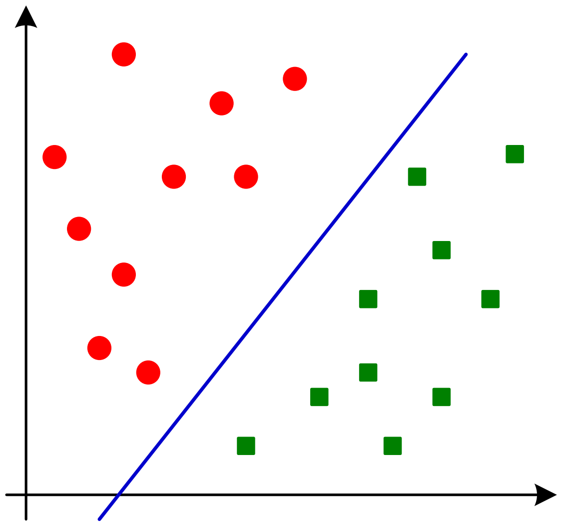
Support Vector Machines

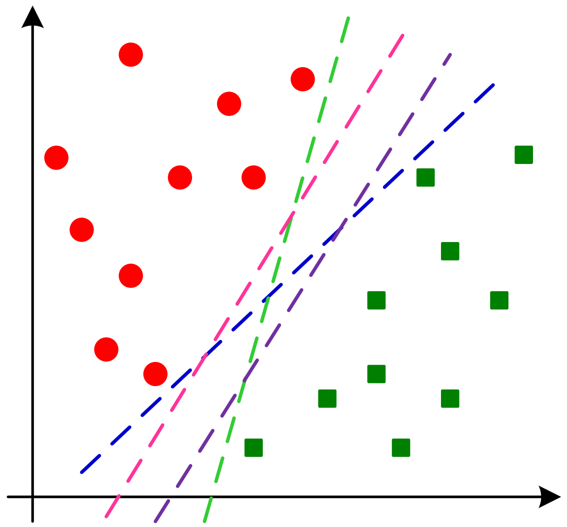
Overview

The Support Vector Machine (SVM) algorithm is a supervised learning algorithm for dividing data into two classes using a hyperplane (line in 2D).



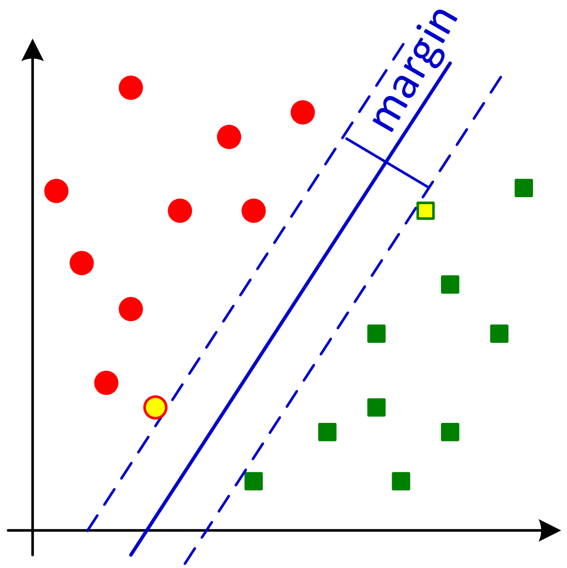
Overview

There may be several hyperplanes that separate the classes.



SVM seeks the hyperplane that ‘best’ separates the classes by giving the greatest distance to points in either category. This is called the margin.

The points closest to the hyperplane in each class are support vectors.



Overview

SVM requires a training set of points that are already labelled with the correct category.

Point	**Feature 1** (x_1)	**Feature 2** (x_2)	**...**	**Class**
1	1.5	3		●
2	5.8	1.9		■
3	7.1	2.4		■
4	8.2	6.4		■
5	4.7	6.4		●
⋮	⋮	⋮		

Hence SVD is a supervised learning algorithm.

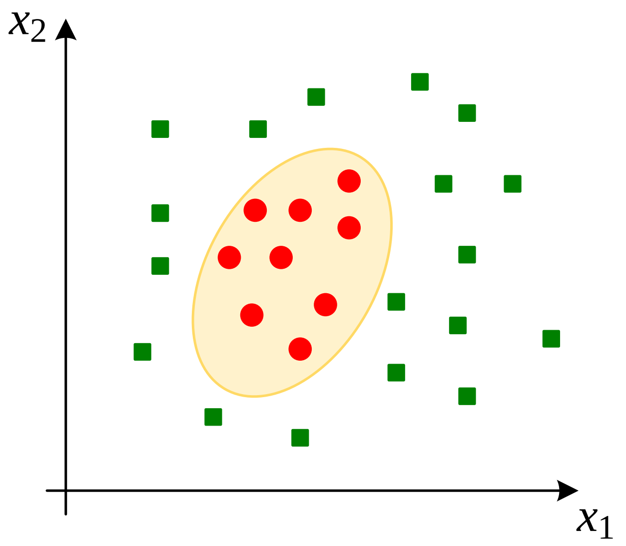
Overview

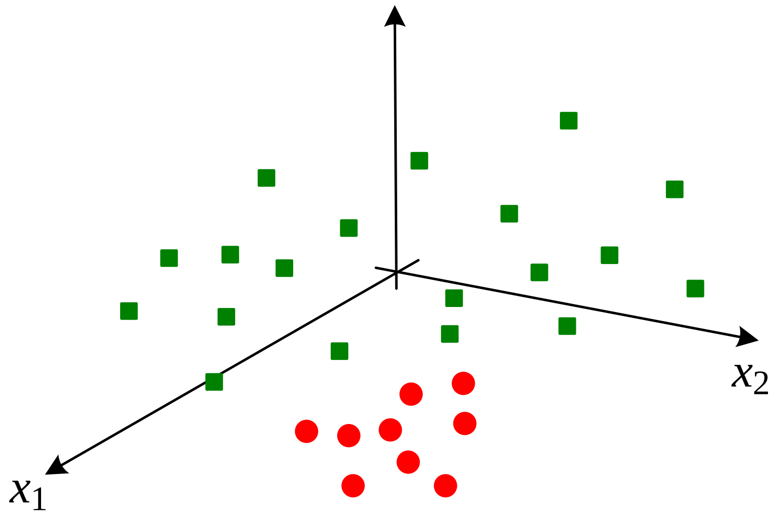
It may not be possible to separate the classes with a hyperplane.

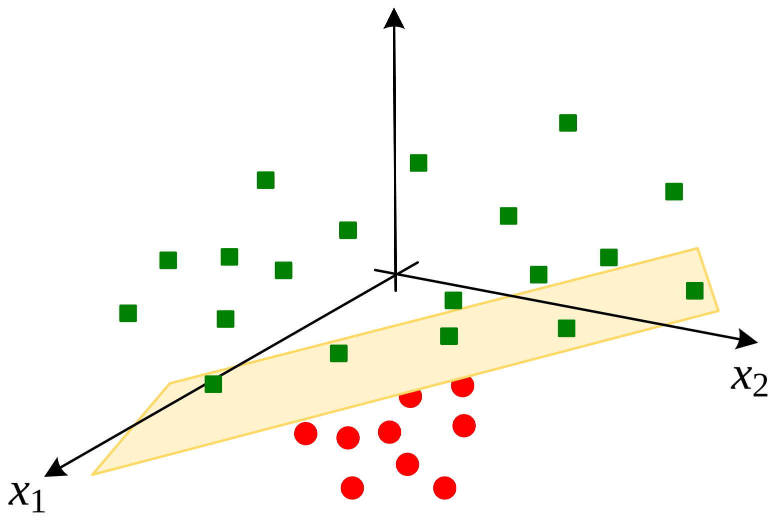
It may be possible to calculate a new attribute.

Then separate with a hyperplane in the higher dimensional space.

Finally project back onto the original space.







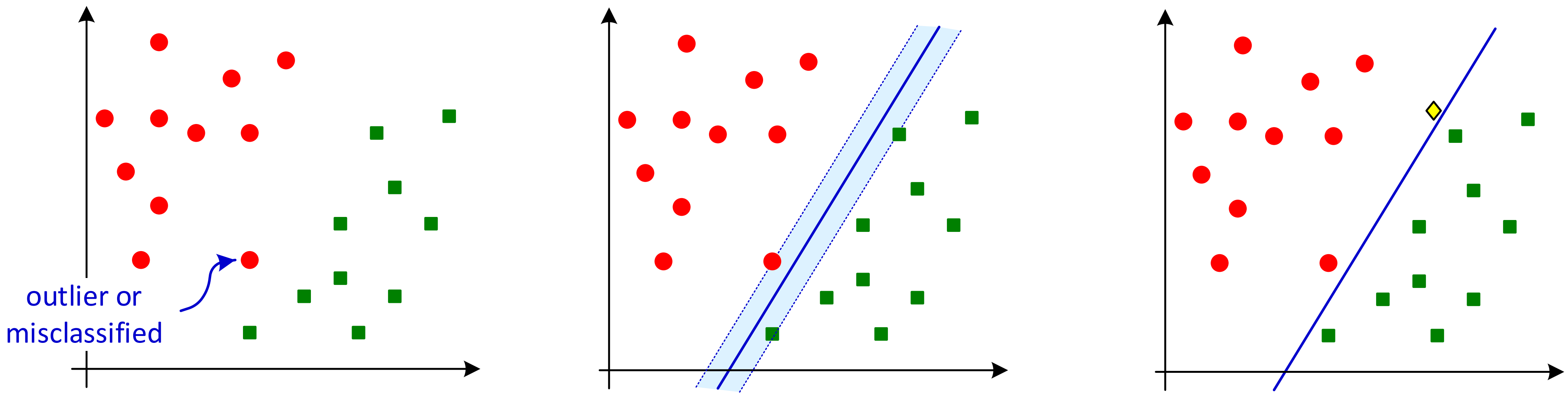
Maximum margin classifier

The maximum margin classifier – the hyperplane that maximises the margin to each class – is sensitive to outliers in the data or incorrectly classified data points.

Consider the following data with a point that is an outlier or misclassified as red.

The hyperplane classifier now gives a very small margin.

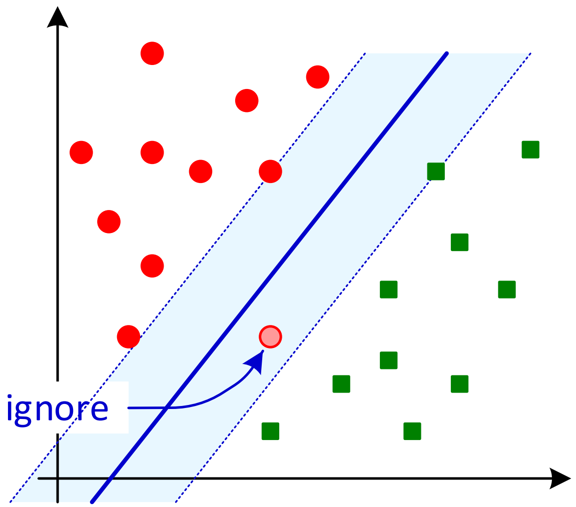
Hence this new observation is likely to be incorrectly classified as red.



Maximum margin classifier

As we will see the SVM algorithm allows us to ignore some outliers or misclassified points when determining the maximum margin classifier.

In the previous example, ignoring the point that is an outlier or misclassified point gives a more sensible maximum margin classifier.

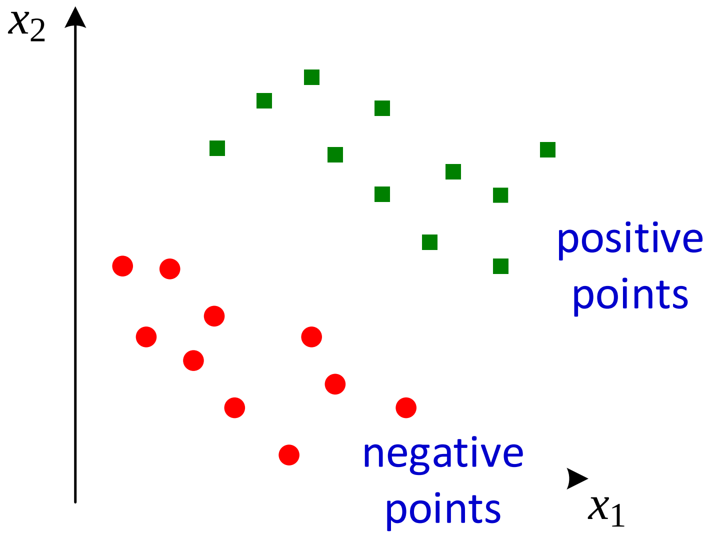


Linear Support Vector Machines

Firstly, we assume that we have data that is linearly separable – that is where a hyperplane will separate the data – and has no outliers or misclassified points.

We will illustrate the method with data that has two attributes x_1 and x_2 so can be represented in the plane $\mathbb{R}^2$.

Suppose the dataset is classified as a set of 'positive points' and a set of 'negative points'.

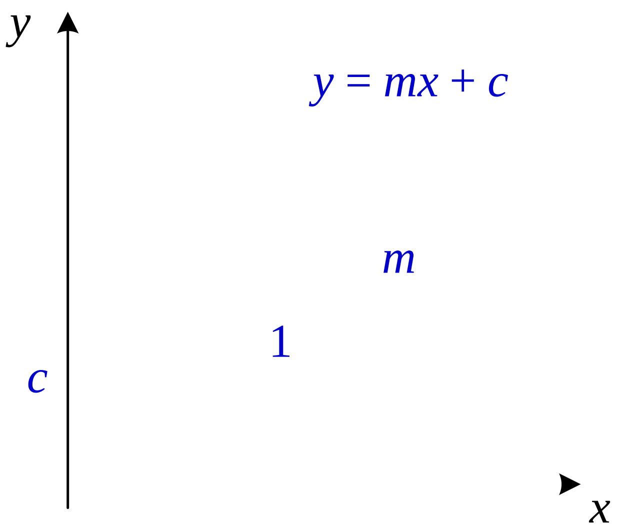


Lines and hyperplanes

We know that a line in $\mathbb{R}^2$ can be represented by equation $y = mx + c$.

The equation $ax + by + c = 0$ $(a = m, b = -1)$ is more symmetric in x and y.

Now $ax + by + c = 0$ can be expressed in vector terms as $\boldsymbol{a} \cdot \boldsymbol{x} + c = 0$ where $\boldsymbol{a} = (a, b)$ and $\boldsymbol{x} = (x, y)$.



In general, a hyperplane in $\mathbb{R}^n$ can be represented by

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$$

where $\boldsymbol{w} = (w_1, w_2, \dots, w_n)$ and $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$.

Think of $\boldsymbol{w}$ as a weight vector and b is a bias term.

Linear Support Vector Machines

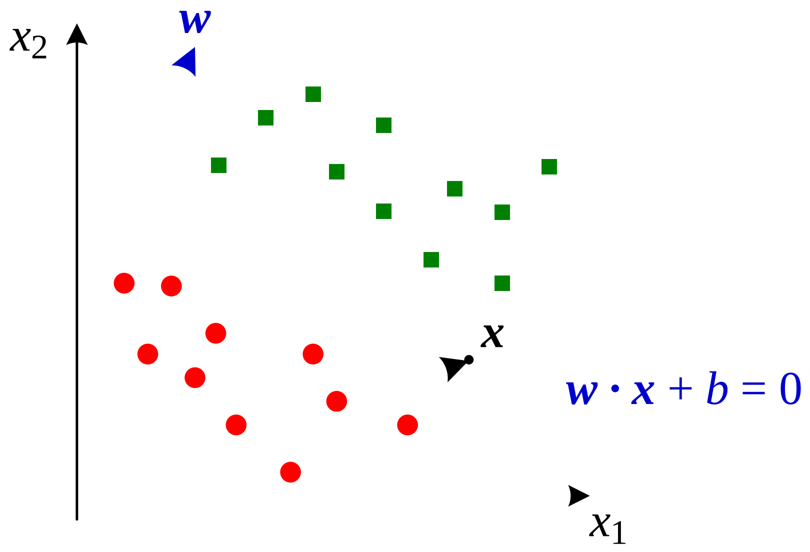
We will represent the separating hyperplane as

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0.$$

Expanding: $$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = 0.$$

This generalises the equation $ax + by + c = 0$ in $\mathbb{R}^2$.

Here $\boldsymbol{w}$ is a vector perpendicular (orthogonal) to the hyperplane.

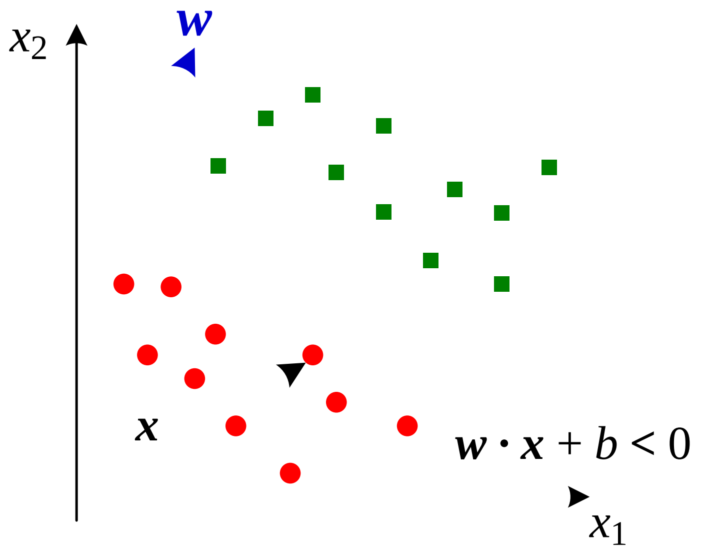


Linear Support Vector Machines

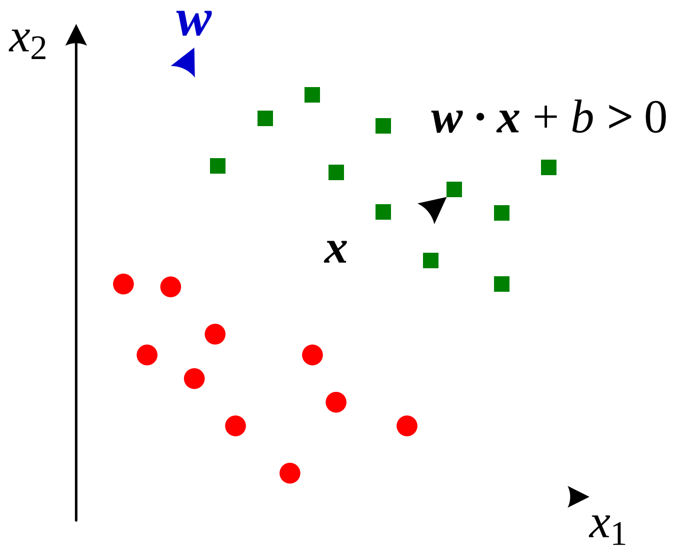
Hyperplane equation: $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0.$

If $\boldsymbol{x}$ is the position vector of a negative point then $\boldsymbol{w} \cdot \boldsymbol{x} + b < 0.$

If $\boldsymbol{x}$ is the position vector of a positive point then $\boldsymbol{w} \cdot \boldsymbol{x} + b > 0.$



Hence the terminology of ‘positive’ points and ‘negative’ points.

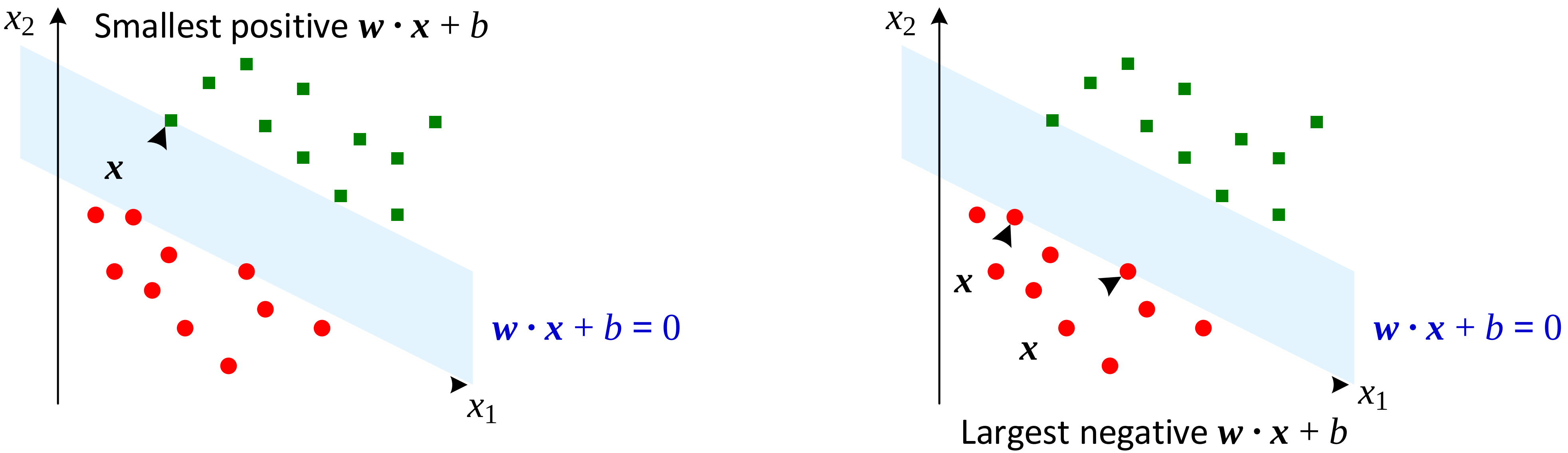


Linear Support Vector Machines

Recall that support vectors are closest to the separating hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$.

A positive support vector has smallest positive value for $\boldsymbol{w} \cdot \boldsymbol{x} + b$.

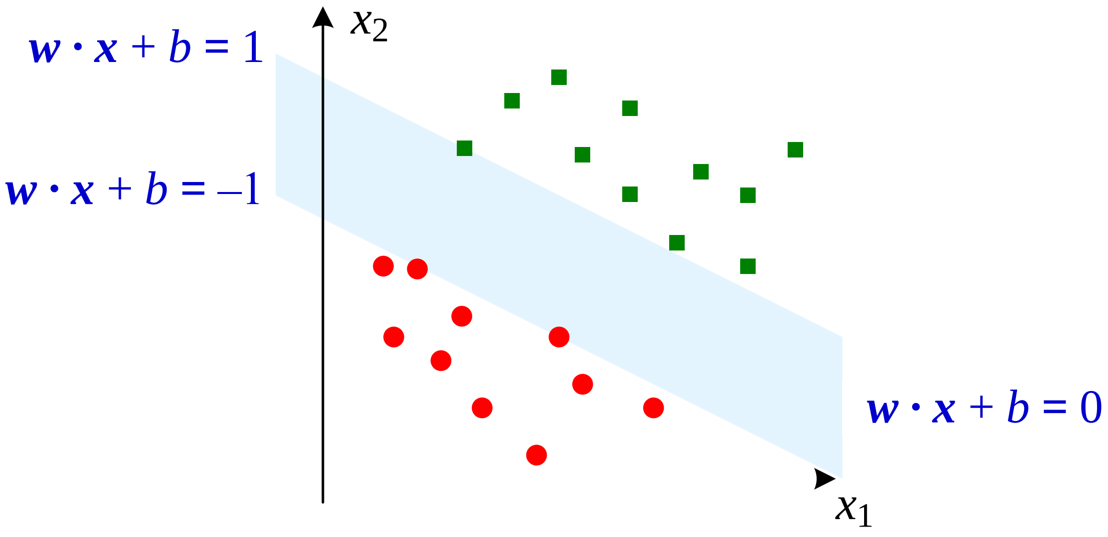
Similarly, a negative support vector has largest negative value for $\boldsymbol{w} \cdot \boldsymbol{x} + b$.



Linear Support Vector Machines

Generally, the separating hyperplane equation $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ is scaled so that

- $\boldsymbol{w} \cdot \boldsymbol{x} + b = 1$ for positive support vectors
- $\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$ for negative support vectors.

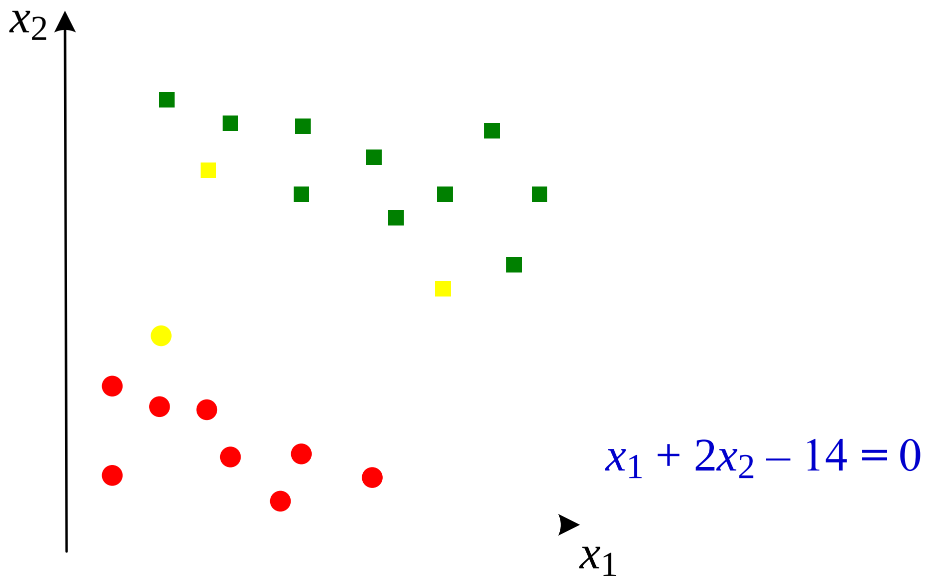


Example

Consider the following example where the hyperplane classifier is

$$x_1 + 2x_2 - 14 = 0.$$

The support vectors – two positive and one negative – are shown in yellow.



Consider positive support vector $(8, 5)$.

Substituting into $x_1 + 2x_2 - 14$ gives

$$8 + 10 - 14 = 4.$$

We want the RHS of the equation for a positive support vector to be $+1$ so we need to scale by a factor $1/4$.

Example

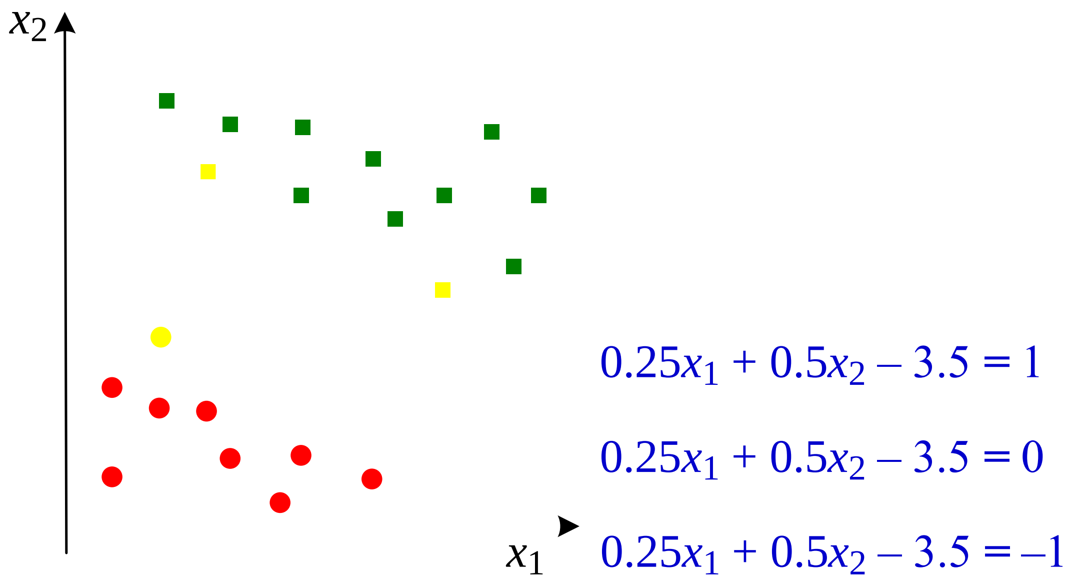
Scaling by $1/4$ the hyperplane classifier equation is $0.25x_1 + 0.5x_2 - 3.5 = 0.$

Positive support vectors:

- $(3,7.5)$ satisfies
$$0.25 \times 3 + 0.5 \times 7.5 - 3.5$$
$$= 0.75 + 3.75 - 3.5 = 1.$$
- $(8,5)$ satisfies
$$0.25 \times 8 + 0.5 \times 5 - 3.5$$
$$= 2 + 2.5 - 3.5 = 1$$

Negative support vector:

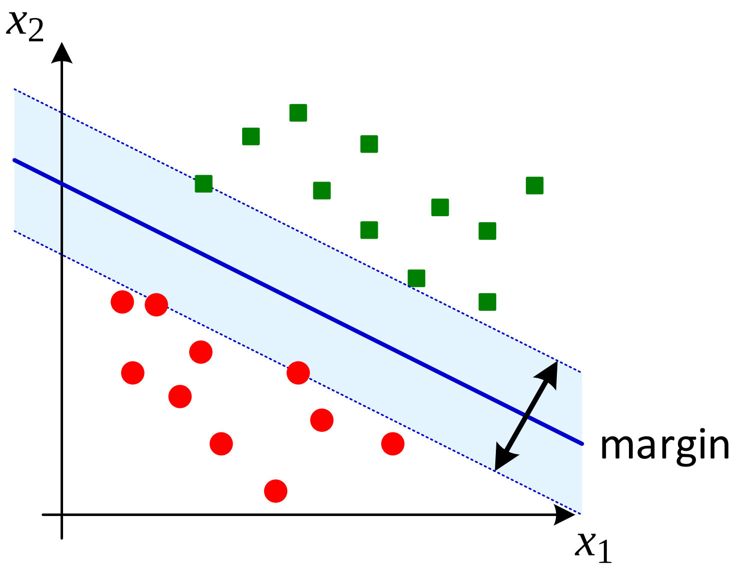
- $(2, 4)$ satisfies $0.25 \times 2 + 0.5 \times 4 - 3.5$
$$= 0.5 + 2 - 3.5 = -1.$$

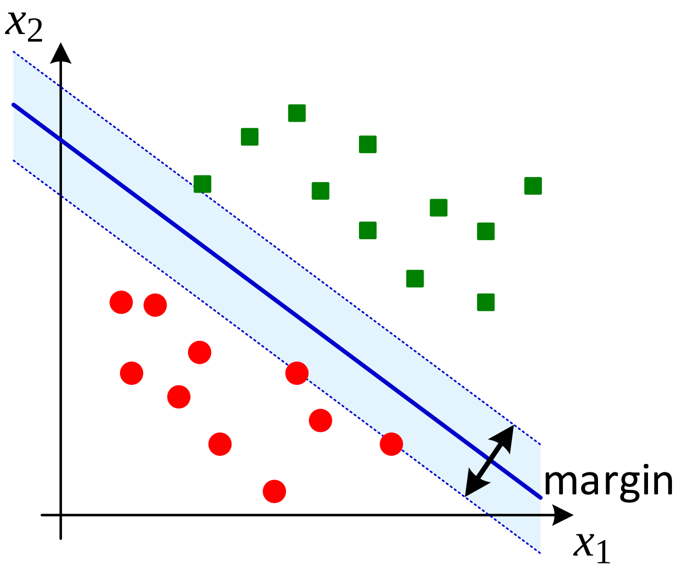


Finding the hyperplane classifier

We have seen that there are many hyperplanes that separate the data into positive and negative points.

We need to find the hyperplane that maximises the margin.





Finding the hyperplane classifier

Since the equation of a hyperplane is $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$, we need to

$$\max_{\boldsymbol{w},b} (\text{margin}) \;\; \text{such that} \; \begin{Bmatrix} \boldsymbol{w} \cdot \boldsymbol{x} + b \geq 1 & \boldsymbol{x} \text{ is a positive point} \\ \boldsymbol{w} \cdot \boldsymbol{x} + b \leq -1 & \boldsymbol{x} \text{ is a negative point} \end{Bmatrix}$$

We can simplify the constraint by setting

$$y_x = \begin{Bmatrix} +1 & \boldsymbol{x} \text{ is a positive point} \\ -1 & \boldsymbol{x} \text{ is a negative point} \end{Bmatrix}$$

Then the constraint just becomes $y_x(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1$ for all points $\boldsymbol{x}$.

So we need to

$$\max_{\boldsymbol{w},b} (\text{margin}) \;\; \text{such that} \;\; y_x(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1.$$

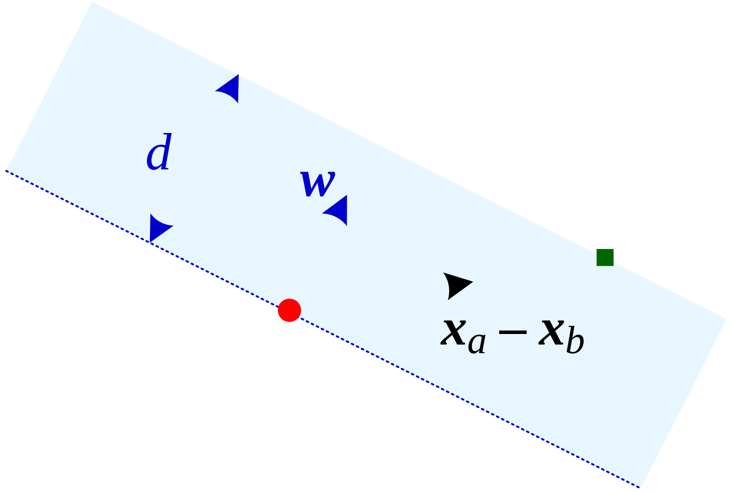
Finding the hyperplane classifier

To find an expression for the margin, let $\boldsymbol{x}_a$ be a positive support vector and let $\boldsymbol{x}_b$ be a negative support vector.

The vector joining $\boldsymbol{x}_a$ to $\boldsymbol{x}_b$ is $\boldsymbol{x}_a - \boldsymbol{x}_b$.

Note that $\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}$ is a unit vector. This means that the width d is

$$d = \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b) = \frac{1}{\|\boldsymbol{w}\|}(\boldsymbol{w} \cdot \boldsymbol{x}_a - \boldsymbol{w} \cdot \boldsymbol{x}_b).$$



Since $\boldsymbol{x}_a$ is a positive support vector $\boldsymbol{w} \cdot \boldsymbol{x}_a + b = 1$

so $\boldsymbol{w} \cdot \boldsymbol{x}_a = 1 - b$.

Since $\boldsymbol{x}_b$ is a negative support vector $\boldsymbol{w} \cdot \boldsymbol{x}_b + b = -1$

So $\boldsymbol{w} \cdot \boldsymbol{x}_b = -1 - b$.

Finding the hyperplane classifier

Summary: $d = \frac{1}{\|\boldsymbol{w}\|}(\boldsymbol{w} \cdot \boldsymbol{x}_a - \boldsymbol{w} \cdot \boldsymbol{x}_b)$ where $\boldsymbol{w} \cdot \boldsymbol{x}_a = 1 - b$ and $\boldsymbol{w} \cdot \boldsymbol{x}_b = -1 - b$.

Substituting

$$d = \frac{1}{\|\boldsymbol{w}\|}(\boldsymbol{w} \cdot \boldsymbol{x}_a - \boldsymbol{w} \cdot \boldsymbol{x}_b) = \frac{1}{\|\boldsymbol{w}\|}(1 - b - (-1 - b)) = \frac{1}{\|\boldsymbol{w}\|}(1 - b + 1 + b) = \frac{2}{\|\boldsymbol{w}\|}.$$

So to find the hyperplane classifier, we need to

$$\max_{\boldsymbol{w},\, b} \frac{2}{\|\boldsymbol{w}\|} \text{ subject to the constraint } y_x(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1 \text{ for all points } \boldsymbol{x}.$$

SVM using sklearn

As usual **sklearn** will solve the constrained optimisation problem $\max_{\boldsymbol{w}, b} \frac{2}{\|\boldsymbol{w}\|}$ subject to the constraint $y_x(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1$ for all points $\boldsymbol{x}$.

The method SVC takes as input two arrays:

- an array X of shape (n_samples, n_features) holding the training samples
- an array y of class labels (strings or integers) of shape (n_samples).

```
from sklearn import svm
X = [[0, 0], [1, 1]]
y = [0, 1]
clf = svm.SVC()
clf.fit(X, y)
SVC()
```

SVM using scikit-learn

After being fitted, the model can then be used to predict new values.

```
clf.predict([[2., 2.]])

array([1])
```

SVM using scikit-learn

Also, some properties of the support vectors can be found in attributes support_vectors_, support_ and n_support_

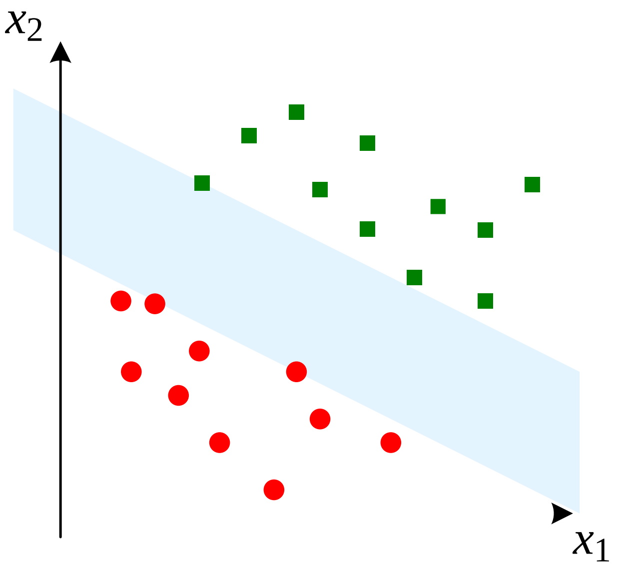
```
# get support vectors
clf.support_vectors_
array([[0., 0.],
       [1., 1.]])
# get indices of support vectors
clf.support_
array([0, 1]...)
# number of support vectors for each class
clf.n_support_
array([1, 1]...)
```

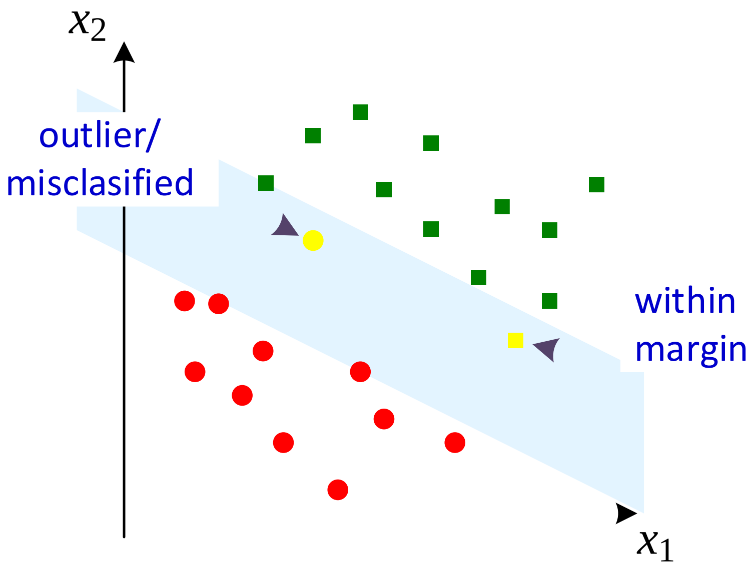
Hard-margin classification

Our description of the hyperplane classifier assumed that all positive points satisfied $\boldsymbol{w} \cdot \boldsymbol{x} + b \geq 1$ and all negative points satisfied $\boldsymbol{w} \cdot \boldsymbol{x} + b \leq -1$.

We assumed no outliers or misclassified points or even correctly classified points within the margin.

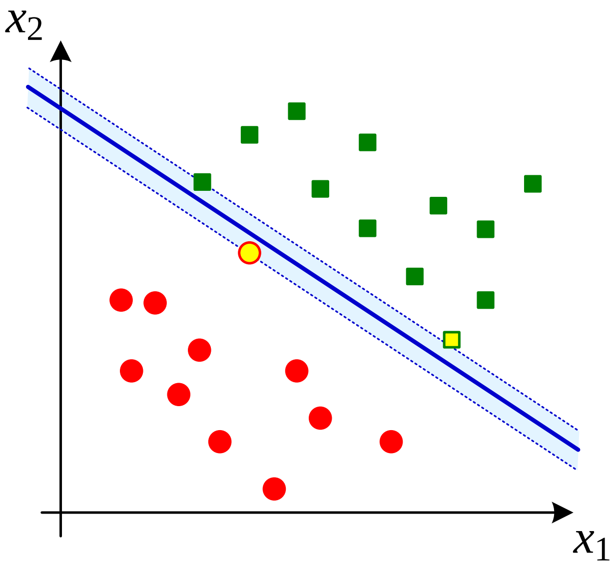
This is called hard-margin classification.



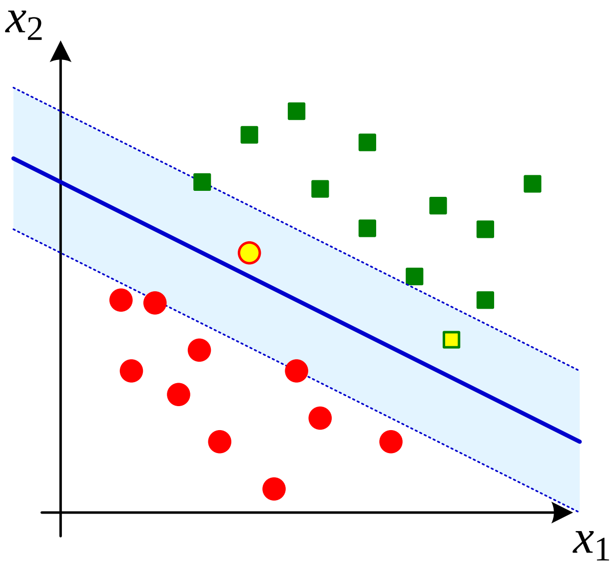


Soft-margin classification

With soft-margin classification, we allow some misclassification (if this improves the margin).



Hard-margin classification



Soft-margin classification

Soft-margin classification

To see how soft-margin classification works, we first change the optimisation problem from

$$\max_{\boldsymbol{w},\, b} \frac{2}{\|\boldsymbol{w}\|} \text{ subject to the constraint } y_x(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1 \text{ for all points } \boldsymbol{x}$$

to

$$\min_{\boldsymbol{w},\, b} \frac{\|\boldsymbol{w}\|}{2} \text{ subject to the constraint } y_x(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1 \text{ for all points } \boldsymbol{x}.$$

Now we add another term to the expression to be minimised:

$$\min_{w,\, b} \left(\frac{\|w\|}{2} + c \sum_{i=1}^{n} \zeta_i \right) \text{ subject to } y_{x_i}(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 - \zeta_i$$

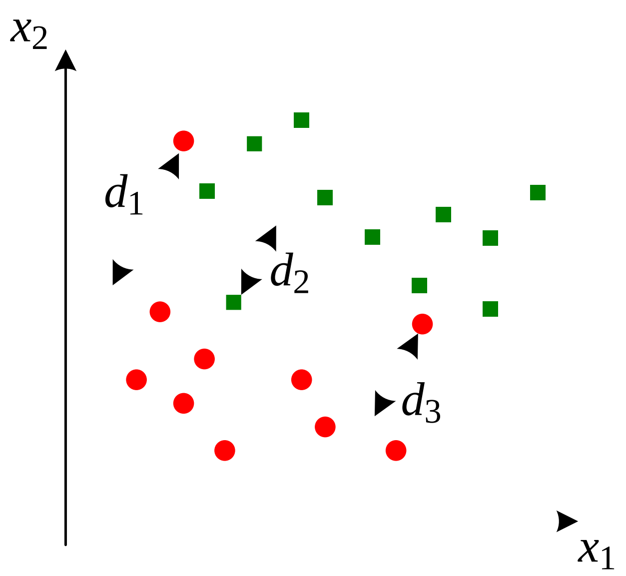
where the summation is over all the data points and ζ is Greek letter zeta.

Soft-margin classification

In the expression being minimised

$$\min_{\boldsymbol{w},\, b}\left(\frac{\|\boldsymbol{w}\|}{2} + c\sum_{i=1}^{n}\zeta_i\right),$$

- for all correctly classified points, $\zeta_i = 0$,
- for misclassified points, ζ_i is the distance from the *correct* boundary hyperplane, and
- c is a hyperparameter.



Soft-margin classification

We can think of the expression to be minimised

$$\frac{\|\boldsymbol{w}\|}{2} + c\sum_{i=1}^{n} \zeta_i$$

as $$\text{SVM error} = \text{margin error} + \text{classification error}.$$

For large values of the hyperparameter c, the classification error term dominates.

In this case, the optimisation will seek to reduce classification errors (as in the hard-margin case).

For smaller values of the hyperparameter c, the optimisation will seek larger margin but allow some misclassification.

Soft-margin classification in scikit-learn

In scikit-learn, we can specify a regularisation parameter C.

By default, C = 1 which gives a hard margin classifier.

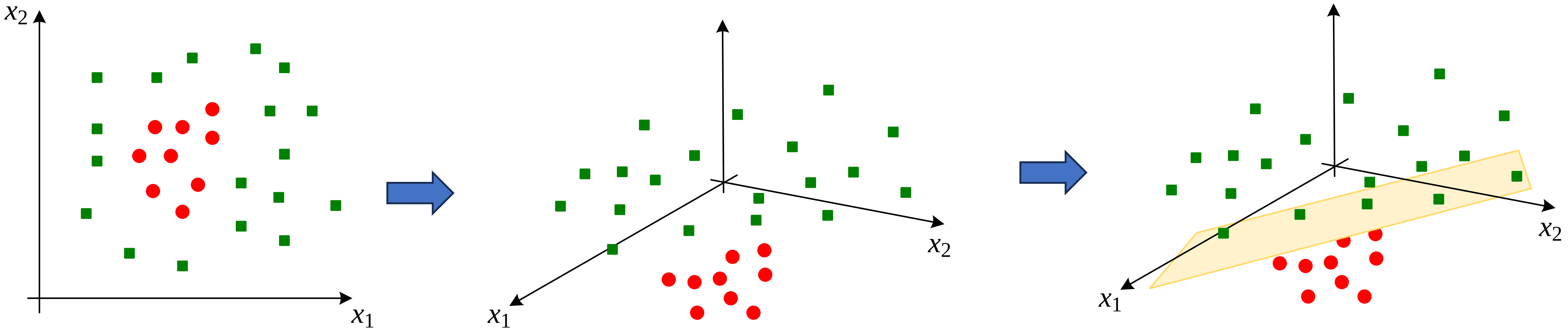
```
from sklearn import svm
clf = SVC(C=1)
```

To use a soft-margin SVM, set a smaller value such as C = 0.1.

```
from sklearn import svm
clf = SVC(C=0.1)
```

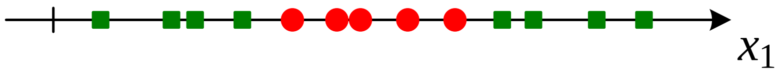
Non-linear SVM

We have seen that, when the data is not linearly separable we may be able to introduce a new parameter so that it becomes linearly separable in higher dimensions.



Non-linear SVM

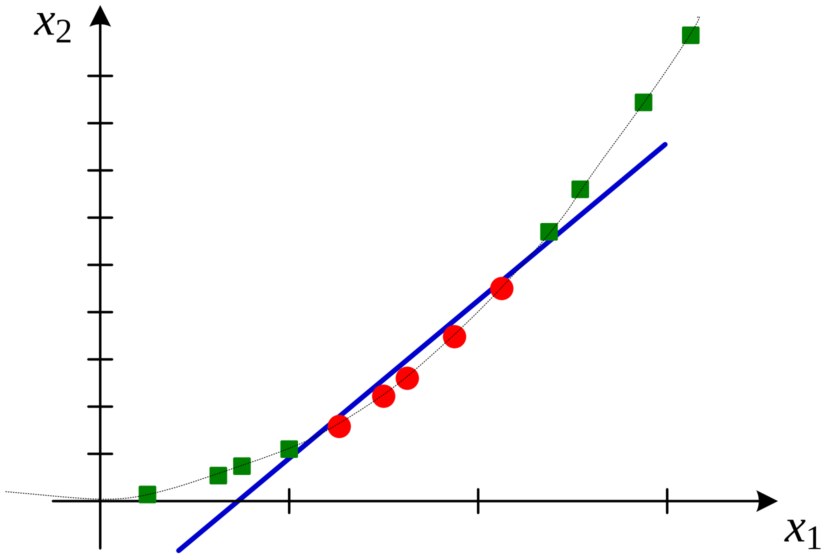
To keep the diagrams simpler, we will illustrate this using data with a single feature which can therefore be illustrated on a line.



Not linearly separable – no single point separates the data

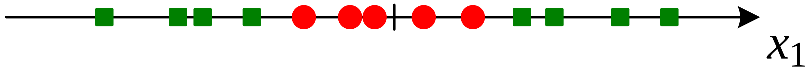
Suppose we introduce a new variable x_1^2.

In 2D, the data (x_1, x_1^2) is linearly separable.



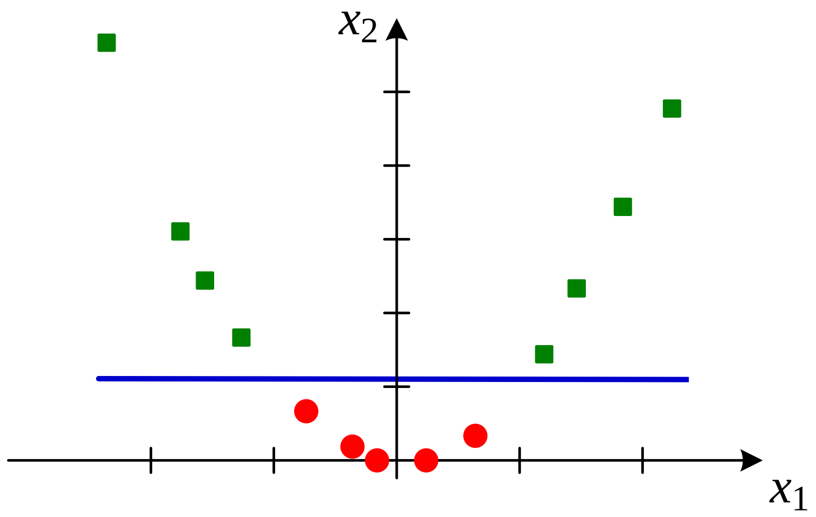
Non-linear SVM

This might look a bit clearer if the red dots are located around the origin.



Again we introduce a new variable x_1^2.

In 2D, the data (x_1, x_1^2) is linearly separable.



Kernel

The function that generated the new attribute – $f(x) = x^2$ in our case – is called the kernel function.

Our kernel is a polynomial of degree 2 and is one of the simplest kernel functions.

If we had two attributes (x_1, x_2) then a general polynomial of degree 2 is more complicated:

$$f(x_1, x_2) = a_0 + a_1x_1 + a_2x_2 + b_1x_1^2 + b_2x_2^2 + cx_1x_2.$$

Higher degree polynomial kernels – degree 3, 4, 5, … – will be more complicated still.

Kernel

In general, to train a non-linear support vector classifier using polynomial kernels, we would have to perform operations with the higher dimensional vectors in the transformed feature space.

In real applications, there might be many features and applying transformations that involve many polynomial combinations of these features will lead to high computational costs.

It turns out that there is a computationally more efficient approach called the kernel trick.

Kernel trick

Suppose our kernel transformation function is $f(\boldsymbol{x})$.

We can then find the classifier in the higher dimensional space: $\boldsymbol{w} \cdot f(\boldsymbol{x}) + b = 0$.

Instead of explicitly applying the transformations $f(\boldsymbol{x})$ and representing the data by the transformed coordinates in a higher dimensional feature space, we only need to consider dot products $\boldsymbol{x} \cdot \boldsymbol{x}'$ of the original data observations in the lower dimensional space.

In other words, the kernel trick allows us to work in the original feature space without computing the coordinates of the data in a higher dimensional space.

An example will illustrate this.

Kernel trick example

Suppose our data has three attributes $\boldsymbol{x} \in (x_1, x_2, x_3) \in \mathbb{R}^3$ and we want a quadratic kernel.

In this case there are 9 quadratic terms so the transformed vectors are

$$f(\boldsymbol{x}) = (x_1^2, x_1x_2, x_1x_3, x_1x_2, x_2^2, x_2x_3, x_1x_3, x_2x_3, x_3^2) \in \mathbb{R}^9.$$

In calculating the classifier in the transformed space we need to evaluate dot products in $\mathbb{R}^9$:

$$f(\boldsymbol{x}) \cdot f(\boldsymbol{y}) = x_1^2y_1^2 + x_1x_2y_1y_2 + \cdots = \sum_{i,j=1}^{3} x_i x_j y_i y_j.$$

Kernel trick example

However, with kernel function $k(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y})^2$ we have

$$k(\boldsymbol{x}, \boldsymbol{y}) = \big((x_1, x_2, x_3) \cdot (y_1, y_2, y_3)\big)^2 = (x_1 y_1 + x_2 y_2 + x_3 y_3)^2 = \sum_{i,j=1}^{3} x_i x_j y_i y_j.$$

Hence $$f(\boldsymbol{x}) \cdot f(\boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y})^2.$$

In calculating $f(\boldsymbol{x}) \cdot f(\boldsymbol{y})$, we need to evaluate the dot product in $\mathbb{R}^9$.

However, in calculating $(\boldsymbol{x} \cdot \boldsymbol{y})^2$, we evaluate the dot product in $\mathbb{R}^3$ and square the corresponding real number. This is computationally more efficient and then effect is magnified further when there are more attributes.

Polynomial kernel

A general polynomial kernel of degree d is of the form

$$k(\boldsymbol{x}, \boldsymbol{y}) = (\gamma(\boldsymbol{x} \cdot \boldsymbol{y}) + c_0)^d.$$

The following code in scikit-learn implements a quadratic ($d = 2$) polynomial kernel.

```
from sklearn.svm import SVC
clf = SVC(kernel='poly', degree=2, coeff0=1, C=1, gamma='scale')
```

- If `gamma='scale'` (default) then it uses value `1/(n_features*X.var())`.
- If `gamma='auto'` then it uses value `1/n_features`.
- `coef0` is a constant and `C=1` means this is a hard-margin classifier.

Radial basis function (RBF) kernel

Apart from polynomials of degree arbitrary degree d there are two other common kernels.

The radial basis function (RBF) or Gaussian kernel

$$k(\boldsymbol{x}, \boldsymbol{y}) = e^{-\gamma \|\boldsymbol{x}-\boldsymbol{y}\|^2} = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{y}\|^2).$$

Here $\|\boldsymbol{x} - \boldsymbol{y}\|$ is the Euclidean distance between inputs.

In scikit-learn:

```
from sklearn.svm import SVC
clf = SVC(kernel='rbf', C=1, gamma=1)
```

Sigmoid kernel

The sigmoid kernel function is

$$k(\boldsymbol{x}, \boldsymbol{y}) = \tanh(\gamma(\boldsymbol{x} \cdot \boldsymbol{y}) + c_0)\,.$$

Here γ controls the slope of the sigmoid function and c_0 is a constant.

In scikit-learn:

```
from sklearn.svm import SVC
clf = SVC(kernel='sigmoid', gamma=1, coef0=1, C=1)
```

Building and evaluating an SVM classifier

The general approach to building an SVM classifier is similar to other supervised learning algorithms.

- Split the data into a training set and a test set; for example, 80% of the data as the training set and 20% as the test set.
- Import an SVM module from a library of your choosing; for example, scikit-learn.
- Use the training set to train the model; test the accuracy of its predictions using the test set.
- Tune hyperparameters using grid search and cross-validation methods; iterate through different kernels, regularisation (C) values, and gamma values to find the best combination.

Multiclass classification using SVM

In its basic form SVM is a binary classifier; it separates data into two classes (positive and negative).

SVM can be used for multiclass classification using techniques such as one-vs-all (or one vs-rest) or one-vs-one.

For example, suppose we wish to classify images of animals into four classes:

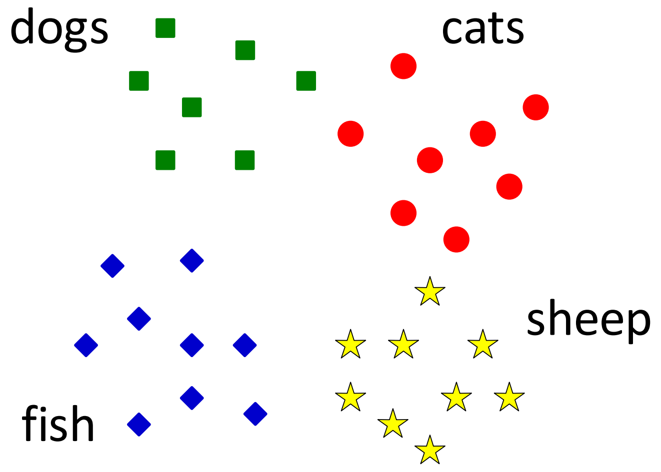
dogs, cats, sheep and fish.

There are two straightforward ways we can use SVM classifiers to achieve this.

Multiclass classification: one vs all

If there are n classes, the one-vs-all method uses n different SVM classifiers – each classifier separates one class from all other classes.

For example, in the (dogs, cats, sheep, fish) example, suppose the data has two attributes so can be represented in $\mathbb{R}^2$.

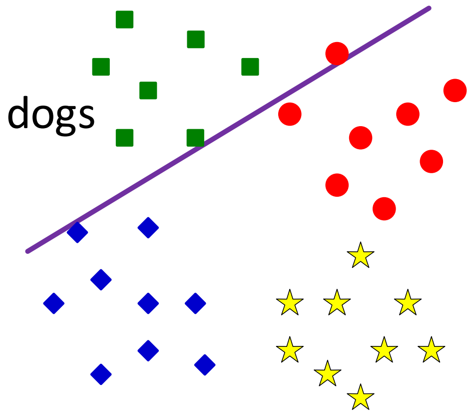


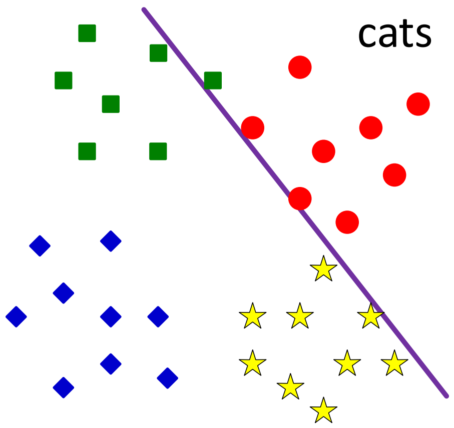
Multiclass classification: one vs all

Note: only the one separating fish is a hard-margin SVM.

We need four different SVM classifiers.

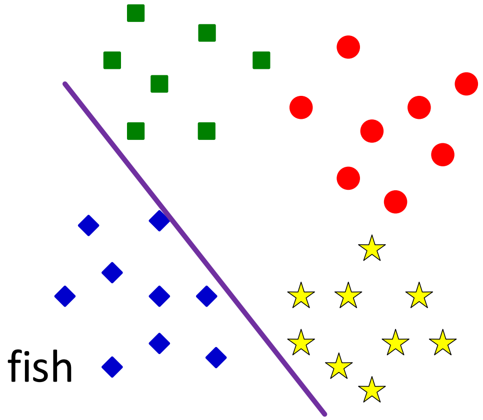
One separating dogs from the rest.

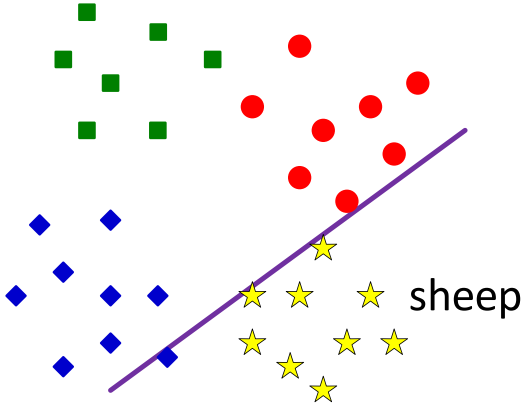




One separating cats from the rest.

One separating fish from the rest.





One separating sheep from the rest.

Multiclass classification: one vs all

The four classifiers are: dogs vs 'non-dogs', cats vs 'non-cats', sheep vs 'non-sheep and fish vs 'non-fish.

To classify a new image, input it into all 4 classifiers, and the classifier that outputs the highest score would be the final prediction for the class of the image.

Note that, in general, if there are n classes, this approach needs to train n different SVM classifiers.

Multiclass classification: one vs one

The second approach, called one-vs-one, trains SVM classifiers to separate each pair of classes.

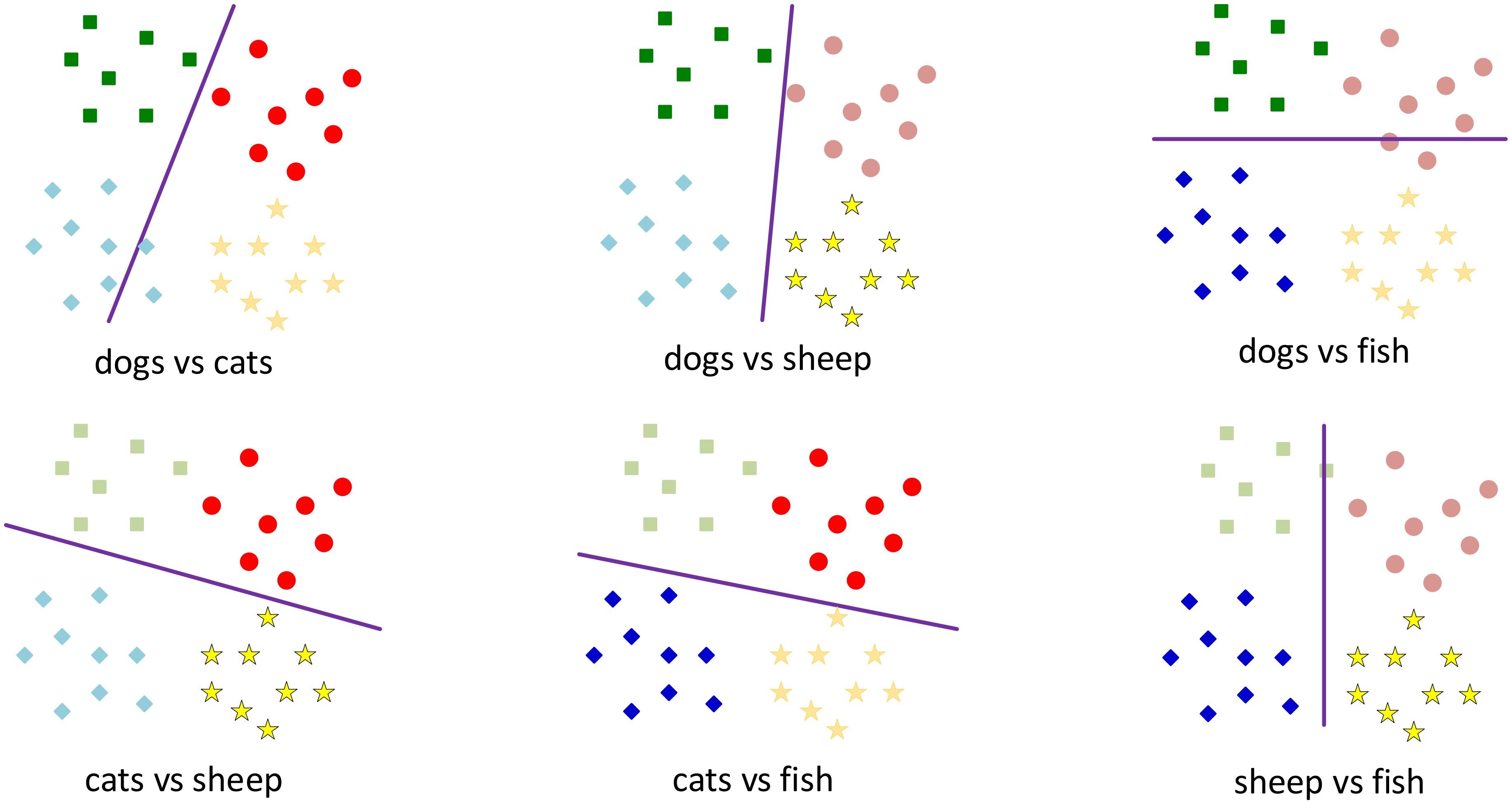
In our example, this requires six different SVM classifiers:

- dogs vs cats
- dogs vs sheep
- dogs vs fish
- cats vs sheep
- cats vs fish
- sheep vs fish.

Multiclass classification: one vs one

Note: all are hard-margin SVMs here.

The six different SVM classifiers:



Multiclass classification: one vs one

To classify a new image, input it into all 6 classifiers. The class predicted by most classifiers gives the final prediction for the class of the image.

Note that, in general for the one-vs-one approach, if there are n classes, this approach needs to train $\frac{1}{2}n(n-1)$ different SVM classifiers.

Compared with the one-vs-all approach, the one-vs-one approach

- requires training of more classifiers but
- may lead to a better performance overall.