

# 1 Linear Decision Boundaries

## 1.1 Plotting Iris Classes

This project is written in Python, using Pandas dataframes to manipulate data and Matplotlib to plot it.

The data is initialized like so,

```
# Read the file into a Pandas dataframe
iris_data = pd.read_csv(filename)
# Capitalize the column names and remove underscores
iris_data.columns = ["Sepal Length", "Sepal Width", "Petal Length",
                    "Petal Width", "Species"]
# Remove any species other than those specified by the assignment
iris_data = iris_data.loc[iris_data["Species"].isin(["versicolor", "virginica"])]
# Select the versicolor and virginica species individually
versi_data = iris_data.loc[iris_data["Species"] == "versicolor"]
virgi_data = iris_data.loc[iris_data["Species"] == "virginica"]
```

The assignment asks us to consider only the versicolor and virginica species. Let's take a look at the first few rows of each.

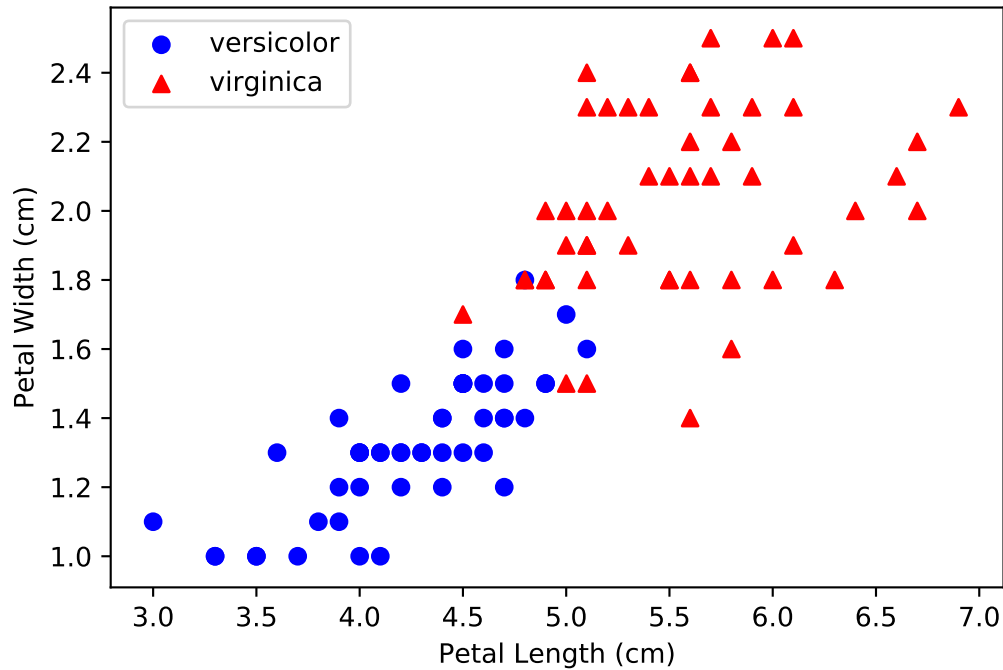
```
In [2]: versi_data.head(3)
   Sepal Length  Sepal Width  Petal Length  Petal Width  Species
50           7.0           3.2           4.7           1.4  versicolor
51           6.4           3.2           4.5           1.5  versicolor
52           6.9           3.1           4.9           1.5  versicolor
```

```
In [3]: virgi_data.head(3)
   Sepal Length  Sepal Width  Petal Length  Petal Width  Species
100           6.3           3.3           6.0           2.5  virginica
101           5.8           2.7           5.1           1.9  virginica
102           7.1           3.0           5.9           2.1  virginica
```

We are interested in the petal dimensions. To plot them:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(x=dataset_1["Petal Length"], y=dataset_1["Petal Width"],
          color="b", marker="o", label="versicolor")
ax.scatter(x=dataset_2["Petal Length"], y=dataset_2["Petal Width"],
          color="r", marker="^", label="virginica")
plt.xlabel("Petal Length (cm)")
plt.ylabel("Petal Width (cm)")
plt.legend(loc="upper left")
plt.savefig("plot_1a.pdf", bbox_inches="tight")
```

This gives us



It is clear that at least two points of different classes overlap, which means that our data is not going to be linearly separable.

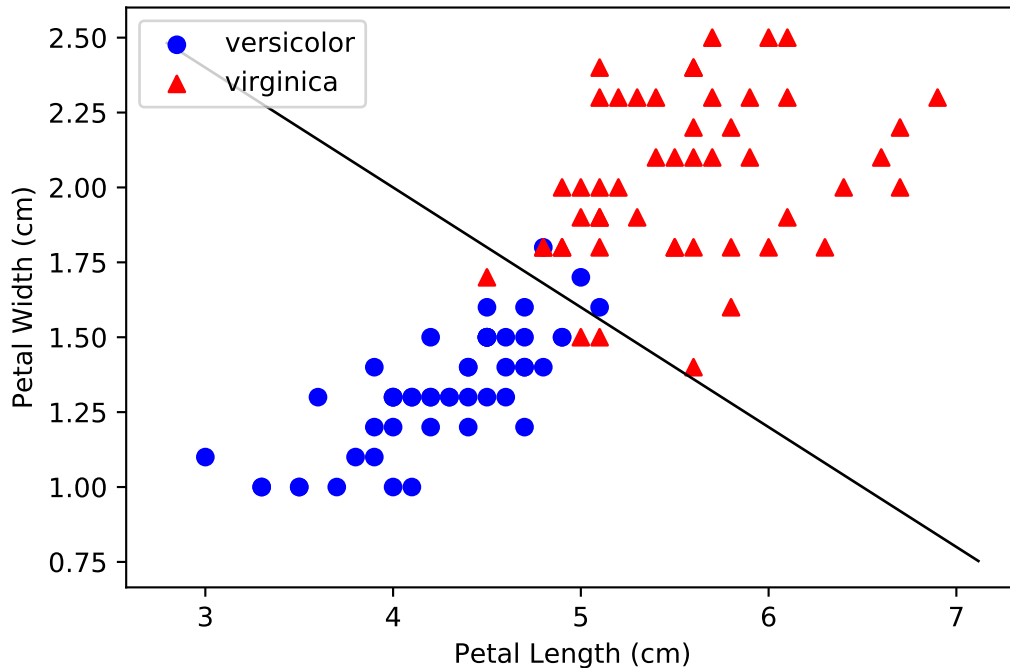
## 1.2 Choosing a Boundary by Hand

I chose a boundary by physically drawing a line on the above plot using an image editor. The line reaches from 2.4 on the y-axis to 6.5 on the x-axis. Then we can get an equation in slope-intercept form from the coordinates  $(3, 2.4)$ ,  $(6.5, 1)$ .  $y = mx + b$  where  $m = -0.4$  and  $b = 3.6$ .

The code to generate this plot is identical to the above plot, except the line

```
plt.plot([3, 6.5], [2.4, 1], color="k", linestyle="-", linewidth=1)
```

is now added before saving.



### 1.3 Defining a Threshold Classifier

We can now define a threshold classifier using the above parameters to compare against our data points. Simply put, if the point is above the line, it is predicted to be virginica. Otherwise it is predicted to be versicolor.

The calculation is shown here, where  $x$  and  $y$  correspond to *petal length* and *petal width*, respectively.

```
""" Returns the distance from point (x,y) to the line in slope-intercept form """
def dist_from_line(x, y, m, b):
    return y - (m * x + b)

""" Returns "virginica" if the point (length, width) is above the line """
def classify_linear(row, slope, intercept):
    if 0 < dist_from_line(row["Petal Length"], row["Petal Width"], slope, intercept):
        return "virginica"
    return "versicolor"
```

To plot this data, we isolate the misclassified points into their own dataframes. Misclassified points are defined as those for which *Species*  $\neq$  *Classification*.

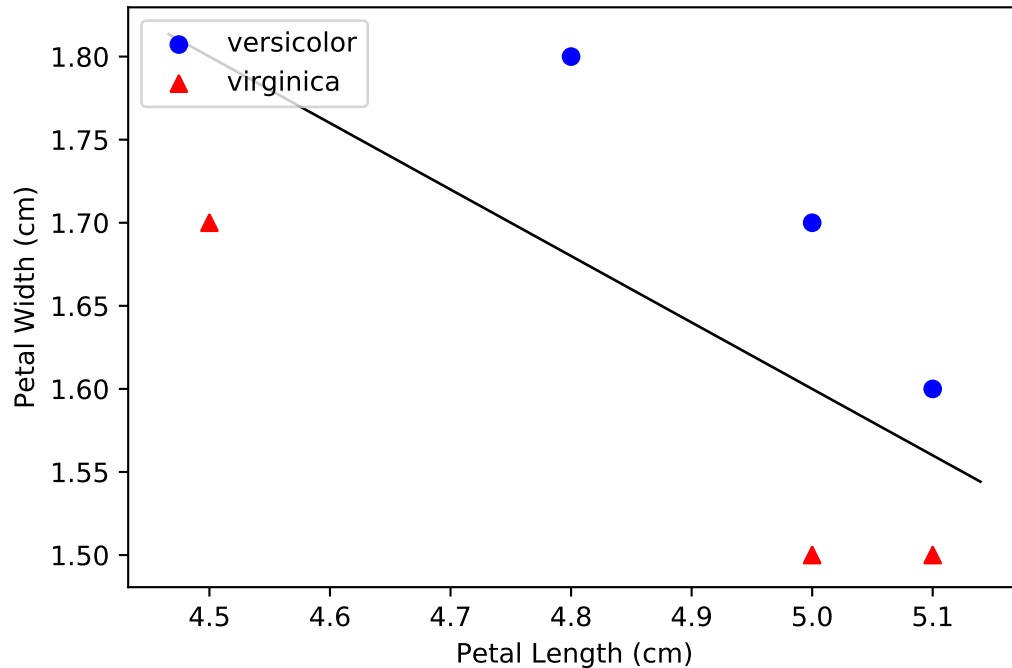
```
# Classify data linearly for part 1c
iris_data["Classification"] = iris_data.apply(
    lambda row: classify_linear(row, -0.4, 3.6), axis=1
)
misclassified = pd.DataFrame(columns=iris_data.columns)

for i, row in iris_data.iterrows():
    if not row["Classification"] == row["Species"]:
        misclassified.loc[len(misclassified.index)] = row
```

```
print("Linear decision accuracy: {}".format(100 - len(misclassified.index) /
      len(iris_data.index) * 100))

versi_class = misclassified.loc[misclassified["Species"] == "versicolor"]
virgi_class = misclassified.loc[misclassified["Species"] == "virginica"]
plot_linear_bound(versi_class, virgi_class, "plot_1c.pdf")
```

This threshold classifier misclassifies six data points, as shown in the next plot. Note that points *above* the line are supposed to be virginica (red triangles), and points *below* the line are supposed to be versicolor (blue circles), but the opposite is true of misclassified points.



## 1.4 Circle Decision Boundaries

Instead of defining our decision boundary with a straight line, we can use a circle instead. The technique is the same, although in this case the decision is made considering whether a point is inside or outside of the circle, rather than above or below a line. The functions are similar to those first shown in section 1.3.

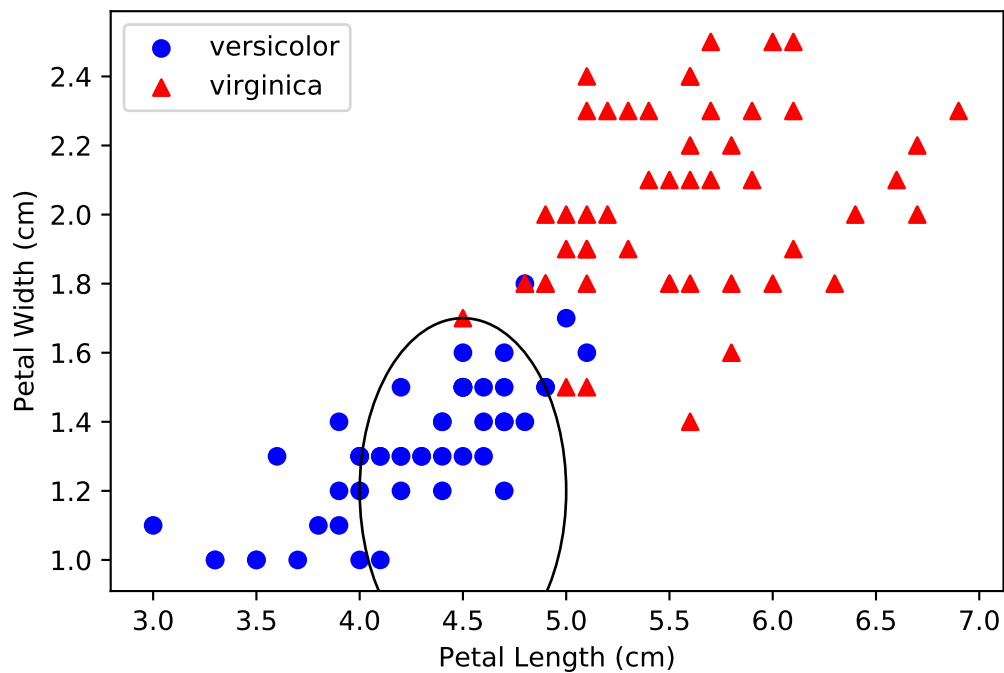
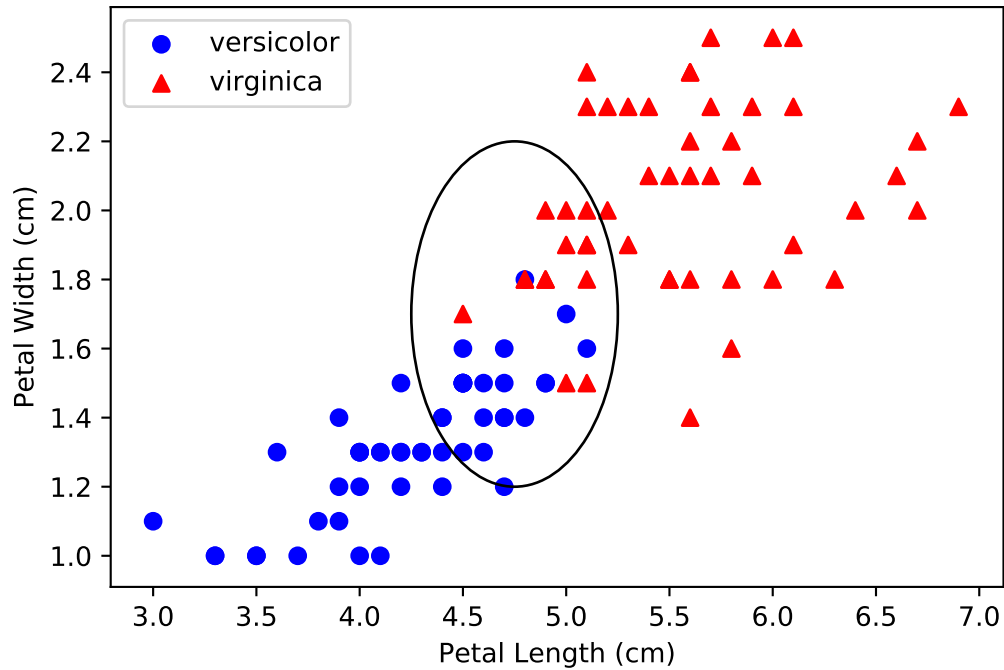
```
""" Returns the distance from point (x,y) to the edge of the circle """
def dist_from_circle(x, y, x_circ, y_circ, r):
    return ((x - x_circ)**2 + (y - y_circ)**2)**0.5 - r

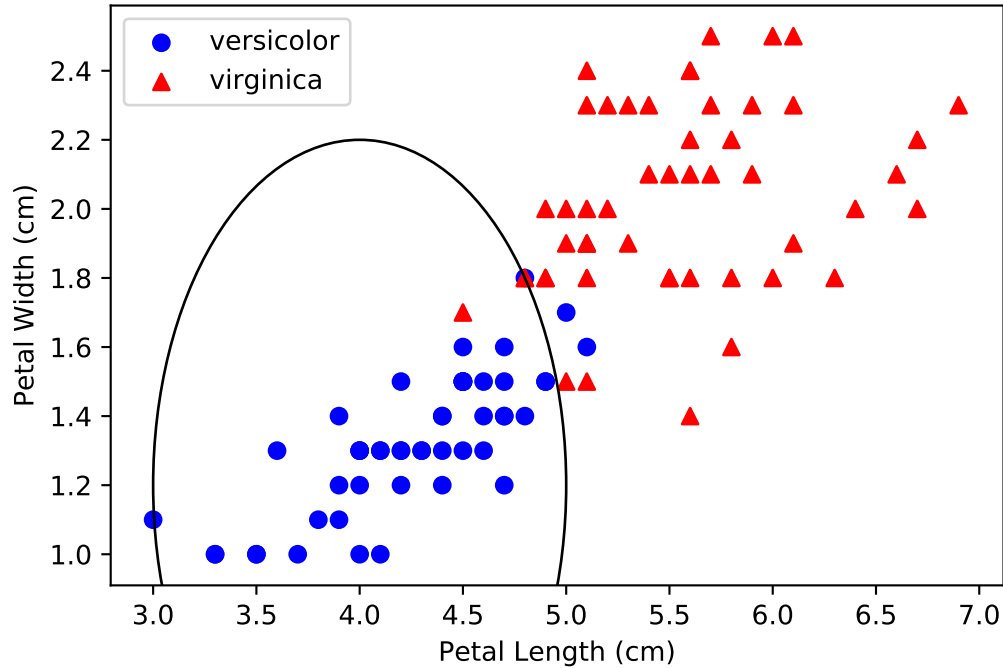
""" Returns "virginica" if the point (length, width) is outside of the circle """
def classify_circular(row, x, y, radius):
    if 0 < dist_from_circle(row["Petal Length"], row["Petal Width"], x, y, radius):
        return "virginica"
    return "versicolor"
```

Now we need to pick parameters for our circle  $(x_0, y_0, r)$ . The assignment instructs us to pick three sets. I started at the center of the plot with an arbitrary radius of  $\frac{1}{2}$ . This, predictably, didn't give great results, but with two adjustments the decision accuracy eventually met that of the linear threshold (94%). The parameters and results are shown below, followed by the plot of each. Note

that because the  $x, y$  scales are not 1 : 1, the circle is *mathematically* circular, *not graphically* circular.

$x$	$y$	$r$	Accuracy
4.75	1.7	0.5	58%
4.5	1.2	0.5	79%
4.0	1.2	1	94%





The plotting code is easy to imagine. Instead of drawing a line, just draw a circle according to

```
circle = plt.Circle((x, y), r, color="k", fill=False)
ax.add_artist(circle)
```

## 2 Objective Functions

### 2.1 Calculating Mean-Squared Error

The mean squared error is given by

$$E = \frac{1}{N} \sum_{n=1}^N (\mathbf{w}^T \mathbf{x}_n - c_n)^2$$

In our case the predicted and actual values are  $\mathbf{w}^T \mathbf{x}_n$  and  $c_n$ , where class of  $x$  in pattern vector  $\mathbf{x}_n$  is given by

$$x \in \begin{cases} \text{class 1} & \text{if } y \geq 0 \\ \text{class 2} & \text{if } y < 0 \end{cases}$$

and we can number the classes 0 or 1 to give  $c_n$  a value according to

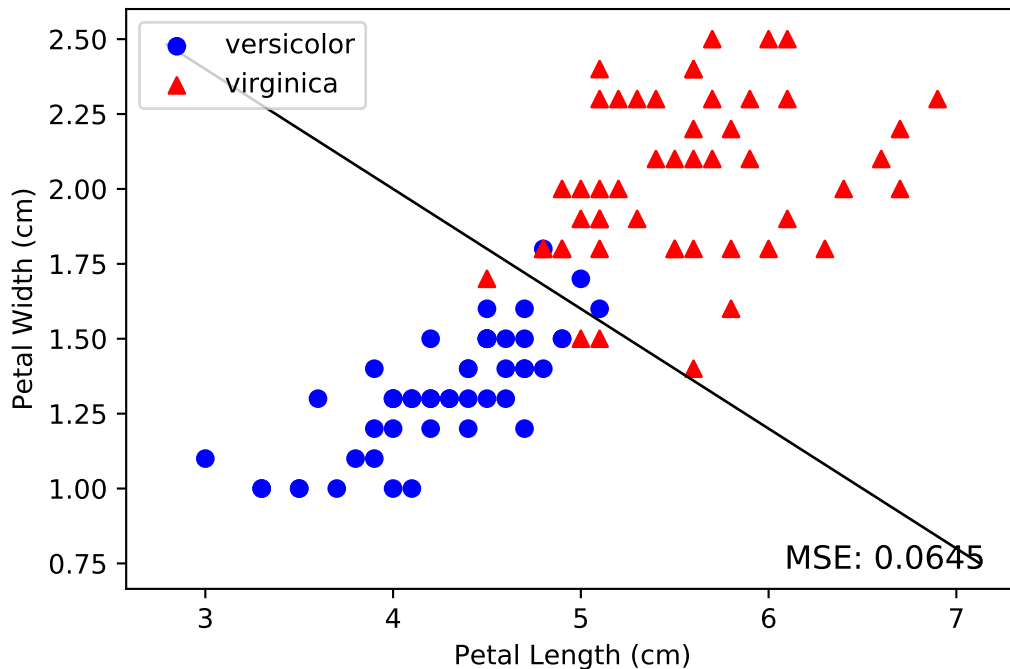
$$c_n = \begin{cases} 0 & \mathbf{x}_n \in \text{class 1} \\ 1 & \mathbf{x}_n \in \text{class 2} \end{cases}$$

However by this definition,  $(\mathbf{x}_n - c_n) \in \{-1, 0, 1\}$  and therefore  $(\mathbf{x}_n - c_n)^2 \in \{0, 1\}$ . The mean of zeroes and ones doesn't give us a very precise error value, and so we can redefine the first term. Rather than assigning a zero or a one based on the predicted class, we can plug distance into a modified version of the sigmoid function, shown below.

$$p = \frac{1}{1 + 10^{-d}}$$

where  $d$  is the distance from the decision boundary and  $p$  is the predictive value.  $p \in [0, 1]$ , which changes the prediction from a discrete set of one and zero to a continuum from zero to one. By making this change, we quantify our predictions with a degree of confidence and theoretically make our MSE value more accurate.

From now on the MSE value will be superimposed over the graph. The current boundary gives a value of 0.0645.



```
if show_mse:
    combined_data = pd.concat([dataset_1, dataset_2])
    mse = mean_squared_error(combined_data, slope, intercept)
    plt.text(0.85, 0.05, "MSE: %.4f" % mse, ha='center', va='center',
             fontsize=12, transform=ax.transAxes)
```

The mean squared error calculation is calculated like so

```
def mean_squared_error(dataset, m, b, classes={"versicolor": 0, "virginica": 1}):
    sum_diffs = 0
    for i, row in dataset.iterrows():
        dist = dist_from_line(m=m, b=b, row=row)
        sum_diffs += (classes[row["Species"]] -
                     classes[classify_prediction(dist)] * sigmoid(dist))**2
    return sum_diffs / len(dataset.index)

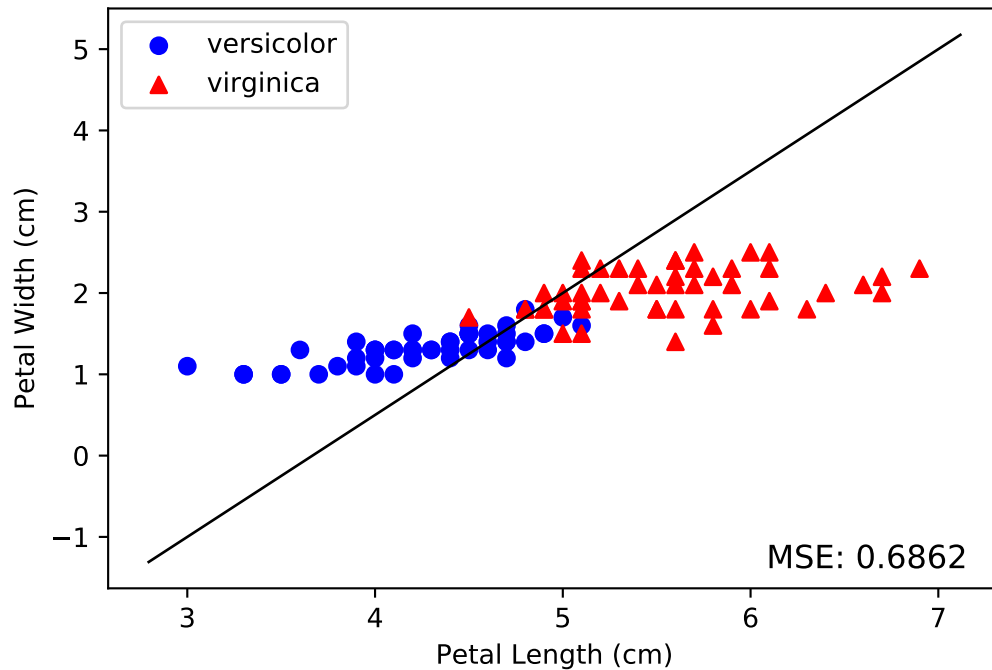
def sigmoid(x):
    return 1 / (1 + 10**(-x))
```

## 2.2 Examples of Large and Small Errors

As was shown in the previous section, the MSE value for a line with  $m = -0.4$ ,  $b = 3.6$  was quite small (0.0645). We can generate a large MSE value (0.6862) by using a positive slope. Here the boundary misclassifies the majority of the data points!

# Very high MSE

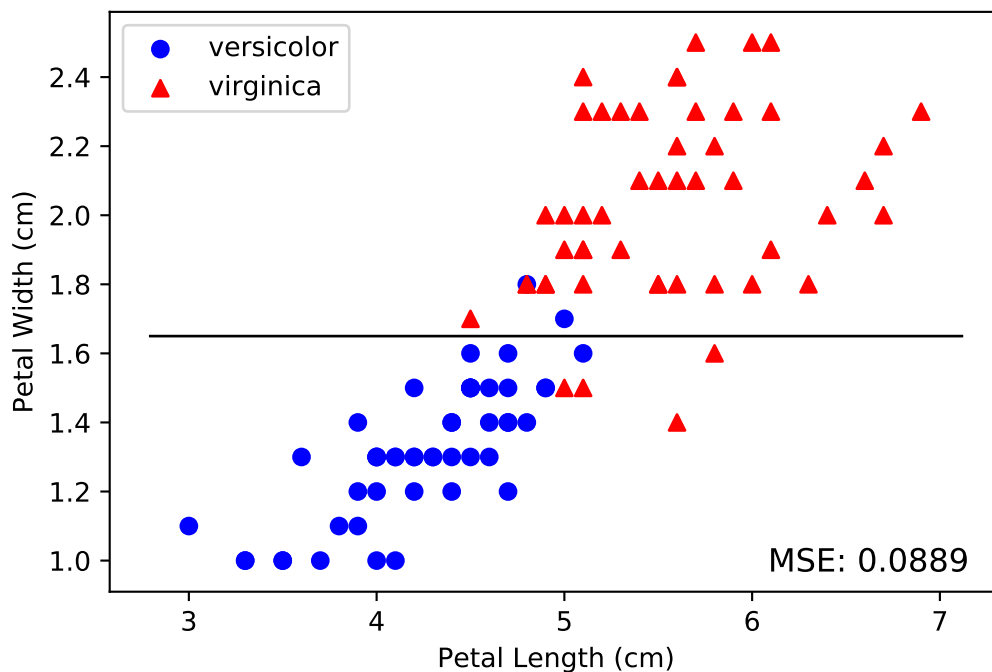
`plot_linear_bound(versi_data, virgi_data, 1.5, -5.5, show_mse=True, filename="plot_2b_1.pdf")`



To lower the MSE, we should move our parameters back to their “approximately optimal” values. Let’s see what a zero slope about halfway through the data gives us.

# Lower MSE

`plot_linear_bound(versi_data, virgi_data, 0, 1.65, show_mse=True, filename="plot_2b_2.pdf")`





0.0889 is much closer to our original value of 0.0645.

### 2.3 Derivation of the Gradient

Starting with the mean square equation from section 2.1,

$$E = \frac{1}{N} \sum_{n=1}^N (\mathbf{w}^T \mathbf{x}_n - c_n)^2$$

Where  $E$  is the mean-squared error,  $N$  is the number of data points (in this case 100),  $\mathbf{w}^T$  is the vector of weights,  $\mathbf{x}_n$  is the vector of data point classifications, and  $c_n$  is its actual class (in this case species).

As was previously stated, I am improving the error function with a modified sigmoid curve. The distance value supplied to that function is calculated with

$$d = W_n - (w_1 \cdot L_n + w_2)$$

where  $W_n$  is the petal width,  $L_n$  is the petal length,  $w_1$  is the first weight corresponding to slope,  $w_2$  is the second weight corresponding to intercept.

The modified sigmoid function is then used to get a bias term ( $w_0$ ).

$$w_0 = S(d) = \frac{1}{1 + 10^{-d}}$$

The derivative of this sigmoid function is

$$S'(d) = S(d) \cdot (1 - S(d)) \cdot \ln|10|$$

If we combine these equations, the MSE equation is written

$$E = \frac{1}{N} \sum_{n=1}^N (S(W_n - (w_1 \cdot L_n + w_2)) \cdot \mathbf{x}_n - c_n)^2$$

The gradient for this MSE involves differentiating its equation, as is done below.

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{w}} &= \frac{2}{N} (1 - S(\mathbf{x}\mathbf{w})) (S(\mathbf{x}\mathbf{w}) - \mathbf{c}) \mathbf{x} \\ &= \frac{2}{N} (1 - \mathbf{p}) (\mathbf{p} - \mathbf{c}) \mathbf{x} \end{aligned}$$

At the end we replace the sigmoid function call with a vector of the predictive values that it returns, notated  $\mathbf{p}$ .

This is the vector form. The scalar form will be shown in the next section.

### 2.4 Gradients in Scalar and Vector Form

We can expand the vector form of the gradient into a scalar form. When expanded, it looks like:

$$\begin{aligned} \text{Scalar: } \frac{\partial E}{\partial w_i} &= \frac{2}{N} \sum_{n=1}^N (x_{i,n} \cdot (1 - S(x_n w)) \cdot (S(x_n w) - c_n)) \\ &= \frac{2}{N} \sum_{n=1}^N (w_0 x_{0,n} + \dots + w_i x_{i,n} + \dots + w_M x_{M,n} - c_n) x_{i,n} \end{aligned}$$

In this form the expression is summed over  $N$  terms. The vector form is similar but does not feature the summation, since it is implicit from the use of vectors instead of scalar variables (e.g.  $x_{i,n}$  and  $\mathbf{x}$ ).

$$\text{Vector: } \frac{\partial E}{\partial \mathbf{w}} = \frac{2}{N}(1 - S(\mathbf{x}\mathbf{w}))(S(\mathbf{x}\mathbf{w}) - c)\mathbf{x}$$

Then to update the gradient, we use an iteration function:

$$w_i^{t+1} = w_i^t - \varepsilon \frac{\partial E}{\partial w_i}$$

Epsilon here is a small value that ensure we eventually converge on a minimum value. At this point the difference between one MSE and the MSE of the next step is negligible.

## 2.5 Summing the Gradient for an Ensemble of Patterns

# 3 Learning a Decision Boundary Through Optimization

## 3.1 Implementing Gradient Descent

## 3.2 Showing the Learning Curve

## 3.3 Illustration with Random Weights

## 3.4 Choosing the Gradient Step Size

## 3.5 Choosing the Stopping Criterion