CHAPTER 8

# Support Vector Machines

Back in Chapter 3 we looked at the Perceptron, a set of McCulloch and Pitts neurons arranged in a single layer. We identified a method by which we could modify the weights so that the network learned, and then saw that the Perceptron was rather limited in that it could only identify straight line classifiers, that is, it could only separate out groups of data if it was possible to draw a straight line (hyperplane in higher dimensions) between them. This meant that it could not learn to distinguish between the two truth classes of the 2D XOR function. However, in Section 3.4.3, we saw that it was possible to modify the problem so that the Perceptron could solve the problem, by changing the data so that it used more dimensions than the original data.

This chapter is concerned with a method that makes use of that insight, amongst other things. The main idea is one that we have seen before, in Section 5.3, which is to modify the data by changing its representation. However, the terminology is different here, and we will introduce kernel functions rather than bases. In principle, it is always possible to transform any set of data so that the classes within it can be separated linearly. To get a bit of a handle on this, think again about what we did with the XOR problem in Section 3.4.3: we added an extra dimension and moved a point that we could not classify properly into that additional dimension so that we could linearly separate the classes. The problem is how to work out which dimensions to use, and that is what kernel methods, which is the class of algorithms that we will talk about in this chapter, do.

We will focus on one particular algorithm, the Support Vector Machine (SVM) , which is one of the most popular algorithms in modern machine learning. It was introduced by Vapnik in 1992 and has taken off radically since then, principally because it often (but not always) provides very impressive classification performance on reasonably sized datasets. SVMs do not work well on extremely large datasets, since (as we shall see) the computations don't scale well with the number of training examples, and so become computationally very expensive. This should be sufficient motivation to master the (quite complex) concepts that are needed to understand the algorithm.

We will develop a simple SVM in this chapter, using `cvxopt`, a freely available solver with a Python interface, to do the heavy work. There are several different implementations of the SVM available on the Internet, and there are references to some of the more popular ones at the end of the chapter. Some of them include wrappers so that they can be used from within Python.

There is rather more to the SVM than the kernel method; the algorithm also reformulates the classification problem in such a way that we can tell a good classifier from a bad one, even if they both give the same results on a particular dataset. It is this distinction that enables the SVM algorithm to be derived, so that is where we will start.

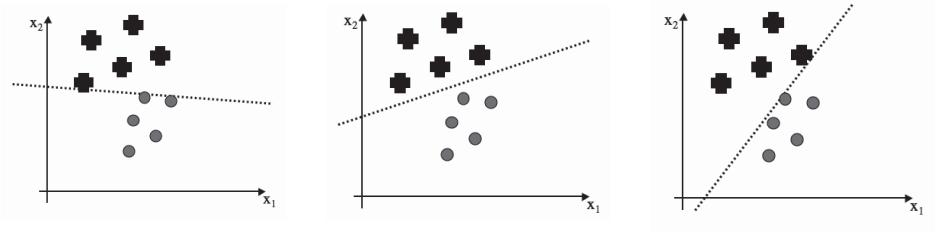


FIGURE 8.1 Three different classification lines. Is there any reason why one is better than the others?

## 8.1 OPTIMAL SEPARATION

Figure 8.1 shows a simple classification problem with three different possible linear classification lines. All three of the lines that are drawn separate out the two classes, so in some sense they are 'correct', and the Perceptron would stop its training if it reached any one of them. However, if you had to pick one of the lines to act as the classifier for a set of test data, I'm guessing that most of you would pick the line shown in the middle picture. It's probably hard to describe exactly why you would do this, but somehow we prefer a line that runs through the middle of the separation between the datapoints from the two classes, staying approximately equidistant from the data in both classes. Of course, if you were feeling smart, then you might have asked what criteria you were meant to pick a line based on, and why one of the lines should be any better than the others.

To answer that, we are going to try to define why the line that runs halfway between the two sets of datapoints is better, and then work out some way to quantify that so we can identify the 'optimal' line, that is, the best line according to our criteria. The data that we have used to identify the classification line is our training data. We believe that these data are indicative of some underlying process that we are trying to learn, and that the testing data that the algorithm will be evaluated on after training comes from the same underlying process. However, we don't expect to see exactly the same datapoints in the test dataset, and inevitably some of the points will be closer to the classifier line, and some will be further away. If we pick the lines shown in the left or right graphs of Figure 8.1, then there is a chance that a datapoint from one class will be on the wrong side of the line, just because we have put the line tight up against some of the datapoints we have seen in the training set. The line in the middle picture doesn't have this problem; like the baby bear's porridge in Goldilocks, it is 'just right'.

### 8.1.1 The Margin and Support Vectors

How can we quantify this? We can measure the distance that we have to travel away from the line (in a direction perpendicular to the line) before we hit a datapoint. Imagine that we put a 'no-man's land' around the line (shown in Figure 8.2), so that any point that lies within that region is declared to be too close to the line to be accurately classified. This region is symmetric about the line, so that it forms a cylinder about the line in 3D, and a hyper-cylinder in higher dimensions. How large could we make the radius of this cylinder until we started to put points into a no-man's land, where we don't know which class they are from? This largest radius is known as the margin, labelled $M$. The margin was mentioned briefly in Section 3.4.1, where it affected the speed at which the Perceptron converged. The classifier in the middle of Figure 8.1 has the largest margin of the three. It

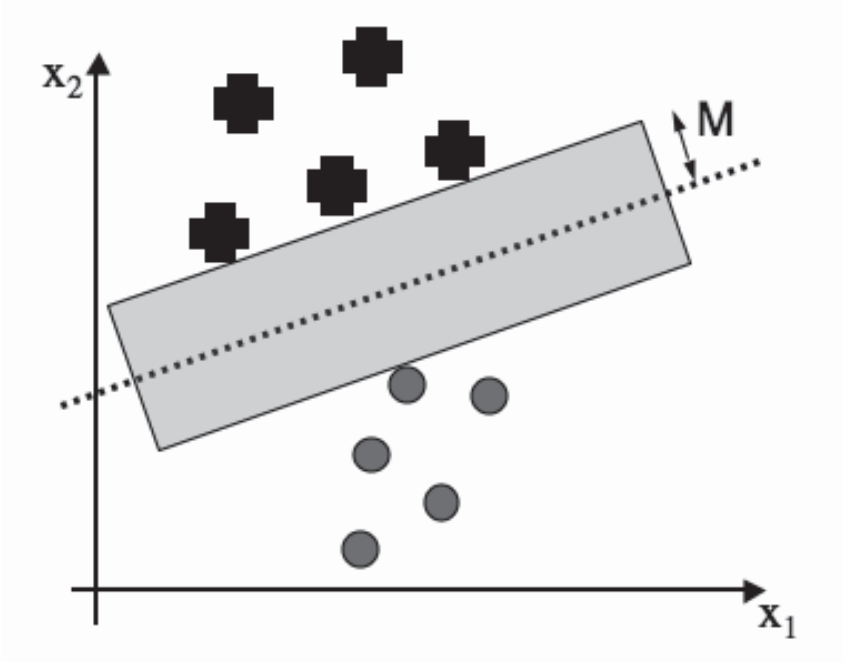


FIGURE 8.2 The margin is the largest region we can put that separates the classes without there being any points inside, where the box is made from two lines that are parallel to the decision boundary.

has the imaginative name of the maximum margin (linear) classifier. The datapoints in each class that lie closest to the classification line have a name as well. They are called support vectors. Using the argument that the best classifier is the one that goes through the middle of no-man's land, we can now make two arguments: first that the margin should be as large as possible, and second that the support vectors are the most useful datapoints because they are the ones that we might get wrong. This leads to an interesting feature of these algorithms: after training we can throw away all of the data except for the support vectors, and use them for classification, which is a useful saving in data storage.

Now that we've got a measurement that we can use to find the optimal decision boundary, we just need to work out how to actually compute it from a given set of datapoints. Let's start by reminding ourselves of some of the things that we worked out in Chapter 3. We have a weight vector (a vector, not a matrix, since there is only one output) and an input vector $\mathbf{x}$. The output we used in Chapter 3 was $y = \mathbf{w} \cdot \mathbf{x} + b$, with $b$ being the contribution from the bias weight. We use the classifier line by saying that any $\mathbf{x}$ value that gives a positive value for $\mathbf{w} \cdot \mathbf{x} + b$ is above the line, and so is an example of the '+' class, while any $\mathbf{x}$ that gives a negative value is in the '$\circ$' class. In our new version of this we want to include our no-man's land. So instead of just looking at whether the value of $\mathbf{w} \cdot \mathbf{x} + b$ is positive or negative, we also check whether the absolute value is less than our margin $M$, which would put it inside the grey box in Figure 8.2. Remember that $\mathbf{w} \cdot \mathbf{x}$ is the inner or scalar product, $\mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i$. This can also be written as $\mathbf{w}^T \mathbf{x}$, since this simply means that we treat the vectors as degenerate matrices and use the normal matrix multiplication rules. This notation will turn out to be simpler, and so will be used from here on.

For a given margin value $M$ we can say that any point $\mathbf{x}$ where $\mathbf{w}^T\mathbf{x} + b \geq M$ is a plus, and any point where $\mathbf{w}^T\mathbf{x} + b \leq -M$ is a circle. The actual separating hyperplane is specified by $\mathbf{w}^T\mathbf{x} + b = 0$. Now suppose that we pick a point $\mathbf{x}^+$ that lies on the '+' class boundary line, so that $\mathbf{w}^T\mathbf{x}^+ = M$. This is a support vector. If we want to find the closest point that lies on the boundary line for the '∘' class, then we travel perpendicular to the '+' boundary line until we hit the '∘' boundary line. The point that we hit is the closest point, and we'll call it $\mathbf{x}^-$. How far did we have to travel in this direction? Figure 8.2 hopefully makes it clear that the distance we travelled is $M$ to get to the separating hyperplane, and then $M$ from there to the opposing support vector. We can use this fact to write down the margin size $M$ in terms of $\mathbf{w}$ if we remember one extra thing from Chapter 3, namely that the weight vector $\mathbf{w}$ is perpendicular to the classifier line. If it is perpendicular to the classifier line, then it is obviously perpendicular to the '+' and '∘' boundary lines too, so the direction we travelled from $\mathbf{x}^+$ to $\mathbf{x}^-$ is along $\mathbf{w}$. Now we need to make $\mathbf{w}$ a unit vector $\mathbf{w}/\|w\|$, and so we see that the margin is $1/\|w\|$. In some texts the margin is actually written as the total distance between the support vectors, so that it would be twice the one that we have computed.

So now, given a classifier line (that is, the vector $\mathbf{w}$ and scalar $b$ that define the line $\mathbf{w}^T\mathbf{x} + b$) we can compute the margin $M$. We can also check that it puts all of the points on the right side of the classification line. Of course, that isn't actually what we want to do: we want to find the $\mathbf{w}$ and $b$ that give us the biggest possible value of $M$. Our knowledge that the width of the margin is $1/\|w\|$ tells us that making $M$ as large as possible is the same as making $\mathbf{w}^T\mathbf{w}$ as small as possible. If that was the only constraint, then we could just set $\mathbf{w} = \mathbf{0}$, and the problem would be solved, but we also want the classification line to separate out the '+' data from the '∘', that is, actually act as a classifier. So we are going to need to try to satisfy two problems simultaneously: find a decision boundary that classifies well, while also making $\mathbf{w}^T\mathbf{w}$ as small as possible. Mathematically, we can write these requirements as: minimise $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ (where the half is there for convenience as in so many other cases) subject to some constraint that says that the data are well matched. The next thing is to work out what these constraints are.

### 8.1.2 A Constrained Optimisation Problem

How do we decide whether or not a classifier is any good? Obviously, the fewer mistakes that it makes, the better. So we can write down a set of constraints that say that the classifier should get the answer right. To do this we make the target answers for our two classes be $\pm 1$, rather than 0 and 1. We can then write down $t_i \times y_i$, that is, the target multiplied by the output, and this will be positive if the two are the same and negative otherwise. We can write down the equation of the straight line again, which is how we computed $y$, to see that we require that $t_i(\mathbf{w}^T\mathbf{x} + b) \geq 1$. This means that the constraints just need to check each datapoint for this condition. So the full problem that we wish to solve is:

$$\text{minimise } \frac{1}{2}\mathbf{w}^T\mathbf{w} \text{ subject to } t_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \text{ for all } i = 1, \ldots n. \tag{8.1}$$

We've put in a lot of effort to write down this equation, but we don't know how to solve it. We could try and use gradient descent, but we would have to put a lot of effort into making it enforce the constraints, and it would be very, very slow and inefficient for the problem. There is a method that is much better suited, which is quadratic programming, which takes advantage of the fact that the problem we have described is quadratic and therefore convex, and has linear constraints. A convex problem is one where if we take any two points on the line and join them with a straight line, then every point on the line will

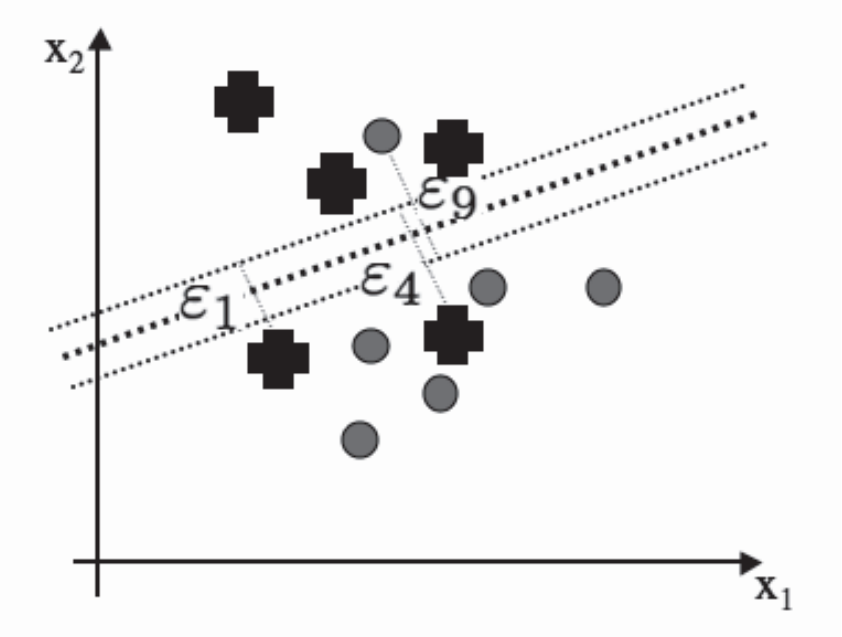


FIGURE 8.3 If the classifier makes some errors, then the distance by which the points are over the border should be used to weight each error in order to decide how bad the classifier is.

be above the curve. Figure 8.4 shows an example of a convex and a non-convex function. Convex functions have a unique minimum, which is fairly easy to see in one dimension, and remains true in any number of dimensions.

The practical upshot of these facts for us is that the types of problem that we are interested in can be solved directly and efficiently (i.e., in polynomial time). There are very effective quadratic programming solvers available, but it is not an algorithm that we will consider writing ourselves. We will, however, work out how to formulate the problem so that it can be presented to a quadratic program solver, and then use one of the programs that other people have been nice enough to prepare and make freely available.

Since the problem is quadratic, there is a unique optimum. When we find that optimal solution, the Karush–Kuhn–Tucker (KKT) conditions will be satisfied. These are (for all values of $i$ from 1 to $n$, and where the $*$ denotes the optimal value of each parameter):

$$\lambda_i^*(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)) = 0 \tag{8.2}$$

$$1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*) \leq 0 \tag{8.3}$$

$$\lambda_i^* \geq 0, \tag{8.4}$$

where the $\lambda_i$ are positive values known as Lagrange multipliers, which are a standard approach to solving equations with equality constraints.

The first of these conditions tells us that if $\lambda_i \neq 0$ then $(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)) = 0$. This is only true for the support vectors (the SVMs provide a sparse representation of the data), and so we only have to consider them, and can ignore the rest. In the jargon, the support vectors

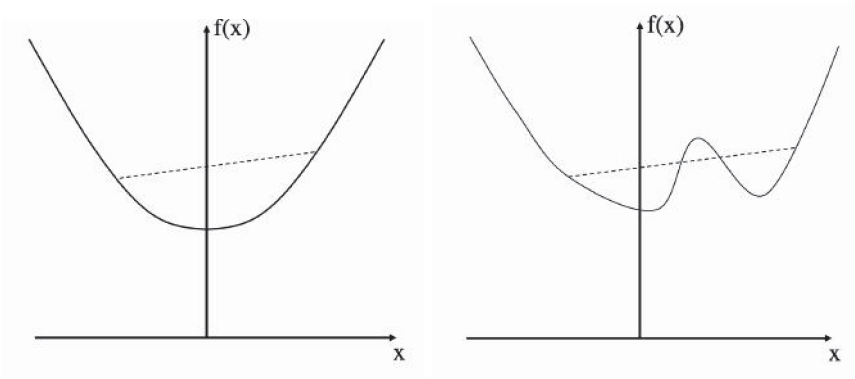


FIGURE 8.4 A function is convex if every straight line that links two points on the curve does not intersect the curve anywhere else. The function on the left is convex, but the one on the right is not, as the dashed line shows.

are those vectors in the active set of constraints. For the support vectors the constraints are equalities instead of inequalities. We can therefore solve the Lagrangian function:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{n} \lambda_i (1 - t_i(\mathbf{w}^T\mathbf{x}_i + b)), \tag{8.5}$$

We differentiate this function with respect to the elements of $\mathbf{w}$ and $b$:

$$\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w} - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i, \tag{8.6}$$

and

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \lambda_i t_i. \tag{8.7}$$

If we set the derivatives to be equal to zero, so that we find the saddle points (maxima) of the function, we see that:

$$\mathbf{w}^* = \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i, \sum_{i=1}^{n} \lambda_i t_i = 0. \tag{8.8}$$

We can substitute these expressions at the optimal values of $\mathbf{w}$ and $b$ into Equation (8.5) and, after a little bit of rearranging, we get (where $\boldsymbol{\lambda}$ is the vector of the $\lambda_i$):

$$\mathcal{L}(\mathbf{w}^*, b^*, \boldsymbol{\lambda}) = \sum_{i=1}^{n} \lambda_i - \sum_{i=1}^{n} \lambda_i t_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_i \lambda_j t_i t_j \mathbf{x}_i^T \mathbf{x}_j, \tag{8.9}$$

and we can notice that using the derivative with respect to $b$ we can treat the middle term as 0. This equation is known as the dual problem, and the aim is to maximise it with respect to the $\lambda_i$ variables. The constraints are that $\lambda_i \geq 0$ for all $i$, and $\sum_{i=1}^{n} \lambda_i t_i = 0$.

Equation (8.8) gives us an expression for $\mathbf{w}^*$, but we also want to know what $b^*$ is. We know that for a support vector $t_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$, and we can substitute the expression for

$\mathbf{w}^*$ into there and substitute in the $(\mathbf{x}, t)$ of one of the support vectors. However, in case of errors this is not very stable, and so it is better to average it over the whole set of $N_s$ support vectors:

$$b^* = \frac{1}{N_s} \sum_{\text{support vectors } j} \left( t_j - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i^T \mathbf{x}_j \right). \tag{8.10}$$

We can also use Equation (8.8) to see how to make a prediction, since for a new point $\mathbf{z}$:

$$\mathbf{w}^{*T}\mathbf{z} + b^* = \left( \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i \right)^T \mathbf{z} + b^*. \tag{8.11}$$

This means that to classify a new point, we just need to compute the inner product between the new datapoint and the support vectors.

### 8.1.3 Slack Variables for Non-Linearly Separable Problems

Everything that we have done so far has assumed that the datatset is linearly separable. We know that this is not always the case, but if we have a non-linearly separable dataset, then we cannot satisfy the constraints for all of the datapoints. The solution is to introduce some slack variables $\eta_i \geq 0$ so that the constraints become $t_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \eta_i$. For inputs that are correct, we set $\eta_i = 0$.

These slack variables are telling us that, when comparing classifiers, we should consider the case where one classifier makes a mistake by putting a point just on the wrong side of the line, and another puts the same point a long way onto the wrong side of the line. The first classifier is better than the second, because the mistake was not as serious, so we should include this information in our minimisation criterion. We can do this by modifying the problem. In fact, we have to do major surgery, since we want to add a term into the minimisation problem so that we will now minimise $\mathbf{w}^T\mathbf{w} + C\times$ (distance of misclassified points from the correct boundary line). Here, $C$ is a tradeoff parameter that decides how much weight to put onto each of the two criteria: small $C$ means we prize a large margin over a few errors, while large $C$ means the opposite. This transforms the problem into a soft-margin classifier, since we are allowing for a few mistakes. Writing this in a more mathematical way, the function that we want to minimise is:

$$L(\mathbf{w}, \boldsymbol{\epsilon}) = \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n} \epsilon_i. \tag{8.12}$$

The derivation of the dual problem that we worked out earlier still holds, except that $0 \leq \lambda_i \leq C$, and the support vectors are now those vectors with $\lambda_i > 0$. The KKT conditions are slightly different, too:

$$\lambda_i^*(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*) - \eta_i) = 0 \tag{8.13}$$

$$(C - \lambda_i^*)\eta_i = 0 \tag{8.14}$$

$$\sum_{i=1}^{n} \lambda_i^* t_i = 0. \tag{8.15}$$

The second condition tells us that, if $\lambda_i < C$, then $\eta_i = 0$, which means that these are

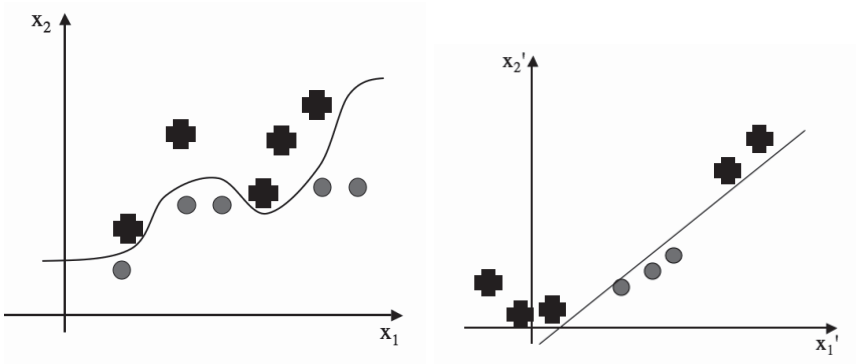


FIGURE 8.5 By modifying the features we hope to find spaces where the data are linearly separable.

the support vectors. If $\lambda_i = C$, then the first condition tells us that if $\eta_i > 1$ then the classifier made a mistake. The problem with this is that it is not as clear how to choose a limited set of vectors, and so most of our training set will be support vectors.

We have now built an optimal linear classifier. However, since most problems are non-linear we seem to have done a lot of work for a case that we could already solve, albeit not as effectively. So while the decision boundary that is found could be better than that found by the Perceptron, if there is not a straight line solution, then the method doesn't work much better than the Perceptron. Not ideal for something that's taken lots of effort to work out! It's time to pull our extra piece of magic out of the hat: transformation of the data.

## 8.2 KERNELS

To see the idea, have a look at Figure 8.5. Basically, we see that if we modify the features in some way, then we might be able to linearly separate the data, as we did for the XOR problem in Section 3.4.3; if we can use more dimensions, then we might be able to find a linear decision boundary that separates the classes. So all that we need to do is work out what extra dimensions we can use. We can't invent new data, so the new features will have to be derived from the current ones in some way. Just like in Section 5.3, we are going to introduce new functions $\phi(\mathbf{x})$ of our input features.

The important thing is that we are just transforming the data, so that we are making some function $\phi(\mathbf{x}_i)$ from input $\mathbf{x}_i$. The reason why this matters is that we want to be able to use the SVM algorithm that we worked out above, particularly Equation (8.11). The good news is that it isn't any worse, since we can replace $\mathbf{x}_i$ by $\phi(\mathbf{x}_i)$ (and $\mathbf{z}$ by $\phi(\mathbf{z})$) and get a prediction quite easily:

$$\mathbf{w}^T\mathbf{x} + b = \left(\sum_{i=1}^{n} \lambda_i t_i \phi(\mathbf{x}_i)\right)^T \phi(\mathbf{z}) + b. \tag{8.16}$$

We still need to pick what functions to use, of course. If we knew something about the data, then we might be able to identify functions that would be a good idea, but this kind of domain knowledge is not always going to be around, and we would like to automate the algorithm. For now, let's think about a basis that consists of the polynomials of everything up to degree 2. It contains the constant value 1, each of the individual (scalar)

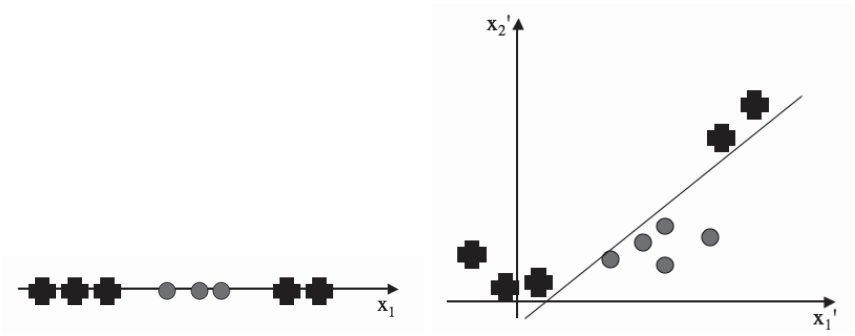


FIGURE 8.6 Using $x_1^2$ as well as $x_1$ allows these two classes to be separated.

input elements $x_1, x_2, \ldots, x_d$, and then the squares of each input element $x_1^2, x_2^2, \ldots, x_d^2$, and finally, the products of each pair of elements $x_1x_2, x_1x_3, \ldots, x_{d-1}x_d$. The total input vector made up of all these things is generally written as $\Phi(\mathbf{x})$; it contains about $d^2/2$ elements. The right of Figure 8.6 shows a 2D version of this (with the constant term suppressed), and I'm going to write it out for the case $d = 3$, with a set of $\sqrt{2}$s in there (the reasons for them will become clear soon):

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_3, x_1^2, x_2^2, x_3^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3). \tag{8.17}$$

If there was just one feature, $x_1$, then we would have changed this from a one-dimensional problem into a three-dimensional one $(1, x_1, x_1^2)$.

The only thing that this has cost us is computational time: the function $\Phi(\mathbf{x}_i)$ has $d^2/2$ elements, and we need to multiply it with another one the same size, and we need to do this many times. This is rather computationally expensive, and if we need to use the powers of the input vector greater than 2 it will be even worse. There is one last piece of trickery that will get us out of this hole: it turns out that we don't actually have to compute $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$. To see how this works, let's work out what $\Phi(\mathbf{x})^T\Phi(\mathbf{y})$ actually is for the example above (where $d = 3$ to match perfectly):

$$\Phi(\mathbf{x})^T\Phi(\mathbf{y}) = 1 + 2\sum_{i=1}^{d} x_iy_i + \sum_{i=1}^{d} x_i^2y_i^2 + 2\sum_{i,j=1;i<j}^{d} x_ix_jy_iy_j. \tag{8.18}$$

You might not recognise that you can factorise this equation, but fortunately somebody did: it can be written as $(1+\mathbf{x}^T\mathbf{y})^2$. The dot product here is in the original space, so it only requires $d$ multiplications, which is obviously much better—this part of the algorithm has now been reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(d)$. The same thing holds true for the polynomials of any degree $s$ that we are making here, where the cost of the naïve algorithm is $\mathcal{O}(d^s)$. The important thing is that we remove the problem of computing the dot products of all the extended basis vectors, which is expensive, with the computation of a kernel matrix (also known as the Gram matrix) $\mathbf{K}$ that is made from the dot product of the original vectors, which is only linear in cost. This is sometimes known as the kernel trick. It means that you don't even have to know what $\Phi(\cdot)$ is, provided you know a kernel. These kernels are the fundamental reason why these methods work, and the reason why we went to all that effort to produce the dual formulation of the problem. They produce a transformation of the data so that they are in a higher-dimensional space, but because the datapoints only

appear inside those inner products, we don't actually have to do any computations in those higher-dimensional spaces, only in the original (relatively cheap) low-dimensional space.

### 8.2.1 Choosing Kernels

So how do we go about finding a suitable kernel? Any symmetric function that is positive definite (meaning that it enforces positivity on the integral of arbitrary functions) can be used as a kernel. This is a result of Mercer's theorem, which also says that it is possible to convolve kernels together and the result will be another kernel. However, there are three different types of basis functions that are commonly used, and they have nice kernels that correspond to them:

- polynomials up to some degree $s$ in the elements $x_k$ of the input vector (e.g., $x_3^3$ or $x_1 \times x_4$) with kernel:
$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x}^T\mathbf{y})^s \tag{8.19}$$
For $s = 1$ this gives a linear kernel
- sigmoid functions of the $x_k$s with parameters $\kappa$ and $\delta$, and kernel:
$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \tanh(\kappa\mathbf{x}^T\mathbf{y} - \delta) \tag{8.20}$$
- radial basis function expansions of the $x_k$s with parameter $\sigma$ and kernel:
$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \exp\left(-(\mathbf{x} - \mathbf{y})^2/2\sigma^2\right) \tag{8.21}$$

Choosing which kernel to use and the parameters in these kernels is a tricky problem. While there is some theory based on something known as the Vapnik–Chernik dimension that can be applied, most people just experiment with different values and find one that works, using a validation set as we did for the MLP in Chapter 4.

There are two things that we still need to worry about for the algorithm. One is something that we've discussed in the context of other machine learning algorithms: overfitting, and the other is how we will do testing. The second one is probably worth a little explaining. We used the kernel trick in order to reduce the computations for the training set. We still need to work out how to do the same thing for the testing set, since otherwise we'll be stuck with doing the $\mathcal{O}(d^s)$ computations. In fact, it isn't too hard to get around this problem, because the forward computation for the weights is $\mathbf{w}^T\Phi(\mathbf{x})$, where:

$$\mathbf{w} = \sum_{i \text{ where } \lambda_i > 0} \lambda_i t_i \Phi(\mathbf{x}_i). \tag{8.22}$$

So we still have the computation of $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$, which we can replace using the kernel as before.

The overfitting problem goes away because of the fact that we are still optimising $\mathbf{w}^T\mathbf{w}$ (remember that from somewhere a long way back?), which tries to keep $\mathbf{w}$ small, which means that many of the parameters are kept close to 0.

### 8.2.2 Example: XOR

We motivated the SVM by thinking about how we solved the XOR function in Section 3.4.3. So will the SVM actually solve the problem? We'll need to modify the problem to have targets -1 and 1 rather than 0 and 1, but that is not difficult. Then we'll introduce a basis of all terms up to quadratic in our two features: $1, \sqrt{2}x_1, \sqrt{2}x_2, x_1x_2, x_1^2, x_2^2$, where the $\sqrt{2}$ is to keep the multiplications simple. Then Equation (8.9) looks like:

$$\sum_{i=1}^{4} \lambda_i - \sum_{i,j}^{4} \lambda_i \lambda_j t_i t_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j), \tag{8.23}$$

subject to the constraints that $\lambda_1 - \lambda_2 + \lambda_3 - \lambda_4 = 0, \lambda_i \geq 0 \; i = 1 \ldots 4$. Solving this (which can be done algebraically) tells us that the classifier line is at $x_1x_2 = 0$. The margin that corresponds to this is $\sqrt{2}$. Unfortunately we can't plot it, since our four points have been transferred into a six-dimensional space. We know that this is not the smallest number that it can be solved in, since we did it in three dimensions in Section 3.4.3, but the dimensionality of the kernel space doesn't matter, as all the computations are in the 2D space anyway.

## 8.3 THE SUPPORT VECTOR MACHINE ALGORITHM

Quadratic programming solvers tend to be very complex (lots of the work is in identifying the active set), and we would be a long way off topic if we tried to write one. Fortunately, general purpose solvers have been written, and so we can take advantage of this. We will use cvxopt, which is a convex optimisation package that includes a wrapper for Python. There is a link to the relevant website on the book webpage. Cvxopt has a nice and clean interface so we can use this to do the computational heavy lifting for an implementation of the SVM. In essence, the approach is fairly simple: we choose a kernel and then for given data, assemble the relevant quadratic problem and its constraints as matrices, and then pass them to the solver, which finds the decision boundary and necessary support vectors for us. These are then used to build a classifier for that training data. This is given as an algorithm next, and then some parts of the implementation are highlighted, particularly those parts where some speed-up can be achieved by some linear algebra.

**The Support Vector Machine Algorithm**

- **Initialisation**
  - for the specified kernel, and kernel parameters, compute the kernel of distances between the datapoints
    - the main work here is the computation $\mathbf{K} = \mathbf{X}\mathbf{X}^T$
    - for the linear kernel, return $\mathbf{K}$, for the polynomial of degree $d$ return $\frac{1}{\sigma}\mathbf{K}^d$
    - for the RBF kernel, compute $\mathbf{K} = \exp(-(\mathbf{x} - \mathbf{x}')^2/2\sigma^2)$
- **Training**
  - assemble the constraint set as matrices to solve:

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T t_i t_j \mathbf{K}\mathbf{x} + \mathbf{q}^T\mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b$$

  - pass these matrices to the solver

  - identify the support vectors as those that are within some specified distance of the closest point and dispose of the rest of the training data
  - compute $b^*$ using equation (8.10)

- **Classification**
  - for the given test data $\mathbf{z}$, use the support vectors to classify the data for the relevant kernel using:
    * compute the inner product of the test data and the support vectors
    * perform the classification as $\sum_{i=1}^{n} \lambda_i t_i \mathbf{K}(\mathbf{x}_i, \mathbf{z}) + b^*$, returning either the label (hard classification) or the value (soft classification)

---

### 8.3.1 Implementation

In order to use the code on the website it is necessary to install the cvxopt package on your computer. There is a link to this on the website. However, we need to work out exactly what we are trying to solve. The key is Equation (8.9), which shows the dual problem, which had constraints $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i t_i = 0$. We need to modify it so that we are dealing with the case for slack variables, and using a kernel. Introducing slack variables changes this surprisingly little, basically swapping the first constraint to be $0 \leq \lambda_i \leq C$, while adding the kernel simply turns $\mathbf{x}_i^T \mathbf{x}_j$ into $\mathbf{K}(\mathbf{x}_i, \mathbf{x}_j)$. So we want to solve:

$$\max_{\boldsymbol{\lambda}} \quad = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \boldsymbol{\lambda}^T \boldsymbol{\lambda} \mathbf{t} \mathbf{t}^T \mathbf{K}(\mathbf{x}_i, \mathbf{x}_j) \boldsymbol{\lambda}, \tag{8.24}$$

$$\text{subject to} \quad 0 \leq \lambda_i \leq C, \sum_{i=1}^{n} \lambda_i t_i = 0. \tag{8.25}$$

The cvxopt quadratic program solver is `cvxopt.solvers.qp()`. This method takes the following inputs `cvxopt.solvers.qp(P, q, G, h, A, b)` and then solves:

$$\min \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b, \tag{8.26}$$

where $\mathbf{x}$ is the variable we are solving for, which is $\boldsymbol{\lambda}$ for us. Note that this solves minimisation problems, whereas we are doing maximisation, which means that we need to multiply the objective function by -1. To make the equations match we set $\mathbf{P} = t_i t_j \mathbf{K}$ and $\mathbf{q}$ is just a column vector containing $-1$s. The second constraint is easy, since if $\mathbf{A} = \boldsymbol{\lambda}$ then we get the right equation. However, for the first constraint we need to do a little bit more work, since we want to include two constraints ($0 \leq \lambda_i$ and $\lambda_i \leq C$). To do this, we double up on the number of constraints, multiplying the ones where we want $\geq$ instead of $\leq$ by -1. In order to do this multiplication efficiently, it will also be better to use a matrix with the elements on the diagonal, so that we make the following matrix:

$$\begin{pmatrix} t_1 & 0 & \dots & 0 \\ 0 & t_2 & \dots & 0 \\ & \dots & & \\ 0 & 0 & \dots & t_n \\ -t_1 & 0 & \dots & 0 \\ 0 & -t_2 & \dots & 0 \\ & \dots & & \\ 0 & 0 & \dots & -t_n \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} C \\ C \\ \dots \\ C \\ 0 \\ 0 \\ \dots \\ 0 \end{pmatrix} \tag{8.27}$$

Assembling these, turning them into the matrices expected by the solver, and then calling it can then be written as:

```
# Assemble the matrices for the constraints
P = targets*targets.transpose()*self.K
q = -np.ones((self.N,1))
if self.C is None:
   G = -np.eye(self.N)
   h = np.zeros((self.N,1))
else:
   G = np.concatenate((np.eye(self.N),-np.eye(self.N)))
   h = np.concatenate((self.C*np.ones((self.N,1)),np.zeros((self.N,1))))
A = targets.reshape(1,self.N)
b = 0.0

# Call the quadratic solver
sol = cvxopt.solvers.qp(cvxopt.matrix(P),cvxopt.matrix(q),cvxopt.matrix(G),
cvxopt.matrix(h), cvxopt.matrix(A), cvxopt.matrix(b))
```

There are a couple of novelties in the implementation. One is that the training method actually returns a function that performs the classification, as can be seen here for the polynomial kernel:

```
if self.kernel == 'poly':
    def classifier(Y,soft=False):
        K = (1. + 1./self.sigma*np.dot(Y,self.X.T))**self.degree

        self.y = np.zeros((np.shape(Y)[0],1))
        for j in range(np.shape(Y)[0]):
            for i in range(self.nsupport):
                self.y[j] += self.lambdas[i]*self.targets[i]*K[j,i]
            self.y[j] += self.b

        if soft:
            return self.y
        else:
            return np.sign(self.y)
```

The reason for this is that the classification function has different forms for the different kernels, and so we need to create this function based on the kernel that is specified. A handle for the classifier is stored in the class, and the method can then be called as:

```
output = sv.classifier(Y,soft=False)
```

The other novelty is that some of the computation of the RBF kernel uses some linear algebra to make the computation faster, since NumPy is better at dealing with matrix manipulations than loops. The elements of the RBF kernel are $K_{ij} = \frac{1}{2\sigma} \exp(-\|x_i - x_j\|^2)$. We could go about forming this by using a pair of loops over $i$ and $j$, but instead we can use some algebra.

The linear kernel has computed $K_{ij} = \mathbf{x}_i^T \mathbf{x}_j$, and the diagonal elements of this matrix are $\|\mathbf{x}_i\|^2$. The trick is to see how to use only these elements to compute the $\|\mathbf{x}_i - \mathbf{x}_j\|^2$ part, and it just requires expanding out the quadratic:

$$(\mathbf{x}_i - \mathbf{x}_j)^2 = \|\mathbf{x}_i\|^2 + \|\mathbf{x}_j^2\| - 2\mathbf{x}_i^T \mathbf{x}_j. \tag{8.28}$$

The only work involved now is to make sure that the matrices are the right shape. This would be easy if it wasn't for the fact that NumPy 'loses' the dimension of some $N \times 1$ matrices, so that they are of size $N$ only, as we have seen before. This means that we need to make a matrix of ones and use the transpose operator a few times, as can be seen in the code fragment below.

```
self.xsquared = (np.diag(self.K)*np.ones((1,self.N))).T
b = np.ones((self.N,1))
self.K -= 0.5*(np.dot(self.xsquared,b.T) + np.dot(b,self.xsquared.T))
self.K = np.exp(self.K/(2.*self.sigma**2))
```

For the classifier we can use the same tricks to compute the product of the kernel and the test data:

```
elif self.kernel == 'rbf':
    def classifier(Y,soft=False):
        K = np.dot(Y,self.X.T)
        c = (1./self.sigma * np.sum(Y**2,axis=1)*np.ones((1,np.shape(Y)[0])))
        .T
        c = np.dot(c,np.ones((1,np.shape(K)[1])))
        aa = np.dot(self.xsquared[self.sv],np.ones((1,np.shape(K)[0]))).T
        K = K - 0.5*c - 0.5*aa
        K = np.exp(K/(2.*self.sigma**2))

        self.y = np.zeros((np.shape(Y)[0],1))
        for j in range(np.shape(Y)[0]):
            for i in range(self.nsupport):
                self.y[j] += self.lambdas[i]*self.targets[i]*K[j,i]
```

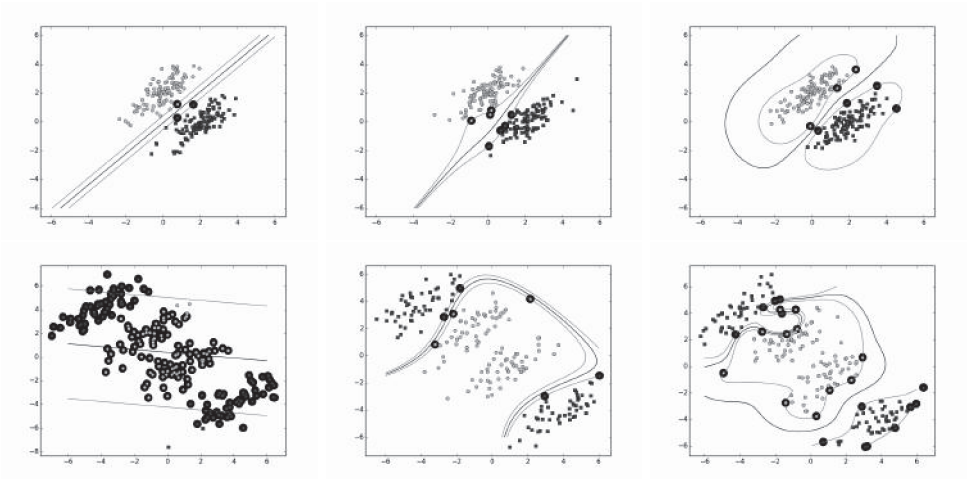

FIGURE 8.7 The SVM learning about a linearly separable dataset (*top row*) and a dataset that needs two straight lines to separate in 2D (*bottom row*) with *left* the linear kernel, *middle* the polynomial kernel of degree 3, and *right* the RBF kernel. $C = 0.1$ in all cases.

```
        self.y[j] += self.b

    if soft:
        return self.y
    else:
        return np.sign(self.y)
```

The first bit of computational work is in computing the kernel (which is $\mathcal{O}(m^2n)$, where $m$ is the number of datapoints and $n$ is the dimensionality), and the second part is inside the solver, which has to factorise a sum of the kernel matrix and a test matrix at each iteration. Factorisation costs $\mathcal{O}(m^3)$ in general, and this is why the SVM is very expensive to use for large datasets. There are some methods by which this can be improved, and there are some references to this at the end of the chapter.

### 8.3.2 Examples

In order to see the SVM working, and to identify the differences between the kernels, we will start with some very simple 2D datasets with two classes.

The first example (shown on the top row of Figure 8.7) simply checks that the SVM can learn accurately about data that is linearly separable, which it does successfully. Note that the different kernels produce different decision boundaries, which are not straight lines in the 2D plot for the polynomial kernel (centre) and RBF kernel (right), and that different numbers of support vectors (highlighted in bold) are needed for the different kernels as well.

On the second line of the figure is a dataset that cannot be separated by a single straight line, and which the linear kernel cannot then separate. However, the polynomial and RBF kernels deal with this data successfully with very few support vectors.

For the second example the data come from the XOR dataset with some spread around each of the four datapoints. The dataset is made by making four sets of random Gaussian samples with a small standard deviation, and means of $(0, 0), (0, 1), (1, 0)$, and $(1, 1)$. Figure 8.8 shows a series of outputs from this dataset with the standard deviations of each cluster being 0.1 on the left, 0.3 in the middle, and 0.4 on the right, and with 100 datapoints for training, and 100 datapoints for testing. The training set for the two classes is shown as black and white circles, with the support vectors marked with a thicker outline. The test set are shown as black and white squares.

The top row of the figure shows the polynomial kernel of degree 3 with no slack variables, while the second row shows the same kernel but with $C = 0.1$; the third row shows the RBF kernel with no slack variables, and the last row shows the RBF kernel with $C = 0.1$. It can be seen that where the classes start to overlap, the inclusion of slack variables leads to far simpler decision boundaries and a better model of the underlying data. Both the polynomial and RBF kernels perform well on this problem.

## 8.4 EXTENSIONS TO THE SVM

### 8.4.1 Multi-Class Classification

We've talked about SVMs in terms of two-class classification. You might be wondering how to use them for more classes, since we can't use the same methods as we have done to work out the current algorithm. In fact, you can't actually do it in a consistent way. The SVM only works for two classes. This might seem like a major problem, but with a little thought it is possible to find ways around the problem. For the problem of $N$-class classification, you train an SVM that learns to classify class one from all other classes, then another that classifies class two from all the others. So for $N$-classes, we have $N$ SVMs. This still leaves one problem: how do we decide which of these SVMs is the one that recognises the particular input? The answer is just to choose the one that makes the strongest prediction, that is, the one where the basis vector input point is the furthest into the positive class region. It might not be clear how to work out which is the strongest prediction. The classifier examples in the code snippets return either the class label (as the sign of $y$) or the value of $y$, and this value of $y$ is telling us how far away from the decision boundary it is, and clearly it will be negative if it is a misclassification. We can therefore use the maximum value of this soft boundary as the best classifier.

```
output = np.zeros((np.shape(test)[0],3))
output[:,0] = svm0.classifier(test[:,:2],soft=True).T
output[:,1] = svm1.classifier(test[:,:2],soft=True).T
output[:,2] = svm2.classifier(test[:,:2],soft=True).T

# Make a decision about which class
# Pick the one with the largest margin
bestclass = np.argmax(output,axis=1)
err = np.where(bestclass!=target)[0]
print len(err)/ np.shape(target)[0]
```

Figure 8.9 shows the first two dimensions of the iris dataset and the class decision boundaries for the three classes. It can be seen that using only two dimensions does not

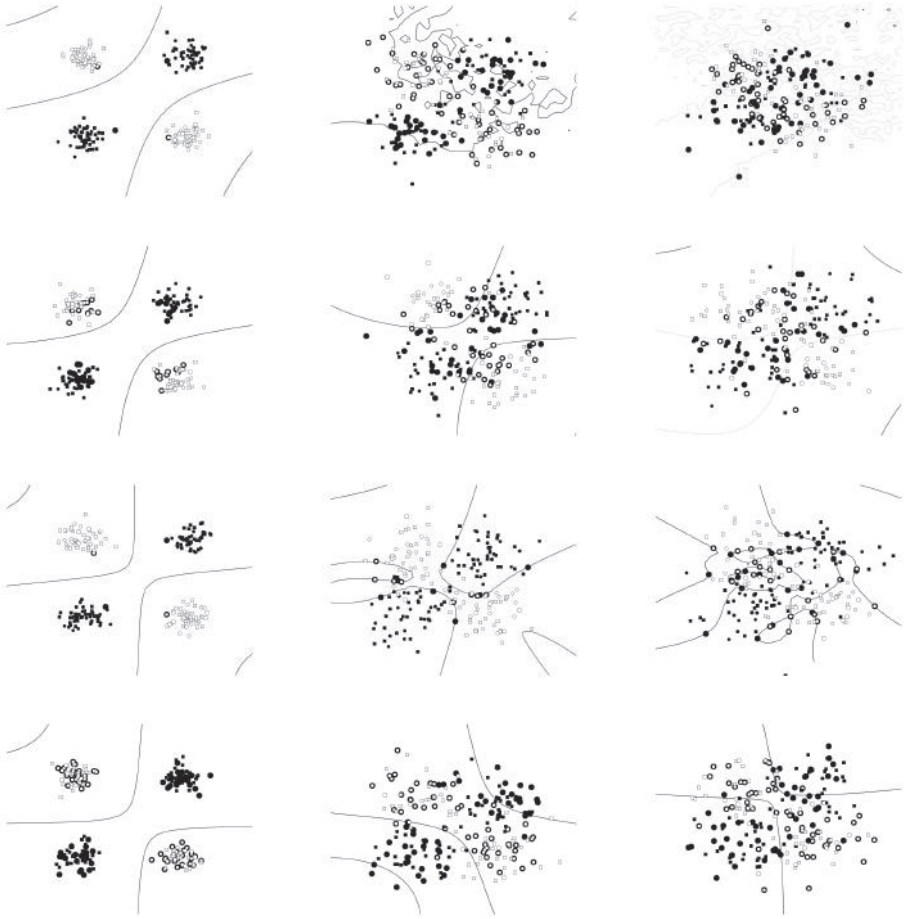

FIGURE 8.8 The effects of different kernels when learning a version of XOR with progressively more overlap (*left* to *right*) between the classes. *Top row:* polynomial kernel of degree 3 with no slack variables, *second row:* polynomial of degree 3 with $C = 0.1$, *third row:* RBF kernel, no slack variables, *bottom row:* RBF kernel with $C = 0.1$. The support vectors are highlighted, and the decision boundary is drawn for each case.

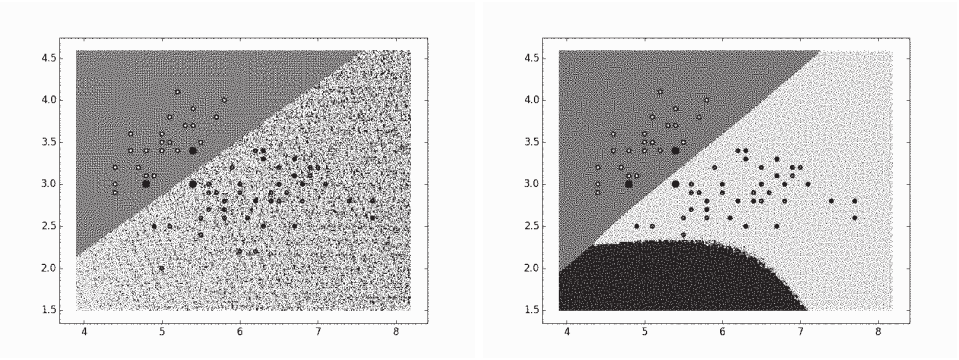


FIGURE 8.9 A linear (*left*) and polynomial, degree 3 (*right*) kernel learning the first two dimensions of the iris dataset, which separates one class very well from the other two, but cannot distinguish between the other two (for good reason). The support vectors are highlighted.

allow good separation of the data, and both kernels get about 33% accuracy, but allowing for all four dimensions, both the RBF and polynomial kernels reliably get about 95% accuracy.

### 8.4.2 SVM Regression

Perhaps rather surprisingly, it is also possible to use the SVM for regression. The key is to take the usual least-squares error function (with the regulariser that keeps the norm of the weights small):

$$\frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2 + \frac{1}{2}\boldsymbol{\lambda}\|\mathbf{w}\|^2, \tag{8.29}$$

and transform it using what is known as an $\epsilon$-insensitive error function ($E_\epsilon$) that gives 0 if the difference between the target and output is less than $\epsilon$ (and subtracts $\epsilon$ in any other case for consistency). The reason for this is that we still want a small number of support vectors, so we are only interested in the points that are not well predicted. Figure 8.10 shows the form of this error function, which is:

$$\sum_{i=1}^{N} E_\epsilon(t_i - y_i) + \boldsymbol{\lambda}\frac{1}{2}\|\mathbf{w}\|^2. \tag{8.30}$$

You might see this written in other texts with the constant $\boldsymbol{\lambda}$ in front of the second term replaced by a $C$ in front of the first term. This is equivalent up to scaling. The picture to think of now is almost the opposite of Figure 8.3: we want the predictions to be inside the tube of radius $\epsilon$ that surrounds the correct line. To allow for errors, we again introduce slack variables for each datapoint ($\epsilon_i$ for datapoint $i$) with their constraints and follow the same procedure of introducing Lagrange multipliers, transferring to the dual problem, using a kernel function and solving the problem with a quadratic solver.

The upshot of all this is that the prediction we make for test point $\mathbf{z}$ is:

$$f(\mathbf{z}) = \sum_{i=1}^{n}(\mu_i - \lambda_i K(\mathbf{x}_i, \mathbf{z}) + b), \tag{8.31}$$

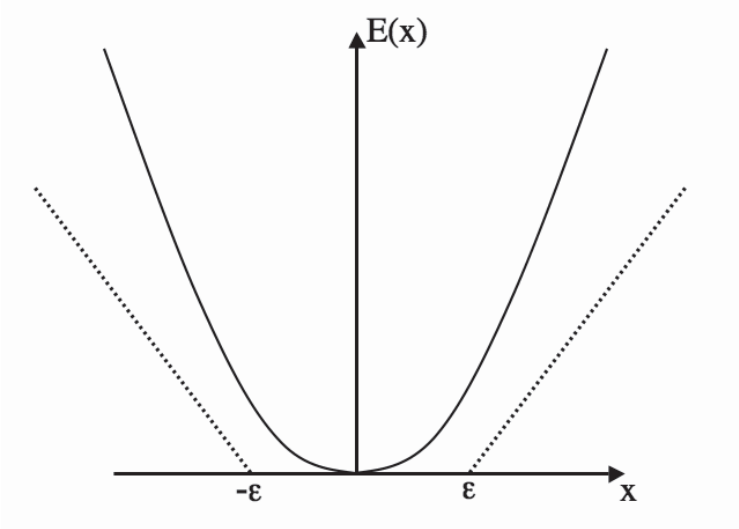


FIGURE 8.10 The $\epsilon$-insensitive error function is zero for any error below $\epsilon$.

where $\mu_i$ and $\lambda_i$ are two sets of constraint variables.

### 8.4.3 Other Advances

There is a lot of advanced work on kernel methods and SVMs. This includes lots of work on the optimisation, including Sequential Minimal Optimisation, and extensions to compute posterior probabilities instead of hard decisions, such as the Relevance Vector Machine. There are some references in the Further Reading section.

There are several SVM implementations available via the Internet that are more advanced than the implementation on the book website. They are mostly written in C, but some include wrappers to be called from other languages, including Python. An Internet search will find you some possibilities to try, but some common choices are SVMLight, LIBSVM, and scikit-learn.

## FURTHER READING

The treatment of SVMs here has only skimmed the surface of the topic. There is a useful tutorial paper on SVMs at:

- C.J. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

If you want more information, then any of the following books will provide it (the first is by the creator of SVMs):

- V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, Berlin, Germany, 1995.
- B. Schölkopf, C.J.C. Burges, and A.J. Smola. *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge, MA, USA, 1999.

- J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.

If you want to know more about quadratic programming, then a good reference is:

- S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.

Other machine learning books that give useful coverage of this area are:

- Chapter 12 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.
- Chapter 7 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.

## PRACTICE QUESTIONS

**Problem 8.1** Suppose that the following are a set of points in two classes:

$$\text{class 1} \quad : \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{8.32}$$

$$\text{class 2} \quad : \quad \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{8.33}$$

Plot them and find the optimal separating line. What are the support vectors, and what is the margin?

**Problem 8.2** Suppose that the points are now:

$$\text{class 1} \quad : \quad \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{8.34}$$

$$\text{class 2} \quad : \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{8.35}$$

Try out the different basis functions that were given in the chapter to see which separate this data and which do not.

**Problem 8.3** Apply it to the `wine` dataset, trying out the different kernels. Compare the results to using an MLP. Do the same for the `yeast` dataset.

**Problem 8.4** Use an SVM on the MNIST dataset.

**Problem 8.5** Verify that introducing the slack variables does not change the dual problem much at all (only changing the constraint to be $0 \leq \lambda_i \leq C$). Start from Equation (8.12) and introduce the Lagrange multipliers and then compare the result to Equations (8.9).