EECS 391 Programming Assignment Two

Will Koehrsen

December 9, 2017

Contents

1 Linear Decision Boundaries

1.1 A. Inspection and Plotting

The first step is to inspect the iris data set. I can load in the data into a dataframe and perform some simple explorations.

```
# Load in data into a dataframe
iris <- read_csv('irisdata.csv')
```

```
## Parsed with column specification:
## cols(
##   sepal_length = col_double(),
##   sepal_width = col_double(),
##   petal_length = col_double(),
##   petal_width = col_double(),
##   species = col_character()
## )
```

```
# Structure of the data
str(iris, give.attr = FALSE)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    150 obs. of  5 variables:
##  $ sepal_length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ sepal_width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ petal_length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ petal_width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ species     : chr  "setosa" "setosa" "setosa" "setosa" ...
```

We can see there are 150 observations of 5 different variables. The variables are sepal length, sepal width, petal length, petal width, and species. For the iris data set, the four numerical measurements are the features and the species is the label.

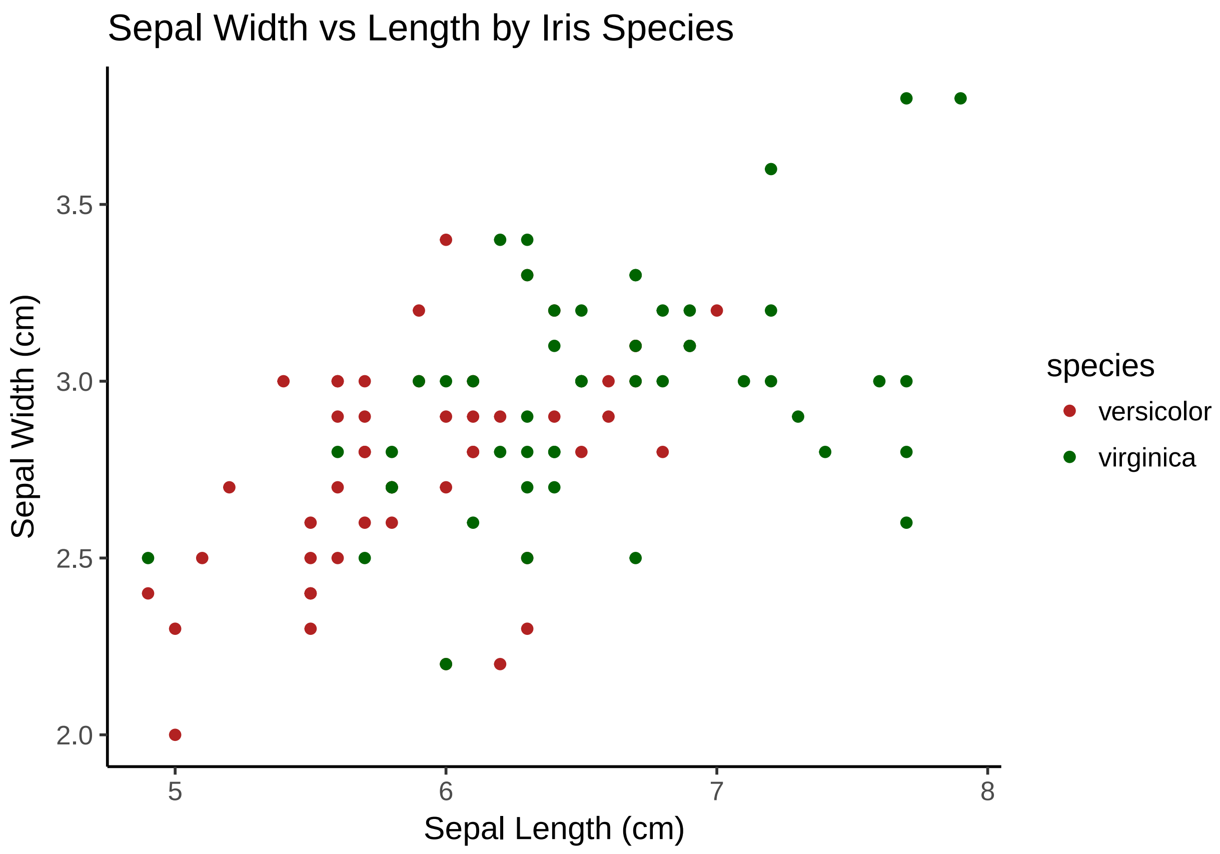
```
table(iris$species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

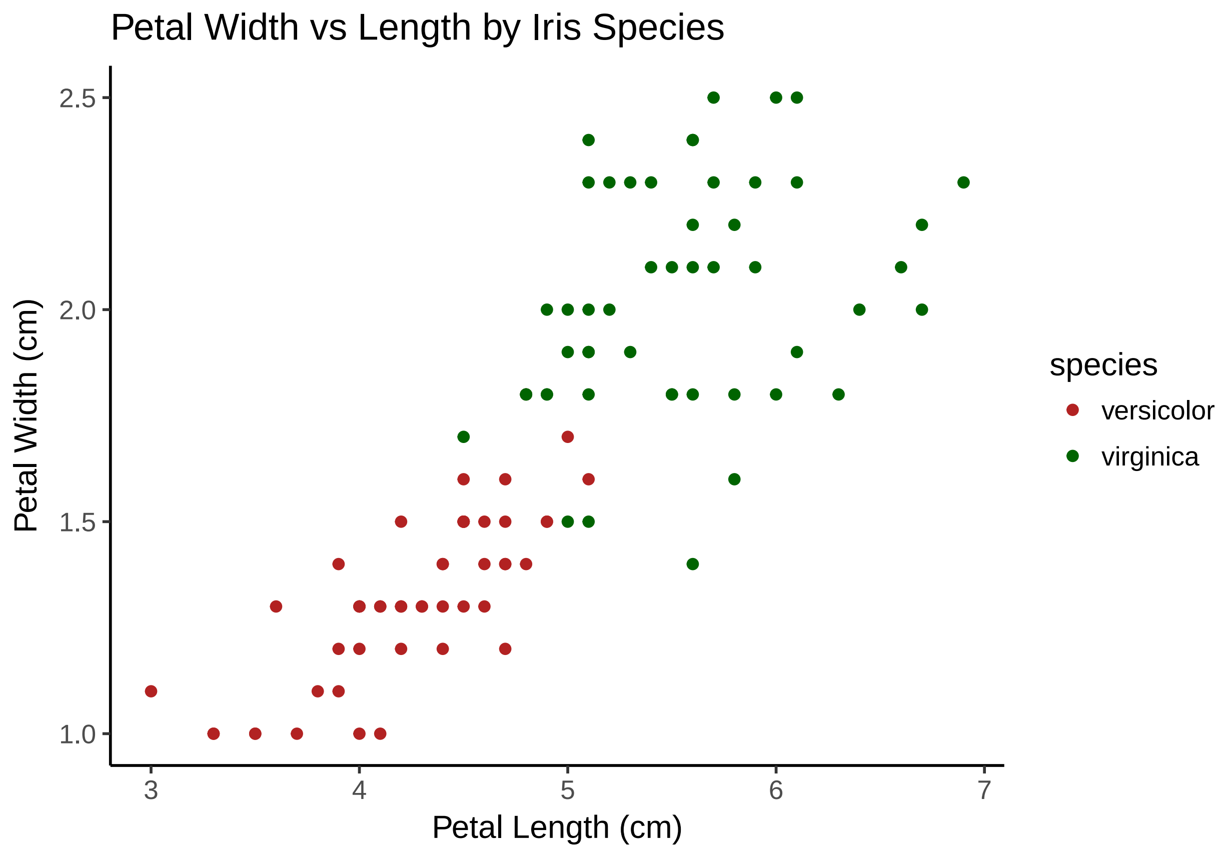
There are three species, each with 50 observations.

Next, we can plot the variables to see the ranges within each species. We are asked to plot only the second and third classes, veriscolor and virginica.

```
# Create a subset with only the relevant classes
iris_subset <- dplyr::filter(iris, species == 'versicolor' | species == 'virginica')

# Plot the sepal features colored by class
ggplot(iris_subset, aes(x = sepal_length, y = sepal_width, color = species)) +
  geom_point() + xlab('Sepal Length (cm)') +
  ylab('Sepal Width (cm)') +
  ggtitle('Sepal Width vs Length by Iris Species') + theme_classic(12) +
  scale_color_manual(values = c('firebrick', 'darkgreen'))
```



```r
# Plot the petal features colored by class
ggplot(iris_subset, aes(x = petal_length, y = petal_width, color = species)) +
  geom_point() + xlab('Petal Length (cm)') +
  ylab('Petal Width (cm)') +
  ggtitle('Petal Width vs Length by Iris Species') + theme_classic(12) +
  scale_color_manual(values = c('firebrick', 'darkgreen'))
```



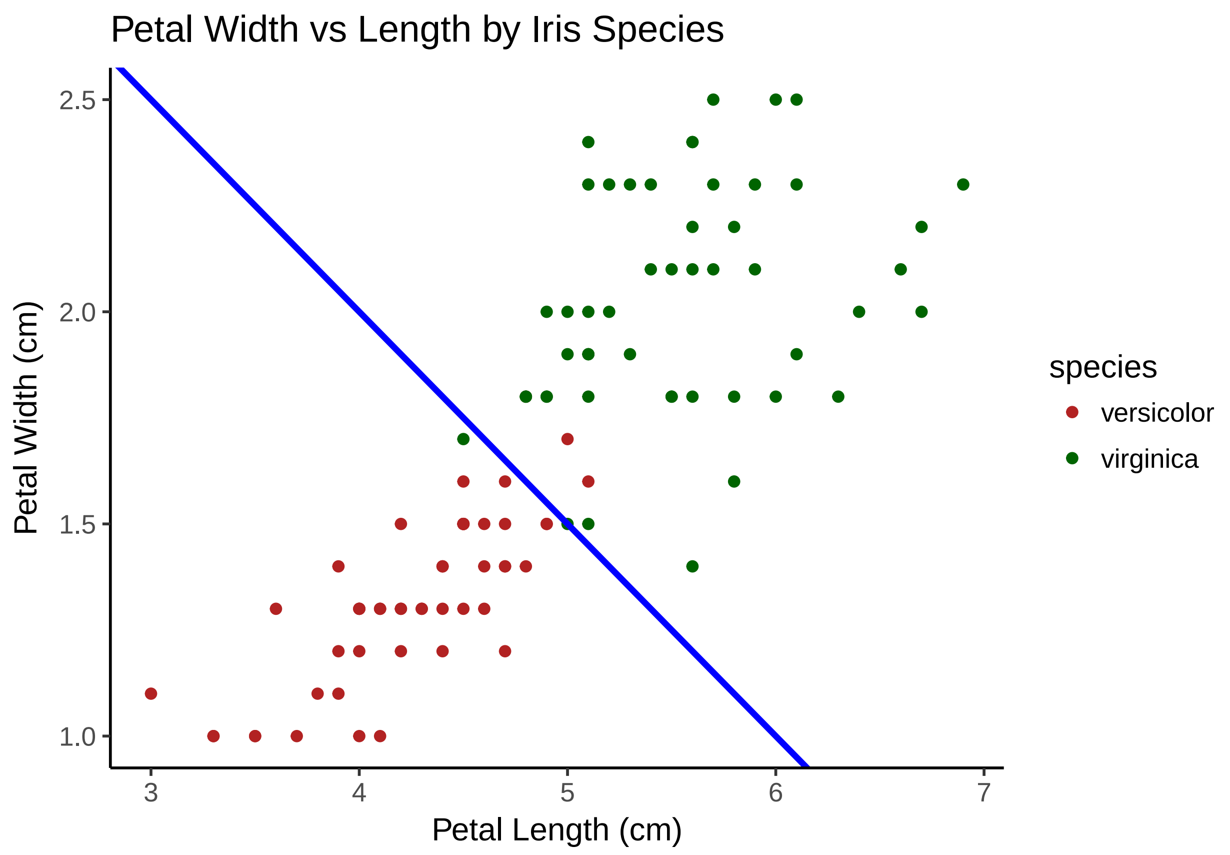
We can see that there is clearly a line that can almost completely separate the two iris classes on the plot of Petal Length vs Width. A simple linear decision boundary will have decent accuracy in separating the versicolor and virginica classes based on petal characteristics, but will not achieve 100% accuracy.

1.2 B. Plot A Linear Decision Boundary

The following function plots a linear decision boundary over the data. The slope and intercept are chosen by inspecting the data and trying to choose a line to minimize the classification error.

```r
# Function to plot a linear decision boundary given slope and intercept
plot_db <- function(iris_data, m, b) {
  ggplot(iris_data, aes(x = petal_length, y = petal_width, color = species)) +
    geom_point() + geom_abline(slope = m, intercept = b, color = 'blue', lwd = 1.1) +
    xlab('Petal Length (cm)') +
  ylab('Petal Width (cm)') +
  ggtitle('Petal Width vs Length by Iris Species') + theme_classic(12) +
    scale_color_manual(values = c('firebrick', 'darkgreen'))
}
```

```r
# Line Parameters
m <- -0.5
b <- 4
# Plot a selected boundary
plot_db(iris_subset, m, b)
```

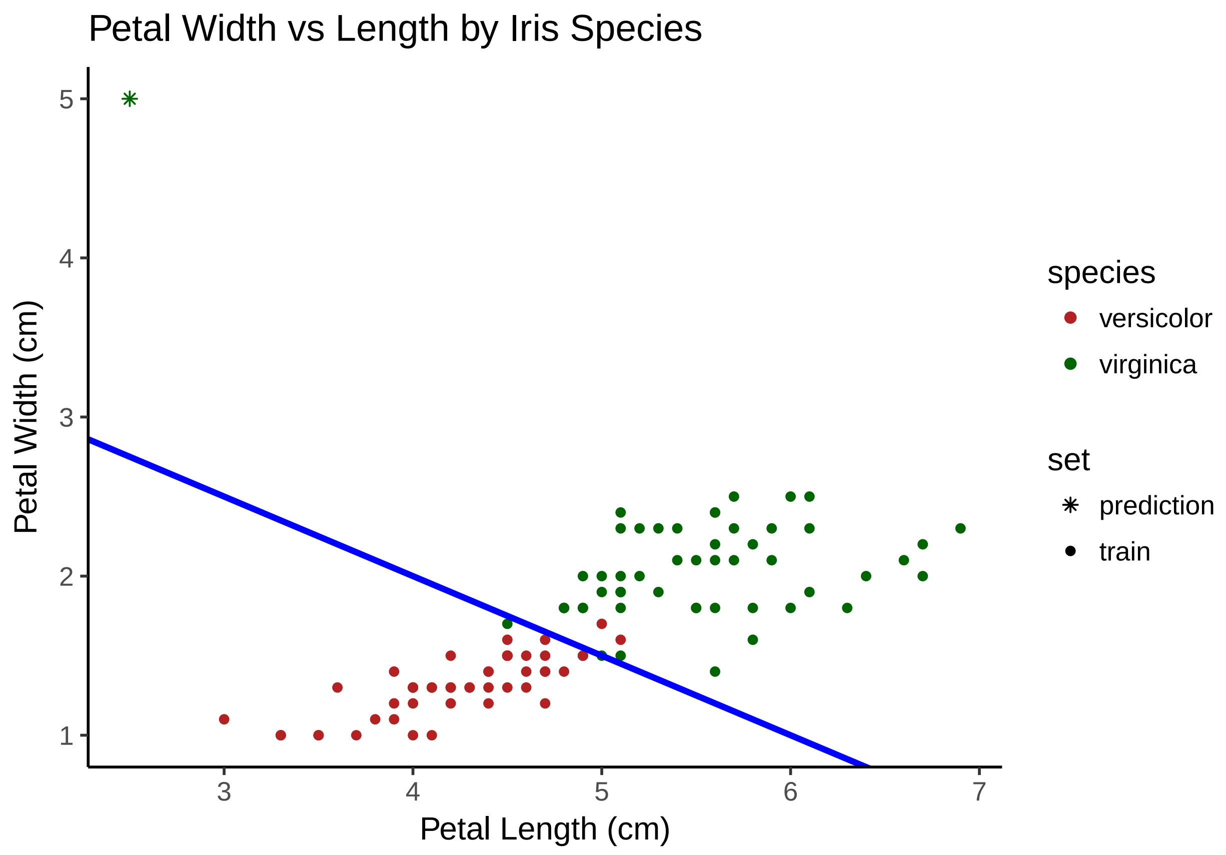


The decision boundary drawn above mis-classifies five points for an overall accuracy of 95%. It will be interesting to see if this is really the best decision boundary that can be drawn, or if a gradient descent algorithm can achieve higher accuracy.

1.3 C. Define a Simple Threshold Classifier Using the above Decision Boundary

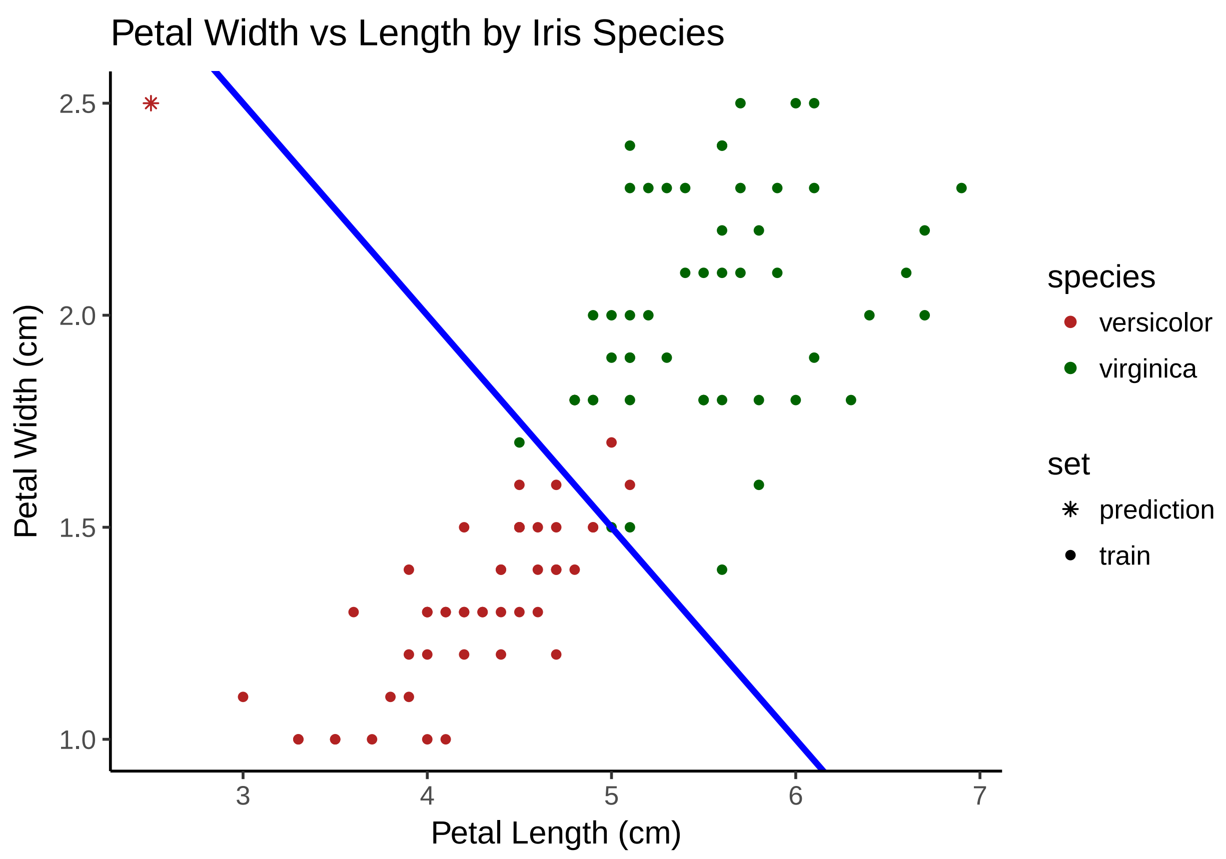
The following function takes in a petal length, petal width, data set, and model parameters (weights) and returns the class (species) of the iris. The threshold classifier multiplies the weights by the observation features, and classifies based on a threshold of zero (greater than 0 is virginica and less than 0 is versicolor. The weights have been set to defaults corresponding to the decision boundary plotted above.

```r
# Function takes in a petal length, petal width, iris data set,
# and model parameters
# and makes a class prediction. The function also plots the
# data and the new observation by default
classify <- function(petal_length, petal_width, iris_data,
                     plot_results = TRUE, w0 = -4, w1 = 0.5, w2 = 1) {
  # Threshold classifier
  if ( (petal_length * w1 + petal_width * w2 + w0) < 0) {
    class = 'versicolor'
  } else if ((petal_length * w1 + petal_width * w2 + w0) >= 0) {
    class = 'virginica'

  }

  iris_data$set <- 'train'

  # Add the prediction to the data set for plotting
  iris_data <- add_row(iris_data, petal_width = petal_width, petal_length = petal_length,
                      sepal_width = NA, sepal_length = NA, species = class, set = 'prediction')

  # Plot the iris data set and the new prediction
  if (plot_results) {
  print(ggplot(iris_data, aes(x = petal_length, y = petal_width, color = species, shape = set)) +
    geom_point() +
      geom_abline(slope = -w1/w2, intercept = -w0/w2, color = 'blue', lwd = 1.1) +
    xlab('Petal Length (cm)') +
  ylab('Petal Width (cm)') +
  ggtitle('Petal Width vs Length by Iris Species') + theme_classic(12) + scale_shape_manual(values = c(8
      scale_color_manual(values = c('firebrick', 'darkgreen')))
  }

  print(sprintf('The class prediction is: %s', class))
}

# Example of new data point
classify(petal_length = 2.5, petal_width = 5, iris_data = iris_subset)
```



```
## [1] "The class prediction is: virginica"
```

```
# Another example of new data point
classify(petal_length = 2.5, petal_width = 2.5, iris_data = iris_subset)
```



```
## [1] "The class prediction is: versicolor"
```

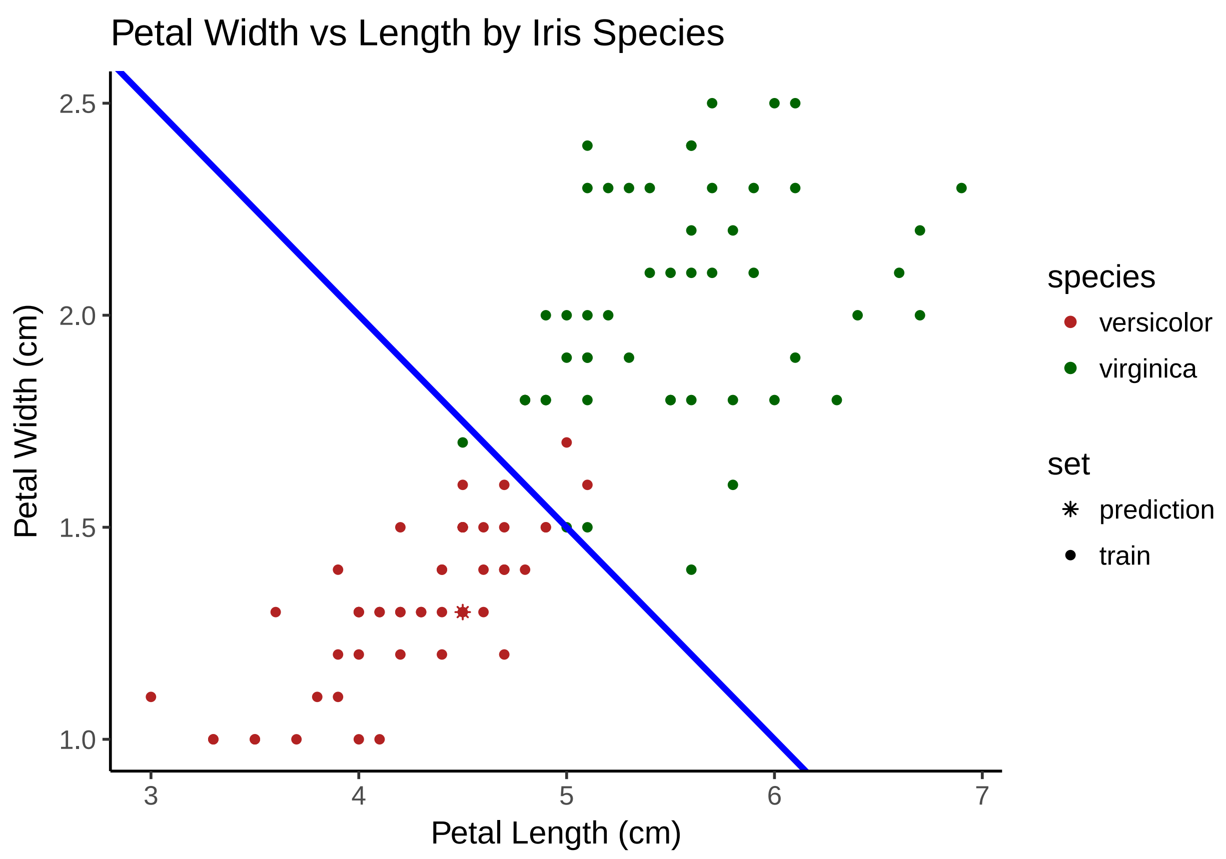
The above examples where not in the actual dataset, but were correctly classified by the model based o the decision boundary.

To demonstrate the ability of the classification function, I can make predictions using some of the points within the data. For these points, we know ahead of time the correct species.

```
# Plot a known versicolor example
sprintf('The true species is: %s', iris_subset$species[6])
```

```
## [1] "The true species is: versicolor"
```

```
petal_length <- iris_subset$petal_length[6]
petal_width <- iris_subset$petal_width[6]

classify(petal_length, petal_width, iris_subset)
```

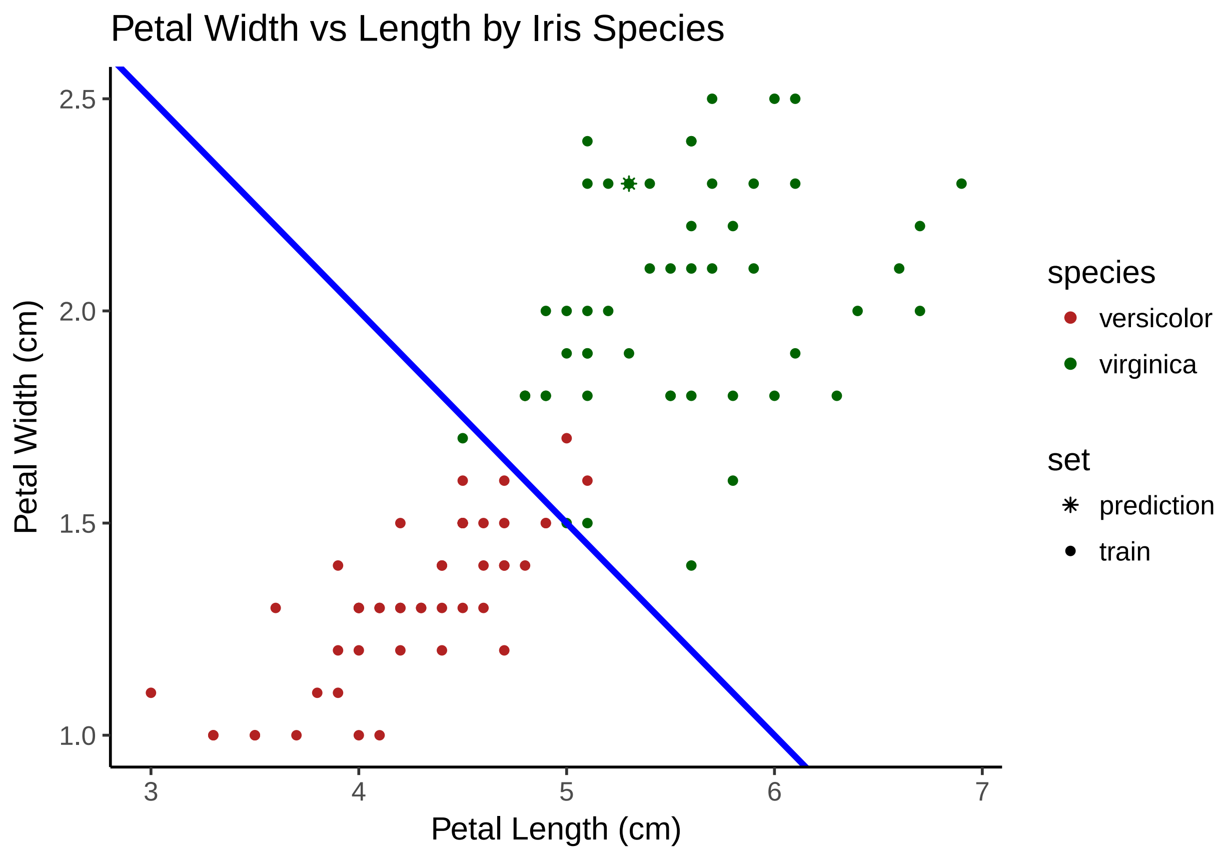


```
## [1] "The class prediction is: versicolor"
```

```
# Plot a known virginica example
sprintf('The true species is: %s', iris_subset$species[66])
```

```
## [1] "The true species is: virginica"
```

```
petal_length <- iris_subset$petal_length[66]
petal_width <- iris_subset$petal_width[66]

classify(petal_length, petal_width, iris_subset)
```



```
## [1] "The class prediction is: virginica"
```

The linear classifier correctly identifies both classes. There are several instances near the boundary the model mis-classifies because the separating line between the two species on a plot of petal width vs petal length is not completely linear.

1.4 D. Define a circle decision boundary using a single point as center

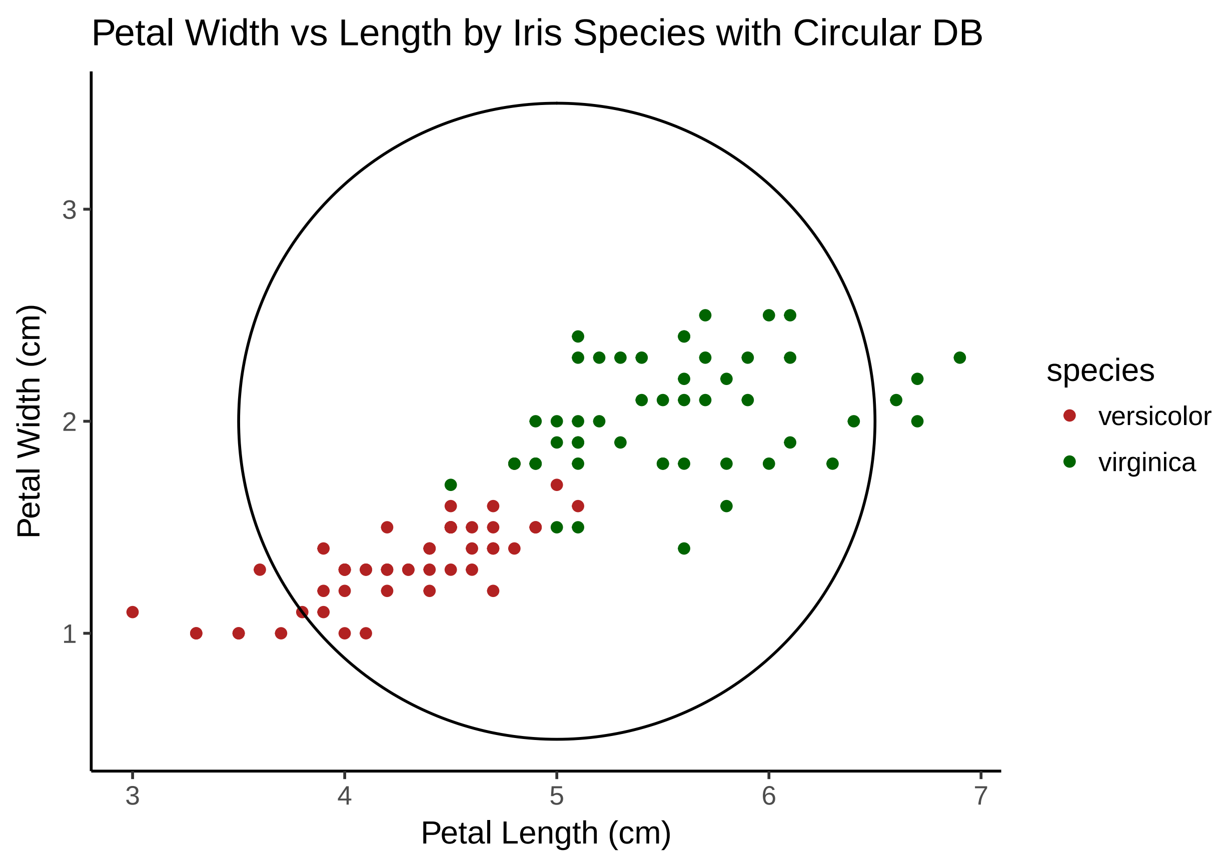
The function below takes a center for the circle (in the form (petal_length, petal_width)) and a radius and draws a circular decision boundary. Points are classified using the Euclidean distance. If points are within the radius from the center of the circle in terms of the Euclidean distance, then they are within the circle. The class of points within the circle is determined by the iris class with the most points in the circle. For example, if there are more veriscolor irises in the circle than viriginica, then all the data points within the circle would be considered versicolor and those outside virginica. Accuracy is assessed as the number of points correctly classified divided by the total number of iris observations.

```
# Takes a circle center (c(x, y)) and a circle radius and predicts the classes
# of the iris_data. Prints the accuracy of the circular classifier
# and displays a plot with the classifier and original data.
circular_db <- function(center, radius, iris_data) {
  classes <- c()

  # Make a classification for each pointbased on the circle parameters
  for (i in 1:nrow(iris_data)) {
    length <- iris_data[[i, 'petal_length']]
    width <- iris_data[[i, 'petal_width']]
```

```r
    # Check if point in circle using Euclidean distance
    in_circle <- ifelse(dist(matrix(c(center[1], center[2], length, width),
                                   nrow = 2, ncol = 2, byrow = TRUE)) < radius, 1, 0)
    classes <- c(classes, in_circle)
  }

  iris_data$in_circle <- classes


  virginica_in <- sum(iris_data$in_circle == 1 &
                      iris_data$species == 'virginica')

  versicolor_in <- sum(iris_data$in_circle == 1 &
                       iris_data$species == 'versicolor')

  # Set in circle class to class with more points in circle
  # If tied for number, choose a random class
  # The tie also handles the cases where no points
  # are in circle and all points are in circle
  circle_class <- ifelse(virginica_in > versicolor_in, 'virginica',
                         ifelse(virginica_in < versicolor_in, 'versicolor',
                                sample(c('virginica', 'versicolor'), 1)))

  # The non-circle class is the other class
  non_circle_class <- ifelse(circle_class == 'virginica', 'versicolor', 'virginica')

  # Prediction is the label of in circle or not
  iris_data$prediction <- ifelse(iris_data$in_circle == 1, circle_class, non_circle_class)

  # Accuracy is the average of correct predictions
  accuracy <- sum(iris_data$prediction == iris_data$species) / nrow(iris_data)

  # Create a dataframe for plotting the circle
  circ_df <- data.frame(x = center[1], y = center[2], radius = radius)

  # Plot the data and the circular decision boundary
  p <- ggplot(iris_subset) + geom_point(aes(x = petal_length, y = petal_width, color = species)) +
    geom_circle(data = circ_df, aes(x0 = x, y0 = y, r = radius)) + coord_fixed() +
    xlab('Petal Length (cm)') +
  ylab('Petal Width (cm)') +
  ggtitle('Petal Width vs Length by Iris Species with Circular DB') + theme_classic(12) +
    scale_color_manual(values = c('firebrick', 'darkgreen'))

  print(p)

 print(sprintf('Accuracy: %0.2f%% with circle at %0.2f, %0.2f with radius %0.2f.',
              accuracy * 100, center[1], center[2], radius))
  }
```

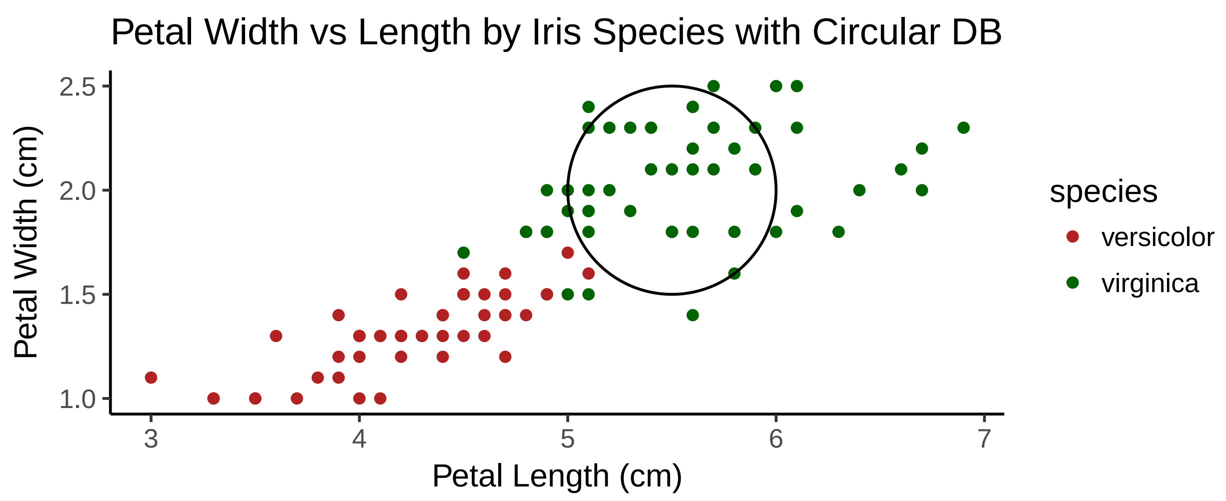
The next step is to test the classification accuracy of the circle with several values for the center and radius.

```r
# Test classification accuracy with different boundaries
circular_db(center = c(5, 2), radius = 1.5, iris_data = iris_subset)
```



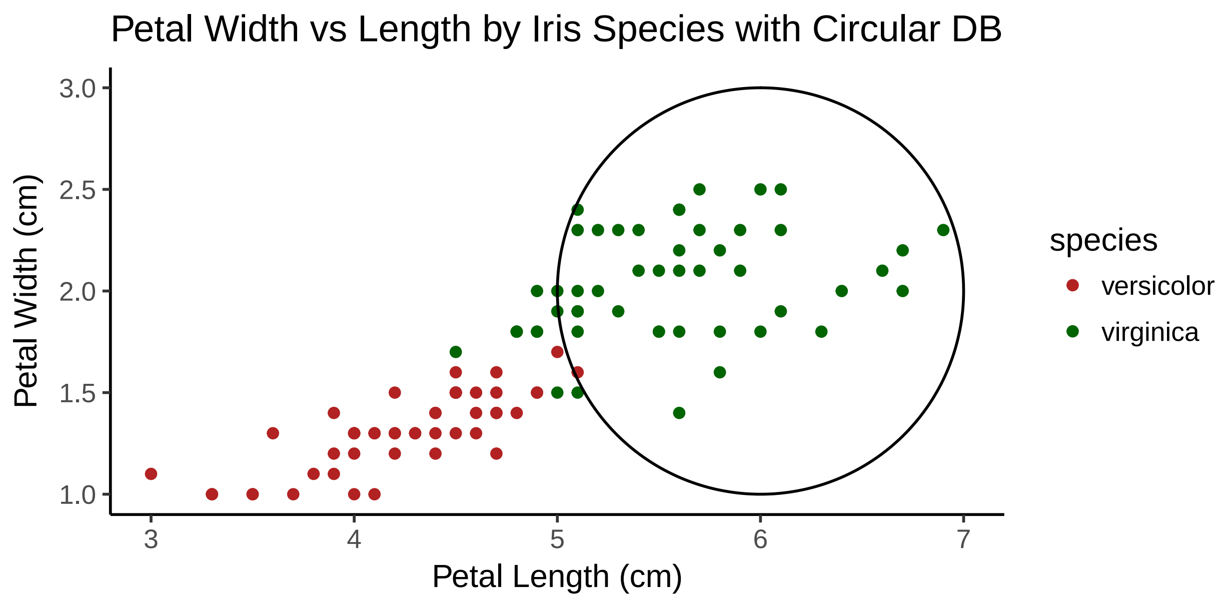
```
## [1] "Accuracy: 54.00% with circle at 5.00, 2.00 with radius 1.50."
```

```
circular_db(center = c(5.5, 2), radius = 0.5, iris_data = iris_subset)
```



```
## [1] "Accuracy: 74.00% with circle at 5.50, 2.00 with radius 0.50."
```

```
# Improve accutracy with manual inspection
circular_db(center = c(6, 2), radius = 1, iris_data = iris_subset)
```



```
## [1] "Accuracy: 89.00% with circle at 6.00, 2.00 with radius 1.00."
```

The final circle achieves the best accuracy. The performance could be further improved by careful adjustment of the decision boundary, but as with the linear classifier, a perfect model is not possible with this data.

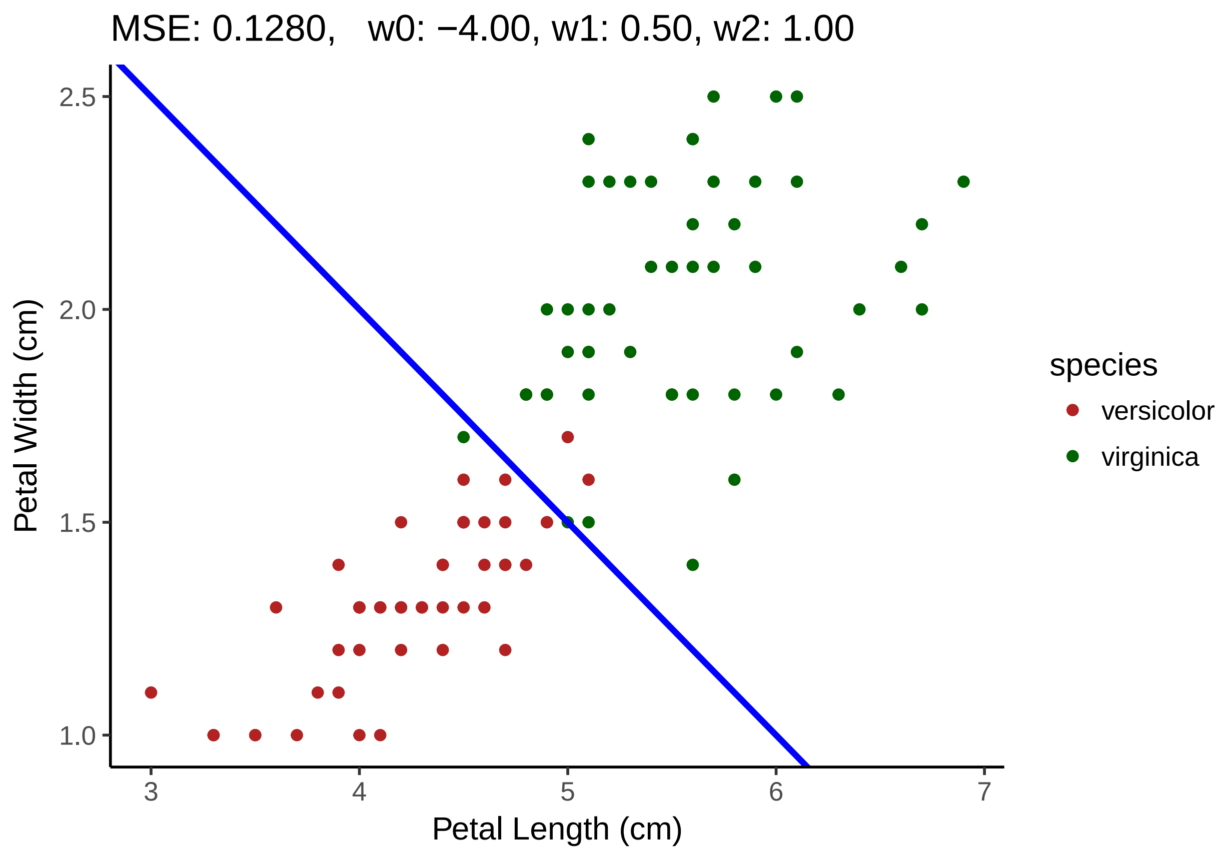
2 Objective Function

2.1 A. Mean Squared Error

The next task is to write a function which calculates the mean-squared error for a given decision boundary. To calculate a mean-squared error, I will replace the threshold activation with a sigmoid function that outputs class probabilities. With a simple threshold (0 or 1) boundary, the mean squared error would is simply the percentage of points mis-classified. Instead, I can use the sigmoid function to return the probability an iris is within a given class and then apply the mean-squared error as the objective function. I like to use the sigmoid activation because it represents the uncertainty in the prediction rather than outputting a simple 0 or 1. The mean-squared is computed as the average of the squared residuals of the predictions where the residuals are the difference between the correct answer and the predicted output. For this problem, the correct class is either 0, versicolor, or 1, virginica. The output from the model will be a probability between 0 and 1 for each class. If the prediction is 0.8, and the true class is 1 (virginica), then the residual is 0.2. The objective is to minimize the mean squared error by optimizing the parameters of the model.

```
# Sigmoid function
sigmoid <- function(x) {return(1 / (1 + exp(-x)) )}

# Takes in data frame with correct values (labels) and weights of decision boundary
```

```r
# Weights should be in form (w0, w1, w2) and labels are 0 (versicolor)
# or 1 (virginica).
create_model <- function(iris_data, weights, plot_results = TRUE) {

  iris_data$intercept <- 1

  # Enforce column order for features and create label vector
  data <- dplyr::select(iris_data, intercept, petal_length, petal_width)
  labels <- ifelse(iris_data$species == 'versicolor', 0, 1)

  predictions <- c()

  # Make predictions on all of the data
  for (i in 1:nrow(data)) {
    prediction <- sigmoid(as.numeric(data[i, ]) %*% weights)[[1]]
    predictions <- c(predictions, prediction)
  }

  # Calculate the mean squared error
  mse <- mean( (predictions - labels) ^ 2)

  # Plot the classes and decision boundary
  if (plot_results) {
  print(ggplot(iris_data, aes(x = petal_length, y = petal_width, color = species)) +
    geom_point() +
      geom_abline(slope = -weights[2]/weights[3], intercept = -weights[1]/weights[3],
                  color = 'blue', lwd = 1.1) +
    xlab('Petal Length (cm)') +
  ylab('Petal Width (cm)') +
  ggtitle(sprintf('MSE: %0.4f,   w0: %0.2f, w1: %0.2f, w2: %0.2f', mse, weights[1], weights[2], weights
  + theme_classic(12) +
      scale_color_manual(values = c('firebrick', 'darkgreen')))
  }

}

# Example application
weights <- c(-4, 0.5, 1)

create_model(iris_subset, weights)
```

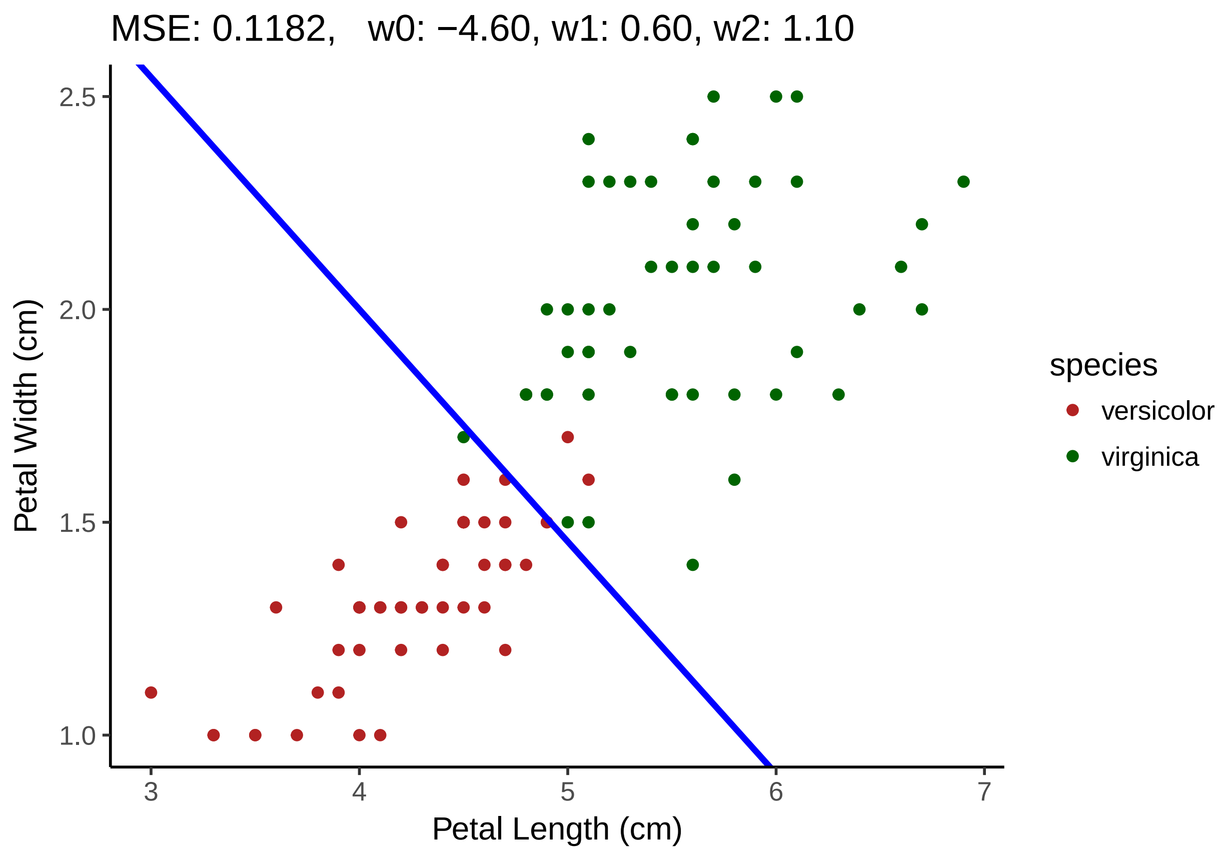


The example shown above is the same decision boundary as was used previously. Classifications are made by multiplying the features of the observation by the model parameters and then applying the sigmoid activation function to calculate the class probability.

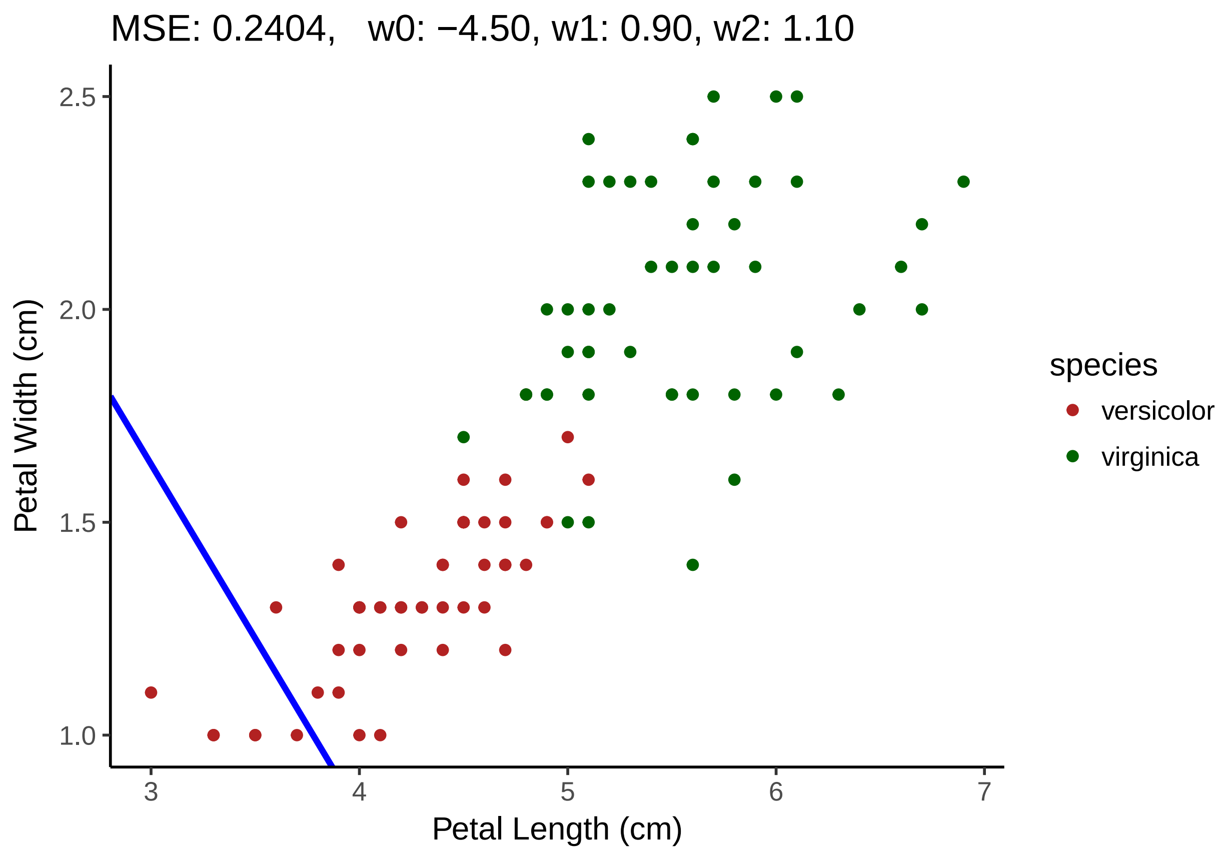
2.2 B. Plot a decision boundary with a high and low MSE

The low MSE corresponds to the linear decision boundary previously drawn that was determined by repeated testing. A high MSE can also be found by some experimentation.

```r
# Low MSE
create_model(iris_subset, weights = c(-4.6, 0.6, 1.1))
```



```r
# High MSE
create_model(iris_subset, weights = c(-4.5, 0.9, 1.1))
```



The first decision boundary has a significantly lower mean-squared error than the second. The number of misclassifications is directly proportional to the mse.

2.3 C. Derivation of MSE with respect to weights

Class probabilities of an observation are made with the following equation:

$$p_n = \sigma(w_0 + w_1 * length_n + w_2 * width_n)$$

Where σ is the sigmoid activation function. The mean squared error is therefore

$$mse = \frac{1}{N}\sum_{n=1}^{N}(\sigma(w_0 + w_1 * L_n + w_2 * W_n) - y_n)^2$$

Where y_n is the known label for the observation. N refers to the total number of observations (data points) which in this problem is 100. The sigmoid activation function has the nice feature that the derivative is:

$$\frac{d\sigma(x)}{dx} = \sigma(x) * (1 - \sigma(x))$$

To simply the equation, we can define the weight column vector as

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

and all of the observations as a matrix

$$\mathbf{x} = \begin{pmatrix} 1 & L_1 & W_1 \\ 1 & L_2 & W_2 \\ \vdots & \vdots & \vdots \\ 1 & L_m & W_m \end{pmatrix}$$

The labels are a column vector

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The matrix algebra works out because the shape of $\mathbf{x}$ is 100 x 3, the shape of $\mathbf{w}$ is 3 x 1, and the shape of the label vector $\mathbf{y}$ is 100 x 1.

The MSE is now expressed as

$$mse = \frac{1}{N}(\sigma(\mathbf{x} * \mathbf{w}) - \mathbf{y})^2$$

To update the model parameters, we need to take the derivative of the objective function respect to the weights. The derivative is then set to 0 to and is minimized using gradient descent.

The gradient of the mse with respect to the weights is

$$\frac{\partial(mse)}{\partial \mathbf{w}} = \frac{2}{N} * \mathbf{x} * (1 - \sigma(\mathbf{x} * \mathbf{w}) * (\sigma(\mathbf{x} * \mathbf{w}) - \mathbf{y})$$

This is the vector form of the equation which is much more efficient to calculate than the scalar version for each component of the weights.

If we substitute in the probability already calculated, the equation becomes:

$$\frac{\partial(mse)}{\partial \mathbf{w}} = \frac{2}{N} * \mathbf{x} * (1 - \mathbf{p}) * (\mathbf{p} - \mathbf{y})$$

Expressing the gradient in this form makes it easier to compute because we already calculate p, the class probability, when making predictions.

2.4 D. Scalar versus Vector Gradient

The full scalar gradient for weight w_i is

$$\frac{\partial(mse)}{\partial w_i} = \frac{2}{N} * \sum_{n=1}^{N}(x_{i,n} * (1 - \sigma(\mathbf{x}_n * \mathbf{w}) * (\sigma(\mathbf{x}_n * \mathbf{w}) - y_n))$$

In this equation, n is the observation, and i is the weight. In this problem, there are 3 weights and 100 observations, so for each iteration, we would need to perform a sum over all 100 observations 3 separate times to update the three weights.

The computation requirements can be greatly reduced by expressing the observations as a (100 by 3) matrix, the weights as a (3 by 1) vector, and the labels as a (100 by 1) vector. The gradient of the mean squared error with respect to the weights as a vector is much simpler:

$$\frac{\partial(mse)}{\partial \mathbf{w}} = \frac{2}{N} * (1 - \sigma(\mathbf{x} * \mathbf{w})) * (\sigma(\mathbf{x} * \mathbf{w}) - \mathbf{y}) * \mathbf{x}$$

The gradient update rule uses the gradient to adjust the weights to minimize the objective function on each iteration.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon * \frac{\partial(mse)}{\partial(\mathbf{w}_t)}$$

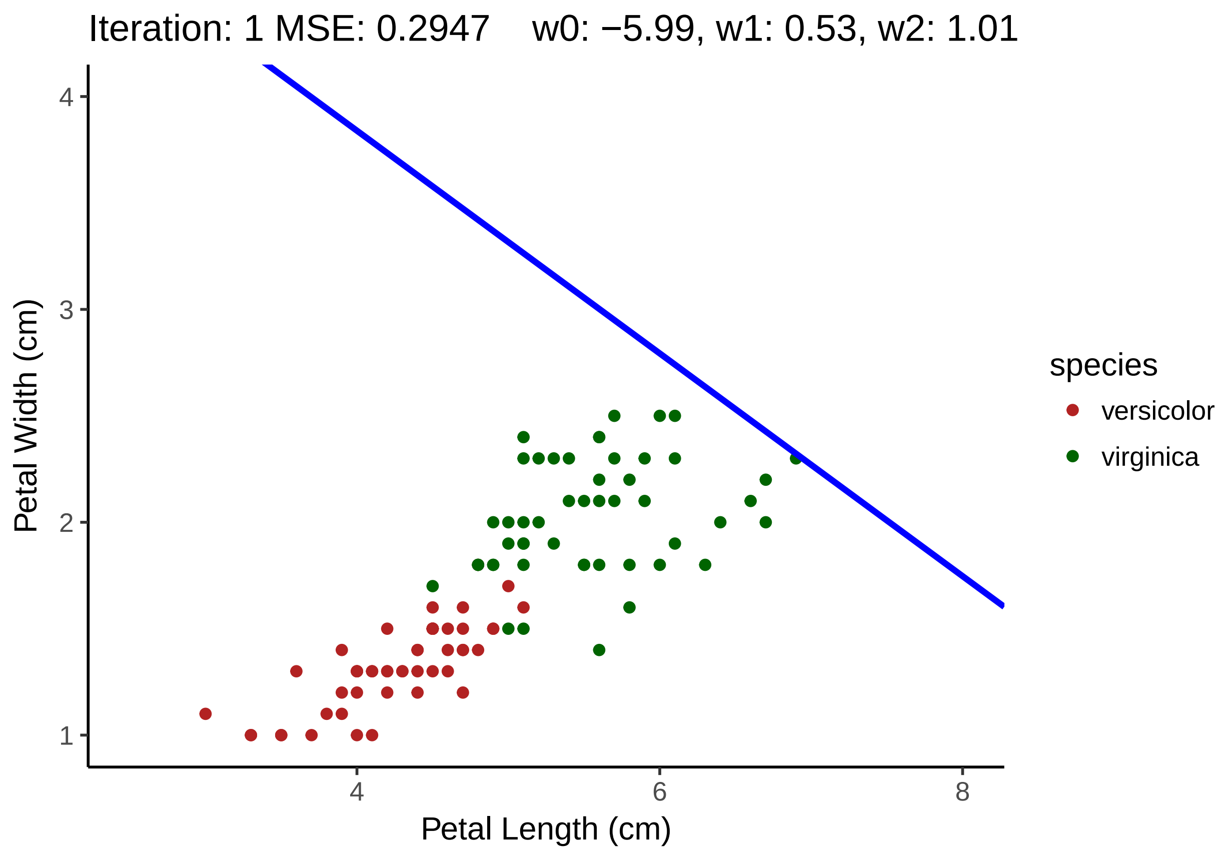
Where ϵ is the learning rate. Eventually, gradient descent will converge on the minimum of the cost function if the step size is small enough.

2.5 E. Calculate the Gradient and Update the Weights

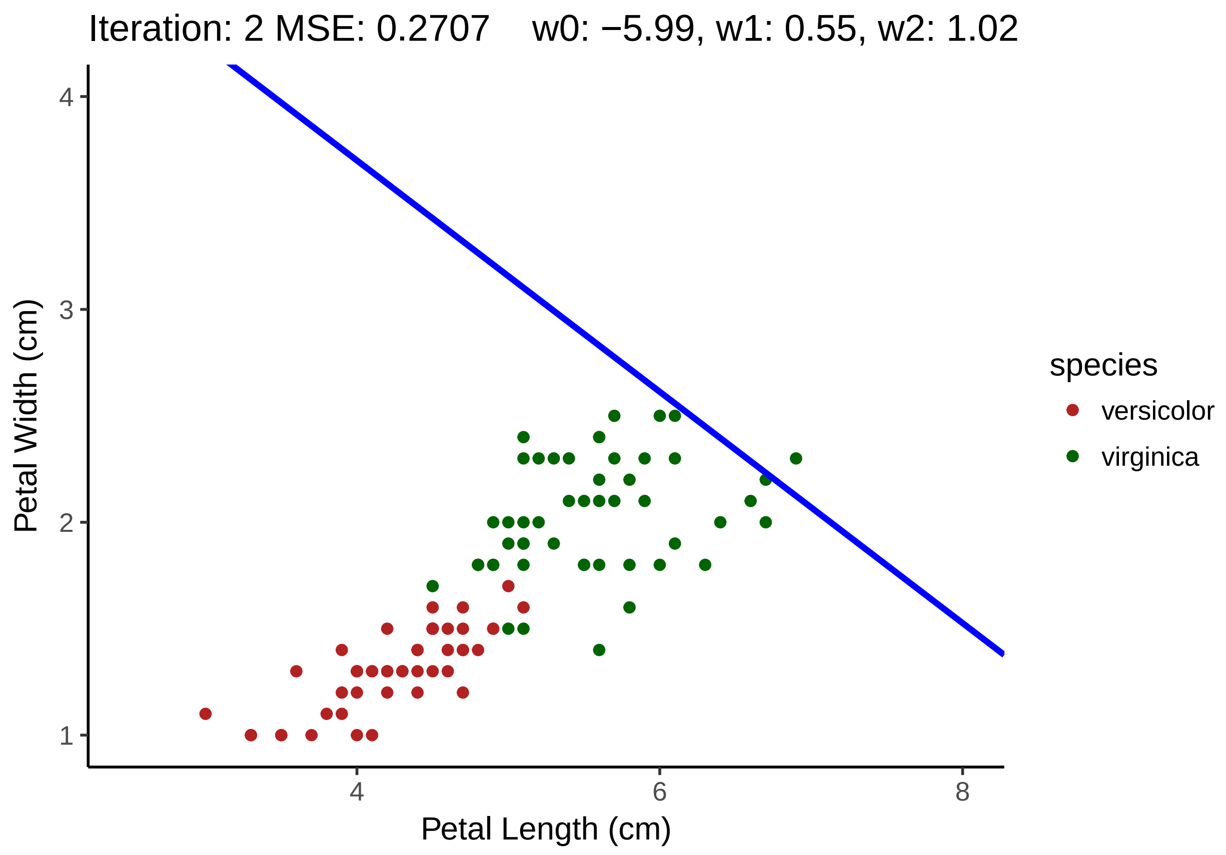
The algorithm is relatively straightforward to implement. I can initialize the weights, make the predictions, calculate the gradient, update the weights, and repeat the process with the updated weights.

```r
# Function takes in the iris data set, initial weights, number of iterations,
# and the learning rate
grad_descent <- function(iris_data, initial_weights, n_steps = 5,
                         plot_decision = TRUE, plot_learning = FALSE,
                         learning_rate = 0.01) {

  # Add intercept to data and determine n
  iris_data$intercept <- 1
  n_obs <- nrow(iris_data)

  # Enforce column order for features and create label vector
  data <- dplyr::select(iris_data, intercept, petal_length, petal_width)
  labels <- ifelse(iris_data$species == 'versicolor', 0, 1)

  weights <- initial_weights
  data <- data.matrix(data)

  # Variables to hold metrics
  results_df <- c()
  all_mse <- c()

  for (i in 1:n_steps) {

    # Make predictions and calculate errors
    predictions <- sapply(data %*% weights, sigmoid)
    errors <- predictions - labels

    # Record mean squared error
    mse <- mean((errors) ^ 2)
    all_mse <- c(all_mse, mse)

    # Calculate the gradient
    gradient <- (2 / n_obs) * matrix((1 - predictions) * (errors), ncol = n_obs) %*% data

    # Update the weights
    weights <- as.numeric(weights - learning_rate * gradient)
    print(sprintf('Iteration: %0.0f, mse: %0.4f', i, mse))

    # Plot the decision boundary if specified
    if (plot_decision) {
    p1 <- ggplot(iris_data, aes(x = petal_length, y = petal_width, color = species)) +
      geom_point() +
        geom_abline(slope = -weights[2]/weights[3], intercept = -weights[1]/weights[3],
                    color = 'blue', lwd = 1.1) +
      xlab('Petal Length (cm)') +
    ylab('Petal Width (cm)') +
    ggtitle(sprintf('Iteration: %0.0f MSE: %0.4f    w0: %0.2f, w1: %0.2f, w2: %0.2f',
                    i, mse, weights[1], weights[2], weights[3])) +
      theme_classic(12) + coord_cartesian(xlim = c(2.5, 8), ylim = c(1.0, 4.0)) +
        scale_color_manual(values = c('firebrick', 'darkgreen'))

    print(p1)
    }
  }

  results_df$mse <- all_mse
  results_df$iteration <- 1:n_steps

    # Plot the learning curve if specified
    if (plot_learning) {
      p2 <- ggplot(results_df, aes(x = iteration, y = all_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', learning_rate)) +
        theme_classic(12)
    }

}

# Example
grad_descent(iris_subset, initial_weights = c(-6, 0.5, 1))
```

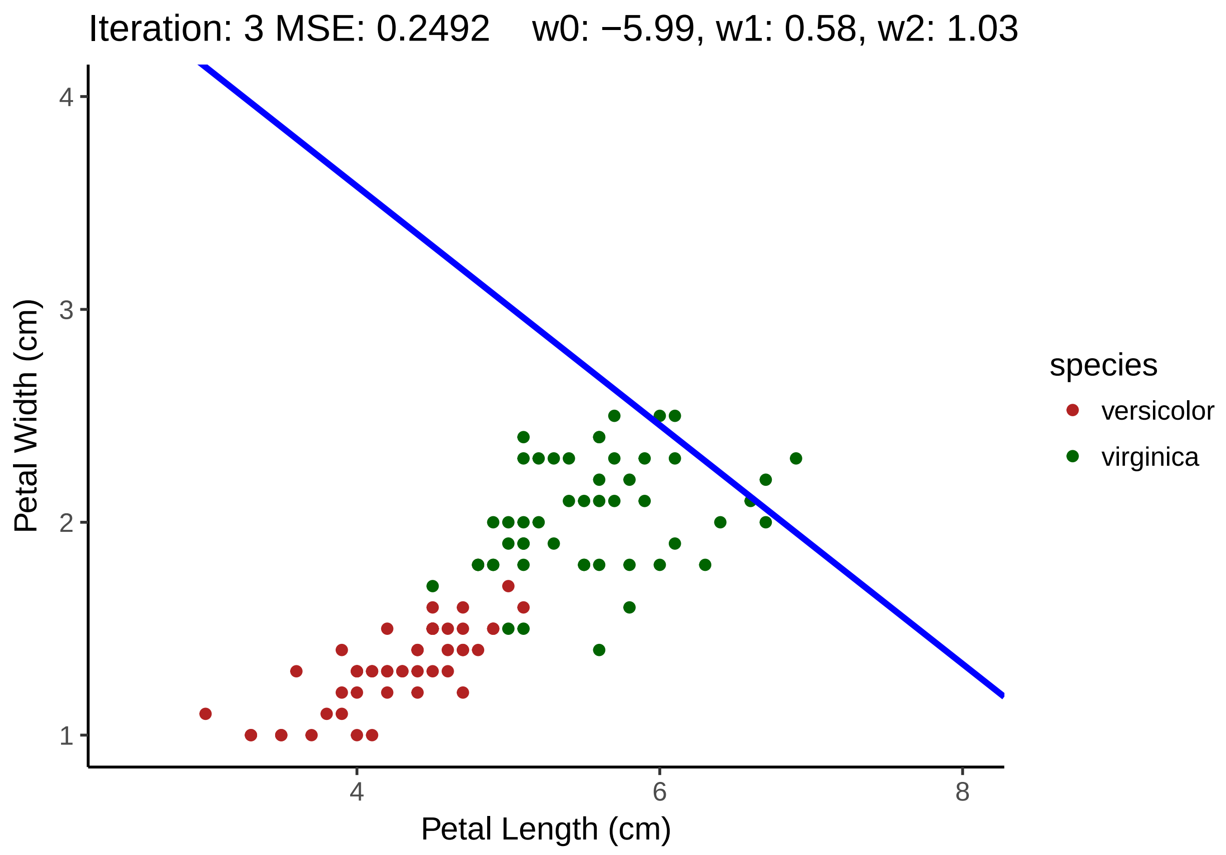
```
## [1] "Iteration: 1, mse: 0.2947"
```



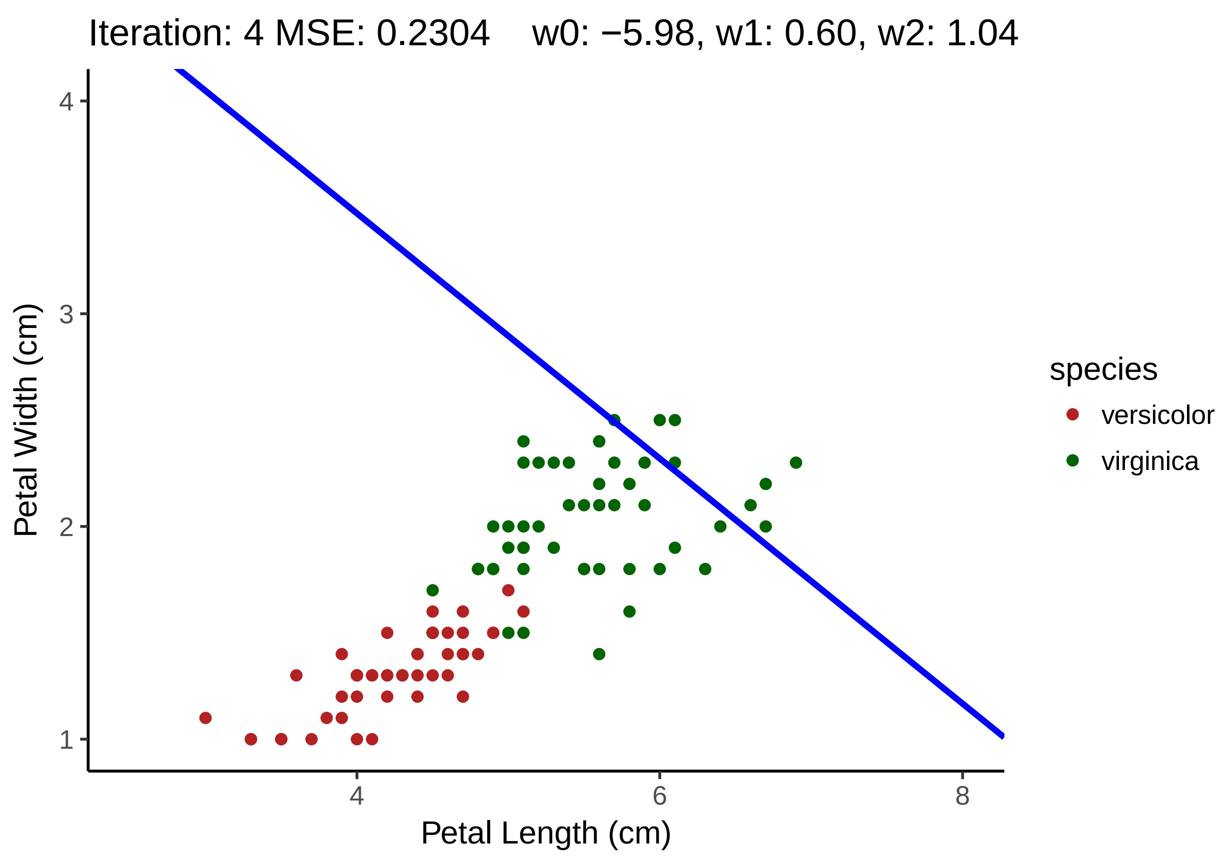
```
## [1] "Iteration: 2, mse: 0.2707"
```



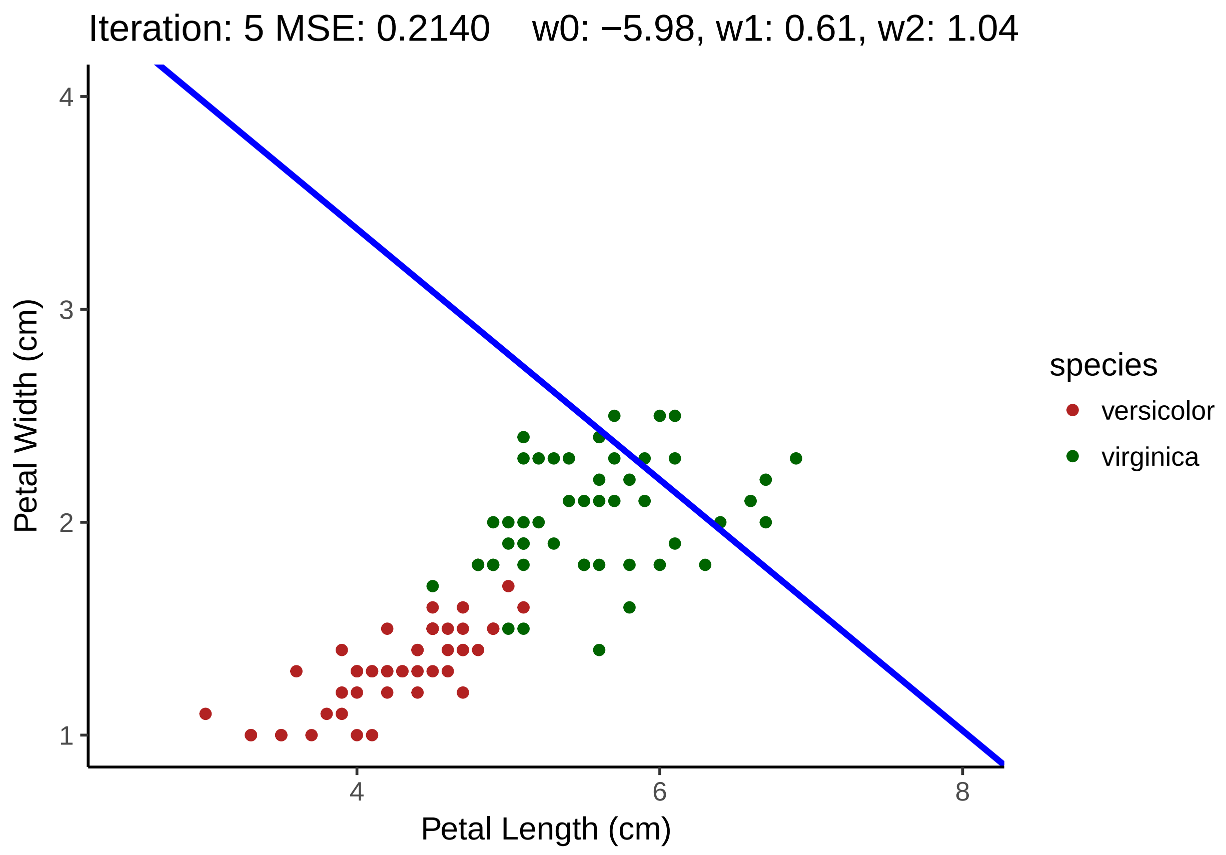
```
## [1] "Iteration: 3, mse: 0.2492"
```



```
## [1] "Iteration: 4, mse: 0.2304"
```

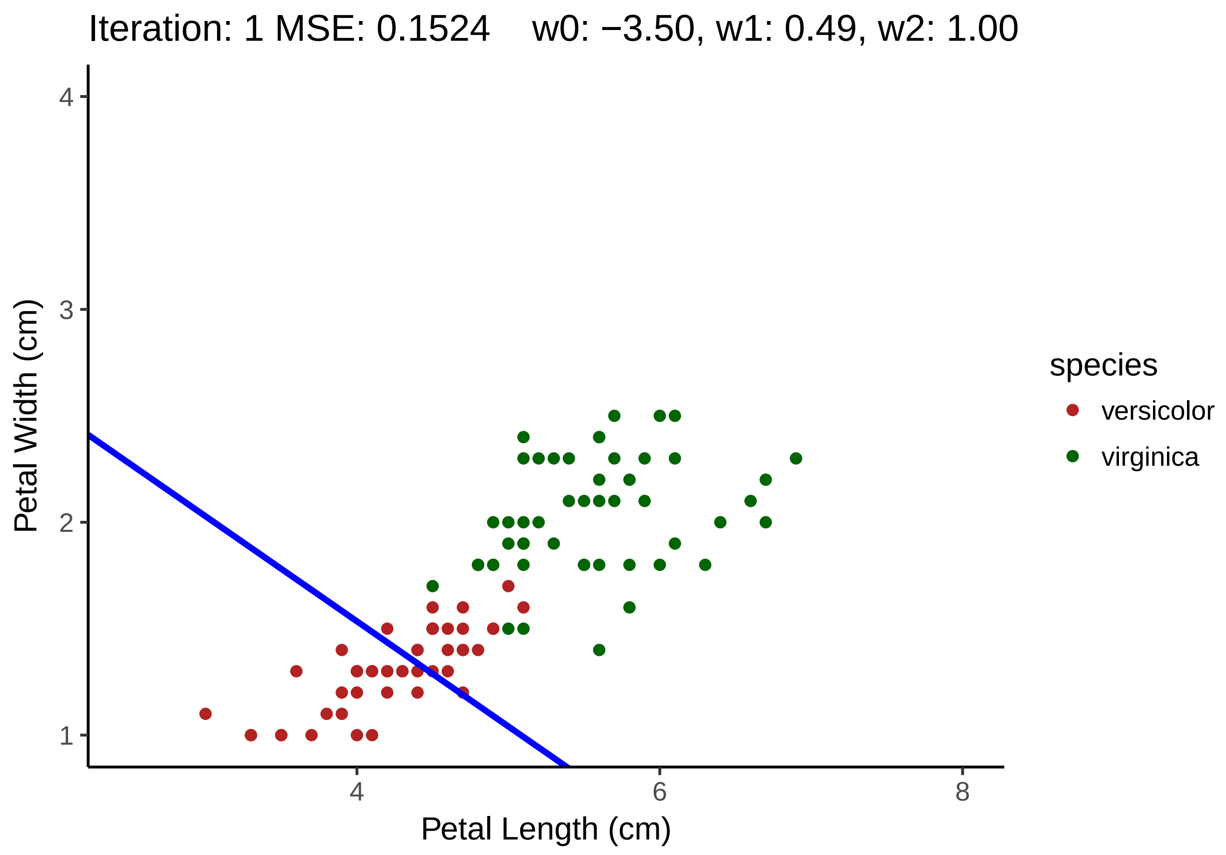


```
## [1] "Iteration: 5, mse: 0.2140"
```

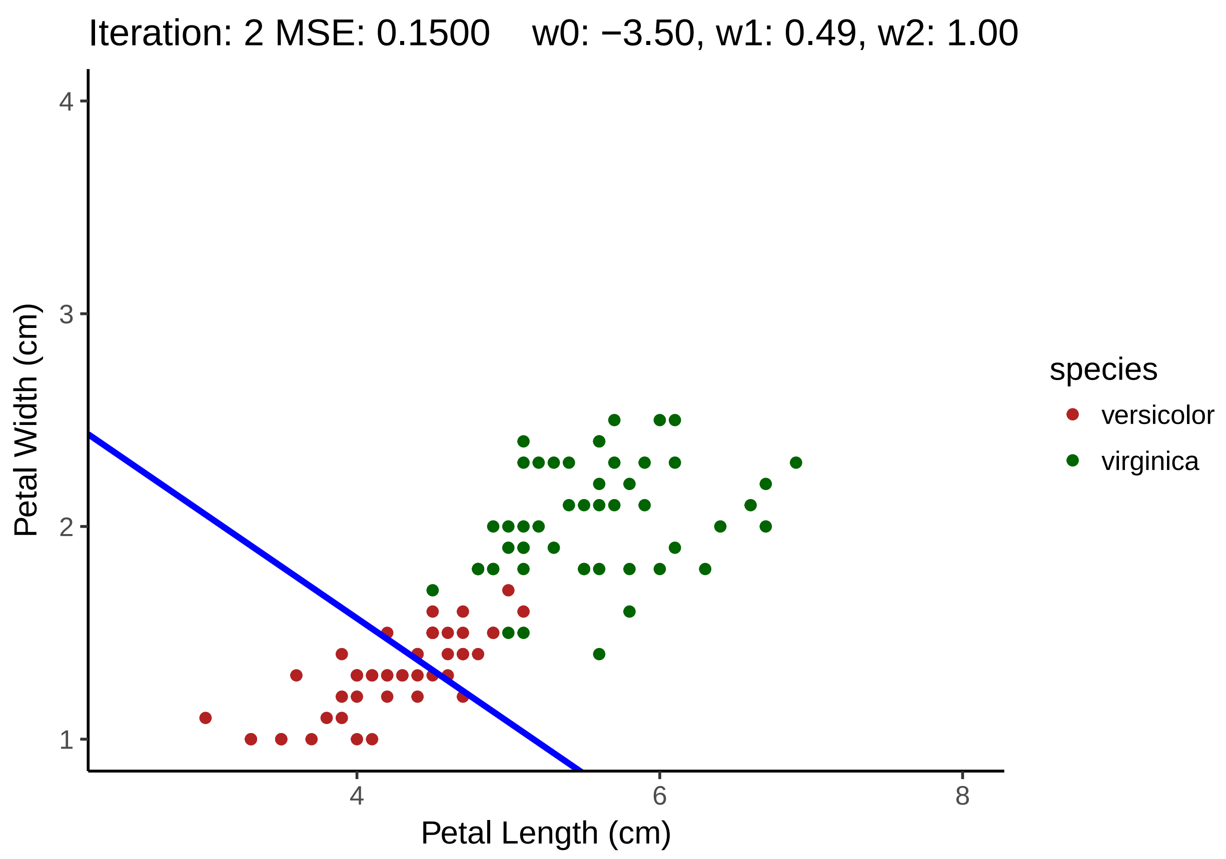


```r
# Example
grad_descent(iris_subset, initial_weights = c(-3.5, 0.5, 1))
```

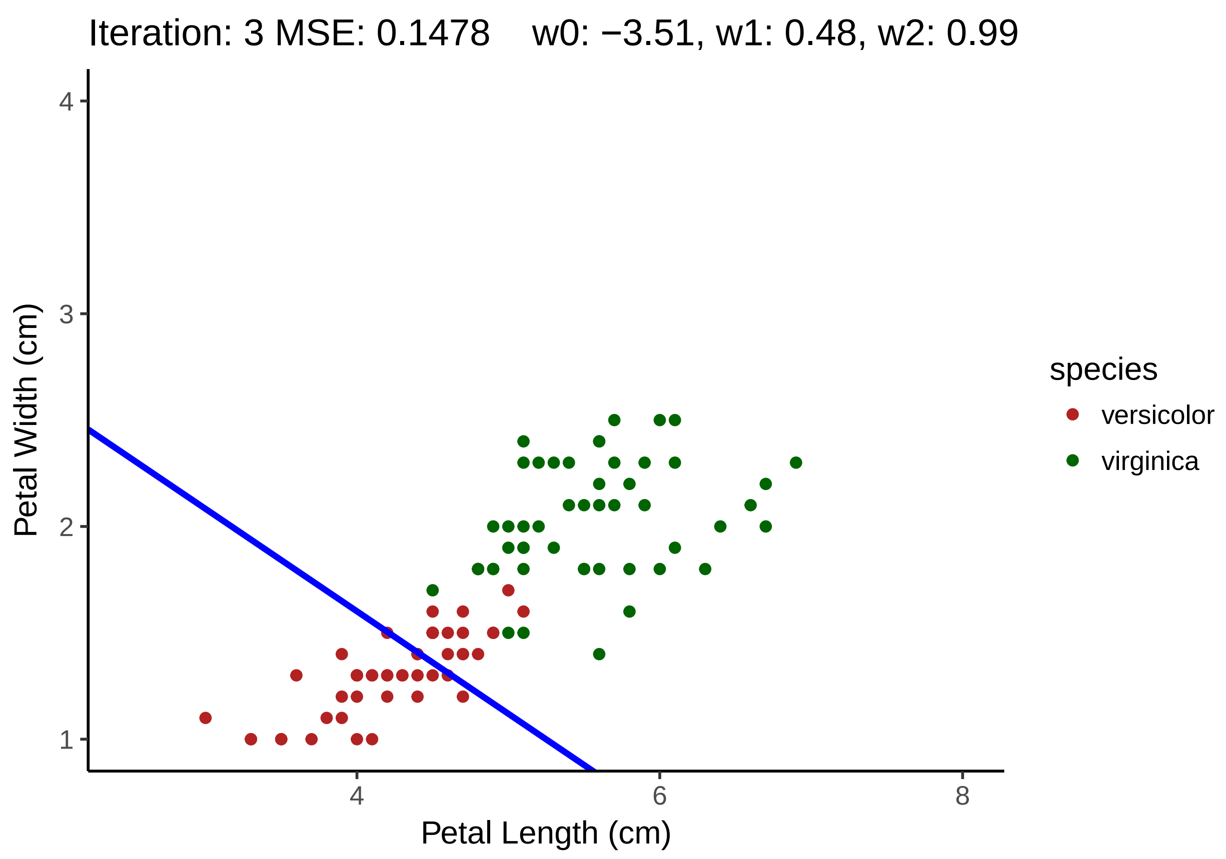
```
## [1] "Iteration: 1, mse: 0.1524"
```



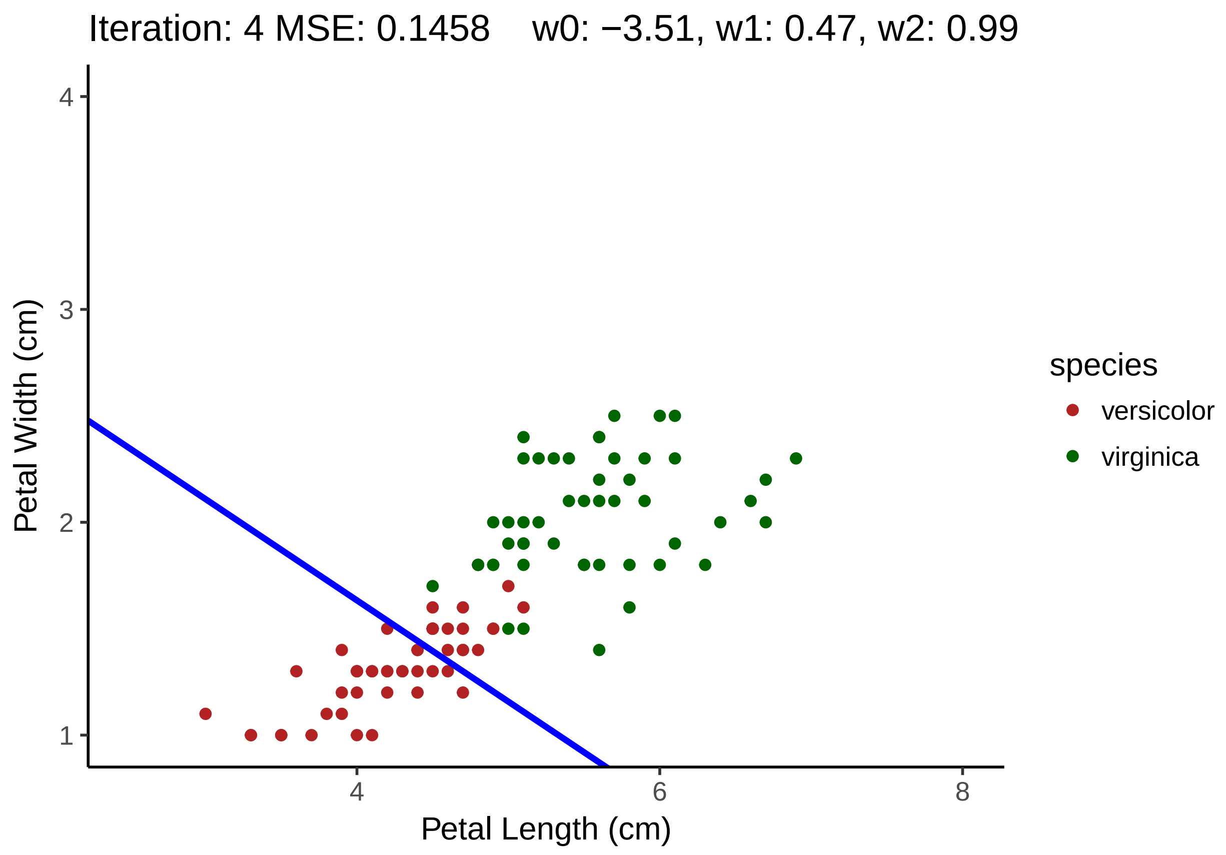
```
## [1] "Iteration: 2, mse: 0.1500"
```



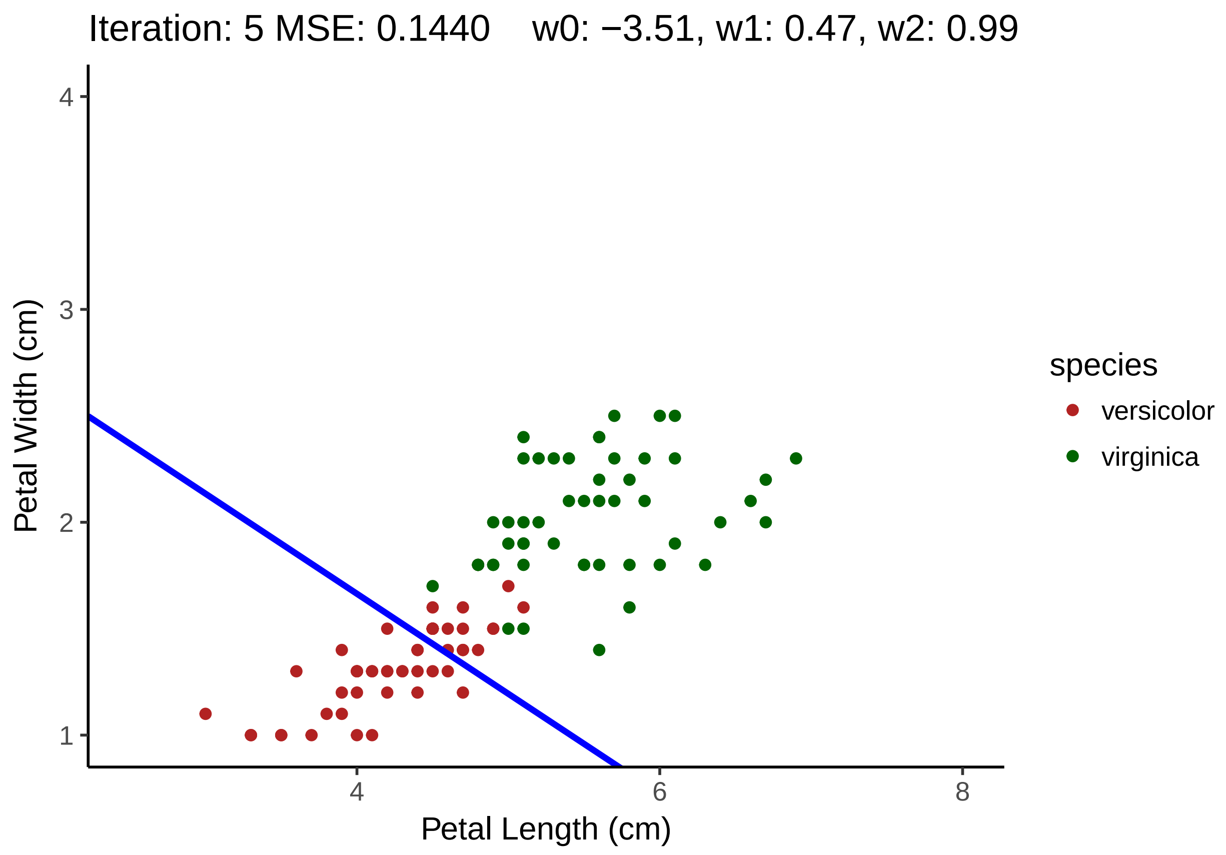
```
## [1] "Iteration: 3, mse: 0.1478"
```



```
## [1] "Iteration: 4, mse: 0.1458"
```



```
## [1] "Iteration: 5, mse: 0.1440"
```



As can be seen, the algorithm successfully updates the decision boundary to decrease the mean squared error with each iteration. If the decision boundary starts off too low, it is raised higher, and when it starts off too high, it is slowly lowered. With enough steps and a small enough learning rate, the algorithm will converge on the optimal decision boundary.

3 Optimization Using Gradient Descent

3.1 A. Implement Gradient Descent

The function above already implements gradient descent on the iris data set. To implement the algorithm, the function needs a set of initial weights (w0, w1, w2), a learning rate (the default is 0.01), a number of iterations (the default is 5), and a tolerance at which to stop iterating.

```
# Function takes in the iris data set, initial weights, number of iterations,
# learning rate, and tolerance
# Return is a dataframe that can be used for plotting
# the decision boundary and learning curve
grad_descent_complete <- function(iris_data, initial_weights, n_steps = 5,
                         learning_rate = 0.01, tolerance = 0.005) {

  # Add intercept to data and determine n
  iris_data$intercept <- 1
  n_obs <- nrow(iris_data)

  # Enforce column order for features and create label vector
```

```r
data <- dplyr::select(iris_data, intercept, petal_length, petal_width)
labels <- ifelse(iris_data$species == 'versicolor', 0, 1)

weights <- initial_weights
data <- data.matrix(data)

# Variables to hold metrics
results_df <- as.data.frame(matrix(ncol = 5, nrow = n_steps))
names(results_df) <- c('iteration', 'mse', 'w0', 'w1', 'w2')

# Iterate for the specified number of steps
for (i in 1:n_steps) {

  # Make predictions and calculate errors
  predictions <- sapply(data %*% weights, sigmoid)
  errors <- predictions - labels

  # Record mean squared error
  mse <- mean((errors) ^ 2)

  # Calculate the gradient
  gradient <- (2 / n_obs) * matrix((1 - predictions) * (errors), ncol = n_obs) %*% data

  # Record results for plotting
  results_df[i, 'iteration'] = i
  results_df[i, 'mse'] = mse
  results_df[i, 'w0'] = weights[1]
  results_df[i, 'w1'] = weights[2]
  results_df[i, 'w2'] = weights[3]


  # First stopping criteria
  if (sqrt(sum(gradient ^ 2)) < tolerance) {
    print("Minimum Tolerance Reached.")
    return(results_df)
  }

  # Second stopping criteria
  all_mse <- results_df$mse[!is.na(results_df$mse)]
  if (length(all_mse) > 3) {
    n_mse <- length(all_mse)
    # Find the third lowest mse and compare to current mse
    third_lowest_mse <- sort(all_mse, decreasing = TRUE)[n_mse - 2]

    if (mse > third_lowest_mse) {
    print('MSE Inceasing.')
    return(results_df)
    }
  }

  # Update the weights
  weights <- as.numeric(weights - learning_rate * gradient)
```

```r
    # Display progress every 50 iterations
    if (i %% 50 == 0) {print(sprintf('Iteration: %0.0f, mse: %0.4f', i, mse))

    }
  }
  # Max iterations reached
  print('Maximum number of iterations reached.')
  return(results_df)
}
```

3.2 B. Plot Decision Boundary and Learning Curve

The code above implements gradient descent to optimize the model parameters and also returns the parameters and mse on each iteration. Using these results, we can plot the decision boundary and mse over the number of iterations to see if the algorithm correctly updates the model parameters.

```r
# Function to plot the decision boundary and the learning curve
plot_decision_learning <- function(results, iris_data) {

  # Plot results from beginning, middle, and end of iterations
  rows <- seq(1, nrow(results), by = nrow(results)/2 - 1)

  for  (i in rows) {
    # Select the data for plotting
    data_row <- results[i, ]
    weights <- c(data_row$w0, data_row$w1, data_row$w2)
    i <- data_row$iteration
    mse <- data_row$mse

  # Plot decision boudary
  print(ggplot(iris_data, aes(x = petal_length, y = petal_width, color = species)) +
      geom_point() +
        geom_abline(slope = -weights[2]/weights[3],
                    intercept = -weights[1]/weights[3],
                    color = 'blue', lwd = 1.1) +
      xlab('Petal Length (cm)') +
    ylab('Petal Width (cm)') +
    ggtitle(sprintf('Iteration: %0.0f MSE: %0.4f    w0: %0.2f, w1: %0.2f, w2: %0.2f',
                    i, mse, weights[1], weights[2], weights[3])) +
      theme_classic(12) + coord_cartesian(xlim = c(2.5, 8), ylim = c(1.0, 4.0)) +
        scale_color_manual(values = c('firebrick', 'darkgreen')))

  learning_data <- results[1:i, ]
  print(ggplot(learning_data, aes(x = iteration, y = mse)) +
    geom_point(color = 'firebrick', shape = 4) +  ggtitle("Learning curve"))
  }

}
```

3.3 C. Show results from the iris data set

I will show two different results of the algorithm, one starting with an initially too high decision boundary, and the other with an initially too low decision boundary. The algorithms are run until the maximum number of steps is reached, the square of the sum of the gradients is below the specified tolerance, or the mse increases for 3 iterations in a row. The results can then be plotted to visualize the change in decision boundary and the learning curve.

```
# Start with a high decision boundary
results_high <- grad_descent_complete(iris_subset, initial_weights = c(-6, 0.4, 1.2),
                      n_steps = 500, learning_rate = 0.005, tolerance = 0.005)
```

```
## [1] "Iteration: 50, mse: 0.1184"
## [1] "Iteration: 100, mse: 0.1086"
## [1] "Iteration: 150, mse: 0.1073"
## [1] "Iteration: 200, mse: 0.1069"
## [1] "Iteration: 250, mse: 0.1067"
## [1] "Iteration: 300, mse: 0.1066"
## [1] "Iteration: 350, mse: 0.1064"
## [1] "Iteration: 400, mse: 0.1062"
## [1] "Iteration: 450, mse: 0.1061"
## [1] "Iteration: 500, mse: 0.1059"
## [1] "Maximum number of iterations reached."
```

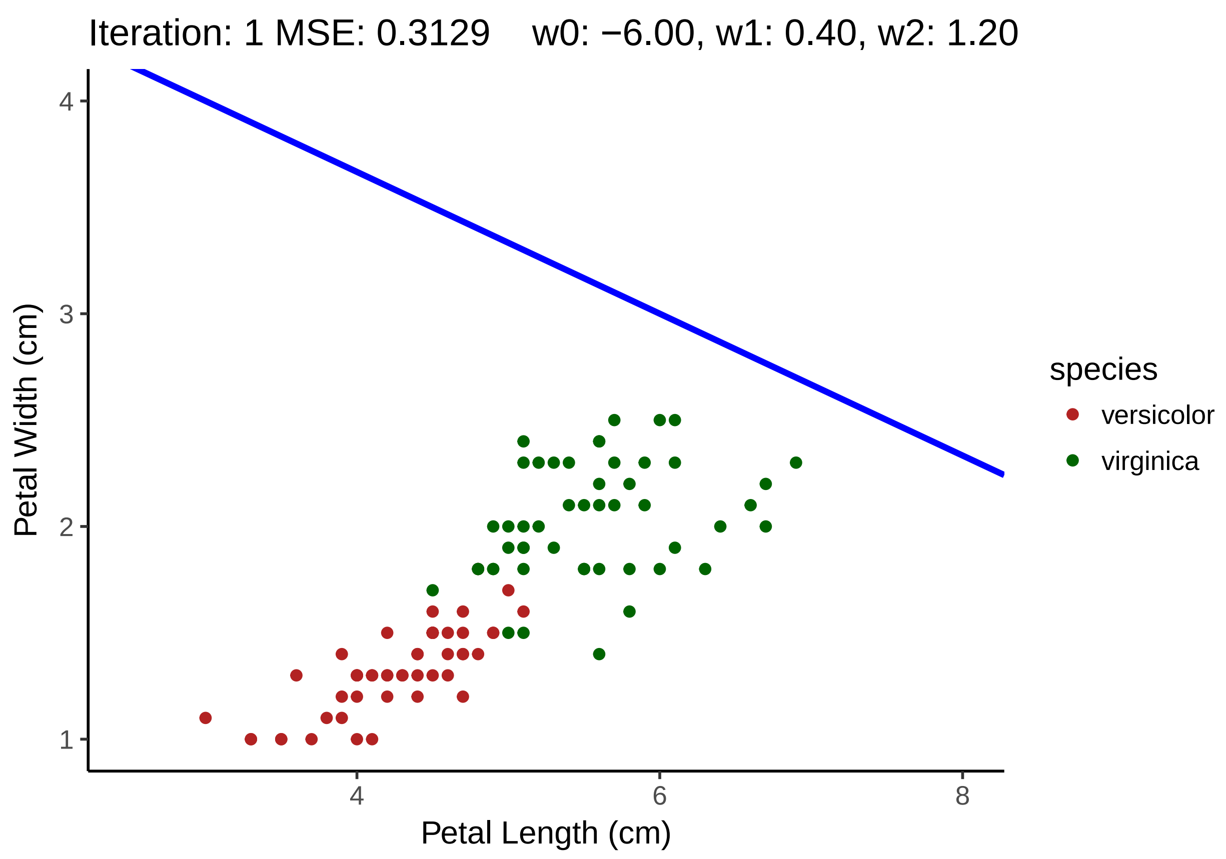
```
# Start with a low decision boundary
results_low <- grad_descent_complete(iris_subset, initial_weights = c(-3.6, 0.6, 1.2),
                      n_steps = 500, learning_rate = 0.005, tolerance = 0.005)
```

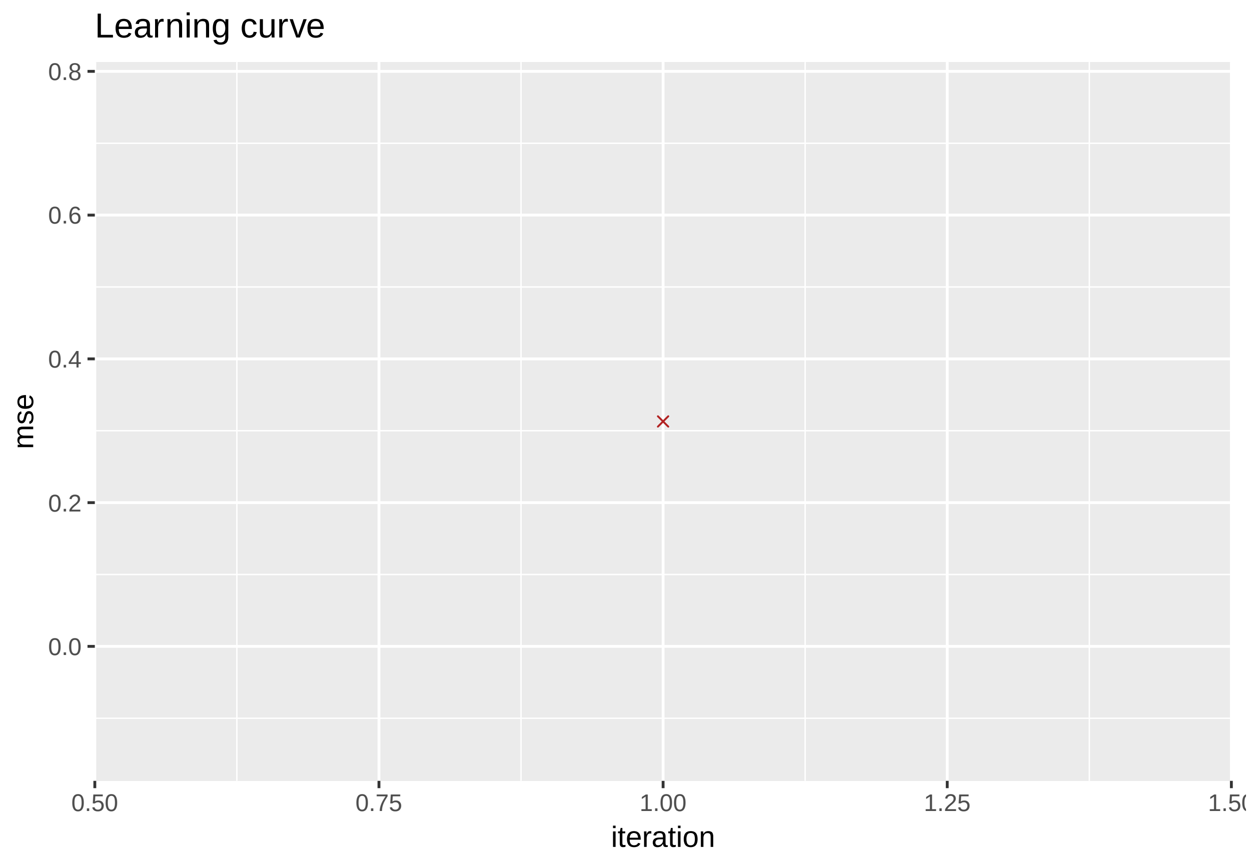
```
## [1] "Iteration: 50, mse: 0.1333"
## [1] "MSE Inceasing."
```

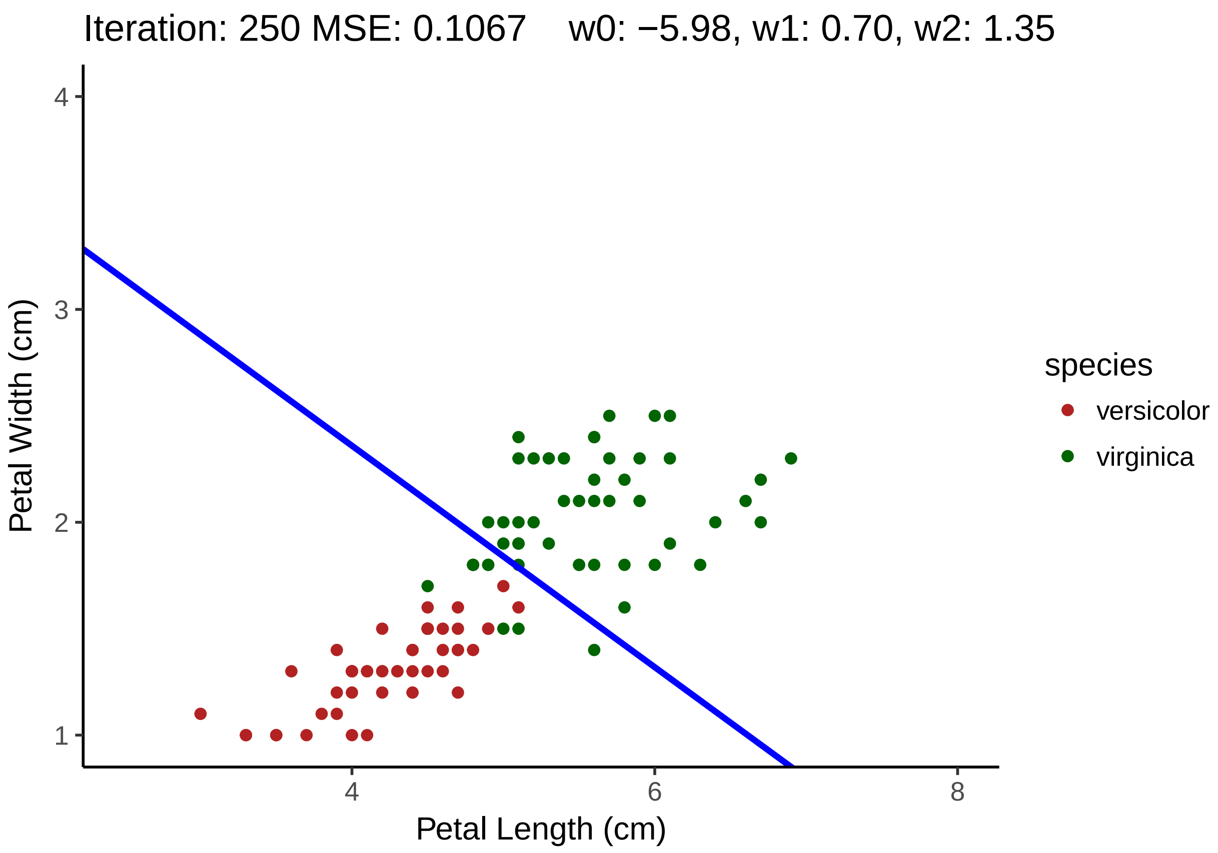
The model that started with the high decision boundary continued to improve until the maximum number of iterations was reached while the decision boundary that started too low stopped after a number of iterations when the mse started to increase, indicating the algorithm was jumping around the minimum. Gradient descent is only guaranteed to find a minimum if the step size is small enough to actually reach the minimum, otherwise it can bounce around the minimum and diverge. This is one reason to use a step size that decreases the number of iterations.

I can now plot the learning curves and decision boundary changes for the first run of the model.

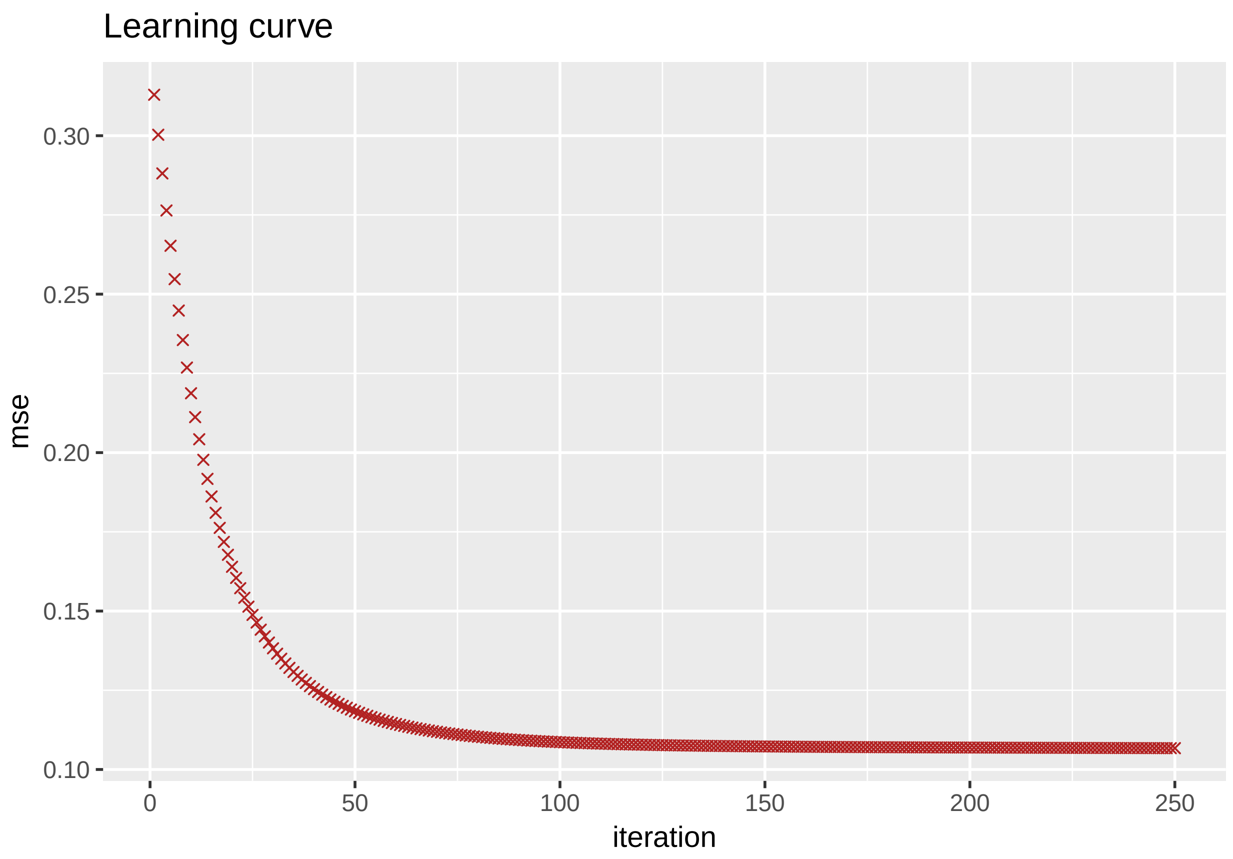
```
# Plot results for first run
results_high <- results_high[complete.cases(results_high), ]
plot_decision_learning(results_high, iris_subset)
```

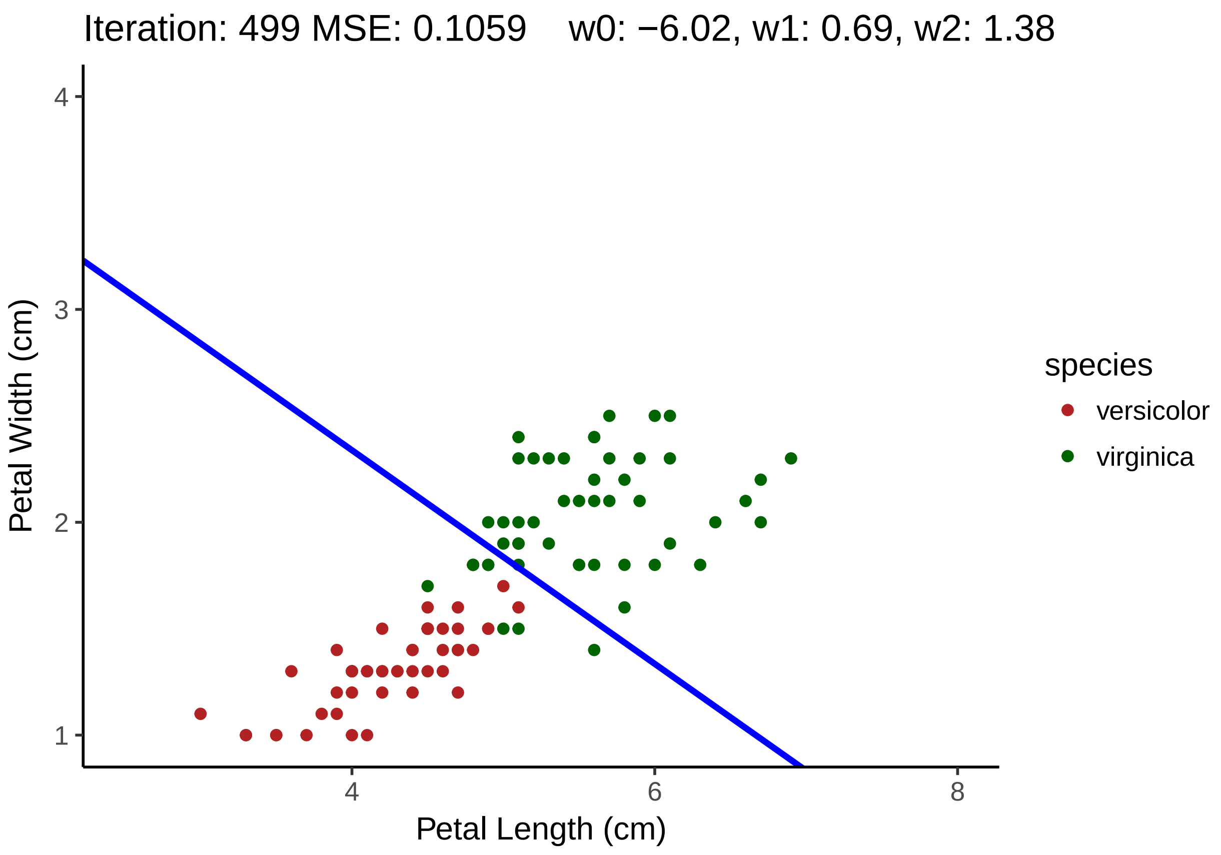

Iteration: 1 MSE: 0.3129   w0: −6.00, w1: 0.40, w2: 1.20
Petal Width (cm)
Petal Length (cm)
1
2
3
4
4
6
8
species
versicolor
virginica

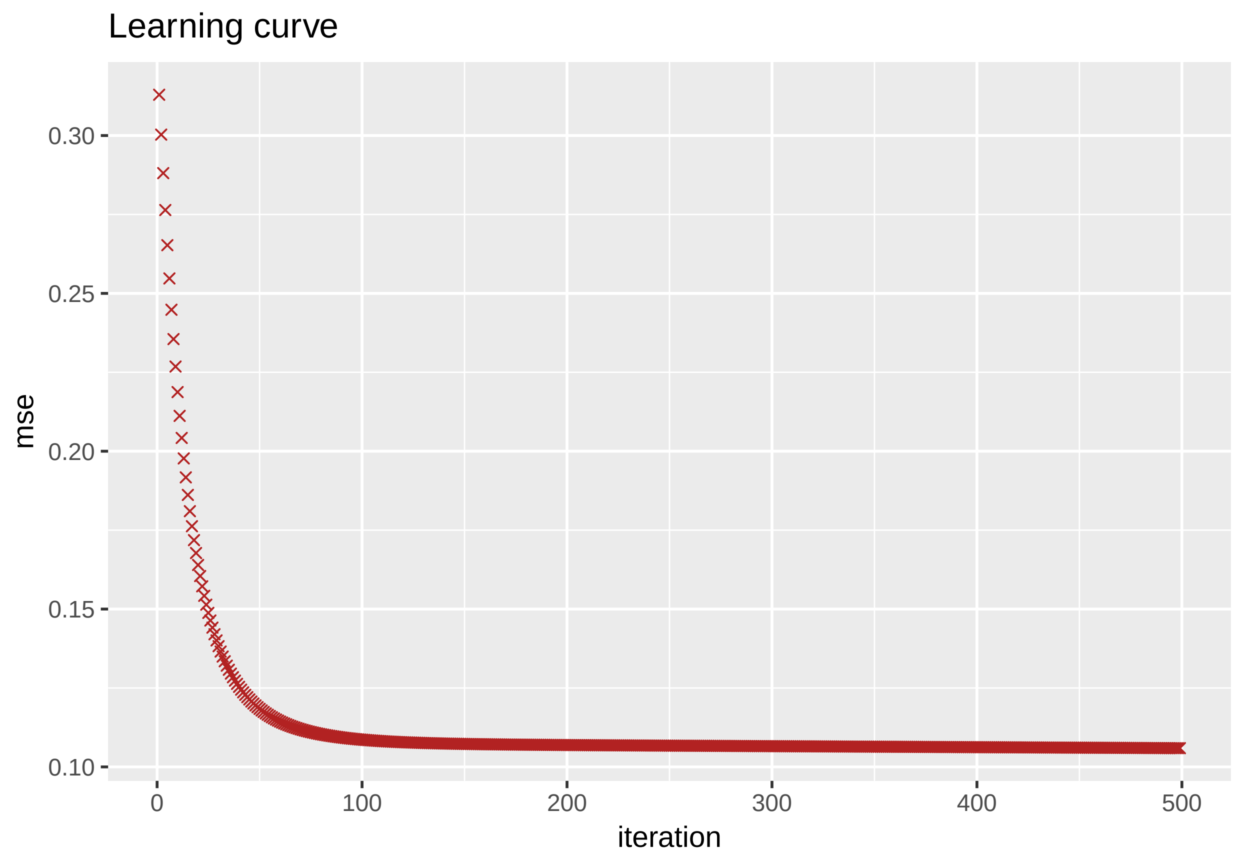

Learning curve
0.8
0.6
0.4
0.2
0.0
mse
0.50
0.75
1.00
1.25
1.50
iteration



Iteration: 250 MSE: 0.1067 w0: −5.98, w1: 0.70, w2: 1.35
Petal Width (cm)
Petal Length (cm)
4
3
2
1
4
6
8
species
versicolor
virginica

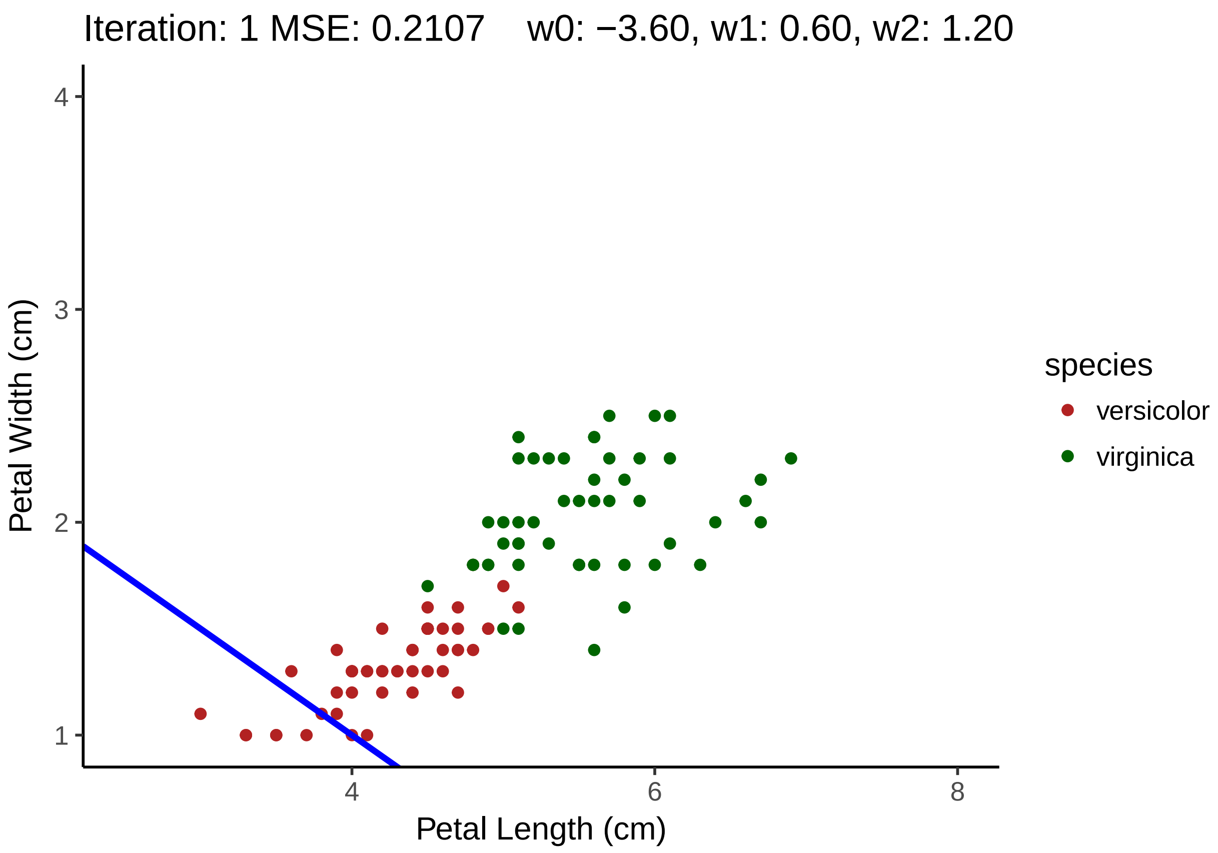

Learning curve
mse
0.30
0.25
0.20
0.15
0.10
iteration
0
50
100
150
200
250

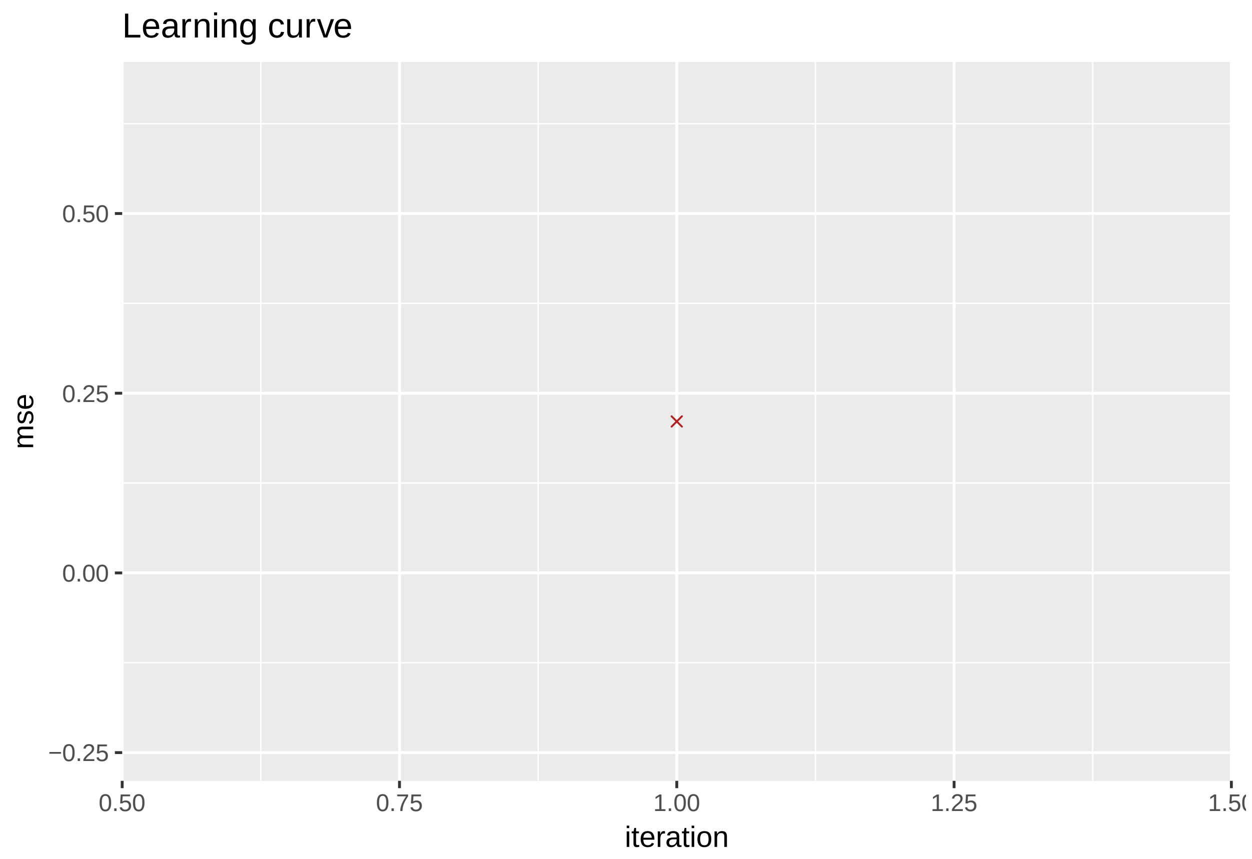

Iteration: 499 MSE: 0.1059   w0: −6.02, w1: 0.69, w2: 1.38
Petal Width (cm)
Petal Length (cm)
4
3
2
1
4
6
8
species
versicolor
virginica

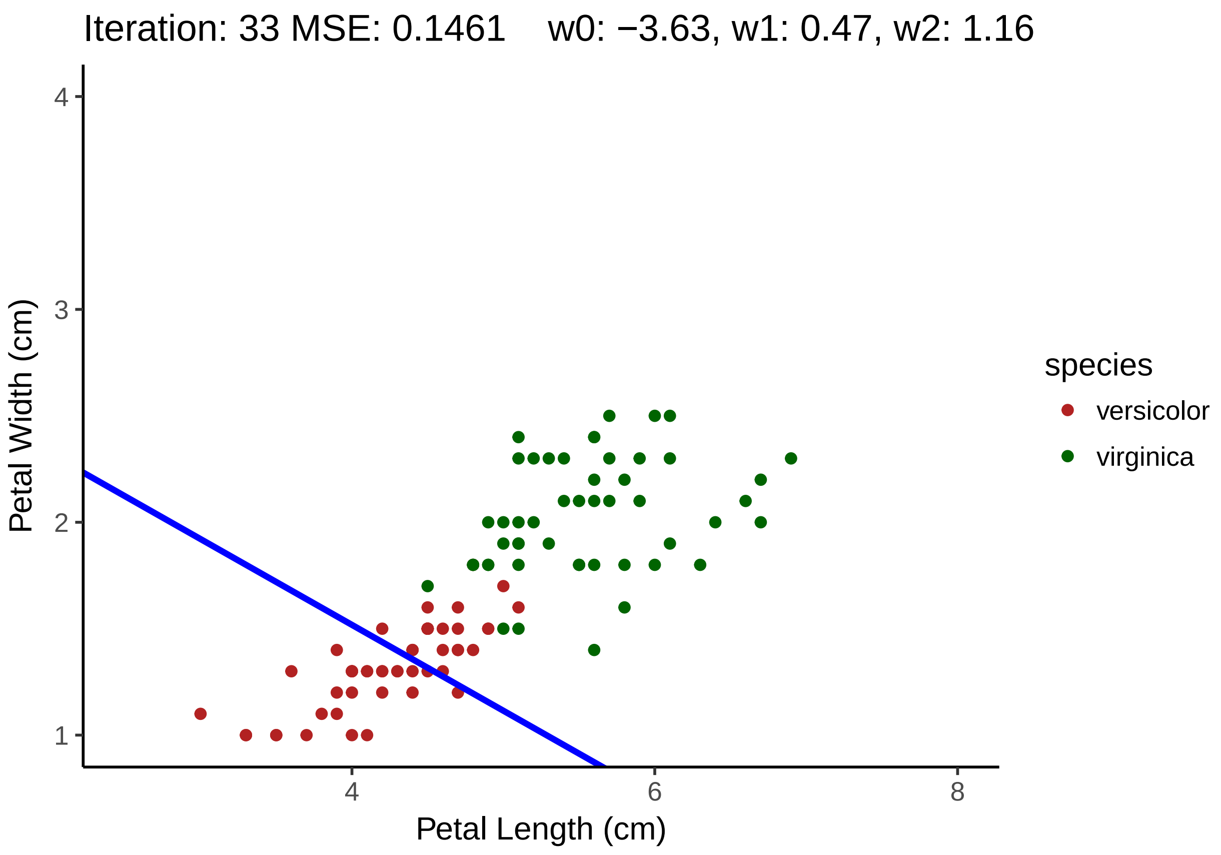


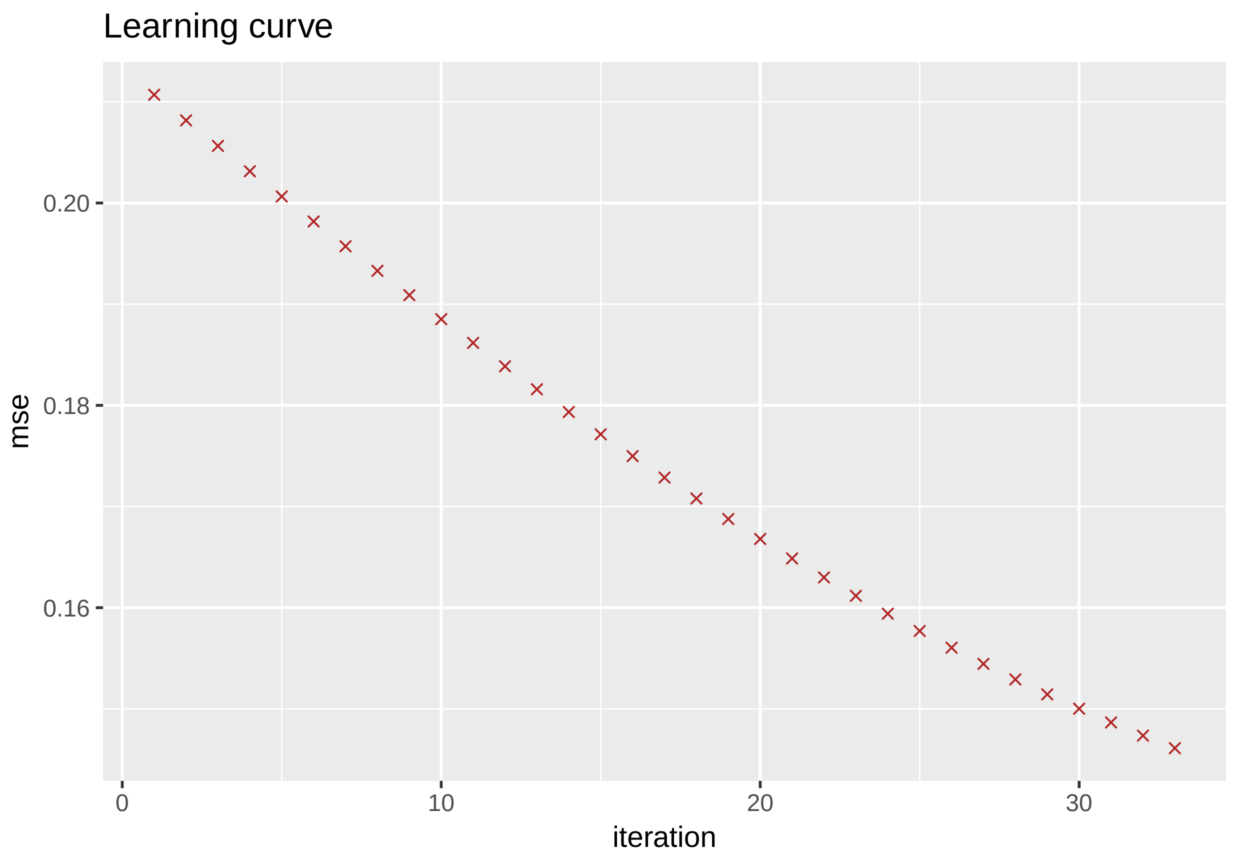
The results clearly show the mse decreasing with the number of iterations. We can also plot an example where the decision boundary starts too low to verify the algorithm works from both directions as expected.

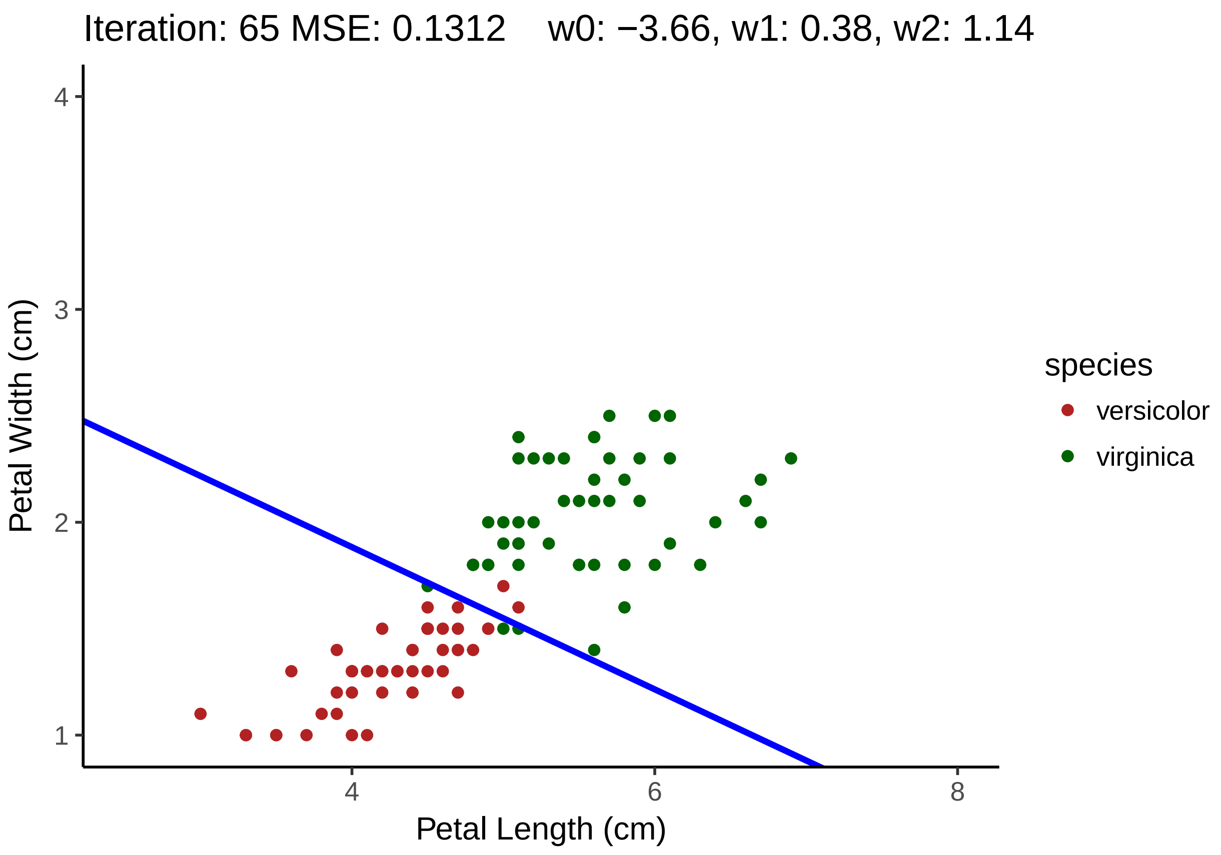
```r
# Plot results for second run
results_low <- results_low[complete.cases(results_low), ]
plot_decision_learning(results_low, iris_subset)
```

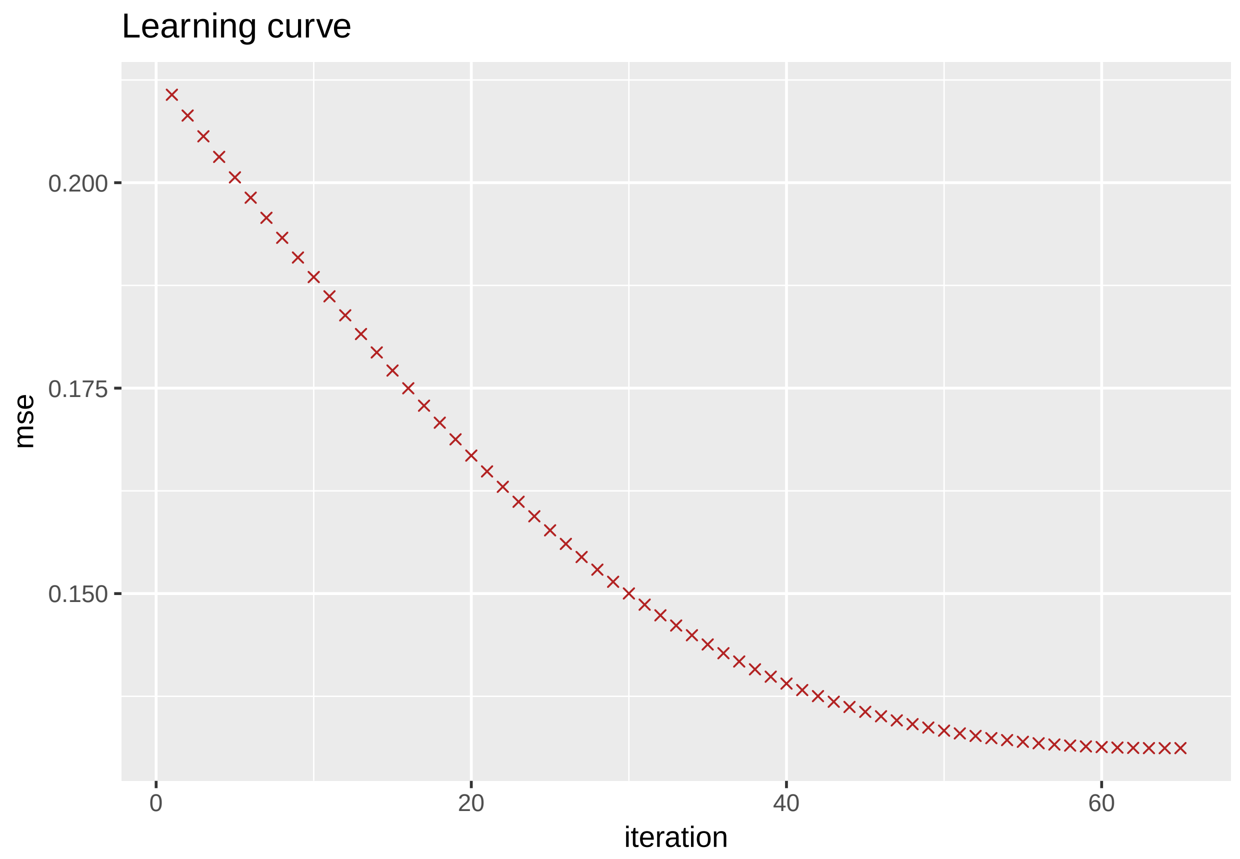

Iteration: 1 MSE: 0.2107   w0: −3.60, w1: 0.60, w2: 1.20
Petal Width (cm)
Petal Length (cm)
1
2
3
4
4
6
8
species
versicolor
virginica


Learning curve
mse
0.50
0.25
0.00
−0.25
iteration
0.50
0.75
1.00
1.25
1.50


Iteration: 33 MSE: 0.1461   w0: −3.63, w1: 0.47, w2: 1.16
Petal Width (cm)
Petal Length (cm)
4
3
2
1
4
6
8
species
versicolor
virginica


Learning curve
mse
0.20
0.18
0.16
iteration
0
10
20
30


Iteration: 65 MSE: 0.1312   w0: −3.66, w1: 0.38, w2: 1.14
Petal Width (cm)
Petal Length (cm)
4
3
2
1
4
6
8
species
versicolor
virginica



In both cases, the algorithm is able to correctly adjust the model parameters to minimize the mean-squared error objective function. The algorithm correctly implements gradient descent to learn the optimal decision boundary for the iris data set.

3.4 D. How to Choose a learning step

The learning rate was selected by trial and error. When choosing an error rate, one can make the error of choosing too large or too small. Too great of a learning rate and the algorithm will never converge but will "jump" around the minimum. Too tiny of a learning rate and the algorithm will converge, but it will take an extremely long time to do so. I experimented with a number of different learning rates and eventually decided on 0.005. This seemed to converge in a relatively short number of iterations and did result in significant divergence from the minimum mse. The learning rate was verified in the above applications of the algorithm. If this were a tougher problem, I would use a learning rate schedule wherein the learning rate decreases with the number of iterations.

Two common learning rate schedules are as follows where ϵ_0 is the initial learning rate.

Linear learning rate decay:

$$\epsilon = \epsilon_0 / t$$

Exponential learning rate decay:

$$\epsilon = \epsilon_0 * e^{-t}$$

The benefit of a learning rate schedule is that one can begin a problem with a higher learning rate to quickly reach the vicinity of the minimum, and then decrease the learning rate to converge on the minimum of the objective function. Choosing the correct learning rate schedule can make a significant difference in the runtime and accuracy of a model.

3.5 E. How to Choose a stopping criteria

I instituted three different checks for the stopping criteria. The first was based on a tolerance threshold for the minimum change in the gradients. If the gradients are not changing above a certain threshold (set at 0.005), then the algorithm should stop. I used the square root of the sum of the gradients squared to measure the magnitude of the gradients. This stopping criterion is effective because the magnitude of the gradients is directly proportional to the error, and therefore, when the gradient decreases below the threshold, that means the error is not decreasing rapidly anymore and further iterations are unnecessary.

The second stopping measure was if the mean-squared error increased for 3 iterations in a row.
This can occur if the algorithm misses the absolute minimum and begins to bounce around the true minimum. The cause of this would be a learning rate that is too great. When running the model with an initially low decision boundary, the algorithm stopped because the mean squared error was increasing. Overall, the algorithm should be able to find a value for the mse within a small distance of the absolute minimum depending on the step size.

The third stopping criteria was the maximum number of iterations. The first run of the model, with the elevated decision boundary, stopped because the maximum number of iterations was reached. However, it is not likely there were significant opportunities for optimization beyond this point because the decrease in the mean-squared error was small. The maximum number of iterations was ultimately set at 500 after a few runs for experimentation. The maximum iterations could be raised though because of the low run time of the model.

The algorithm developed in this report was able to find an optimal decision boundary for the iris classification task to the extent that a linear classifier can separate the two species. In order to achieve 100% accuracy at this task, a non-linear decision boundary would be required, which would necessitate a flexible algorithm, such as a random forest or a piecewise linear model.