12_svm

March 14, 2023

1 Support Vector Machines

variationalform https://variationalform.github.io/

Just Enough: progress at pace https://variationalform.github.io/

https://github.com/variationalform

Simon Shaw https://www.brunel.ac.uk/people/simon-shaw.

This document uses python

and also makes use of LaTeX

in Markdown

1.1 What this is about:

- Binary Classification.
- Linear separability.
- Separating planes, hyperplanes: **decision boundaries**.
- Support Vectors, and SVM (Support Vector Machine) classification.

As usual our emphasis will be on *doing* rather than *proving*: *just enough: progress at pace*

For this material you are recommended Pages 1-43 of [SVMS] - **this was set as homework**. Also recommended is Chapter 12 of [MML] and, less so, Chapter 8.5 of [MLFCES].

- SVMS: Support Vector Machines Succinctly by Alexandre Kowalczyk. https://www.syncfusion.com/succinctly-free-ebooks/support-vector-machines-succinctly
- MML: Mathematics for Machine Learning, by Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. Cambridge University Press. https://mml-book.github.io.
- MLFCES: Machine Learning: A First Course for Engineers and Scientists, by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön. Cambridge University Press. http://smlbook.org.

These can be accessed legally and without cost.

There are also these useful references for coding:

- PT: `python`: https://docs.python.org/3/tutorial
- NP: `numpy`: https://numpy.org/doc/stable/user/quickstart.html
- MPL: `matplotlib`: https://matplotlib.org

1.2 Context

In the last session we moved from classification to regression and then, with logistic regression, we moved back to classification again.

In this session we will continue with the classification theme and briefly discuss **Support Vector Machines** (SVM's). We will then be able to move quickly on to discuss the **perceptron** which will set us up for **deep neural networks**.

In the homework you were asked to read https://www.syncfusion.com/succinctly-free-ebooks/support-vector-machines-succinctly up page 43. Pages 1-20 are a revision of the vector material we have already covered.

The following material assumes familiarity with that source.

We're going to start with the Iris Data Set, with the **virginica** data removed - just as with the logistic regression example in the last session.

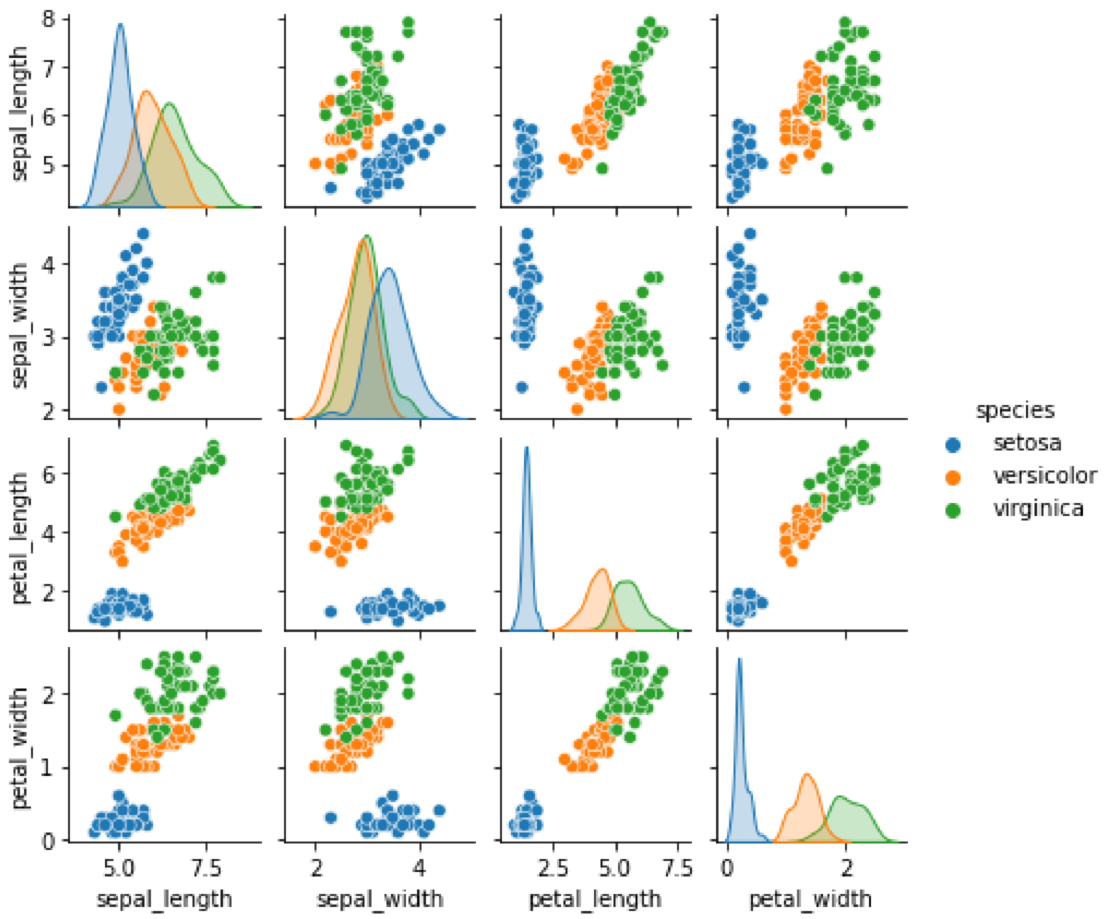
[261]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score
```

[262]:
```
sns.get_dataset_names()
dfi = sns.load_dataset('iris')
dfi.head()
```

[262]:
```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

[263]:
```
sns.pairplot(dfi, hue='species', height = 1.5)
```

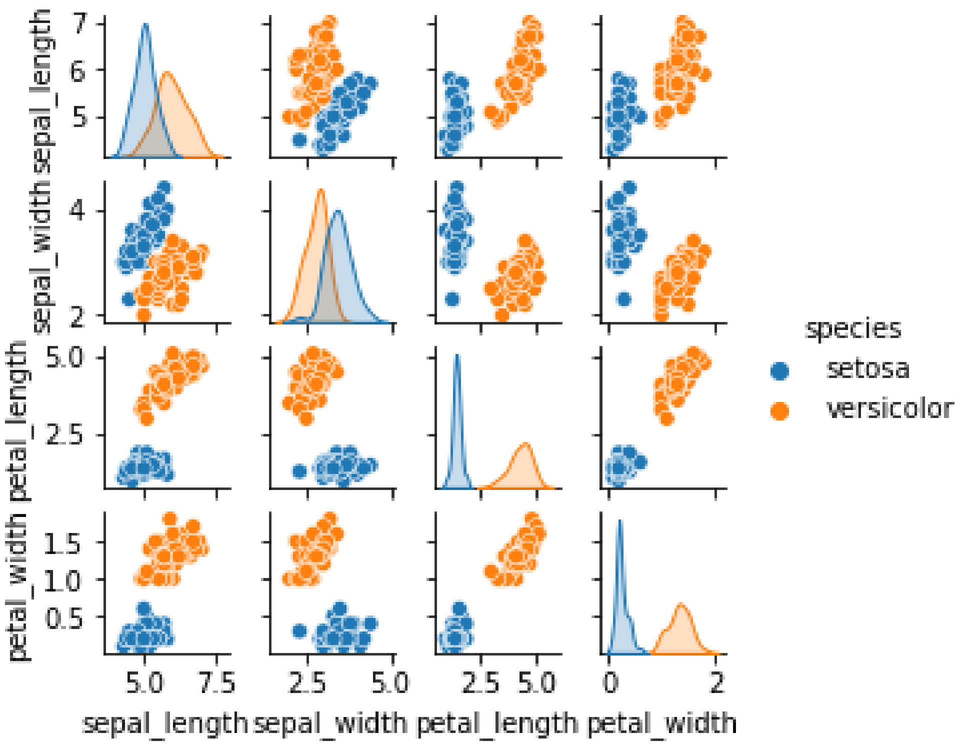
[263]:
```
<seaborn.axisgrid.PairGrid at 0x7fdabeb027f0>
```



We want a binary classifier so we drop the virginica data.

[264]:
```
# we want a binary classifier, so drop the virginica data
dfid = dfi[ (dfi['species'] != 'virginica') == True ]
# this leave just two classes.
sns.pairplot(dfid, hue='species', height = 1.0)
```

[264]:
```
<seaborn.axisgrid.PairGrid at 0x7fdad9c529e8>
```



Again, we focus on the well separated petal length and sepal width for our features.

We'll use them to predict species: setosa or versicolor.

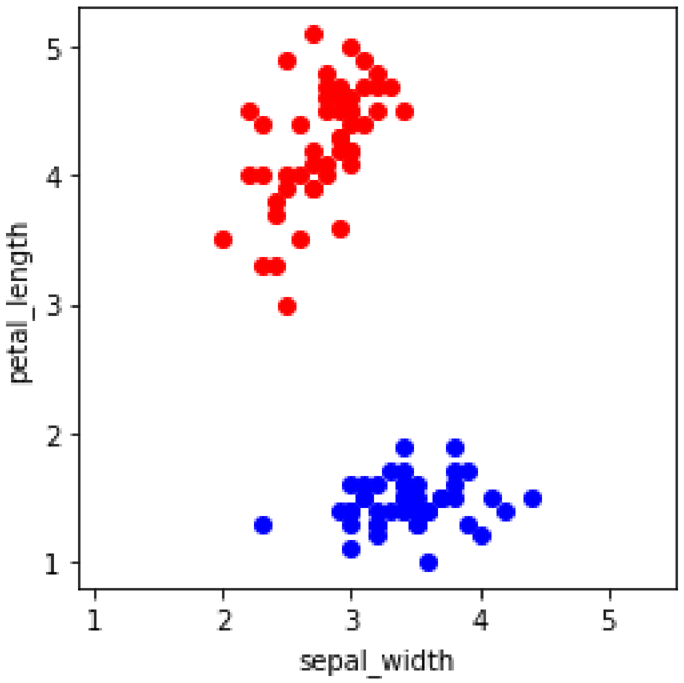
```
[265]: dfid.head()
```

```
[265]:    sepal_length  sepal_width  petal_length  petal_width species
       0           5.1          3.5           1.4          0.2  setosa
       1           4.9          3.0           1.4          0.2  setosa
       2           4.7          3.2           1.3          0.2  setosa
       3           4.6          3.1           1.5          0.2  setosa
       4           5.0          3.6           1.4          0.2  setosa
```

```
[266]: # we use petal length and sepal width as our features
       X = dfid.iloc[:,[1,2]].values
       # and species as our label
       y = dfid.iloc[:, 4].values
```

```
[267]: # let's plot them in different colours - find the species array index sets
       indxS = np.where(y == 'setosa')[0]
       indxV = np.where(y != 'setosa')[0]
       ax = plt.figure(figsize=(4,4))
       plt.scatter(X[indxS,0], X[indxS,1], color='blue')
       plt.scatter(X[indxV,0], X[indxV,1], color='red')
       plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length')
```

[267]:
```
Text(0, 0.5, 'petal_length')
```



We want a decision boundary - in this case it will be a straight line that separates the classes. Then we can classify new data accrding to which side of the line it appears. This is just like we described with **Logistic Regression**.

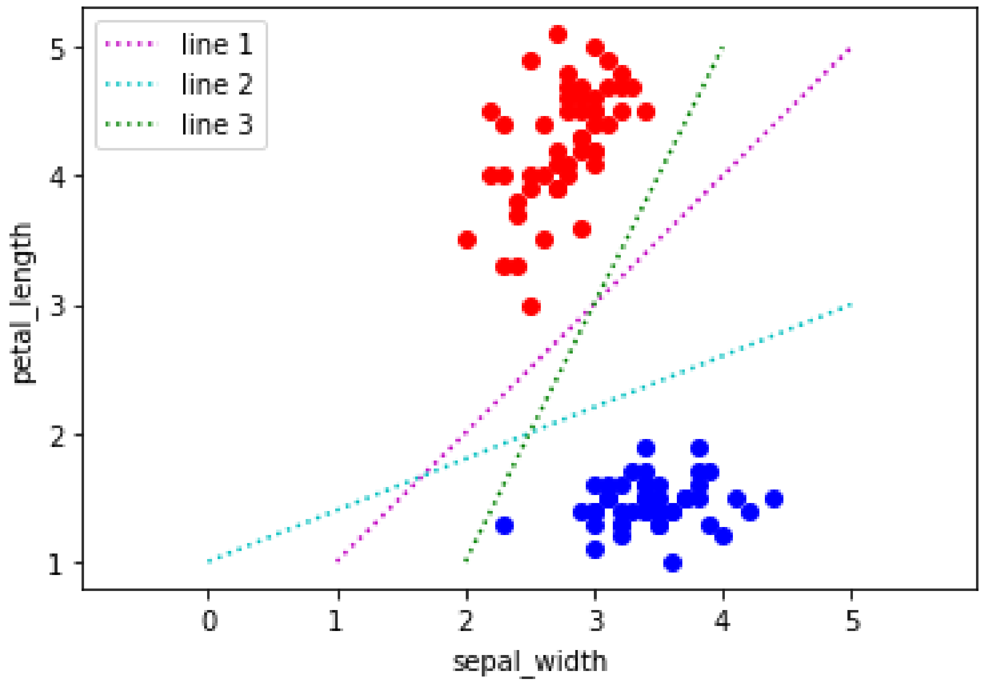
Note that we can clearly see that it is possible to separate the classes with a single line. Data sets for which this is true are called **linearly separable**.

You can read more about this concept in, for example, [MML, Chapter 12]. Data sets which cannot be seprated in this way are more difficult to work and may required so-called kernel methods. They are beyond our scope.

Back to our data. Let's plot a few lines - **which one would you prefer to use?**

[268]:
```
ax = plt.figure(figsize=(6,4))
plt.scatter(X[indxS,0], X[indxS,1], color='blue')
plt.scatter(X[indxV,0], X[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length')
plt.plot([1,5],[1,5],':m', label='line 1')
plt.plot([0,5],[1,3],':c', label='line 2')
plt.plot([2,4],[1,5],':g', label='line 3')
plt.legend()
```

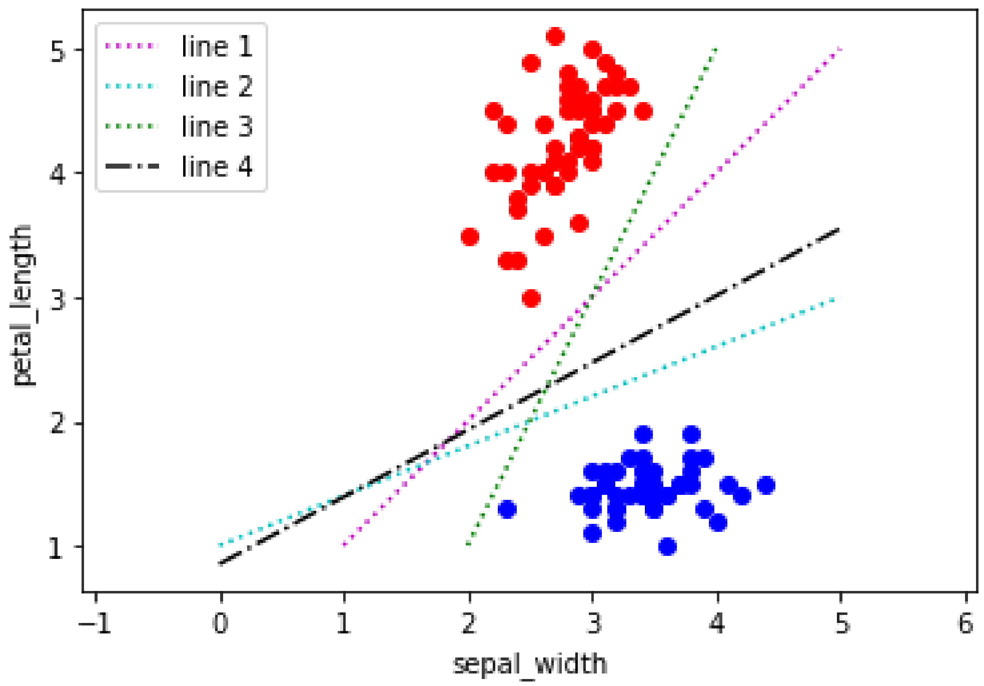
```
[268]: <matplotlib.legend.Legend at 0x7fdacae08cc0>
```



Which one now?

```
[269]: ax = plt.figure(figsize=(6,4))
       plt.scatter(X[indxS,0], X[indxS,1], color='blue')
       plt.scatter(X[indxV,0], X[indxV,1], color='red')
       plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length')
       plt.plot([1,5],[1,5],':m', label='line 1')
       plt.plot([0,5],[1,3],':c', label='line 2')
       plt.plot([2,4],[1,5],':g', label='line 3')
       plt.plot([0,5],[.85,3.55],'-.k', label='line 4')
       plt.legend()
```

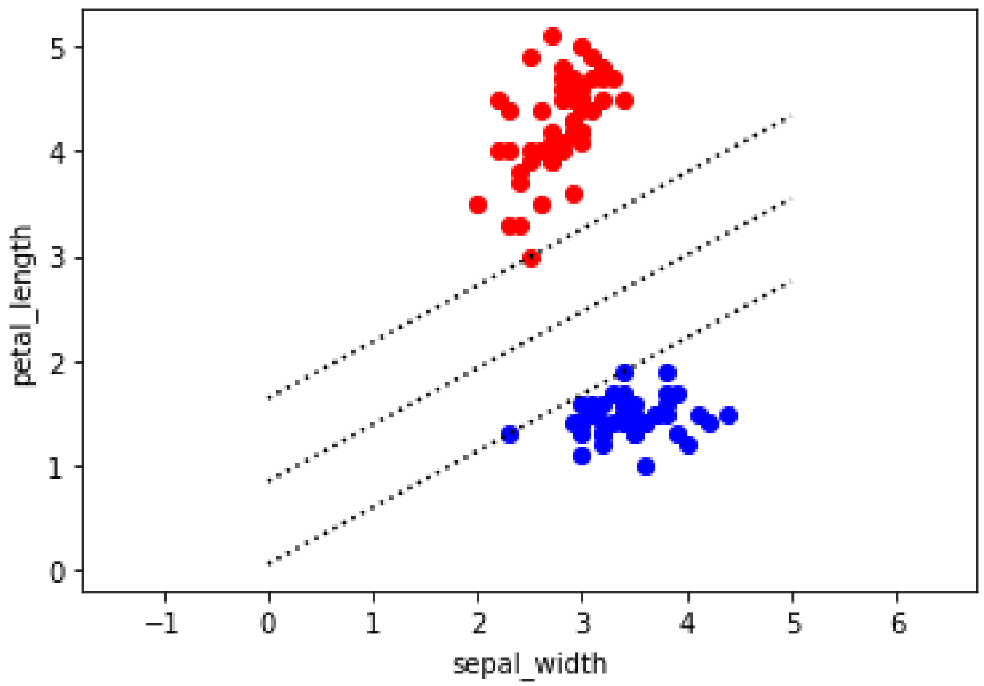
```
[269]: <matplotlib.legend.Legend at 0x7fdacae4c3c8>
```



Let's explore this... Does this give the widest gap? Or maximum margin?

```
[270]: ax = plt.figure(figsize=(6,4))
       plt.scatter(X[indxS,0], X[indxS,1], color='blue')
       plt.scatter(X[indxV,0], X[indxV,1], color='red')
       plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length')
       plt.plot([0,5],[.85,3.55],':k')
       plt.plot([0,5],[0.85+0.79,3.55+0.79],':k')
       plt.plot([0,5],[0.85-0.79,3.55-0.79],':k')
```

```
[270]: [<matplotlib.lines.Line2D at 0x7fda9af2c400>]
```



1.2.1 Maximum Margin

The previous picture illustrates the central concept behind the SVM (**Support Vector Machine**) classifier.

The idea is to find the **decision boundary** as the mid-line between the two parallel separating lines that are as far apart as possible. This separating gap is called the **separating margin**.

We aim to find the **maximum separating margin** using the training data and then, because the gap is as wide as possible, we hope that all of the unseen test and future data will fall the correct side of the decision boundary.

This method is appropriate for **linearly separable data** as described in the homework reading [SVMS], https://www.syncfusion.com/succinctly-free-ebooks/support-vector-machines-succinctly on page 21 onwards. The use of **kernels** can make this method applicable to non-linearly separable data but that is an advanced topic and beyond our scope.

1.2.2 The General Set-Up

We illustrated the maximum margin with 2D data. In general though our data will lie in higher dimensional space, and for that we will need more than 1D lines to describe the separating margin.

If the data were in 3D then we would separate it using the 2D version of a line, which is a plane.

Think of a two adjacent rooms - the wall between is like a plane separating the larger combined space into the two rooms. If we used just a line then it would be like a thread of cotton running from one wall to another - this would not make it clear how the room is divided into two.

Mental images and intuition fail us when the data live in higher dimensional space. But maths saves use - we use **hyperplanes**.

1.2.3 Points, Lines, Planes and Hyperplanes

In two dimensions we are very familiar with the idea of a straight line having equation $y = mx + c$ where m is called the gradient and c the y-intercept.

These are useful but for us it will be more useful to write the vector $\boldsymbol{x} = (x_1, x_2)^T$ in place of (x, y) and define our straight line by the equation

$$\boldsymbol{w} \cdot \boldsymbol{x} = \phi$$

for given **weights**, $\boldsymbol{w} = (w_1, w_2)^T$, and some constant ϕ.

For example, the equation $y = \frac{4}{5}x + 7$ becomes

$$x_2 = \frac{4}{5}x_1 + 7 \quad \Longrightarrow \quad -4x_1 + 5x_2 = 35 \quad \Longrightarrow \quad \boldsymbol{w} \cdot \boldsymbol{x} = \phi$$

for $\boldsymbol{w} = (-4, 5)^T$ and $\phi = 35$.

Let $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$ be two distinct points on the line - as represented by vectors pointing from the origin and ending on the line - then,

$$\boldsymbol{w} \cdot \boldsymbol{x}_a - \boldsymbol{w} \cdot \boldsymbol{x}_b = \phi - \phi = 0 \quad \Longrightarrow \quad \boldsymbol{w} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b) = 0.$$

> **THINK ABOUT:** what does this mean geometrically? What is the angle between the line and the vector $\boldsymbol{w}$?

In more than two dimensions, in $\mathbb{R}^n$ say, we would have $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$ and a vector of *weights* $\boldsymbol{w}$ for which

$$\boldsymbol{w} \cdot \boldsymbol{x} = \phi$$

just as above. If $\boldsymbol{x} = (x_1, x_2, x_3)^T$ then this is

$$\boldsymbol{w}_1\boldsymbol{x}_1 + \boldsymbol{w}_2\boldsymbol{x}_2 + \boldsymbol{w}_3\boldsymbol{x}_3 = \phi$$

which defines a *plane* - an infinite flat surface. In higher dimensions we call it a *hyperplane.*

Suppose a point Q is described by the vector $\boldsymbol{q}$ rooted at the origin. We can ask: **What is the shortest distance from this point to the hyperplane?**

To answer this imagine you are standing on one side of a busy road and want to cross to the other side. You would want to minimise the time you spend on the road and so would choose to cross by moving at a right angle to the road.

That is just the case with our point Q. The shortest distance to the hyperplane is along the path that meets the plane at a right angle in all directions.

It's like a tall tree growing vertically from the plane ground. However: this is just an analogy, not a plug for **flat earth theory**.

To figure out this distance we move from the origin to Q, by moving to the end of the vector $\boldsymbol{q}$. We then move a distance d along a line parallel to $\boldsymbol{w}$ in the direction towards the plane.

$$\text{This takes us to the point} \qquad \boldsymbol{y} = \boldsymbol{q} + \frac{d\,\boldsymbol{w}}{\|\boldsymbol{w}\|_2}.$$

THINK ABOUT: why is the length of $\boldsymbol{w}$ in the denominator?

If we insist that the point $\boldsymbol{y}$ is actually a point $\boldsymbol{x}$ on the hyperplane, then $\boldsymbol{w}^T\boldsymbol{x} = \phi$ and

$$\boldsymbol{y} = \boldsymbol{x} \qquad \text{means} \qquad \phi = \boldsymbol{w}^T\boldsymbol{y} = \boldsymbol{w}^T\boldsymbol{x} = \boldsymbol{w}^T\boldsymbol{q} + d\frac{\boldsymbol{w}^T\boldsymbol{w}}{\|\boldsymbol{w}\|_2} = \boldsymbol{w}^T\boldsymbol{q} + d\|\boldsymbol{w}\|_2.$$

The shortest distance d from Q to the hyperplane is then

$$d = \frac{\phi - \boldsymbol{w}^T\boldsymbol{q}}{\|\boldsymbol{w}\|_2}.$$

Note that this distance is signed. See also [SVMS, page 46] for an alternative derivaiton.

We've already seen why this is important in data science with the earlier **Iris Data Set** example. The situation is ubiquitous. Here's another example of why this geometrical view is relevant.

We have seen how data sets comprise of rows of observations with each having several characteristics or **features**.

Each stock in the FTSE 100 for example has a daily high, a daily low, percentage change, a yield, a volume traded, market capitalization and so on. Each item can therefore be thought of as a point in high dimensional space. For example, for a given stock, the six quantities, or *features*, just listed could be represented by this *feature vector*,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} \text{daily high} \\ \text{daily low} \\ \text{percentage change} \\ \text{yield} \\ \text{volume traded} \\ \text{market cap.} \end{pmatrix}$$

We have seen with our Iris data above that a way to classify such data is to try and separate these points into two distinct groups.

If we can do that we have in some sense reduced our mass of data down to two essential clusters.

We can do that by trying to find a hyperplane that passes between the two clusters. We would want the hyperplane to be as far away from all of the points as possible so that it makes a clear distinction between the two clusters.

This would mean finding d for each point, each feature vector, and making sure that the minimum value of all such d-values is as large as possible.

THINK ABOUT: can you sketch this situation in 2D? Does this seem to be a difficult thing to do in general?

Fortunately we only have to understand this in the simplest case. We will be using software to solve the 'real' problems.

1.2.4 Optimization Problem

We now have a mathematical expression for the signed distance d between a point $\boldsymbol{q}$ and the hyperplane $\boldsymbol{w}^T\boldsymbol{x} = \phi$.

The two classes in the training data are labelled as positive or negative. A data point in the positive class is labelled as $y = +1$ and a point in the negative class is labelled as $y = -1$.

The negative class corresponds to negative distances, d.

We imagine visiting every training data point $\boldsymbol{q}_k$, for $k = 1, \ldots, N$ (say there are N in total), finding the label y_k and distance d_k, and calculating $D_k = d_k y_k \geq 0$. Then:

Determine ϕ and $\boldsymbol{w}$ such that the minimum of D_k is maximized.

This **optimization problem** is mathematically very challenging. See [SVMS, Chapter 4] and [MML, Chapter 12] if you want some insight as to why.

We'll use software…

1.2.5 SVM using `sklearn`

Recall that we already set up `X` and `y` for the petal length and sepal width iris data above with

```
X = dfid.iloc[:,[1,2]].values
y = dfid.iloc[:, 4].values
```

[271]:
```
dfid.head(2)
```

[271]:
```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
```

The Support Vector Machine, or SVM, classification method uses the setting described above to determine a maximum separating margin from training data. The mid-plane (in general hyperplane) is then a decision boundary between the two classes.

The rationale for this approach is that because the separation is maximized, we expect that even unseen test and future data will fall on the correct side of the decision boundary. This will be enough for correct classification - even though it may stray into the separating margin.

We'll use the `sklearn` SVM capability: https://scikit-learn.org/stable/modules/svm.html

[272]:
```
# We'll use 50% of the data to test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50)
```

Should we scale the data? On the face of it, it seems that it isn't needed. We'll use a `python if` statement to leave the option open.

[273]:
```
if False:
  scaler = StandardScaler()
  # initialise the scaler by feeding it the training data
  scaler.fit(X_train)
  # now carry out the transformation of all of the feature data
  X_train = scaler.transform(X_train)
  X_test  = scaler.transform(X_test)
```

Toggle `if False:` and `if True:` to switch scaling off/on.

[274]:
```
# import the SVM classifier
from sklearn import svm
# instance it
svmclf = svm.SVC(kernel='linear')
# and fit the training data
svmclf.fit(X_train, y_train)
```

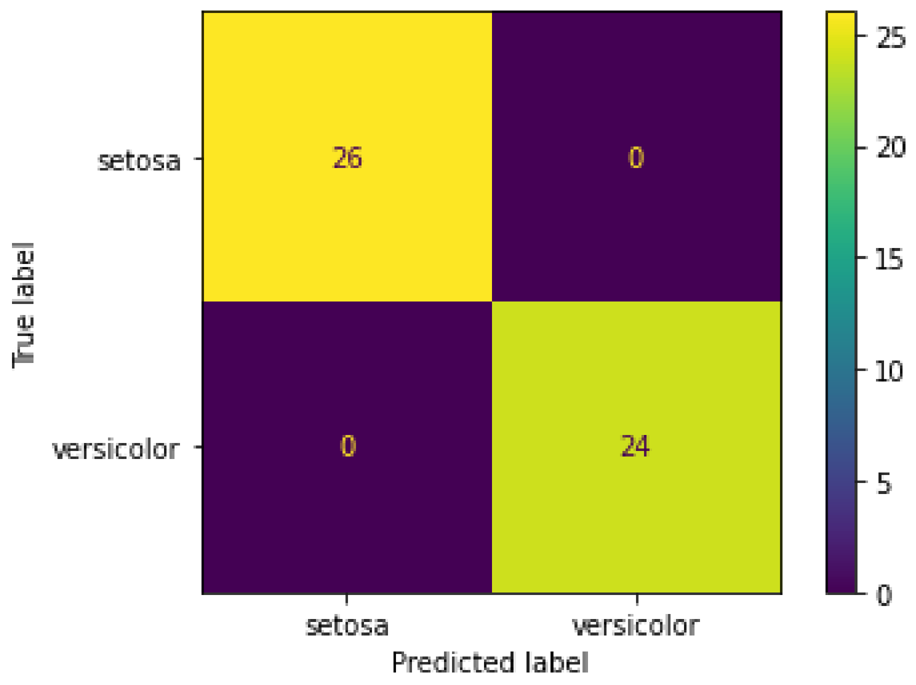
[274]:
```
SVC(kernel='linear')
```

[275]:
```
# make predictions on the test set
y_pred = svmclf.predict(X_test)
```

[276]:
```
# get the confusion matrix and accuracy data
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
accsc = accuracy_score(y_test,y_pred)
print("Accuracy:", accsc)
```

```
Confusion Matrix:
[[26  0]
 [ 0 24]]
Accuracy: 1.0
```

[277]:
```
# plot a nicer confusion matrix
cmplot = ConfusionMatrixDisplay(cm, display_labels=svmclf.classes_)
cmplot.plot()
plt.show()
```



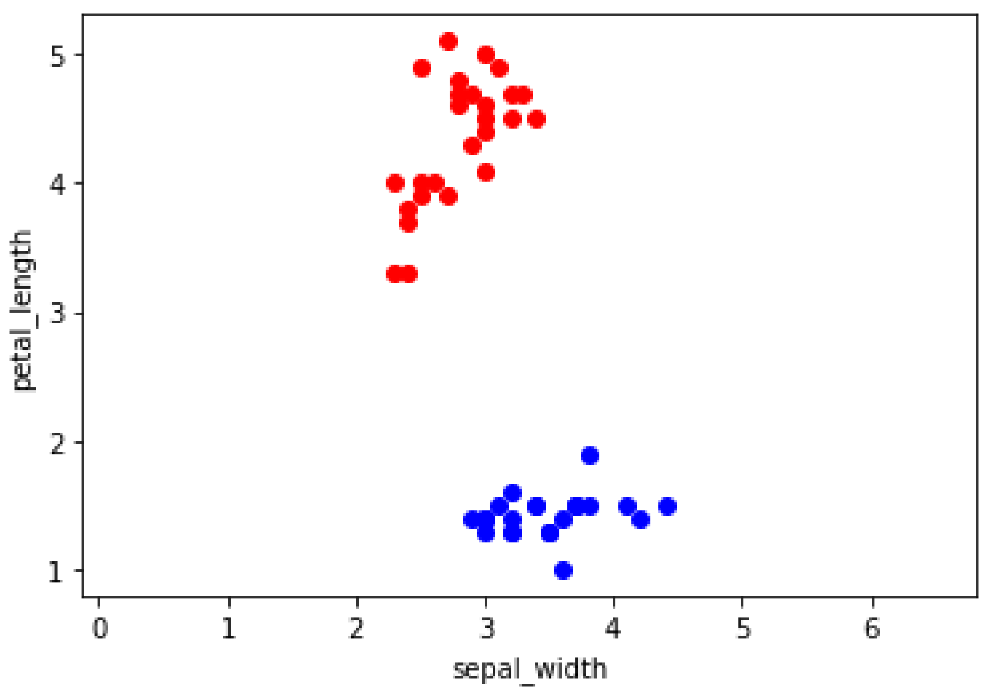
This is perfect - but it ought to be because the original data were well separated.

Let's look at the data and the SVM in pictures - we will get more insight.

we'll start with the training data - these are the data points the SVM used.

[278]:
```
indxS = np.where(y_train == 'setosa')[0]
indxV = np.where(y_train != 'setosa')[0]
```

[279]:
```
plt.scatter(X_train[indxS,0], X_train[indxS,1], color='blue')
plt.scatter(X_train[indxV,0], X_train[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
```



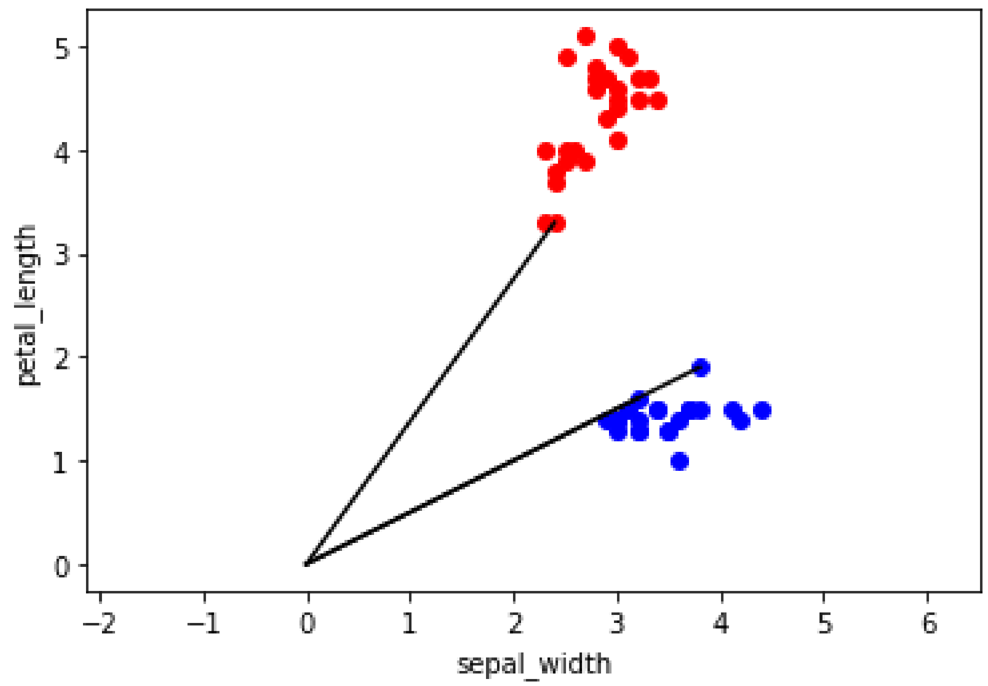
The SVM classifier provides **support vectors** … These are not easy to explain. The optimization process determines a subset of the data set that can be used to determine the maximum separating margin. Those points in the subset are called **support vectors**.

[280]:
```
# get support vectors
SVecs = svmclf.support_vectors_
print('The (transposed) support vectors are:\n', SVecs.T)
# get number of support vectors for each class
NumSVecs = svmclf.n_support_
print('There are these many per class', NumSVecs)
```

```
The (transposed) support vectors are:
 [[3.2 3.8 2.4]
 [1.6 1.9 3.3]]
There are these many per class [2 1]
```

The support vectors tell us which data points are forming the margin. (It's actually a bit more complicated than that - we'll come back to this below.)

[281]:
```
plt.scatter(X_train[indxS,0], X_train[indxS,1], color='blue')
plt.scatter(X_train[indxV,0], X_train[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
# plot the support vectors
for k in range(NumSVecs.sum()):
  plt.plot([0,SVecs[k,0]],[0,SVecs[k,1]],'k')
```



The SVM classifier can also tell us the equation of the hyperplane decision boundary.

It has the form $ax_1 + bx_2 + c = 0$ where...

[282]:
```
# a x1 + b x2 + c = 0
a = svmclf.coef_[0,0]
b = svmclf.coef_[0,1]
c = svmclf.intercept_[0]
print(f'a={a}, b={b}, c={c}')
```

```
a=-0.47594229165317875, b=0.9524978485590939, c=-1.001042610426726
```

Let's add this to the plot using $x_2 = -(ax_1 + c)/b$ (which requires $b \neq 0$).

We can set up two points and join them with a straight line:

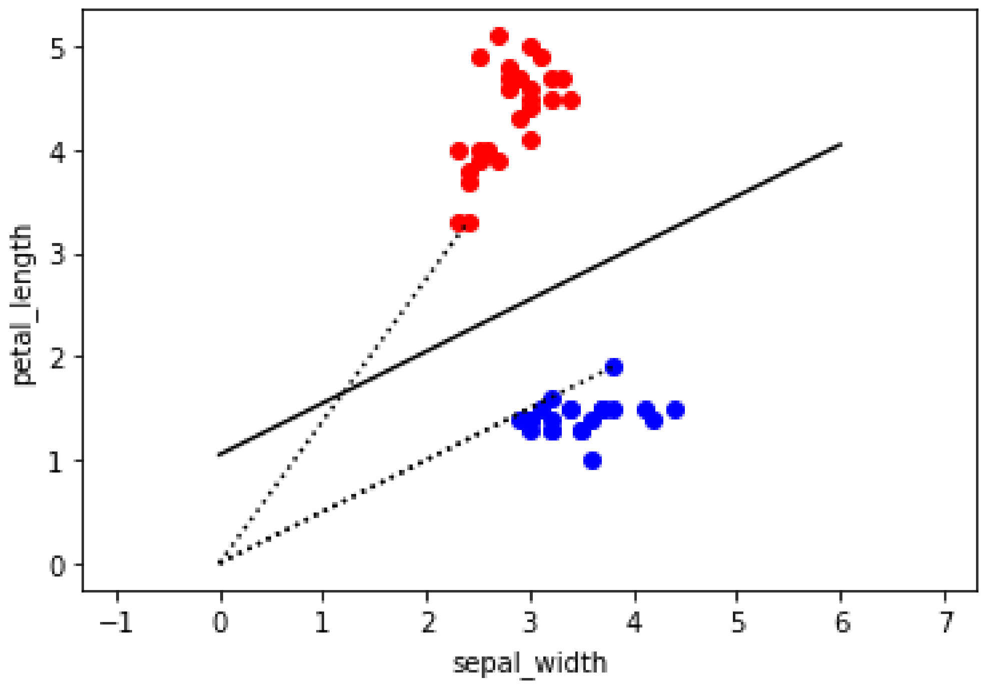
$$P_1: \ (x_1, x_2) = (0, -c/b) \qquad \text{and} \qquad P_2: \ (x_1, x_2) = (6, -(6a + c)/b).$$

[283]:
```
P1 = np.array([0, -c/b]); P2=np.array([6, -(6*a + c)/b])
```

[284]:
```
plt.scatter(X_train[indxS,0], X_train[indxS,1], color='blue')
plt.scatter(X_train[indxV,0], X_train[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
# plot the support vectors
for k in range(NumSVecs.sum()):
  plt.plot([0,SVecs[k,0]],[0,SVecs[k,1]],':k')
```

```
# the decision boundary
plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
```

[284]:
```
[<matplotlib.lines.Line2D at 0x7fdae9588358>]
```



We can also draw parallel lines through the support vector data points.

In general, with the decision boundary given by $ax_1 + bx_2 + c = 0$, a parallel line passing through $\boldsymbol{q} = (q_1, q_2)^T$ satisfies

$$a(x_1 - q_1) + b(x_2 - q_2) = 0$$

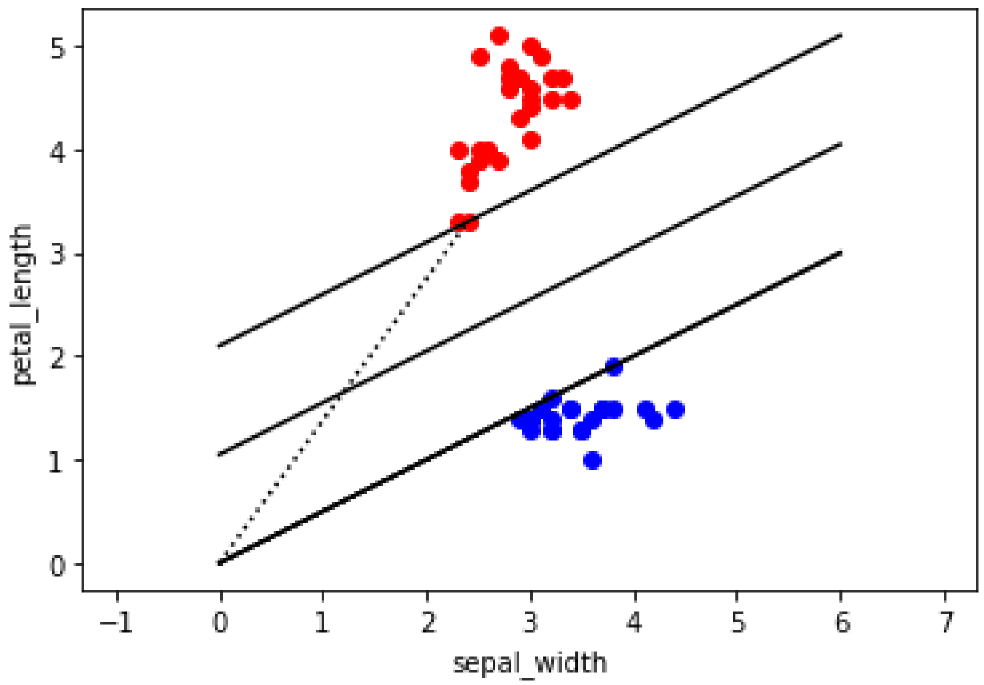
because a and b control the gradient and so must be the same.

Hence,

$$ax_1 + bx_2 + c_q = 0 \quad \text{where} \quad c_q = -aq_1 - bq_2.$$

We can add the parallel margin edges to the plot with this...

[285]:
```
plt.scatter(X_train[indxS,0], X_train[indxS,1], color='blue')
plt.scatter(X_train[indxV,0], X_train[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
for k in range(NumSVecs.sum()):
  plt.plot([0,SVecs[k,0]],[0,SVecs[k,1]],':k')
  q = np.array([SVecs[k,0],SVecs[k,1]])
  cq = -a*q[0]-b*q[1]; P1[1]=-(0*a + cq)/b; P2[1]=-(6*a + cq)/b
  plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
```



NOTE

The code above may seem a bit overwhelming but in the end we are just identifying points and joining them up with straight lines.

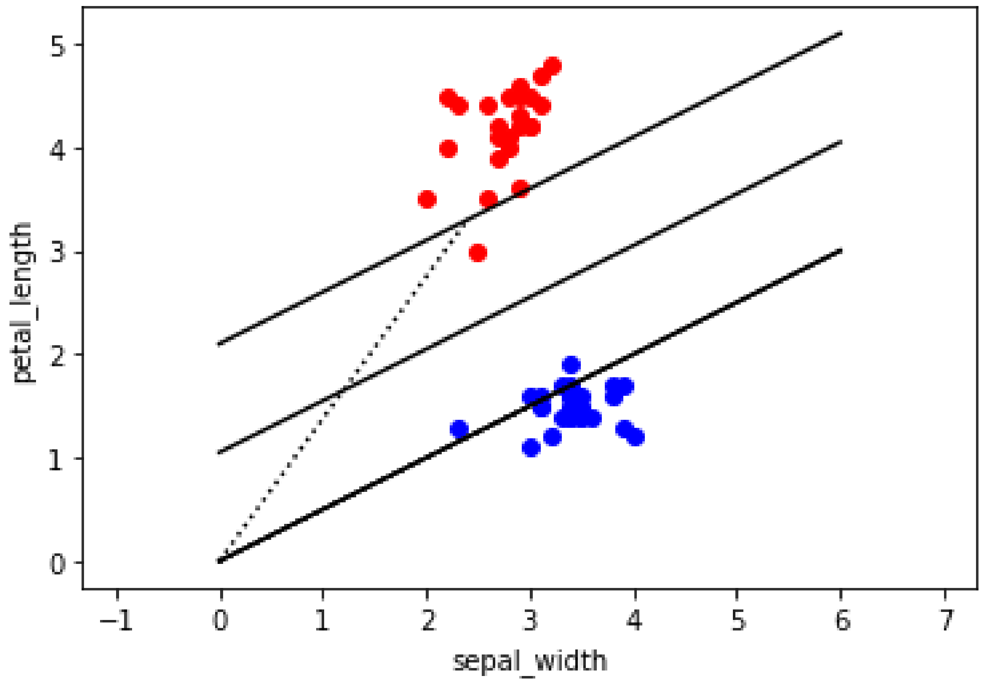
The most important thing to gain from this are the pictures, for they explain how the SVM classifier works.

Let's plot the same pictures, but with the predicted values from the test data.

[286]:
```
indxS = np.where(y_pred == 'setosa')[0]
indxV = np.where(y_pred != 'setosa')[0]
P1 = np.array([0, -c/b]); P2=np.array([6, -(6*a + c)/b])
```

[287]:
```
plt.scatter(X_test[indxS,0], X_test[indxS,1], color='blue')
plt.scatter(X_test[indxV,0], X_test[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
for k in range(NumSVecs.sum()):
  plt.plot([0,SVecs[k,0]],[0,SVecs[k,1]],':k')
  q = np.array([SVecs[k,0],SVecs[k,1]])
```

```
    cq = -a*q[0]-b*q[1]; P1[1]=-(0*a + cq)/b; P2[1]=-(6*a + cq)/b
    plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
```

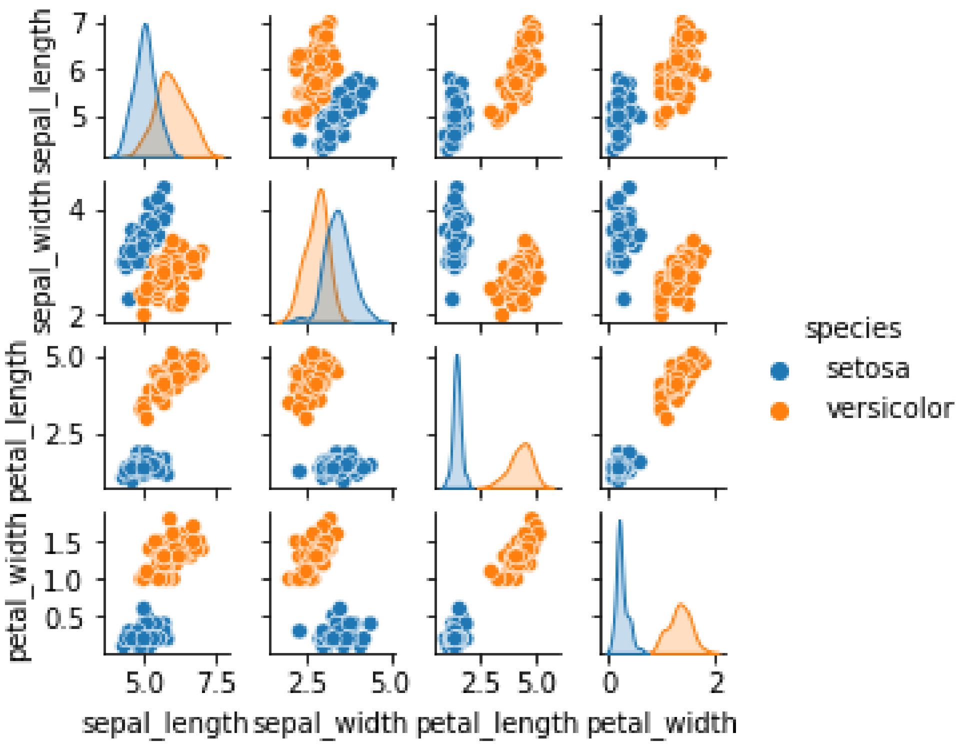


This is working well because we have very highly separated data.

Let's go back to the data set and pick another pair of features for which the separation is not as good.

```
[288]: sns.pairplot(dfid, hue='species', height = 1.0)
```

```
[288]: <seaborn.axisgrid.PairGrid at 0x7fdaaa639748>
```



Let's choose *sepal width* and *sepal length*.

[289]:
```
dfid.head(2)
```

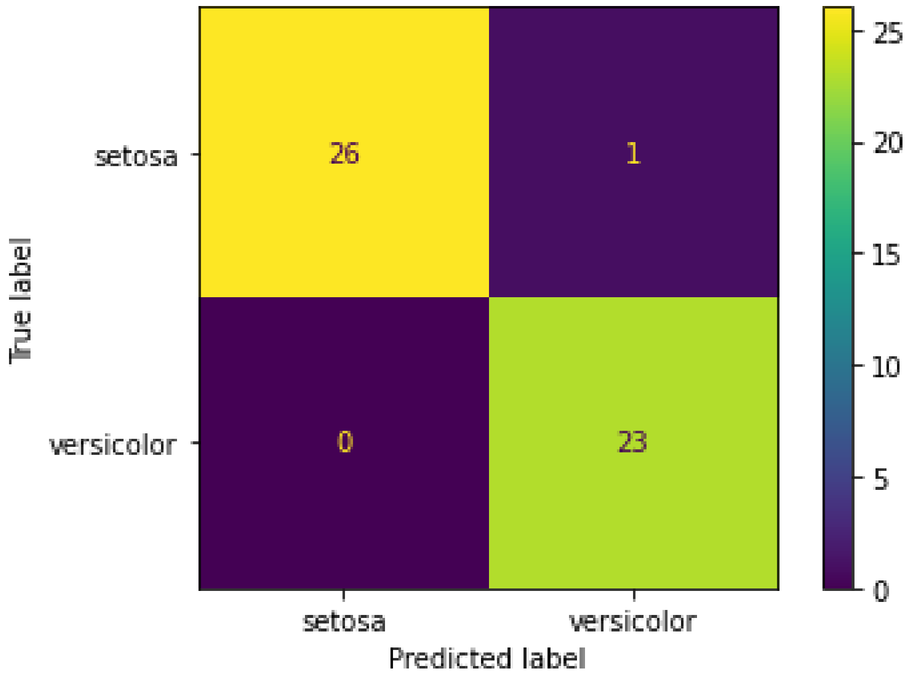
[289]:
```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
```

[290]:
```
# we use sepal length and sepal width as our features
X = dfid.iloc[:,[1,0]].values
# and species as our label
y = dfid.iloc[:, 4].values

# We'll use 50% of the data to test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50)

# instance the SVM - C affects the 'goodness' of the decision boundary
svmclf = svm.SVC(kernel='linear', C=100)
# and fit the training data
svmclf.fit(X_train, y_train)

# make predictions on the test set
y_pred = svmclf.predict(X_test)
```

[291]:
```
# get the confusion matrix and accuracy data
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
accsc = accuracy_score(y_test,y_pred)
print("Accuracy:", accsc)
```

```
Confusion Matrix:
[[26  1]
 [ 0 23]]
Accuracy: 0.98
```

[292]:
```
# plot a nicer confusion matrix
cmplot = ConfusionMatrixDisplay(cm, display_labels=svmclf.classes_)
cmplot.plot()
plt.show()
```



There are a lot more support vectors this time. These are the ones used in the optimization process.

[293]:
```
indxS = np.where(y_train == 'setosa')[0]
indxV = np.where(y_train != 'setosa')[0]
# get support vectors and number of them for each class
NumSVecs = svmclf.n_support_
print('There are these many per class', NumSVecs)
SVecs = svmclf.support_vectors_
```

```
print('The transposed support vectors are:\n', SVecs.T)
```

```
There are these many per class [1 2]
The transposed support vectors are:
 [[3.4 3.4 2.7]
 [5.4 6.  5.2]]
```

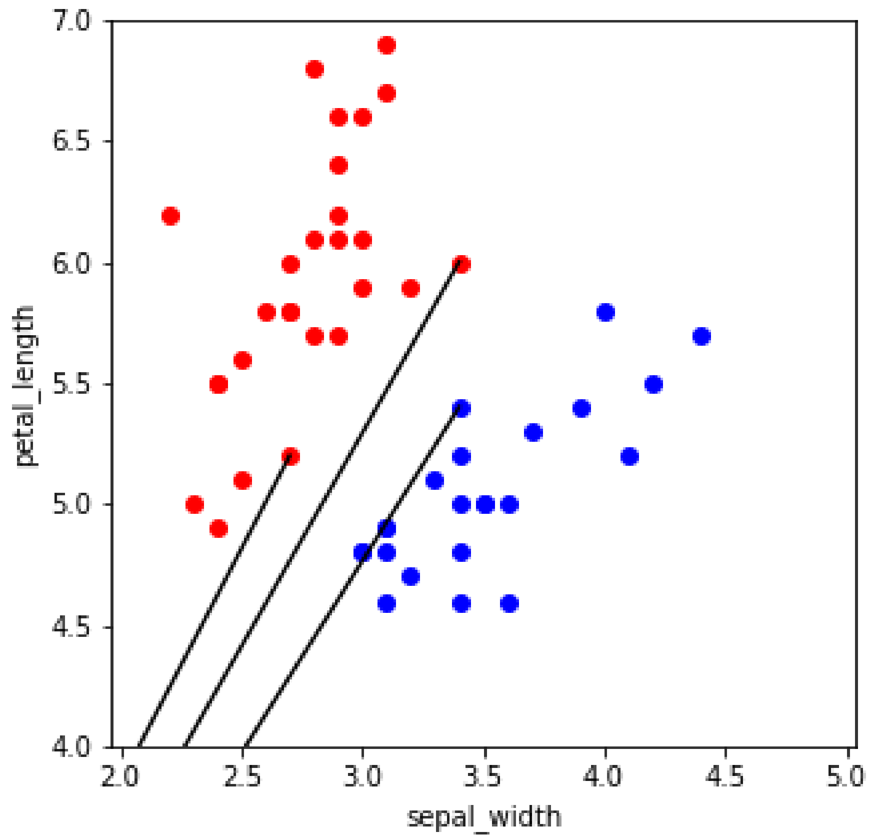
We can get the equation of the decision boundary as before…

```
[294]: # a x1 + b x2 + c = 0 => x2 = -(a x1 + c)/b
print('svmclf.intercept_ = ', svmclf.intercept_.shape)
a = svmclf.coef_[0,0]
b = svmclf.coef_[0,1]
c = svmclf.intercept_[0]
print(f'a={a}, b={b}, c={c}')

P1 = np.array([2, -(2*a + c)/b]); P2=np.array([4, -(4*a + c)/b])
print('P1 = ', P1, ', P2 = ', P2)
```

```
svmclf.intercept_ =  (1,)
a=-3.8078677079905554, b=3.3318988285270525, c=-6.044937598341233
P1 =  [2.         4.09996633] , P2 =  [4.         6.38567061]
```

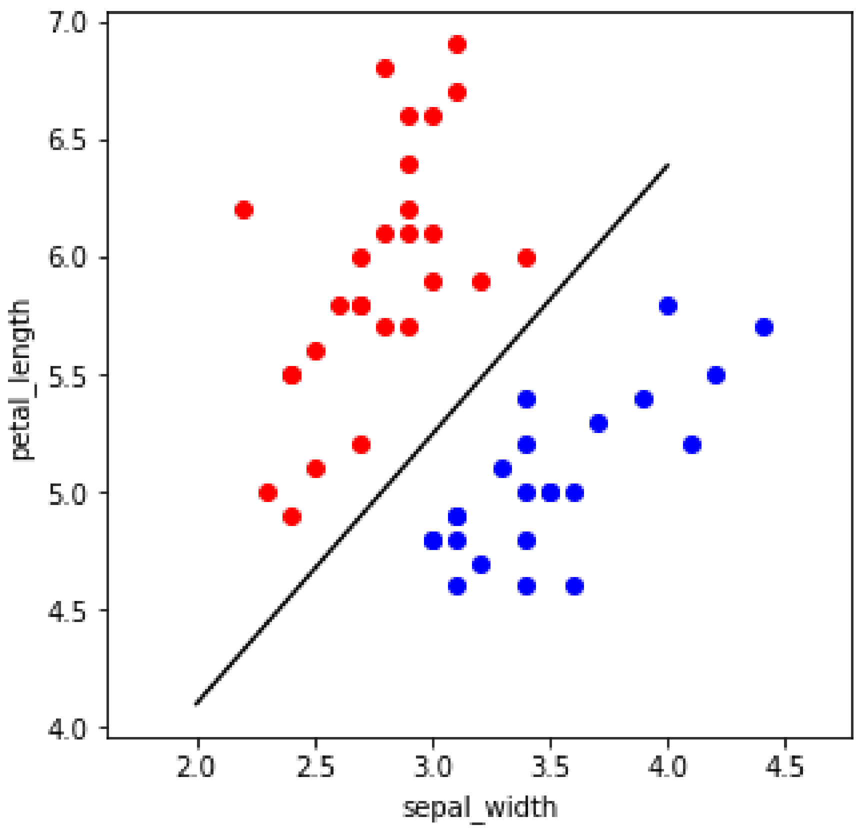
```
[295]: # plot the training data with the support vectors
plt.figure(figsize=(5,5));
plt.scatter(X_train[indxS,0], X_train[indxS,1], color='blue')
plt.scatter(X_train[indxV,0], X_train[indxV,1], color='red')
plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
# plot the support vectors in black - stemming form the origin
for k in range(NumSVecs.sum()):
  plt.plot([0,SVecs[k,0]],[0,SVecs[k,1]],'k')
plt.xlim(2,5); plt.ylim(4,7)
```

```
[295]: (4.0, 7.0)
```



```
[296]: # plot training data with decision boundary
       plt.figure(figsize=(5,5))
       plt.scatter(X_train[indxS,0], X_train[indxS,1], color='blue')
       plt.scatter(X_train[indxV,0], X_train[indxV,1], color='red')
       plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
       plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
       #plt.plot(xx, yy, "y-")
```

```
[296]: [<matplotlib.lines.Line2D at 0x7fda9aacc550>]
```

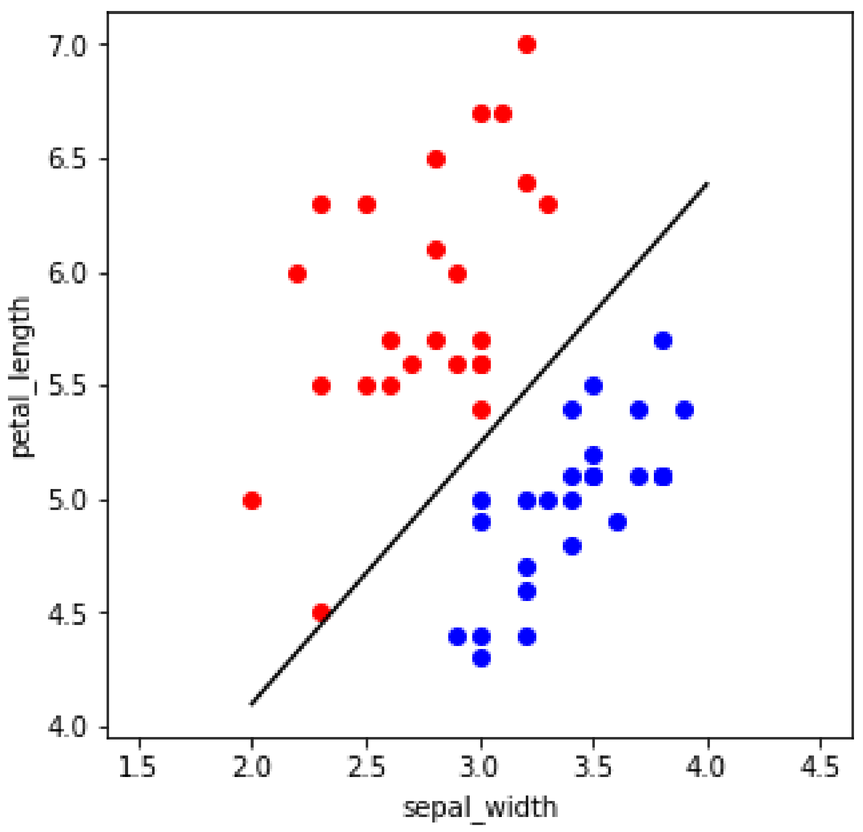


Let's look at the predictions

```
[297]: indxS = np.where(y_pred == 'setosa')[0]
       indxV = np.where(y_pred != 'setosa')[0]
       P1 = np.array([2, -(2*a + c)/b]); P2=np.array([4, -(4*a + c)/b])
```

```
[298]: # test data with predictions and decision boundary
       plt.figure(figsize=(5,5))
       plt.scatter(X_test[indxS,0], X_test[indxS,1], color='blue')
       plt.scatter(X_test[indxV,0], X_test[indxV,1], color='red')
       plt.axis('equal'); plt.xlabel('sepal_width'); plt.ylabel('petal_length');
       plt.plot([P1[0],P2[0]],[P1[1],P2[1]],'k')
```

```
[298]: [<matplotlib.lines.Line2D at 0x7fdac9bc8fd0>]
```



The separation of the data is less than before, so the SVM might not do as well.

1.2.6 Review

We covered *just enough*, to make *progress at pace*. We looked at

- lines, planes, hyperplanes
- distance from a point to a hyperplane: optimisation
- Binary Classification with SVM

1.2.7 Next...

The SVM is an important tool, and like many of the other things we have discussed, we could talk much more about it. It can for example be used in multi-class applications and also for regression.

However, we have a much bigger goal: **deep neural networks**.

For that we need to move on to the **perceptron**.

1.3 Technical Notes, Production and Archiving

Ignore the material below. What follows is not relevant to the material being taught.

Production Workflow

- Finalise the notebook material above
- Clear and fresh run of entire notebook
- Create html slide show:
    - `jupyter nbconvert --to slides 12_svm.ipynb`
- Set `OUTPUTTING=1` below
- Comment out the display of web-sourced diagrams
- Clear and fresh run of entire notebook
- Comment back in the display of web-sourced diagrams
- Clear all cell output
- Set `OUTPUTTING=0` below
- Save
- git add, commit and push to FML
- copy PDF, HTML etc to web site
    - git add, commit and push
- rebuild binder

Some of this originated from

https://stackoverflow.com/questions/38540326/save-html-of-a-jupyter-notebook-from-within-the-r

These lines create a back up of the notebook. They can be ignored.

At some point this is better as a bash script outside of the notebook

[39]:
```
%%bash
NBROOTNAME=12_svm
OUTPUTTING=1

if [ $OUTPUTTING -eq 1 ]; then
  jupyter nbconvert --to html $NBROOTNAME.ipynb
  cp $NBROOTNAME.html ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.html
  mv -f $NBROOTNAME.html ./formats/html/

  jupyter nbconvert --to pdf $NBROOTNAME.ipynb
  cp $NBROOTNAME.pdf ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.pdf
  mv -f $NBROOTNAME.pdf ./formats/pdf/

  jupyter nbconvert --to script $NBROOTNAME.ipynb
  cp $NBROOTNAME.py ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.py
  mv -f $NBROOTNAME.py ./formats/py/
else
  echo 'Not Generating html, pdf and py output versions'
fi
```

```
Not Generating html, pdf and py output versions
```