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"]
```

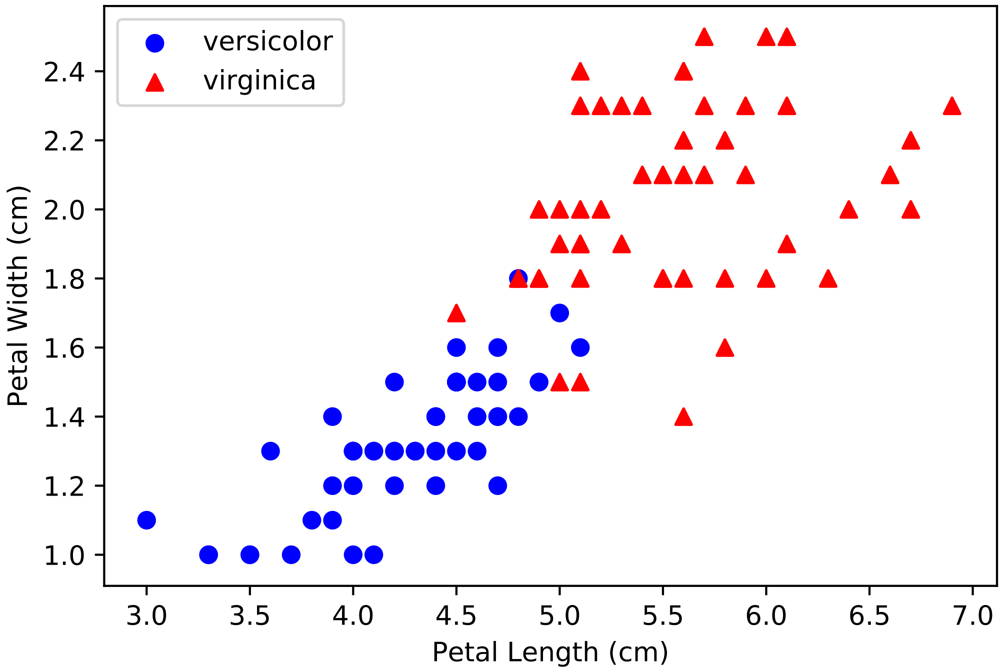
The assignment asks us to consider only the versicolor and virginica species. Let's take a look at the first few rows of each.

```
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.

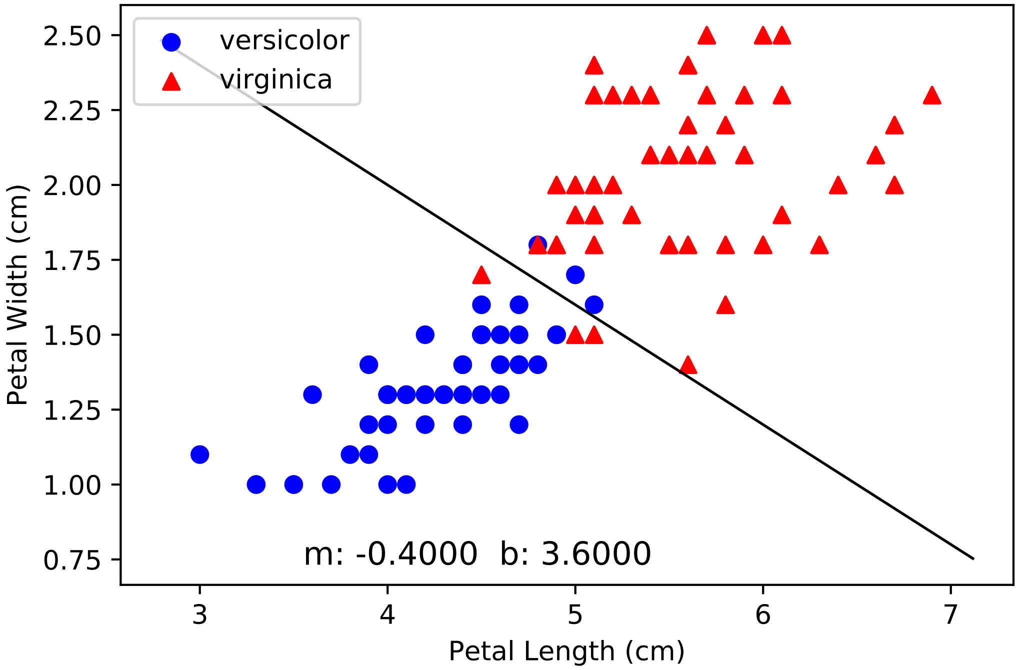
1.2 Choosing a Boundary by Hand

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.

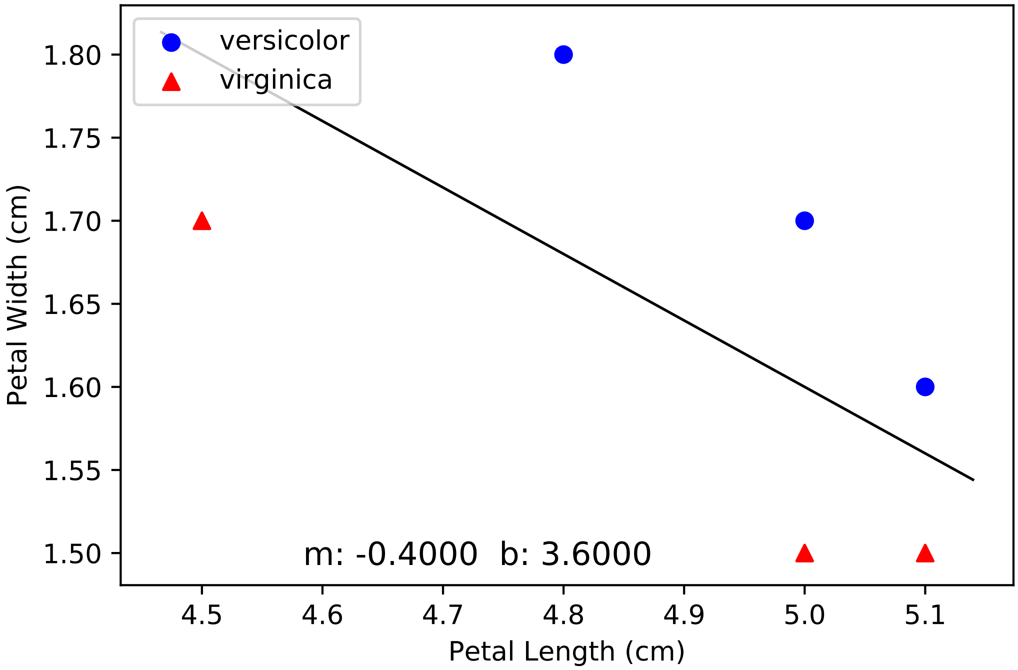
```
""" Returns the distance from point (x,y) to the line in slope-intercept form """
def dist_from_line(x, y, m, b):
    return y - (m * x + b)

""" 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.



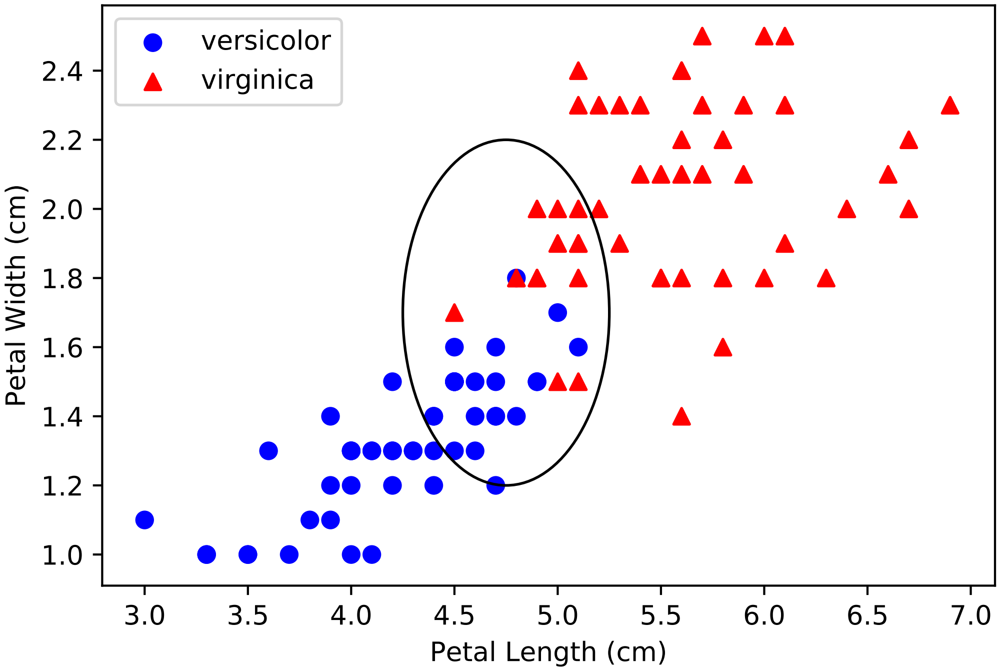
1.4 Circle Decision Boundaries

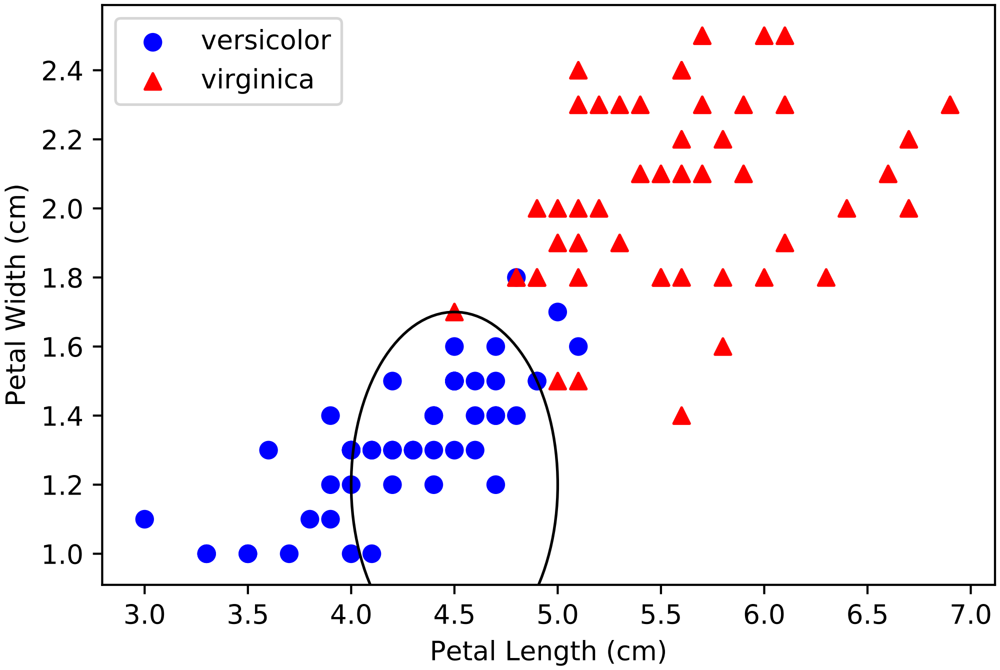
Instead of defining our decision boundary with a straight line, we can use a circle instead. The technique is the same, although in this case the decision is made considering whether a point is inside or outside of the circle, rather than above or below a line. The functions are similar to those first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

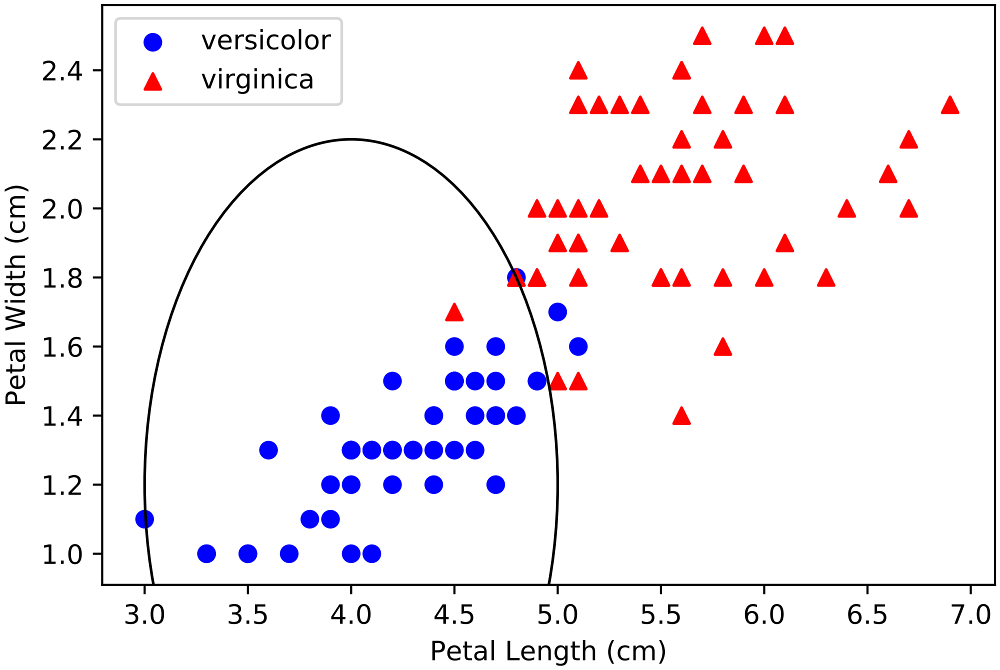
```
""" Returns the distance from point (x,y) to the edge of the circle """
def dist_from_circle(x, y, x_circ, y_circ, r):
    return ((x - x_circ)**2 + (y - y_circ)**2)**0.5 - r

""" 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

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

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

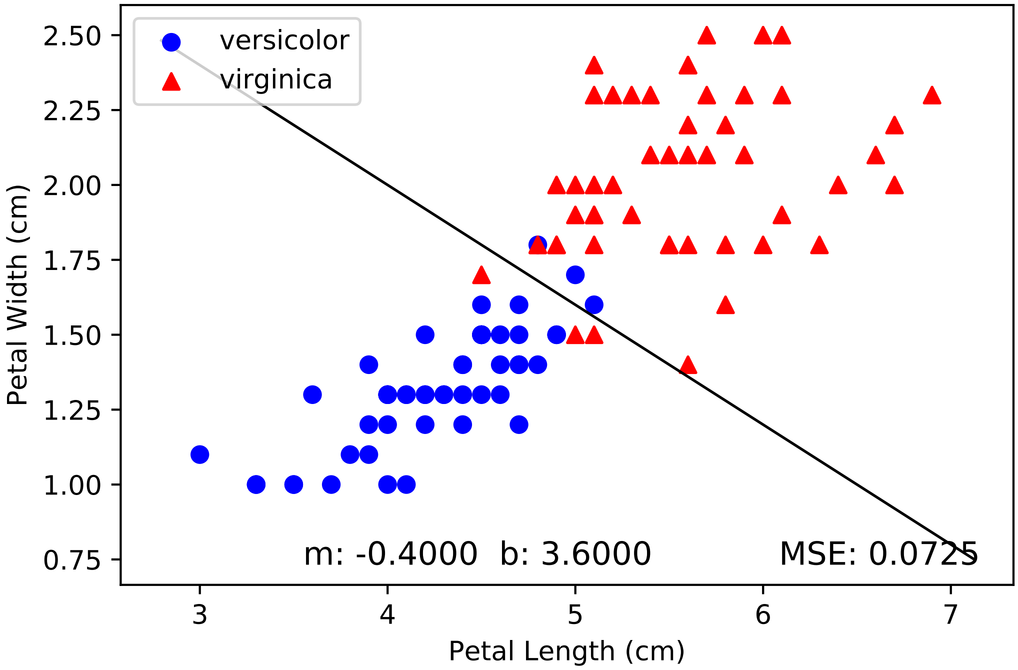
$$c_n = \begin{cases} 0 & \boldsymbol{x}_n \in \text{class 1} \\ 1 & \boldsymbol{x}_n \in \text{class 2} \end{cases}$$

However by this definition, $(\boldsymbol{x}_n - c_n) \subset \{-1, 0, 1\}$ and therefore $(\boldsymbol{x}_n - c_n)^2 \subset \{0, 1\}$. The mean of zeroes and ones doesn't give us a very percise 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 qualify 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)
```

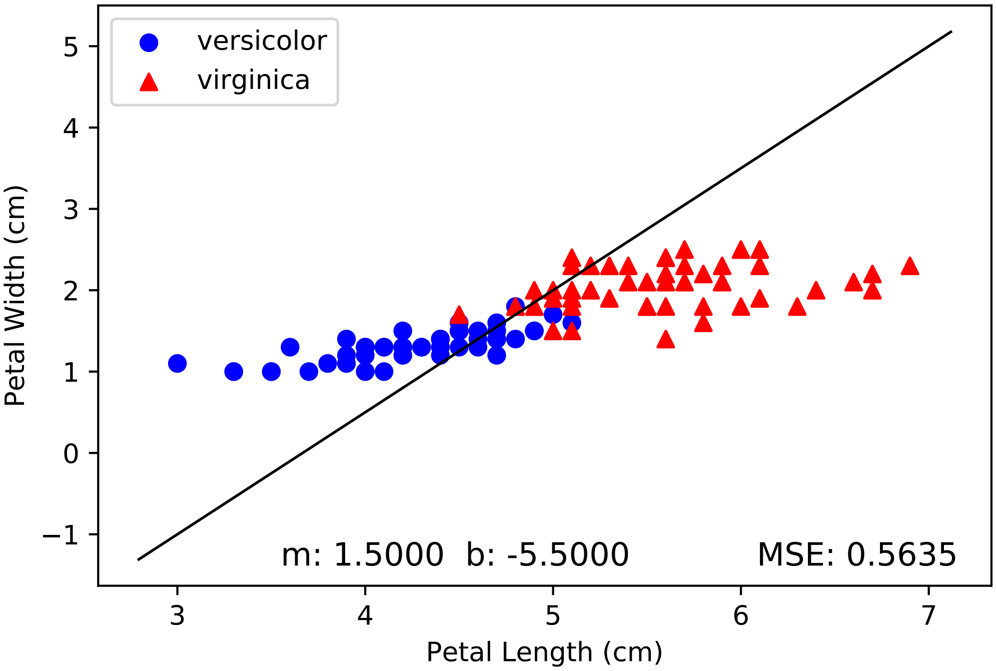
The mean squared error calculation is calculated like so

```
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

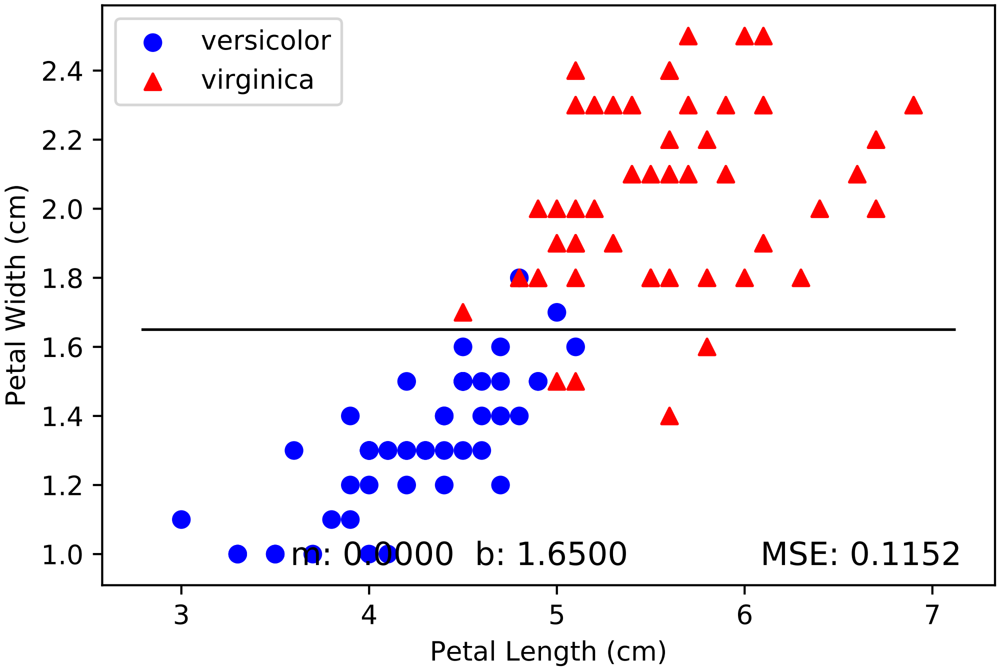
As was shown in the previous section, the MSE value for a line with $m = -0.4$, $b = 3.6$ was quite small (0.0645). We can generate a large MSE value (0.6862) by using a positive slope. Here the boundary misclassifies the majority of the data points!

```
# 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,

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

Where E is the mean-squared error, N is the number of data points (in this case 100), $\boldsymbol{w}^T$ is the vector of weights, $\boldsymbol{x}_n$ 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

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

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

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

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

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:

$$\text{Scalar:}\quad \frac{\partial E}{\partial w_i} = \frac{2}{N}\sum_{n=1}^{N}(x_{i,n} \cdot (1 - S(x_n w) \cdot (S(x_n w) - c_n))$$

$$\frac{2}{N}\sum_{n=1}^{N}(w_0 x_{0,n} + \ldots + w_i x_{i,n} + \ldots + w_M x_{M,n} - c_n)x_{i,n}$$

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 $\boldsymbol{x}$).

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

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

$$w_i^{t+1} = w_1^t - \varepsilon \frac{\partial E}{\partial w_i}$$

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.

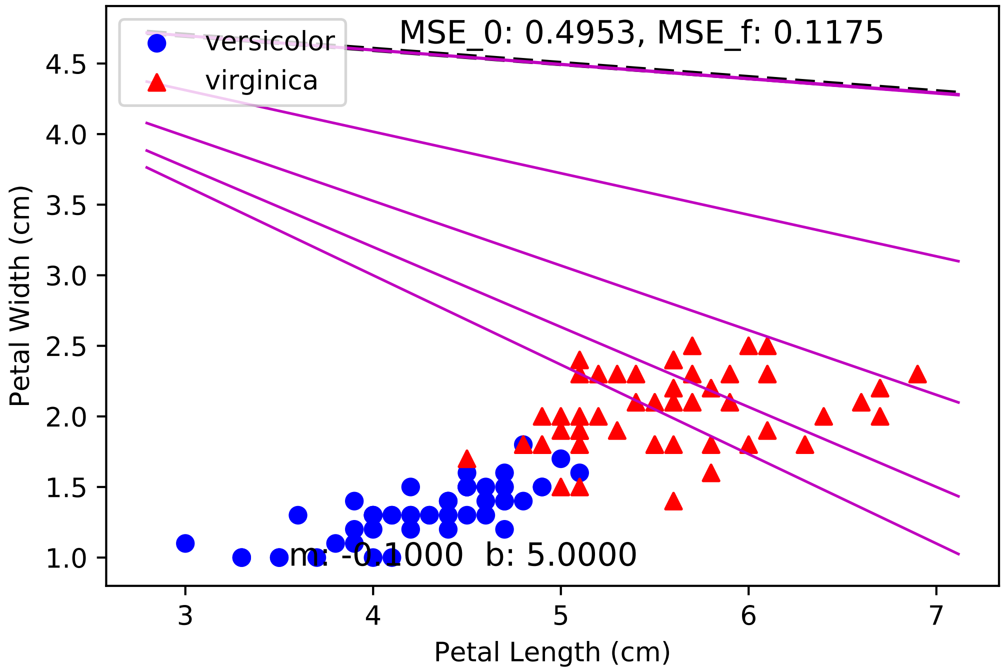
2.5 Summing the Gradient for an Ensemble of Patterns

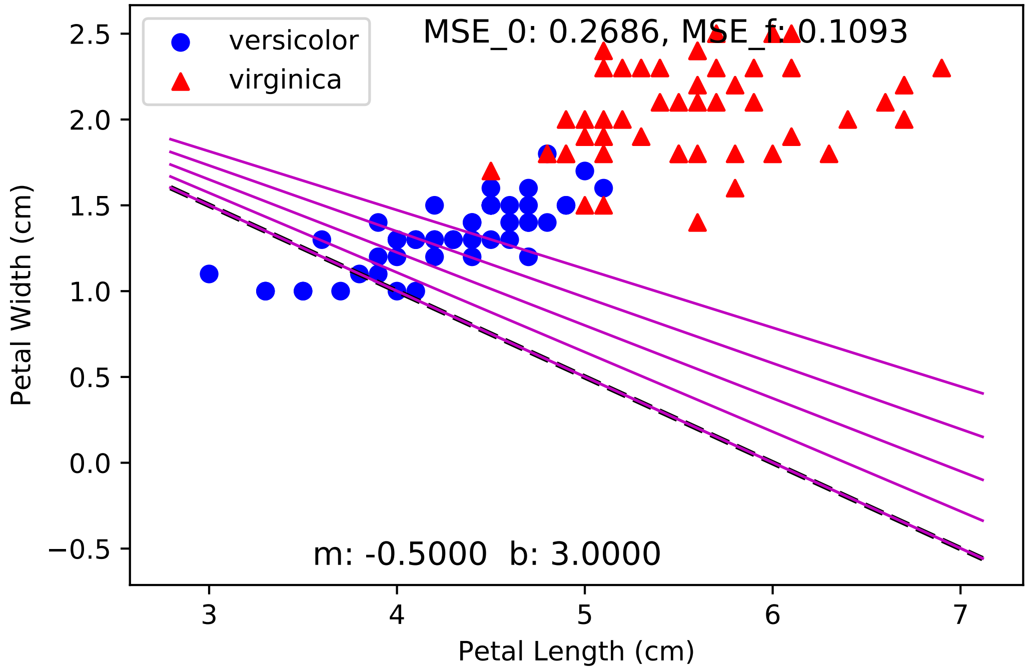
We can now use a gradient to take steps that should ultimately minimize mean squared error. To do this, we simply combine the tools outlined above. Namely, we need the predictive function by which to compare the relative confidence in each categorization, and the derivative, which tells us how to improve this confidence (thereby lowering MSE).

```
def sum_gradient(dataset, m, b, classes={"versicolor": 0, "virginica": 1}):
    epsilon = 0.1 / len(dataset.index)
    gradient = 0
    for i, row in dataset.iterrows():
        pred = sigmoid(dist_from_line(m=m, b=b, row=row))
        error = pred - classes[row["Species"]]
        gradient += (2 / len(dataset.index)) * (1 - pred) * error
    if m < 0:
        new_m = m + epsilon * gradient
    else:
        new_m = m - epsilon * gradient
    new_b = b - epsilon * gradient
    return new_m, new_b
```

The above function calculates the new slope and intercept for weights in the next step in a gradient descent. Below are function calls to plotting routines that produce the following plots. The plots show sequential steps (magenta) from an initial decision boundary (black, dashed).

```
plot_gradient_descent(versi_data, virgi_data, -0.1, 5, snapshots=5, filename="plot_2e_1.pdf")
plot_gradient_descent(versi_data, virgi_data, -0.5, 3, snapshots=5, filename="plot_2e_2.pdf")
```





3 Learning a Decision Boundary Through Optimization

3.1 Implementing Gradient Descent

This section is basically the one above, since in the above plots I considered it more helpful to show multiple steps at once. However since this section covers the gradient descent implementation, I will point out that on the previous graphs we notice initial and final MSE values, also reproduced in the table below. The final value comes after 1000 steps, though not all steps are always necessary.

Initial MSE	Final MSE	Reduction
0.4953	0.1175	76.3%
0.2586	0.1093	57.7%

We can also take a look at the descent function.

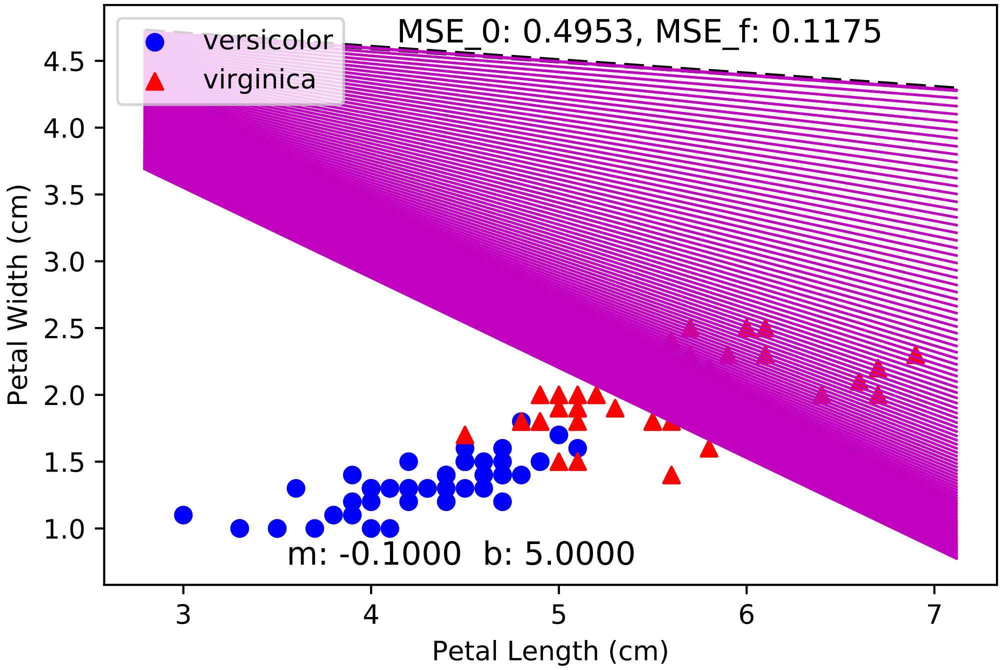
```
mse_0 = mean_squared_error(combined_data, slope, intercept)
m = slope
b = intercept
steps = 1000
for i in range(steps):
    m, b = sum_gradient(combined_data, m, b)
    if (i == 1) or (i % (steps / snapshots) == 0):
        # x_vals = np.array(ax.get_xlim())
        y_vals = b + m * x_vals
        plt.plot(x_vals, y_vals, color="m", linestyle='-', linewidth=1)
mse_f = mean_squared_error(combined_data, m, b)
```

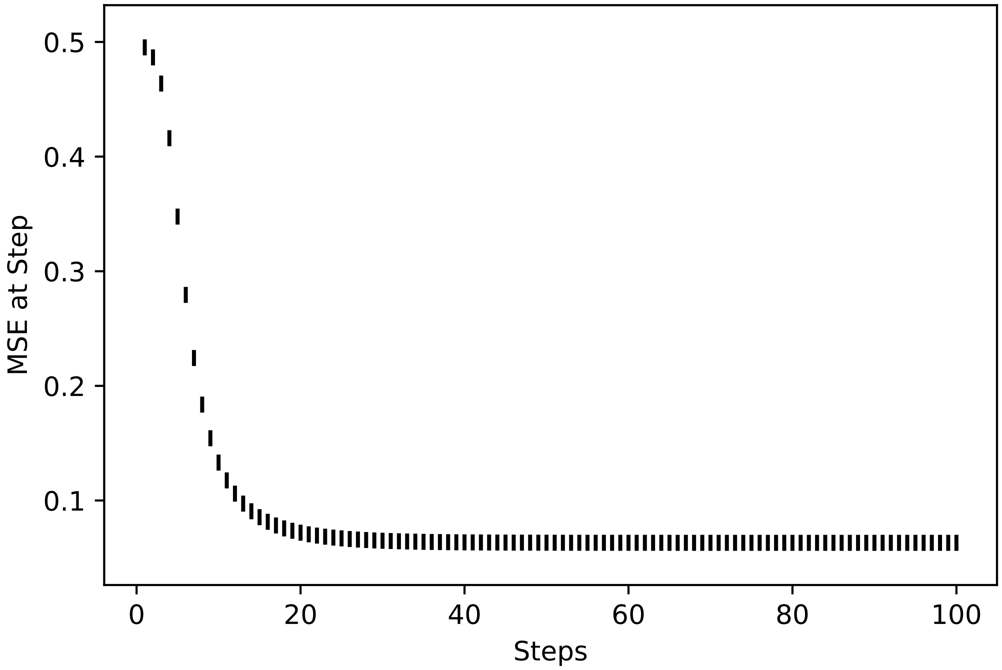
First the mean squared error is calculated for the inital weights. This is MSE_0. Then we take 1000 steps, and for every step a new gradient sum is calculated, m and b are updated and plotted. After the 1000^{th} step, we calculated mean squared error once more. This is MSE_f.

The main point here is that there can be a significant reduction in MSE by gradient descent.

3.2 Showing the Learning Curve

Now let us look at a learning curve for the first descent plot. This time it is shown with a 100 steps visible, rather than just 5. You can see how the steps “slow down” as the boundary converges. Plus it might make a cool Moir pattern on your screen.





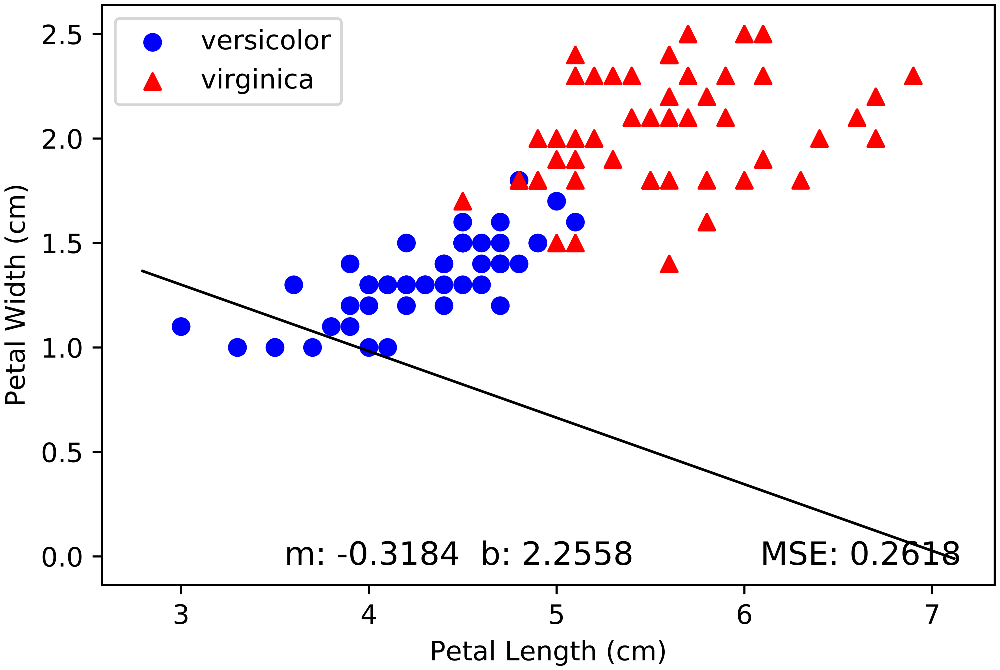
The learning curve is shown above.

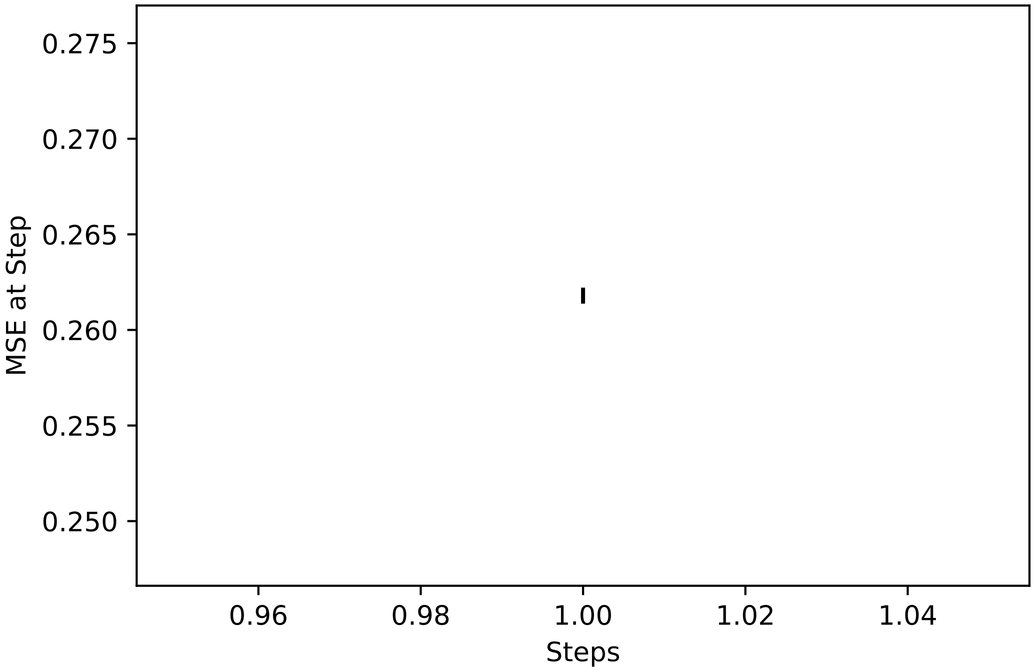
```
mse_vals = []
m = slope
b = intercept
steps = 10000
for i in range(steps):
    if i % 100 == 0:
        mse_vals.append(mean_squared_error(dataset, m, b))
    m, b = sum_gradient(dataset, m, b)

fig = plt.figure()
ax = fig.add_subplot(111)
# mse_vals = mse_vals[int(-len(mse_vals)/2):]
ax.scatter(x=list(range(1, len(mse_vals) + 1)), y=mse_vals, color="k", marker="|")
plt.xlabel("Steps")
plt.ylabel("MSE at Step")
plt.savefig(filename, bbox_inches="tight")
plt.close(fig)
```

3.3 Illustration with Random Weights

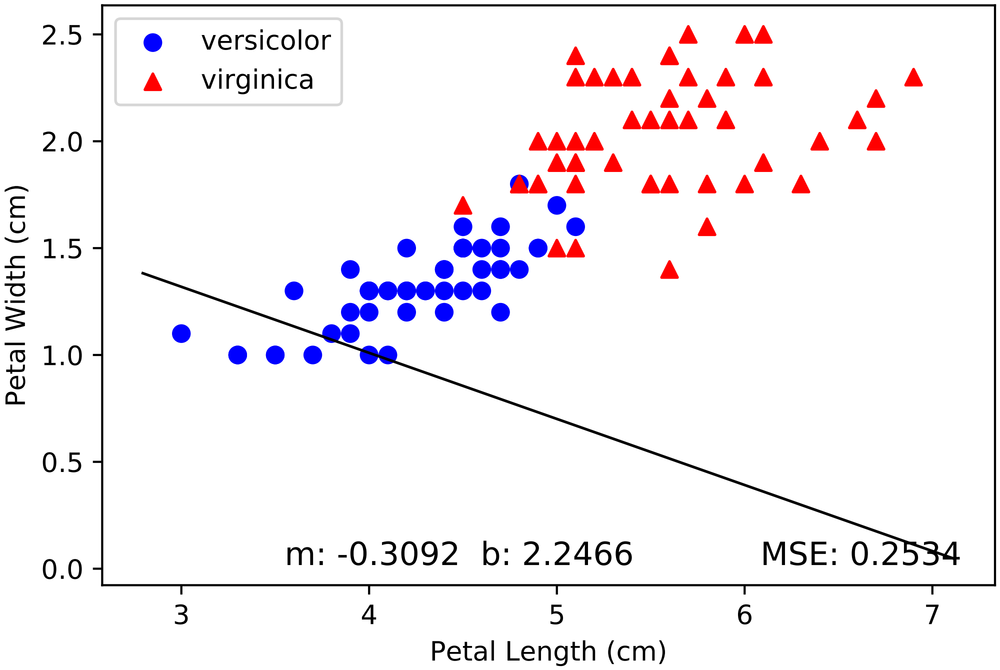
Since this section uses a random element, the results will be different every time the code is run. Therefore I can't comment on specifics here. Enjoy the plots.

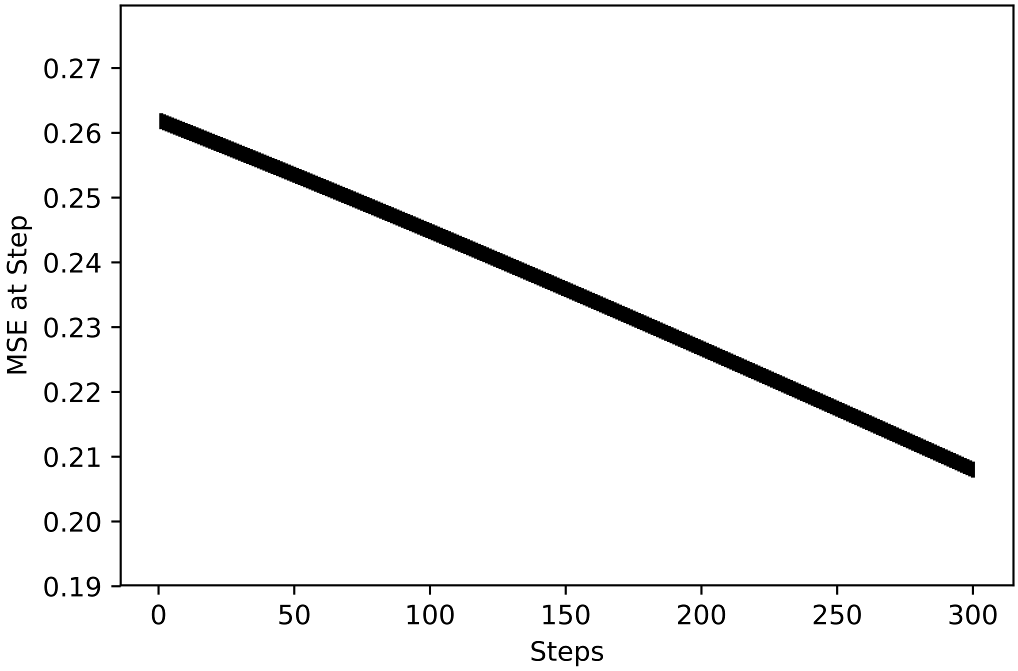
3.3.1 Initial


versicolor
virginica
Petal Width (cm)
2.5
2.0
1.5
1.0
0.5
0.0
3
4
5
6
7
Petal Length (cm)
m: -0.3184 b: 2.2558
MSE: 0.2618

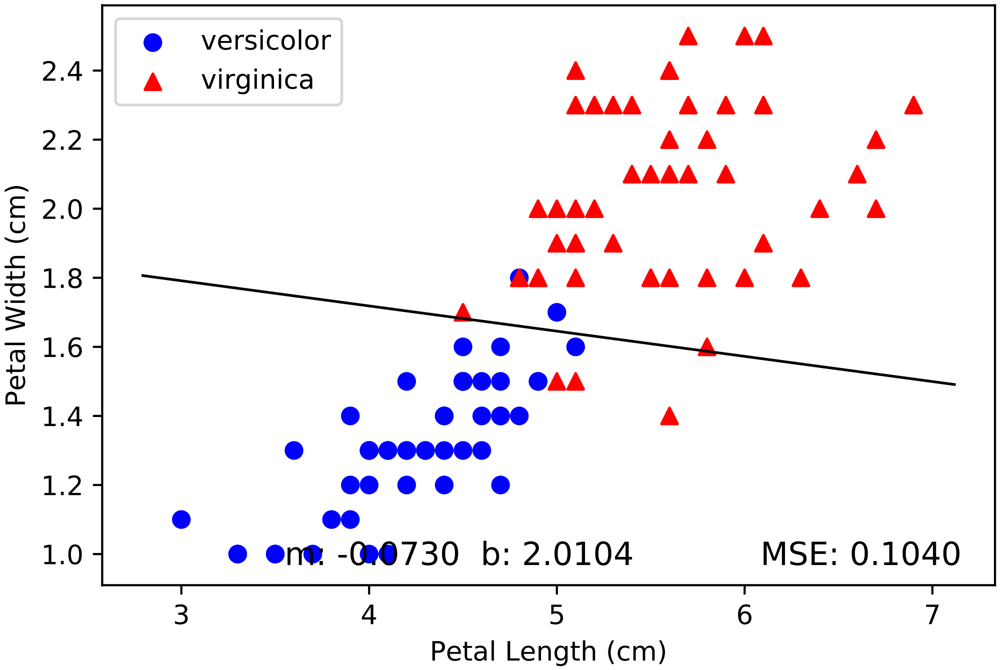

MSE at Step
0.275
0.270
0.265
0.260
0.255
0.250
0.96
0.98
1.00
1.02
1.04
Steps

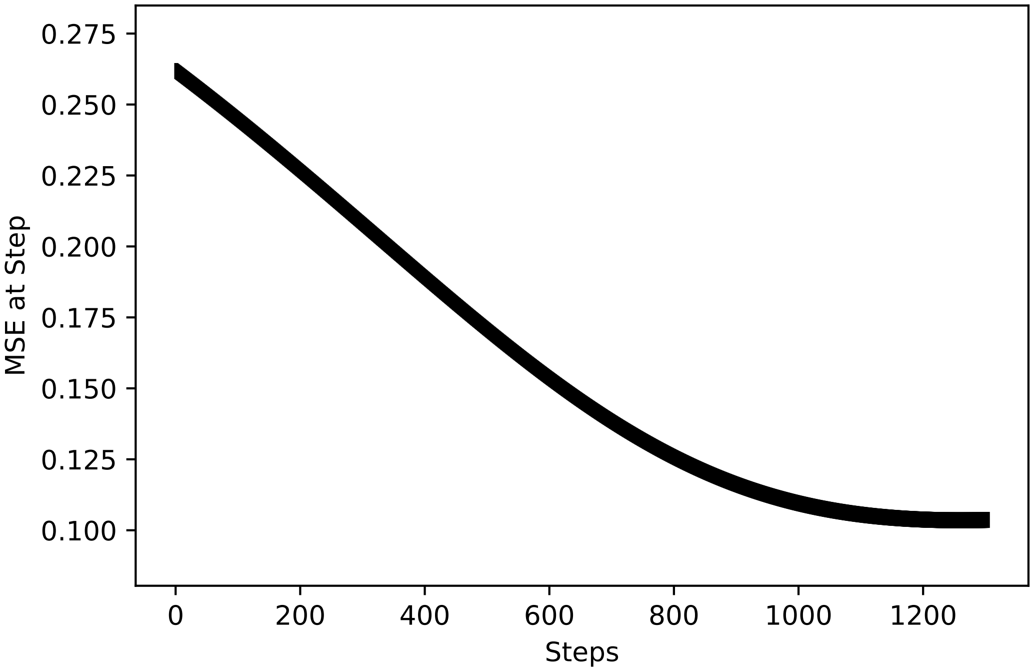
3.3.2 Middle


versicolor
virginica
2.5
2.0
1.5
1.0
0.5
0.0
Petal Width (cm)
3
4
5
6
7
Petal Length (cm)
m: -0.3092 b: 2.2466
MSE: 0.2534


0.27
0.26
0.25
0.24
0.23
0.22
0.21
0.20
0.19
MSE at Step
0
50
100
150
200
250
300
Steps

3.3.3 Final





3.4 Choosing the Gradient Step Size

The gradient step size was selected similarly to an example discussed in lecture. It is determined by the size of the data, namely

$$\varepsilon = \frac{0.1}{N}$$

where N is the size of the dataset, in this case 100. This value was chosen because if the step size is too large, there is an increased probability that the learning will diverge. On the other hand, if the step size is too small, learning will take too long to converge. By making the step size a function of the data size, we guarantee that the step could adapt to a change in size of the data.

For this dataset and algorithm, I found that the step size was somewhat lenient. I tested between 0.001 and 0.01 and both of these values worked properly.

3.5 Choosing the Stopping Criterion

I tested three different stopping criteria, keeping in mind that once the learning converges, there is no point in continuing further.

First was to detect a lack of change in the mean squared error (MSE), since it seems reasonable to claim that once the MSE stops changing no more steps are needed. However in practice this assumption is not always true. If the decision boundary is ever completely outside of the clusters of points (*all* points fall on one side or another), then this method will falsely detect convergence, since shifting the boundary will usually keep the MSE at around 0.50. Even if the boundary is within the data, the data are granular enough that you can sometimes shift the boundary without changing any classifications.

Second was to detect a lack of change in an "upstream" variable. This method is more sensitive to stillness. I decided that if the boundary does not move at all within some degree of freedom, then the learning has converged, and it ended up working successfully for my code. The degree of freedom was adjusted over testing. A very small number was initially used, then when the runtime was long, it was sequentially reduced until it produced good results in an acceptible time.

```
while 0.0001 < b_2 - b:
    (continue calculation)
```

When this criterion is used, it stops slightly after 1300 steps.

Third we can set a limit on number of steps. This is arguably the best way to do it if you have the empirical data to back it up. Since we are working with a single dataset and not in a general application, this is easy to calculate, however it is technically wasteful in some cases, as the convergence will occur before the step limit is reached, and for this reason I chose the second method instead.