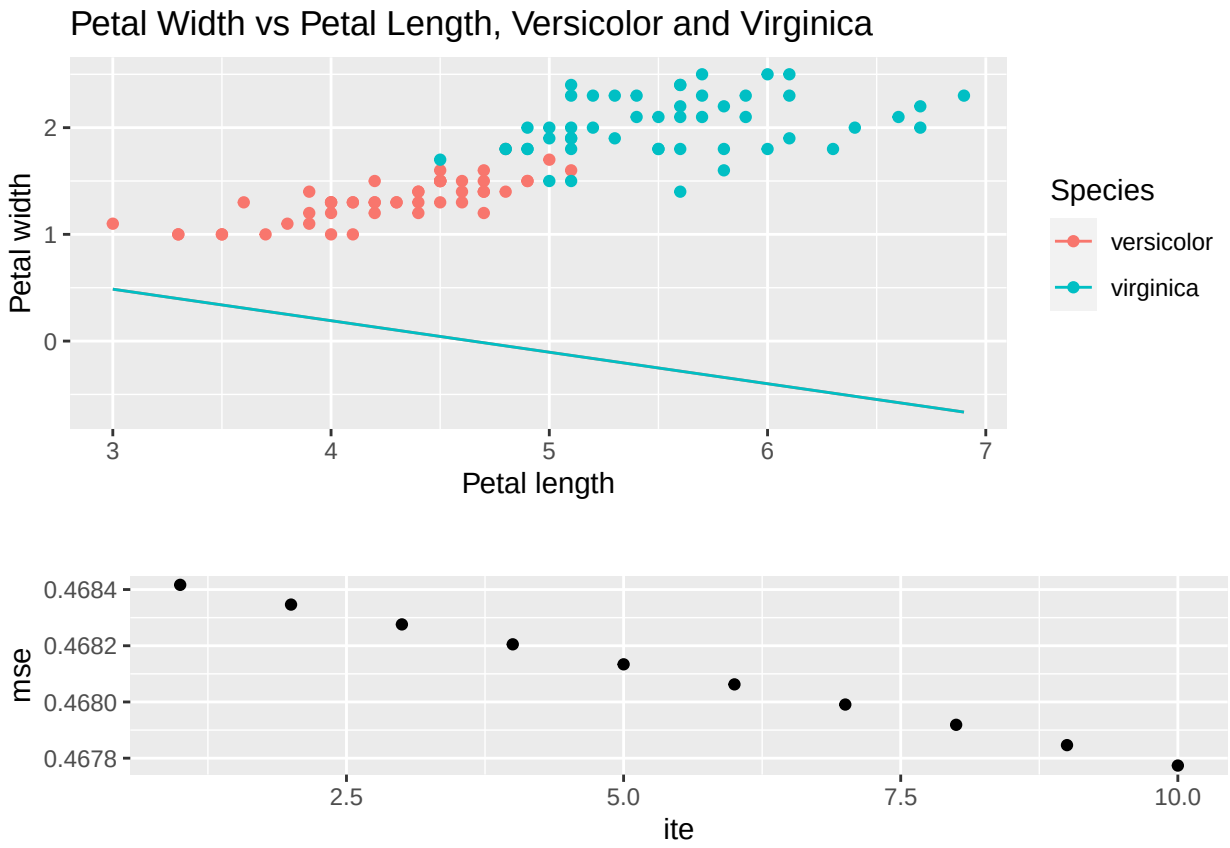
Knowing how a relatively accurate decision boundary should look like, I limited the degree of randomness of the weights so they fit into the plot. You can tell by looking at the printed mean-squared error and the position of the decision boundary that the decision is more and more refined.

```
randW0 <- runif(1, min=-5, max=0)
randW1 <- runif(1, min=0, max=1)
randW2 <- runif(1, min=0, max=5)
optimizer(randW0, randW1, randW2, 0.005, 10)
```

```
## [1] -4.3330786  0.9324933  3.1579452
## [1] "MSE:  0.467700825359484"
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```

```
## Warning: Graphs cannot be vertically aligned unless the axis parameter is set.
## Placing graphs unaligned.
```

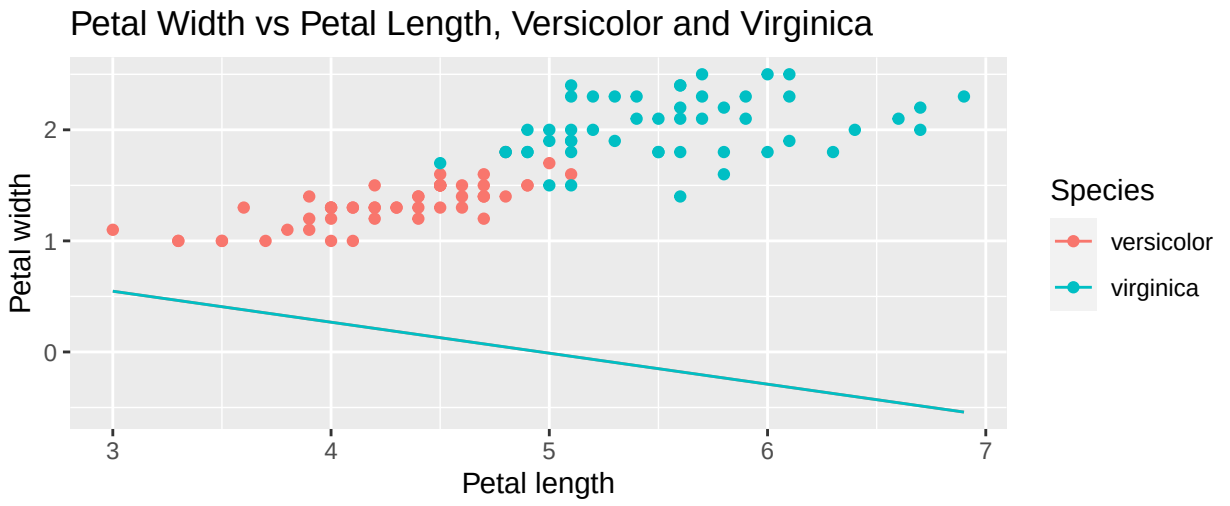


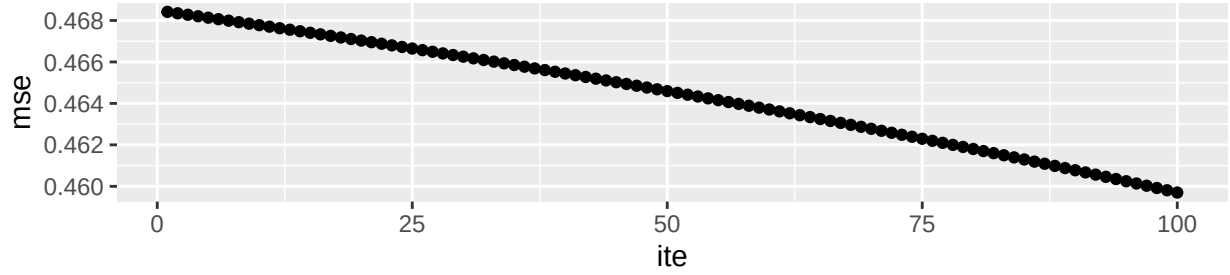

```
optimizer(randW0, randW1, randW2, 0.005, 100)
```

```
## [1] -4.3474901  0.8764614  3.1410423
## [1] "MSE:  0.45958892135694"
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```

```
## Warning: Graphs cannot be vertically aligned unless the axis parameter is set.
## Placing graphs unaligned.
```



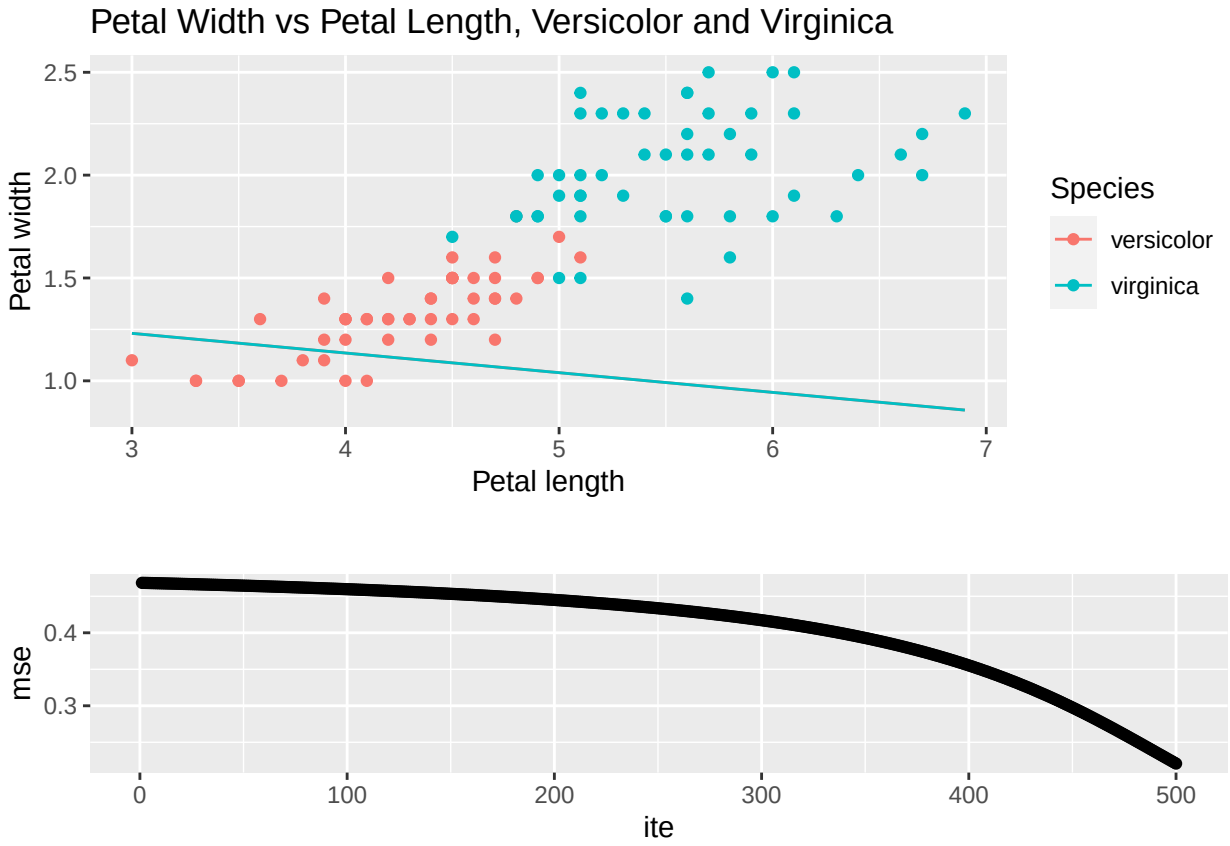



```
optimizer(randW0, randW1, randW2, 0.005, 500)
```

```
## [1] -4.493036  0.283177  2.959990
## [1] "MSE:  0.219284704593743"
```

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```

```
## Warning: Graphs cannot be vertically aligned unless the axis parameter is set.
## Placing graphs unaligned.
```



(d)

I chose the gradient step size based on multiple tryouts of different values. I first plugged in 0.05, which was too big since I could tell from looking at the learning curve that it never converges, i.e. reach the peak. I then tried 0.001, which was a significantly smaller step size, and the improvement wasn't visible. I realized that this could be that the step size is so small that it would take lots of iterarions to converge. I then adjusted the step size and was happy with 0.005.

(e)

I had two different checks for the stopping criteria. The first was based on the minimum change in the gradient vector. If the sum in the change of the gradient vector is smaller than 0.0001, meaning the improvement is almost insignificant, the algorithm will stop. This stopping criterion is effective because without it, the algorithm will keep landing on the same spot without knowing it has converged. This check is included in the code.

The second stopping measure was that if the step size is too large and the gradient keeps jumping around a minimum, the algorithm will be forced to stop. This can also be seen from the learning curve, since when this happens, the learning curve will fluctuate instead of gradually decreasing.

Problem 4

The code is included as a .py file in the zip file. As I found a good tutorial on how to use keras to train models, I decided to use the keras toolbox. I will include the tutorial that I based my work on in the zip file as well. Since the tutorial was made a few years ago and the framework for backend that I chose, TensorFlow,

has now changed, I slightly modified it and included comments in the code to explain the process and my neural network model structure.

Using the sequential model, the sigmoid as an activation function and rmsprop as the optimizer, I saw a 96% accuracy in the training process and a 92% in the validation process after 2000 epochs. I will include the screenshot of results and the plot in the zip file.

In the code, I included comments on why I chose rmsprop as my optimizer, and for my tutorial, I will explore the different non-linearities the keras toolbox has to offer, and why I decided to choose the sigmoid function.

In keras, non-linearities serves as another layer in the neural network (called "activation layer", or "activation function"). The available activation functions are:

- **relu** function: Applies the rectified linear unit fucnction. What this means is thatthe relu function will output the input directly if it is positive, otherwise, it will output zero. The relu function works well in the scenario that the gradient vanishes after trials, but for our purpose, it is not what we are looking for.
- **sigmoid** function: Sigmoid activation function, $sigmoid(x) = \frac{1}{1+exp(-x)}$. Basically, it takes an input and outputs a value between 0 and 1.
- **softmax** function: Softmax converts a real vector to a vector of categorical probabilities. The elements of the output vector are in range (0, 1) and sum to 1, and the i^{th} element is calculated as follows: $softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_i}}$. It is somewhat similar to the sigmoid function, but it is used for vectors, and we are dealing with scalars.
- **softplus** and **softsign** function: Just by looking at the formula, we can tell that these functions won't work. $softplus(x) = log(exp(x)+1)$, and $softsign(x) = \frac{x}{abs(x)+1}$. They do not convey the probabilistic meaning behind each computed value from the data and weights.
- **tanh** function: A hyperbolic tangent activation function, so although it returns a value between 0 and 1, it is irrelevant to our purpose.
- **selu** and **elu** function: The Scaled Exponential Linear Unit (SELU) activation function returns $scale*x$ if $x < 0$ and $scale * alpha * (exp(x) - 1)$ if $x > 0$, with scale and alpha being predefined constants. Again, it is irrelevant to our purpose. The elu function is similar, except for slightly modified formula for the output.
- **exponential** function: It simply return $exp(x)$.

Based on what I have research, I wrote this short explanation as to why I chose the sigmoid function to be the non-linearity in my model. This can serve as a tutorial for people who are new to keras and don't know which activation function to select.