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.

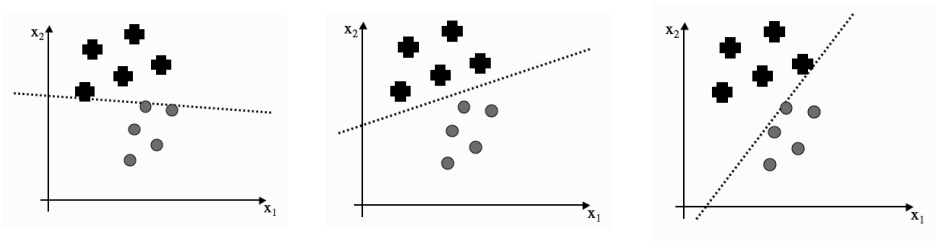

FIGURE 8.1 Three different classification lines. Is there any reason why one is better than the others?

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

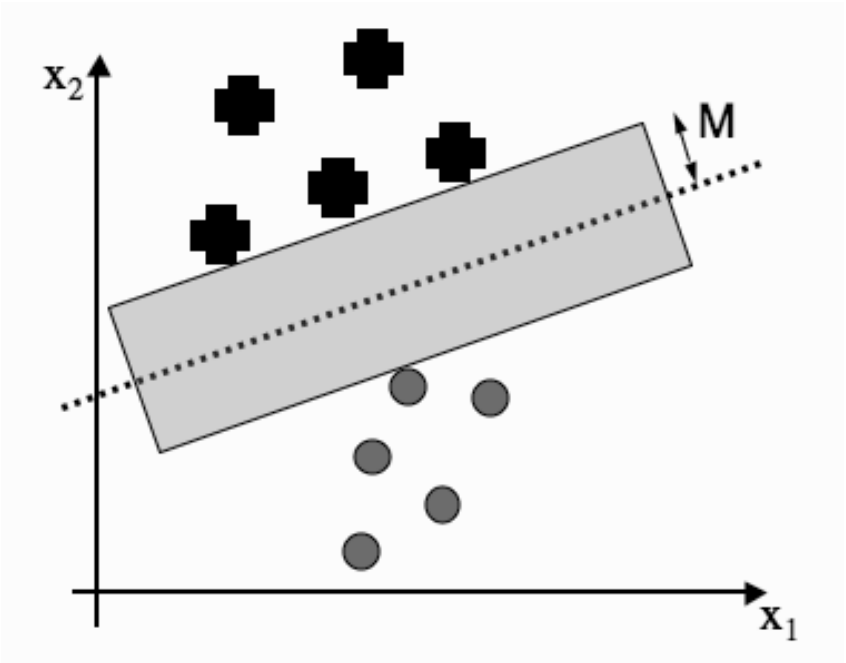

FIGURE 8.2 The margin is the largest region we can put that separates the classes without there being any points inside, where the box is made from two lines that are parallel to the decision boundary.

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 '○' 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 '$\circ$' class, then we travel perpendicular to the '+' boundary line until we hit the '$\circ$' 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 '$\circ$' 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 '$\circ$', 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 ± 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

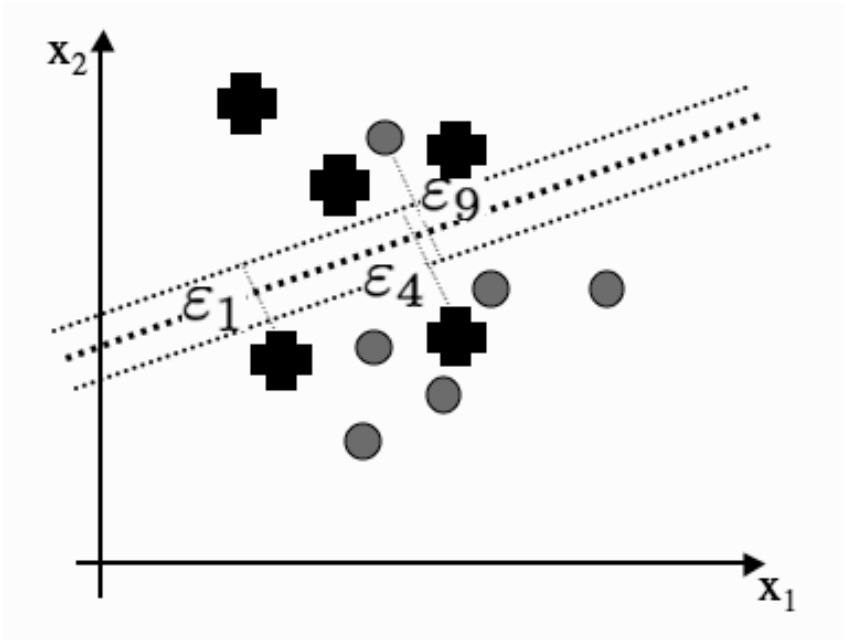

FIGURE 8.3 If the classifier makes some errors, then the distance by which the points are over the border should be used to weight each error in order to decide how bad the classifier is.

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 λ_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

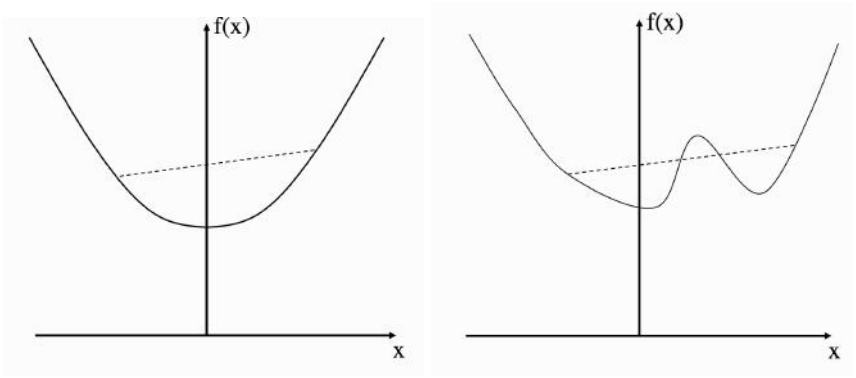

FIGURE 8.4 A function is convex if every straight line that links two points on the curve does not intersect the curve anywhere else. The function on the left is convex, but the one on the right is not, as the dashed line shows.

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 λ_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 λ_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

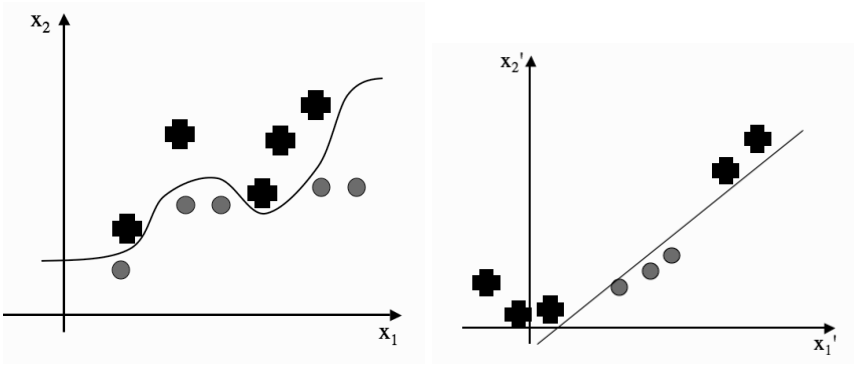

FIGURE 8.5 By modifying the features we hope to find spaces where the data are linearly separable.

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)

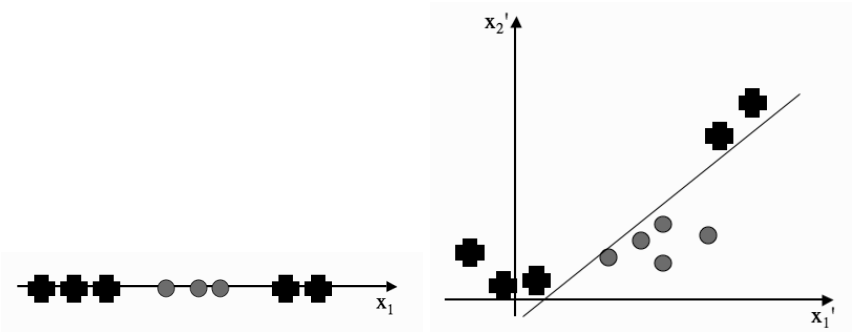

FIGURE 8.6 Using x_1^2 as well as x_1 allows these two classes to be separated.

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_ks with parameters κ and δ, 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_ks with parameter σ 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 -1s. 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]
```

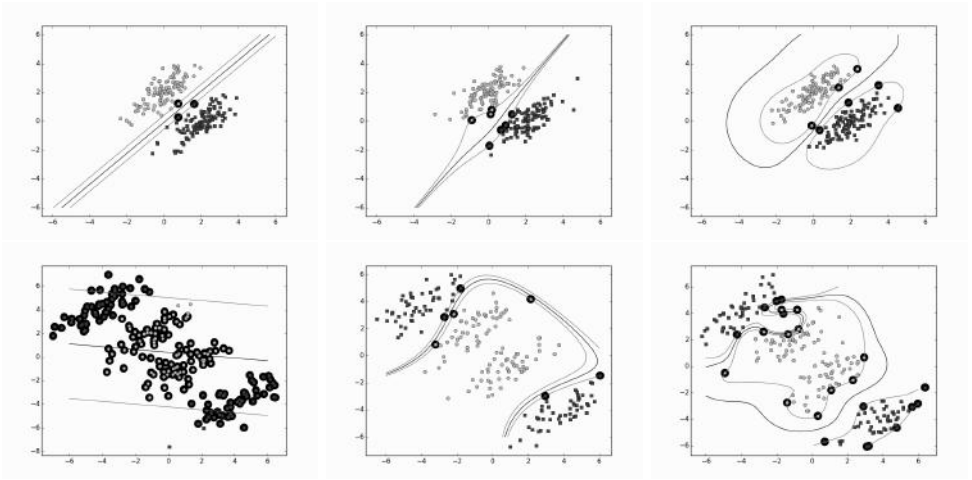

FIGURE 8.7 The SVM learning about a linearly separable dataset (*top row*) and a dataset that needs two straight lines to separate in 2D (*bottom row*) with *left* the linear kernel, *middle* the polynomial kernel of degree 3, and *right* the RBF kernel. $C = 0.1$ in all cases.

```
            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 $\mathcal{O}(m^2 n)$, 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

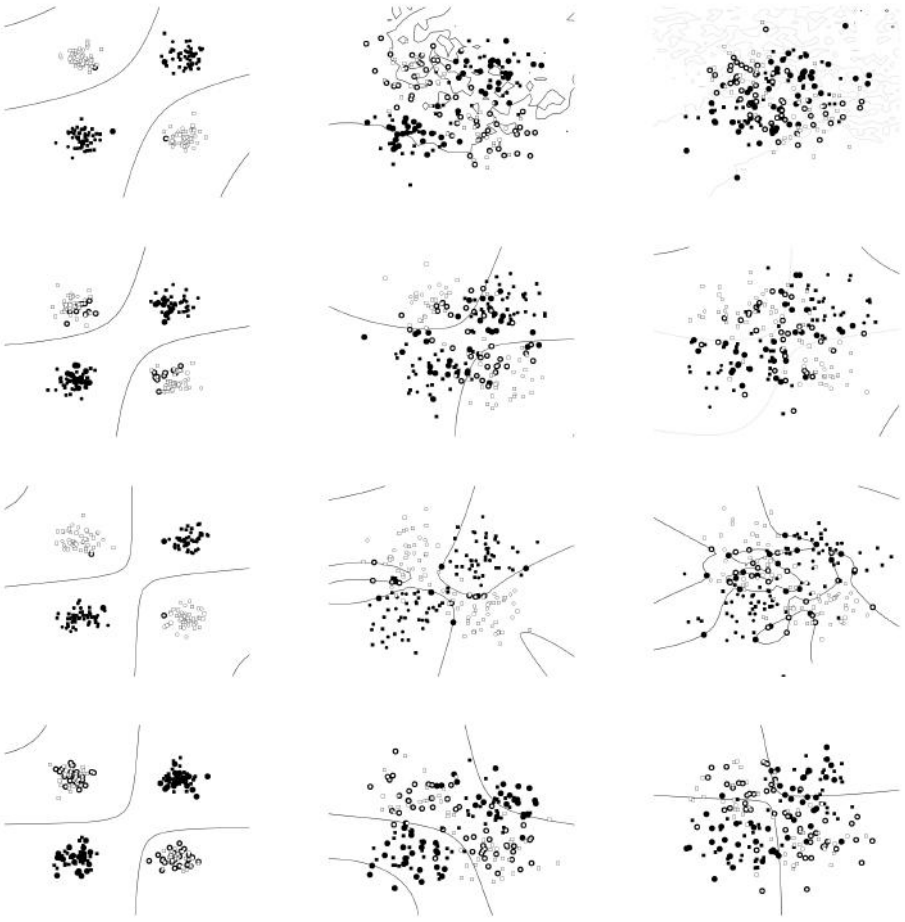

FIGURE 8.8 The effects of different kernels when learning a version of XOR with progressively more overlap (*left* to *right*) between the classes. *Top row:* polynomial kernel of degree 3 with no slack variables, *second row:* polynomial of degree 3 with $C = 0.1$, *third row:* RBF kernel, no slack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

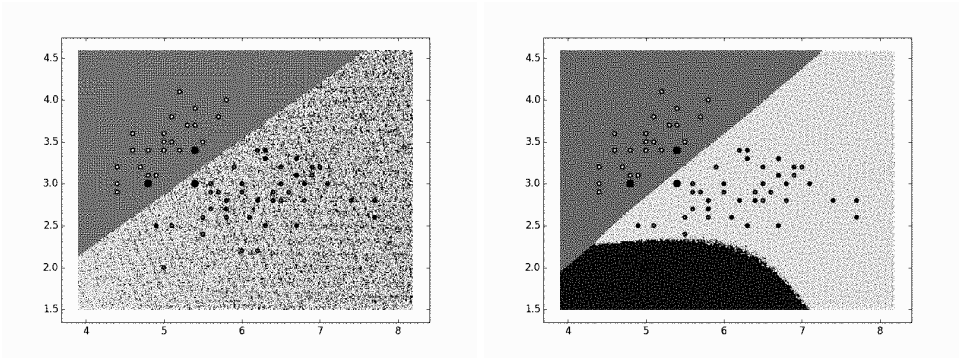

FIGURE 8.9 A linear (*left*) and polynomial, degree 3 (*right*) kernel learning the first two dimensions of the iris dataset, which separates one class very well from the other two, but cannot distinguish between the other two (for good reason). The support vectors are highlighted.

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 ϵ-insensitive error function (E_ϵ) that gives 0 if the difference between the target and output is less than ϵ (and subtracts ϵ 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 ϵ that surrounds the correct line. To allow for errors, we again introduce slack variables for each datapoint (ϵ_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:

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

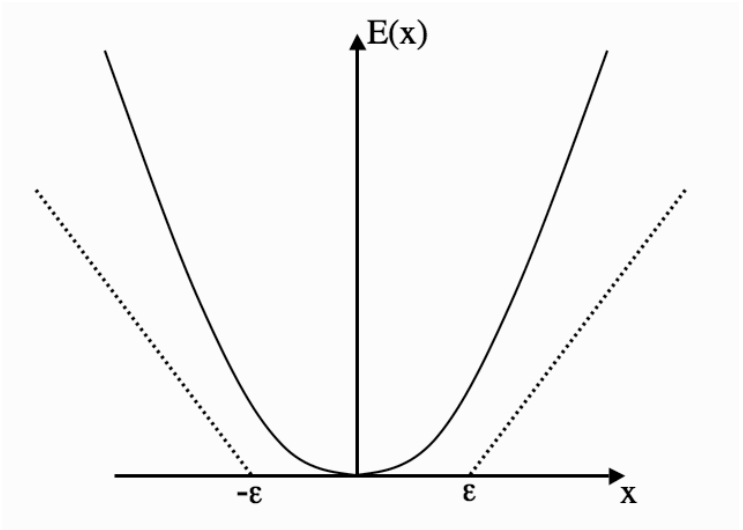

FIGURE 8.10 The ϵ-insensitive error function is zero for any error below ϵ.

where μ_i and λ_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).

CHAPTER 9

Optimisation and Search

In almost all of the algorithms that we've looked at in the previous chapters there has been some element of optimisation, generally by defining some sort of error function, and attempting to minimise it. We've talked about gradient descent, which is the optimisation method that forms the basis of many machine learning algorithms. In this chapter, we will look at formalising the gradient descent algorithm and understanding how it works, and then we will look at what we can do when there are no gradients in the problem, and so gradient descent doesn't work.

Whatever method we have used to solve the optimisation problem, the basic methodology has been the same: to compute the derivative of the error function to get the gradient and follow it downhill. What if that derivative doesn't exist? This is actually common in many problems—discrete problems are not defined on continuous functions, and hence can't be differentiated, and so gradient descent can't be used. In theory, it is possible to check all of the cases for a discrete problem to find the optimum, but the computations are infeasible for any interesting problem. We therefore need to think of some other approaches. Some examples of discrete problems are:

Chip design Lay a circuit onto a computer chip so that none of the tracks cross.

Timetabling Given a list of courses and which students are on each course, find a timetable with the minimum number of clashes (or given a number of planes and routes, schedule the planes onto the routes).

The Travelling Salesman Problem Given a set of cities, find a tour (that is, a solution that visits every city exactly once, and returns to the starting point) that minimises the total distance travelled.

One thing that is worth noting is that there is no one ideal solution to the search problem. That is, there is no one search algorithm that is guaranteed to perform the best on every problem that is presented to it—you always have to put some work into choosing the algorithm that will be most effective for your problem, and phrasing your problem to make the algorithm work as efficiently as possible. This is called the No Free Lunch theorem.

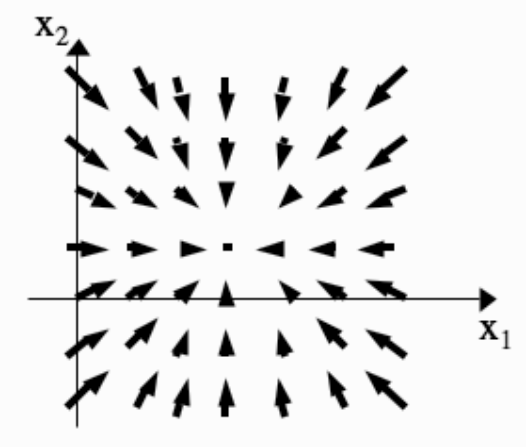

FIGURE 9.1 The downhill gradients to minimise a function. At the solution the gradient is 0. This is a nice example without local minima; they would also have gradient 0.

9.1 GOING DOWNHILL

We will start by trying to derive a better understanding of gradient descent, and seeing the algorithms that can be used for finding local optima for general problems. We will also look at the specific case of solving least-squares optimisation problems, which are the most common examples in machine learning.

The basic idea, as we have already seen, is that we want to minimise a function $f(\mathbf{x})$, where $\mathbf{x}$ is a vector $(x_1, x_2, \ldots, x_n)$ that has elements for each feature value, starting from some initial guess $\mathbf{x}(0)$. We try to find a sequence of new points $\mathbf{x}(i)$ that move downhill towards a solution. The methods that we are going to look at work in any number of dimensions. We will therefore have to take derivatives of the function in each of the different dimensions of $\mathbf{x}$. We write down this whole set of functions as $\nabla f(\mathbf{x})$, which is a vector with elements $(\frac{\partial f}{\partial \mathbf{x_1}}, \frac{\partial f}{\partial \mathbf{x_2}}, \ldots, \frac{\partial f}{\partial \mathbf{x_n}})$, so that it gives us the gradient in each dimension separately. Figure 9.1 shows a set of directions in two dimensions in order to minimise some function.

The first thing to think about is how we know when we have found a solution; in other words, how will we know when to stop? This is relatively easy: it is when $\nabla f = 0$, since then there is no more downhill to go. If you are walking down a hill, then you have reached the bottom when everything is flat around you (which might not be a very large space before things start going up again, but if the function is continuous, as we will assume here, then there must be a point where it is 0 inbetween where it is going down and where it starts going up). So we will know when to terminate the algorithm by checking whether or not $\nabla f = 0$. In practice, the algorithms will always have some numerical inaccuracy, since they are floating point numbers inside the computer, so we usually stop if $|\nabla f| < \epsilon$ where ϵ is some small number, maybe 10^{-5}. There is another concept that it can be useful to think about, which is the places that we can travel to without going up or down, i.e., the places that are at the same level as we are. The full set of places that have the same function value are known as level sets of the function, and some examples are shown in Figure 9.2. Often there will be several discrete parts to a level set, so it is not possible to explore it all without stepping off the set itself.

So from the current point $\mathbf{x}_i$ there are two things that we need to decide: what direction should we move in to go downhill as fast as possible, and how far should we move? Looking

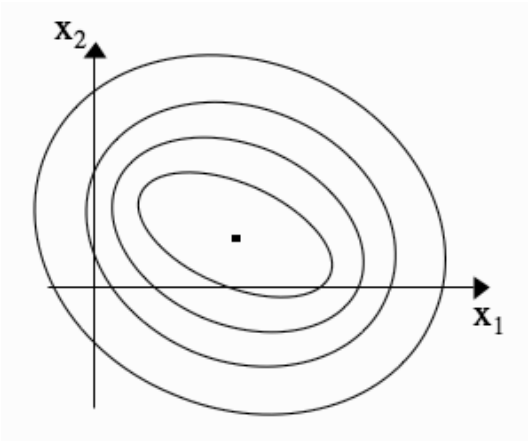

FIGURE 9.2 The lines show contours of equal value (level sets) for a function.

at the second of these questions first, there are two types of methods that can be used to solve it. The simplest approach is a line search: if we know what direction to look in, then we move along it until we reach the minimum in this direction. So this is just a search along the line we are moving along. Writing this down mathematically, if we are currently at $\mathbf{x}_k$ then the next guess will be $\mathbf{x}_{k+1}$, which is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{9.1}$$

where $\mathbf{p}_k$ is the direction we have chosen to move in and α_k is the distance to travel in that direction, chosen by the line search. Finding a value for α_k can be computationally expensive and inaccurate, so it is generally just estimated.

The other method of choosing how far to move is known as a trust region. It is more complex, since it consists of making a local model of the function as a quadratic form and finding the minimum of that model. We will see one example of a trust region method in Section 9.2, and more information about general trust region methods is available in the books listed at the end of the chapter.

The direction $\mathbf{p}_k$ can also be chosen in several ways. The left of Figure 9.3 shows the ideal situation, which is that we point directly to the minimum, in which case the line search finds it straight away. Since we don't know the minimum (it is what we are trying to find!) this is virtually impossible. One thing that we can do is to make greedy choices and always go downhill as fast as possible at each point. This is known as steepest descent, and it means that $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$. The problem with it can be seen on the right of Figure 9.3, which is that many of the directions that it travels in are not directly towards the centre. In extreme cases they can be very different: across the valley, rather than down towards the global minimum (we saw this in Figure 4.7).

If we don't worry about the stepsize, and just set it as $\alpha_k = 1$, then we can perform the search using Equation (9.1) with a very simple program. All that is needed is to iterate the line search until the solution stops changing (or you decide that there have been too many iterations). The only other thing that you have to compute is the derivative of the function, which is the direction $\mathbf{p}_k$. This is the problem-specific part of the algorithm, and as a small example, we consider a simple three-dimensional function $f(\mathbf{x}) = (0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2)$. We can differentiate once to compute the vector of derivatives, $\nabla f(\mathbf{x}) = (x_1, 0.4x_2, 1.2x_3)$, which is returned by the `gradient()` function in the code below:

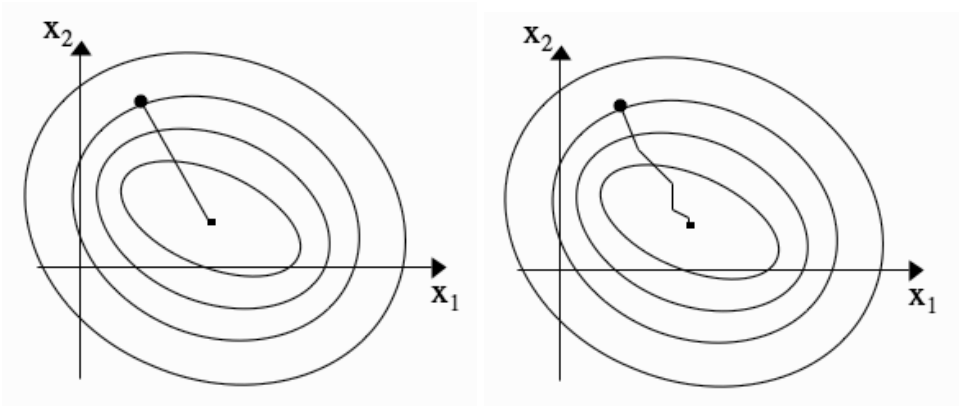

FIGURE 9.3 *Left:* In an ideal world we would know how to go to the minimum directly. In practice, we don't, so we have to approximate it by something like *right:* moving in the direction of steepest descent at each stage.

```
def gradient(x):
    return np.array([x[0], 0.4*x[1], 1.2*x[2]])

def steepest(x0):
    i = 0
    iMax = 10
    x = x0
    Delta = 1
    alpha = 1

    while i<iMax and Delta>10**(-5):
        p = -Jacobian(x)
        xOld = x
        x = x + alpha*p
        Delta = np.sum((x-xOld)**2)
        print x
        i += 1
```

To compute the minimum we now need to pick a start point, for example, $\mathbf{x}(0) = (-2, 2, -2)$, and then we can compute the steepest downhill direction as $(-2, 0.8, -2.4)$. Using the steepest descent method for this example gives fairly poor results, taking several steps before the answer gets close to the correct answer of $(0, 0, 0)$, and even then it is not that close:

```
[ 0. 1.20      0.40]
[ 0. 0.72     -0.08]
[ 0. 0.43      0.01]
[ 0. 0.26     -0.00]
[ 0. 0.16      0.00]
[ 0. 0.09     -0.00]
[ 0. 5.69-02   2.56-05]
```

To see how we can improve on this we need to examine the basics of function approximation.

9.1.1 Taylor Expansion

Steepest descent is based on the Taylor expansion of the function, which is a method of approximating the value of a function at a point in terms of its derivatives: a function $f(\mathbf{x})$ can be approximated by:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{J}(f(\mathbf{x}))|_{\mathbf{x}_0}(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(f(\mathbf{x}))|_{\mathbf{x}_0}(\mathbf{x} - \mathbf{x}_0) + \ldots, \tag{9.2}$$

where $\mathbf{x}_0$ is a common, but potentially slightly confusing notation for the initial guess $\mathbf{x}(0)$, the $|_{\mathbf{x}_0}$ notation means that the function is evaluated at that point, and the $\mathbf{J}(\mathbf{x})$ term is the Jacobian, which is the vector of first derivatives:

$$\mathbf{J}(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right) \tag{9.3}$$

and $\mathbf{H}(\mathbf{x})$ is the Hessian matrix of second derivatives (the Jacobian of the gradient), which for a single function $f(x_1, x_2, \ldots x_n)$ is defined as:

$$\mathbf{H}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}_i} \frac{\partial}{\partial \mathbf{x}_j} f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ & \cdots & & \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial f_n(\mathbf{x})}{\partial x_n^2} \end{pmatrix}. \tag{9.4}$$

If $f(\mathbf{x})$ is a scalar function (so that it returns just 1 number) then $\mathbf{J}(\mathbf{x}) = \nabla f(\mathbf{x})$ and is a vector and $\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x})$ is a two-dimensional matrix. For a vector $\mathbf{f}(\mathbf{x})$ with components $f_1(\mathbf{x}), f_2(\mathbf{x})$, etc., $\mathbf{J}(\mathbf{x})$ is a two-dimensional matrix and $\mathbf{H}(\mathbf{x})$ is three-dimensional.

If we ignore the Hessian term in Equation (9.2), then for scalar $f(\mathbf{x})$ we get precisely the steepest descent step. However, if we choose to minimise Equation (9.2) exactly as it is written (i.e., ignoring third derivatives and higher), then we find the Newton direction at the kth iteration to be: $\mathbf{p}_k = -(\nabla^2 f(\mathbf{x}_k))^{-1} \nabla f(\mathbf{x}_k)$. There is something important to notice about this equation, which is that we actually use the inverse of the Hessian. Computing this is generally of order $\mathcal{O}(N^3)$ (where N is the number of elements in the matrix) which makes this a computationally expensive method. The compensation for this cost is that we don't really have to worry about the stepsize at all; it is always set to 1.

Implementing this requires only 1 line of change to our basic steepest descent algorithm, plus the addition of a function that computes the Hessian. The line to change is the one that computes $\mathbf{p}_k$, which becomes:

```
p = -np.dot(np.linalg.inv(Hessian(x)),Jacobian(x))
```

For this simple example, this algorithm goes straight to the correct answer in one step, which is much better than the steepest descent method that we saw earlier. However, for more complicated functions it won't work as well, because the estimate of the Hessian is not as accurate. There are particular cases where we can, however, do better, as we shall see.

9.2 LEAST-SQUARES OPTIMISATION

For many of the algorithms that we have derived we have used a least-squares error function, such as the error of the MLP and the linear regressor. Least-squares problems turn out to be the most common optimisation problems in many fields, and this means that they have been very well studied and, fortunately, they have special structure in the problem that makes solving them easier than other problems. This leads to a set of special algorithms for solving least-squares problems, although they are mostly special cases of standard methods. One of these has become very well known, the Levenberg–Marquardt method, which is a trust region optimisation algorithm. We will derive the Levenberg–Marquardt algorithm, beginning by identifying why least-squares optimisation is a special case.

9.2.1 The Levenberg–Marquardt Algorithm

For least-squares problems, the objective function that we are optimising is:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\mathbf{x}) = \frac{1}{2}\|\mathbf{r}(\mathbf{x})\|_2^2, \tag{9.5}$$

where the $\frac{1}{2}$ makes the derivative nicer, and $\mathbf{r}(\mathbf{x}) = (r_1(\mathbf{x}), r_2(\mathbf{x}), \ldots, r_m(\mathbf{x}))^T$. In this last version, we can write the (transpose of the) Jacobian of $\mathbf{r}$ as:

$$\mathbf{J}^T(\mathbf{x}) = \left\{ \begin{array}{cccc} \frac{\partial r_1}{\partial x_1} & \frac{\partial r_2}{\partial x_1} & \cdots & \frac{\partial r_m}{\partial x_1} \\ \frac{\partial r_1}{\partial x_2} & \frac{\partial r_2}{\partial x_2} & \cdots & \frac{\partial r_m}{\partial x_2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial r_1}{\partial x_n} & \frac{\partial r_2}{\partial x_n} & \cdots & \frac{\partial r_m}{\partial x_n} \end{array} \right\} = \left[\frac{\partial r_j}{\partial x_i}\right]_{j=1,\ldots,m,\ i=1,\ldots,n}, \tag{9.6}$$

which is useful because the function gradients that we want can mostly be computed directly:

$$\nabla f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T\mathbf{r}(\mathbf{x}) \tag{9.7}$$

$$\nabla^2 f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T\mathbf{J}(\mathbf{x}) + \sum_{j=1}^{m} r_j(\mathbf{x})\nabla^2 r_j(\mathbf{x}). \tag{9.8}$$

The upshot of this is that knowing the Jacobian gives you the first (and usually, most important) part of the Hessian effectively without any additional computational cost, and it is this that special algorithms can exploit to solve least-squares problems efficiently. To see this, remember that, as in all of the other gradient-descent algorithms that we have looked at, we are approximating the function by the Taylor series (Equation (9.2)) up to second-order (Hessian) terms.

If $\|\mathbf{r}(\mathbf{x})\|$ is a linear function of $\mathbf{x}$ (which means that $f(\mathbf{x})$ is quadratic), then the Jacobian is constant and $\nabla^2 r_j(\mathbf{x}) = 0$ for all j. In this case, substituting Equations (9.7) and (9.8) into Equation (9.2) and taking derivatives, we see that at a solution:

$$\nabla f(\mathbf{x}) = \mathbf{J}^T(\mathbf{J}\mathbf{x} + \mathbf{r}) = 0, \tag{9.9}$$

and so:

$$\mathbf{J}^T\mathbf{J}\mathbf{x} = -\mathbf{J}^T\mathbf{r}(\mathbf{x}). \tag{9.10}$$

This is a linear least-squares problem and can be solved. In an ideal world we would be able to see that it is effectively just the statement $\mathbf{A}\mathbf{x} = \mathbf{b}$ (where $\mathbf{A} = \mathbf{J}^T\mathbf{J}$ is a square matrix and $\mathbf{b} = -\mathbf{J}^T\mathbf{r}(\mathbf{x})$ and so solve it directly as:

$$\mathbf{x} = -(\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{r}. \tag{9.11}$$

However, this is computationally expensive and numerically very unstable, so we need to use linear algebra to find $\mathbf{x}$ in a variety of different ways, such as Cholesky factorisation, QR factorisation, or using the Singular Value Decomposition. We will look at the last of these methods, since it uses eigenvectors, which we have already seen in Chapter 6, although we will see the first method in Chapter 18.

The Singular Value Decomposition (SVD) is the decomposition of a matrix $\mathbf{A}$ of size $m \times n$ into:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T, \tag{9.12}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices (i.e., the inverse of the matrix is its transpose, so $\mathbf{U}^T\mathbf{U} = \mathbf{U}\mathbf{U}^T = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix). $\mathbf{U}$ is of size $m \times m$ and $\mathbf{V}$ is of size $n \times n$. $\mathbf{S}$ is a diagonal matrix of size $m \times n$, with the elements of this matrix, σ_i, being known as singular values.

To apply this to the linear least-squares problem we compute the SVD of $\mathbf{J}^T\mathbf{J}$ and substitute it into Equation (9.11):

$$\begin{aligned} \mathbf{x} &= \left[(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T(\mathbf{U}\mathbf{S}\mathbf{V}^T)\right]^{-1}(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T\mathbf{J}^T\mathbf{r} && (9.13)\\ &= \mathbf{V}\mathbf{S}\mathbf{U}^T\mathbf{J}^T\mathbf{r} && (9.14)\end{aligned}$$

using the fact that $\mathbf{A}\mathbf{B}^T = \mathbf{B}^T\mathbf{A}^T$ and similar linear algebraic identities.

We can actually go a bit further, and deal with the fact that $\mathbf{J}$ is probably not a square matrix. The size of the various matrices will be $m \times m$ for $\mathbf{U}$ and $m \times m$ for the other two (where m and n are defined in Equation (9.6); generally $n < m$). We can split $\mathbf{U}$ into two parts, $\mathbf{U}_1$ of size $m \times n$ and then the last few columns into a second part $\mathbf{U}_2$ of size $(n - m) \times n$. This lets us solve the linear least-squares equation as:

$$\mathbf{x} = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}_1^T\mathbf{J}\mathbf{r}. \tag{9.15}$$

NumPy has an algorithm for linear least-squares in `np.linalg.lstsq()` and can compute the SVD decomposition using `np.linalg.svd()`.

We can now use this derivation to look at the most well-known method for solving non-linear least-squares problems, the Levenberg–Marquardt algorithm. The principal approximation that the algorithm makes is to ignore the residual terms in Equation (9.8), making each iteration a linear least-squares problem, so that $\nabla^2 f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T\mathbf{J}(\mathbf{x})$. Then the problem to be solved is:

$$\min_{\mathbf{p}} \frac{1}{2}\|\mathbf{J}_k\mathbf{p} + \mathbf{r}_k\|_2^2, \ \ \|\mathbf{p}\| \leq \Delta_k, \tag{9.16}$$

where Δ_k is the radius of the trust region, which is the region where it is assumed that this approximation holds well. In normal trust region methods, the size of the region (Δ_k) is controlled explicitly, but in Levenberg–Marquardt it is used to control a parameter $\nu \geq 0$ that is added to the diagonal elements of the Jacobian matrix and is known as the damping factor. The minimum $\mathbf{p}$ then satisfies:

$$(\mathbf{J}^T\mathbf{J} + \nu\mathbf{I})\mathbf{p} = -\mathbf{J}^T\mathbf{r}. \tag{9.17}$$

This is a very similar equation to the one that we solved for the linear least-squares method, and so we can just use that solver here; effectively non-linear least-squares solvers solve a lot of linear problems to find the non-linear solution. There are very efficient Levenberg–Marquardt solvers, since it is possible to avoid computing the $\mathbf{J}^T\mathbf{J}$ term explicitly using the SVD composition that we worked out above.

The basic idea of the trust region method is to assume that the solution is quadratic about the current point, and use that assumption to minimise the current step. You then compute the difference between the actual reduction and the predicted one, based on the model, and make the trust region larger or smaller depending upon how well these two match, and if they do not match at all, then you reject that update. The Levenberg–Marquardt algorithm itself is very general, but it needs to have the function to be minimised, along with its gradient and Jacobian passed into it. The entire algorithm can be written as:

The Levenberg–Marquardt Algorithm

- Given start point $\mathbf{x}_0$
- While $\mathbf{J}^T\mathbf{r}(\mathbf{x})$ >tolerance and maximum number of iterations not exceeded:
  - repeat
    - solve $(\mathbf{J}^T\mathbf{J} + \nu\mathbf{I})\mathbf{dx} = -\mathbf{J}^T\mathbf{r}$ for $\mathbf{dx}$ using linear least-squares
    - set $\mathbf{x}_{\text{new}} = \mathbf{x} + \mathbf{dx}$
    - compute the ratio of the actual and prediction reductions:
      - actual = $\|f(\mathbf{x}) - f(\mathbf{x}_{new})\|$
      - predicted = $\nabla f^T(\mathbf{x}) \times \mathbf{x}_{new} - \mathbf{x}$
      - ρ = actual/predicted
    - if $0 < \rho < 0.25$:
      - accept step: $\mathbf{x} = \mathbf{x}_{\text{new}}$
    - else if $\rho > 0.25$:
      - accept step: $\mathbf{x} = \mathbf{x}_{\text{new}}$
      - increase trust region size (reduce ν)
    - else:
      - reject step
      - reduce trust region (increase ν)
  - until $\mathbf{x}$ is updated or maximum number of iterations is exceeded

In SciPy the Levenberg–Marquardt optimiser is in the `optimize` module, and it can

be called using `scipy.optimize.leastsq()`. Some general details about using the SciPy optimisers are given in Section 9.3.2.

We will look at two examples of using non-linear least-squares. One is a simple case of finding the minimum of a function that consists of two quadratic terms added together, i.e., a sum-of-squares problem, while the second is to minimise the fitting of a function to data.

The function that we will attempt to minimise is Rosenbrock's function:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{9.18}$$

This is a common problem to try since it has a long narrow valley, so finding the optimal solution is not especially easy (except by hand: if you look at the problem, then guessing that $x_1 = 1, x_2 = 1$ is the minimum is fairly obvious). You need to work out how to encode this in the form required for a sum-of-squares problem, which is basically to write:

$$\mathbf{r} = (10(x_2 - x_1^2), 1 - x_1)^T. \tag{9.19}$$

The Jacobian is then:

$$\mathbf{J} = \begin{pmatrix} -20x_1 & 10 \\ -1 & 0 \end{pmatrix}. \tag{9.20}$$

In this notation, $f(x_1, x_2) = \mathbf{r}^T\mathbf{r}$ and the gradient is $\mathbf{J}^T\mathbf{r}$. All of which can be written as a simple Python function:

```
def function(p):
    r = np.array([10*(p[1]-p[0]**2),(1-p[0])])
    fp = np.dot(transpose(r),r)
    J = (np.array([[-20*p[0],10],[-1,0]]))
    grad = np.dot(J.T,r.T)
    return fp,r,grad,J
```

Running the algorithm with starting point $(-1.92, 2)$ leads to the following outputs, where the numbers printed on each line are the function value, the parameters that gave it, the gradient, and the value of ν.

```
f(x)    Params       Grad        nu
292.92  [ 0.66 -6.22] 672.00  0.001
4421.20 [ 0.99  0.87] 1099.51 0.0001
1.21    [ 1.00  1.00] 24.40   1e-05
8.67-07 [ 1.00  1.00] 0.02    1e-06
6.18-17 [ 1.00  1.00] 1.57-07 1e-07
```

The second example is fitting a function to data. The function is a moderately complicated beast that is definitely not amenable to linear least-squares fitting:

$$y = f(p_1, p_2) = p_1 \cos(p_2 x) + p_2 \sin(p_1 x), \tag{9.21}$$

where the p_i are the parameters to be fitted and x is a datapoint from a set that are

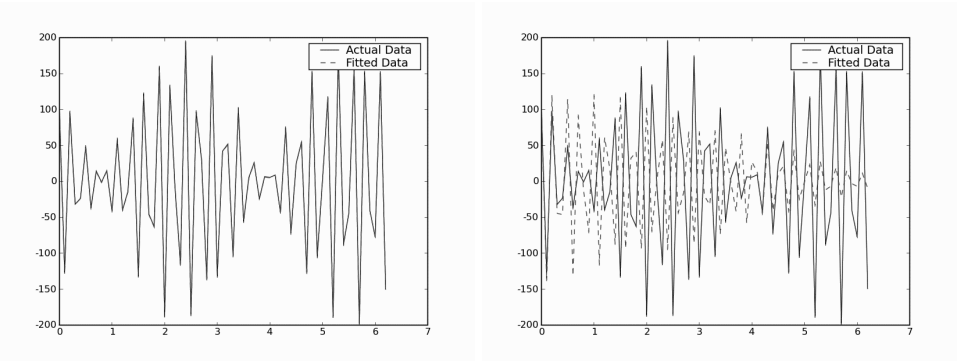

FIGURE 9.4 Using Levenberg–Marquardt for least-squares data fitting of data from Equation (9.21). The example on the left converges to the correct solution, while the one on the right, which still starts from a point close to the correct solution, fails to find it, resulting in significantly different output.

used to construct the function to be fitted. This is a difficult function to fit because it has lots of minima (since sin and cos are periodic, with period 2π). For data fitting problems, the assumption is often that data are generated at regular x points by a noisy process that produces the y values. Then the sum-of-squares error that we wish to minimise is the difference between the data (y) and the current fit (parameter estimates $\hat{p}_1, \hat{p}_2$):

$$\mathbf{r} = y - \hat{p}_1 \cos(\hat{p}_2 x) + \hat{p}_2 \sin(\hat{p}_1 x). \tag{9.22}$$

The Jacobian for this function requires some careful differentiating, and then the whole problem can be left to the optimiser. Figure 9.4 shows two examples of trying to recover values $p_1 = 100, p_2 = 102$. On the left, the starting point is $(100.5, 102.5)$, while on the right it is $(101, 101)$. It can be seen that on this problem, Levenberg–Marquardt is very susceptible to local minima, since while the example on the left works (converging after only 8 iterations), the example on the right, which still starts with parameter values very close to the correct ones, gets stuck and fails, with final parameter values $(100.89, 101.13)$.

9.3 CONJUGATE GRADIENTS

Not every problem that we want to solve is a least-squares problem. The good news is that we can do rather better than steepest descent even when we want to minimise an arbitrary objective function. The key to this is to look again at Figure 9.3, where you can see that there are several of the steepest gradient lines that are in pretty much the same direction. We would only need to go in that direction once if we knew how far to go the first time. And then we would go in a direction orthogonal (at right angles) to that one and, in two dimensions, we would be finished, as is shown on the right of Figure 9.5, where one step in the x direction and one in the y direction are enough to complete the minimisation. In n dimensions we would have to take n steps, and then we would have finished. This amazing scenario is the aim of the method of conjugate gradients. It manages to achieve it in the linear case, but in most non-linear cases, which are the kind we are usually interested in, it usually requires a few more iterations than it theoretically should, although still many less steps than most other methods for real problems.

It turns out that making the lines be orthogonal is generally impossible, since you don't

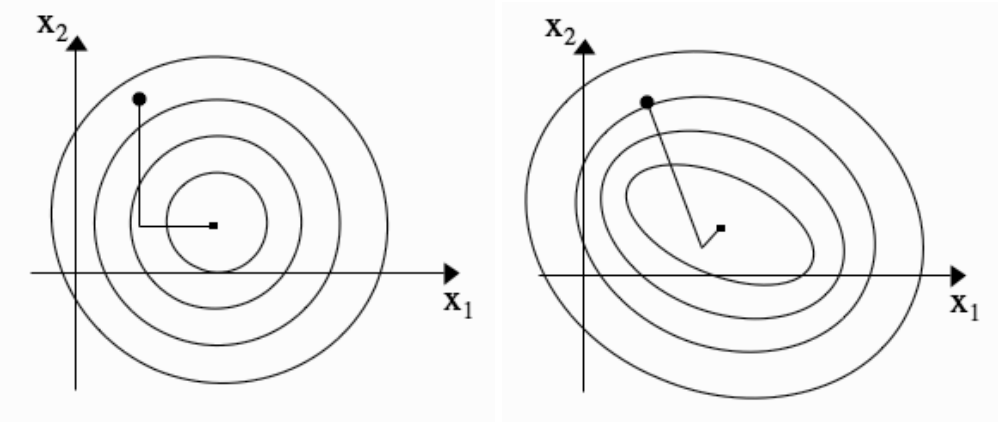

FIGURE 9.5 *Left:* If the directions are orthogonal to each other and the stepsize is correct, then only one step is needed for each dimension in the data, here two. *Right:* The conjugate directions are not orthogonal to each other on the ellipse.

have enough information about the solution space. However, it is possible to make them conjugate or A-orthogonal. Two vectors $\mathbf{p}_i, \mathbf{p}_j$ are conjugate if $\mathbf{p}_i^T \mathbf{A} \mathbf{p}_j = 0$ for some matrix $\mathbf{A}$. Conjugate lines for the ellipse contours in Figure 9.2 are shown on the right of Figure 9.5. Amazingly, the line search that we wrote down in Equation (9.1) is soluble along these directions, since they do not interfere with each other, with solution:

$$\alpha_i = \frac{\mathbf{p}_i^T(-\nabla f(\mathbf{x}_{i-1}))}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}. \tag{9.23}$$

We then need to use a function to find the zeros of this. The Newton–Raphson iteration, which is one method that will do it, is described below. So if we can find conjugate directions, then the line search is much better. The only question that remains is how to find them. This requires a Gram–Schmidt process, which constructs each new direction by taking a candidate solution and then subtracting off any part that lies along any of the directions that have already been used. We start by picking a set of mutually orthogonal vectors $\mathbf{u}_i$ (the basic coordinate axes will do; there are better options, but they are beyond the scope of this book) and then using:

$$\mathbf{p}_k = \mathbf{u}_k + \sum_{i=0}^{k-1} \beta_{ki} \mathbf{p}_i. \tag{9.24}$$

There are two possible β terms that can be used. They are both based on the ratios between the squared Jacobian before and after an update. The Fletcher–Reeves formula is:

$$\beta_{i+1} = \frac{\nabla f(\mathbf{x}_{x+1})^T \nabla f(\mathbf{x}_{i+1})}{\nabla f(\mathbf{x}_i)^T \nabla f(\mathbf{x}_i)}, \tag{9.25}$$

while the Polak–Ribiere formula is:

$$\beta_{i+1} = \frac{\nabla f(\mathbf{x}_{i+1})^T \left(\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x})_i\right)}{\nabla f(\mathbf{x}_i)^T \nabla f(\mathbf{x}_i)}. \tag{9.26}$$

The second one is often faster, but sometimes fails to converge (reach a stopping point).

We can put these things together to form a complete algorithm. It starts by computing an initial search direction $\mathbf{p}_0$ (steepest descent will do), then finding the α_i that minimises the function $f(\mathbf{x}_i + \alpha_i \mathbf{p}_i)$, and using it to set $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i$. The next direction is then $\mathbf{p}_{i+1} = -\nabla f(\mathbf{x}_{i+1}) + \beta_{i+1}\mathbf{p}_i$ where β is set by one of the two formulas above.

It is common to restart the algorithm every n iterations (where n is the number of dimensions in the problem) because the algorithm has now generated the whole set of conjugate directions. The algorithm will then cycle through the directions again making incremental improvements.

The only thing that we don't know how to do yet is to find the α_is. The usual method of doing that is the Newton–Raphson iteration, which is a method of finding the zero points of a polynomial. It works by computing the Taylor expansion of the function $f(\mathbf{x} + \alpha\mathbf{p})$, which is:

$$f(\mathbf{x}+\alpha\mathbf{p}) \approx f(\mathbf{x})+\alpha\mathbf{p}\left(\frac{d}{d\alpha}f(\mathbf{x}+\alpha\mathbf{p})\right)\Big|_{\alpha=0} + \frac{\alpha^2}{2}\mathbf{p}\cdot\mathbf{p}\left(\frac{d^2}{d\alpha^2}f(\mathbf{x}+\alpha\mathbf{p})\right)\Big|_{\alpha=0} + \ldots, \quad (9.27)$$

and differentiating it with respect to α, which requires the Jacobian and Hessian matrices (here, these matrices are derivatives of $f(\cdot)$, not $\mathbf{r}$ as they were in Section 9.2):

$$\frac{d}{d\alpha}f(\mathbf{x}+\alpha\mathbf{p}) \approx \mathbf{J}(\mathbf{x})\mathbf{p} + \alpha\mathbf{p}^T\mathbf{H}(\mathbf{x})\mathbf{p}. \quad (9.28)$$

Setting this equal to zero tells us that the minimiser of $f(\mathbf{x} + \alpha\mathbf{p})$ is:

$$\alpha = \frac{\mathbf{J}(\mathbf{x})^T\mathbf{p}}{\mathbf{p}^T\mathbf{H}(\mathbf{x})\mathbf{p}}. \quad (9.29)$$

Unless $f(\mathbf{x})$ is an especially nice function, the second derivative approximation that we have made here won't get us to the bottom in one step, so we will have to iterate this step a few times to find the zero point, which is why it is known as the Newton–Raphson *iteration*, i.e., you have to put it into a loop that runs until the iterate stops changing.

Putting all of those things together gives the entire algorithm, which we'll look at before we work on an example:

The Conjugate Gradients Algorithm

- Given start point $\mathbf{x}_0$, and stopping parameter ϵ, set $\mathbf{p}_0 = -\nabla f(\mathbf{x})$
- Set $\mathbf{p} = \mathbf{p}_0$
- While $\mathbf{p} > \epsilon^2\mathbf{p}_0$:
  - compute α_k and $\mathbf{x}_{\text{new}} = \mathbf{x} + \alpha_k\mathbf{p}$ using the Newton–Raphson iteration:
    - while $\alpha^2 dp > \epsilon^2$:
      - $\alpha = -(\nabla f(\mathbf{x})^T\mathbf{p})/(\mathbf{p}^T\mathbf{H}(\mathbf{x})\mathbf{p})$
      - $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$
      - $dp = \mathbf{p}^T\mathbf{p}$
  - evaluate $\nabla f(\mathbf{x}_{\text{new}})$
  - compute β_{k+1} using Equation (9.25) or (9.26)
  - update $\mathbf{p} \leftarrow \nabla f(\mathbf{x}_{\text{new}}) + \beta_{k+1}\mathbf{p}$
  - check for restarts

9.3.1 Conjugate Gradients Example

Computing the conjugate gradients solution to the function $f(\mathbf{x}) = (0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2)$ makes use of the Jacobian and Hessian again. The first Newton–Raphson step yields an α value of 0.931, so that the next step is:

$$\mathbf{x}(1) = \begin{pmatrix} -2 \\ 2 \\ 0 \end{pmatrix} + 0.931 \times \begin{pmatrix} 2 \\ -0.8 \\ 2.4 \end{pmatrix} = \begin{pmatrix} -0.138 \\ 1.255 \\ 0.235 \end{pmatrix} \tag{9.30}$$

Then $\beta = 0.0337$, so that the direction is:

$$\mathbf{p}(1) = \begin{pmatrix} 0.138 \\ -0.502 \\ -0.282 \end{pmatrix} + 0.0337 \times \begin{pmatrix} 2 \\ -0.8 \\ 2.4 \end{pmatrix} = \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} \tag{9.31}$$

In the second step, $\alpha = 1.731$,

$$\mathbf{x}(2) = \begin{pmatrix} -0.138 \\ 1.255 \\ 0.235 \end{pmatrix} + 1.731 \times \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} = \begin{pmatrix} -0.217 \\ -0.136 \\ 0.136 \end{pmatrix} \tag{9.32}$$

and the update is:

$$p(2) = \begin{pmatrix} -0.217 \\ -0.136 \\ 0.136 \end{pmatrix} + 0.240 \times \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} = \begin{pmatrix} -0.168 \\ -0.263 \\ 0.088 \end{pmatrix} \tag{9.33}$$

A third step then gives the final answer as $(0, 0, 0)$.

9.3.2 Conjugate Gradients and the MLP

The scientific Python libraries SciPy include a set of general purpose optimisation algorithms in `scipy.optimize`, including an interface function (`scipy.optimize.minimize()`) that can call the others. In this section we will investigate using the methods that are provided within that library, particularly the conjugate gradient optimiser, in order to find the weights of the Multi-layer Perceptron (MLP) that was the main algorithm of Chapter 4. In that chapter we derived an algorithm based on gradient descent of the back-propagated error from first principles, but here we can use general methods.

In order to use any gradient descent algorithm we need to work out a function to minimise, an initial guess for where to start searching, and (preferably) the gradient of that function with respect to the variables. The reason for saying 'preferably the gradient' is that many of the algorithms will create a numerical estimate of the gradient if an explicit version is not given. However, since the gradient is fairly easy to compute for the MLP, numerical estimation is not necessary. We used the sum-of-squares error for the MLP, so we just need to work out the derivatives of that function for the three different activation functions that we allow: the normal logistic function, the linear activation that was used for regression problems, and the soft-max activation, and we've already done that in Section 4.6.5.

As was mentioned above, there is an interface function for most of the SciPy optimisers, which has the following form:

```
scipy.optimize.minimize(fun, x0, args=(), method='BFGS', jac=None,
hess=None,
hessp=None, bounds=None, constraints=(), tol=None, callback=None,
options=None)}.
```

The choice of method, which is the actual gradient descent algorithm used can include `'BFGS'` (which is the Broyden, Fletcher, Goldfarb, and Shanno algorithm, a variation on Newton's method from Section 9.1.1 that computes an approximation to the Hessian rather than requiring the programmer to supply it) and `CG` which is the conjugate gradient algorithm.

Looking at the code snippet again we see that we need to pass in an error function and the function to compute the derivatives. Both of these functions take arguments, specifically the inputs to the network, and the targets that those inputs are meant to produce. There is one issue that we have to deal with here, which is that the SciPy optimisers find the minimum value for a vector of parameters, and we currently have two separate weights matrices. We need to reshape these two matrices into vectors and then concatenate them before they can be used, using:

```
w = np.concatenate((self.weights1.flatten(),self.weights2.flatten()))
```

and something similar for the gradients. When the optimiser has run we will need to separate them and put the values back into the weight matrices using:

```
split = (self.nin+1)*self.nhidden
self.weights1 = np.reshape(wopt[:split],(self.nin+1,self.nhidden))
self.weights2 = np.reshape(wopt[split:],(self.nhidden+1,self.nout))
```

The optimiser also needs an initial guess `x0` for the weights, but this is not an issue since in the original algorithm they are already set to have small positive and negative values, so we can just use those values.

In fact, there is a numerical detail that we need to deal with as well; technically it could be a problem with the version of the MLP that we implemented in Chapter 4 as well, but it doesn't usually seem to be an issue there. The problem is that when we use the sigmoid function and take the exponential we can get overflow in the floating point number, either from it becoming too large, or too close to 0. This is a particular problem when we use the cross-entropy error function of Section 4.6.6, because we then take the logarithm, and we need to make sure that the input is in the range of the log function. NumPy provides some useful constants to make these checks, and they can be seen in use in the following code snippet, which replaces the error calculation in the original MLP:

```
# Different types of output neurons
if self.outtype == 'linear':
    error = 0.5*np.sum((outputs-targets)**2)
elif self.outtype == 'logistic':
    # Non-zero checks
    maxval = -np.log(np.finfo(np.float64).eps)
    minval = -np.log(1./np.finfo(np.float64).tiny - 1.)
    outputs = np.where(outputs<maxval,outputs,maxval)
    outputs = np.where(outputs>minval,outputs,minval)
    outputs = 1./(1. + np.exp(-outputs))
    error = - np.sum(targets*np.log(outputs) + (1 - targets)*np.log(1 - outputs))
elif self.outtype == 'softmax':
    nout = np.shape(outputs)[1]
    maxval = np.log(np.finfo(np.float64).max) - np.log(nout)
    minval = np.log(np.finfo(np.float32).tiny)
    outputs = np.where(outputs<maxval,outputs,maxval)
    outputs = np.where(outputs>minval,outputs,minval)
    normalisers = np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)[0]))
    y =  np.transpose(np.transpose(np.exp(outputs))/normalisers)
    y[y<np.finfo(np.float64).tiny] = np.finfo(np.float32).tiny
    error = - np.sum(targets*np.log(y));
```

Finally, we need to decide how accurate we want the result to be, and how many iterations we are going to allow the algorithm to run for before calling a halt to the optimisation. When the algorithm has reached a minimum the gradient function will be 0, and so the normal convergence criterium is that the gradient is close to zero. The default value for this parameter is 1×10^{-5} and we will leave this unchanged. We will also specify that the algorithm can run for no more than 10,000 steps. Together, these lead to the following function call to the conjugate gradient optimiser (here the code uses an explicit call to the conjugate gradient method rather than the interface, but there is no real difference):

```
out =        so.fmin_cg(self.mlperror,        w,        fprime=self.mlpgrad,
args=(inputs,targets)
, maxiter=10000, full_output=True, disp=1)
```

The `full_output` and `disp` parameters tell the optimiser to give a report on whether or not it was successful and how much work it did, something like:

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 7.487182
         Iterations: 10000
         Function evaluations: 250695
```

```
        Gradient evaluations: 140930
```

Now all that remains is to extract the new weight values from the values that the optimiser returns, which are in `out[0]`, and we are ready to use the algorithm. The demonstrations that we used in Chapter 4 are all perfectly suitable, of course, and all that needs changing is to import the conjugate gradient version of the MLP instead of the earlier version.

There are other methods of doing gradient descent, some of which are more effective on certain problems (but note that the No Free Lunch theorem tells us that no one solver will be the most effective for every problem). For example, the convex optimisation that was used for the Support Vector Machine in Chapter 8 is a gradient descent method for a particular type of constrained problem. We will next consider what happens when the problems that we wish to solve are discrete, which means that there is no gradient to find.

9.4 SEARCH: THREE BASIC APPROACHES

We are going to discuss three different ways to attempt optimisation without gradients. For each one, we will see how it works on the Travelling Salesman Problem (TSP), which is a classic discrete optimisation problem that consists of trying to find the shortest route through a set of cities that visits each city exactly once and returns to the start. For the first (starting) city we can choose any of the N that are available. For the next, there are $N - 1$ choices, and for the next $N - 2$. Using a brute force search in this way provides a $\mathcal{O}(N!)$ solution, which is obviously infeasible.

In fact, the TSP is an NP-hard problem. The best-known solution that is guaranteed to find the global maximum is using dynamic programming and its computational cost is $\mathcal{O}(n^2 2^n)$, but we won't be considering that here—the TSP is an example, not a problem we really want to solve here. The basic search methods are described next.

9.4.1 Exhaustive Search

Try out every solution and pick the best one. While this is obviously guaranteed to find the global optimum, because it checks every single solution, it is impractical for any reasonable size problem. For the TSP it would involve testing out every single possible way of ordering the cities, and calculating the distance for each ordering, so the computational complexity is $\mathcal{O}(N!)$, which is worse than exponential.

It is computationally infeasible to do the computations for more than about $N = 10$ cities. The basic part of the algorithm uses a helper function `permutation()` that computes possible orderings of the cities, but is otherwise fairly obvious:

```
for newOrder in permutation(range(nCities)):
    possibleDistanceTravelled = 0
    for i in range(nCities-1):
        possibleDistanceTravelled += distances[newOrder[i],newOrder[i+1]]
    possibleDistanceTravelled += distances[newOrder[nCities-1],0]

    if possibleDistanceTravelled < distanceTravelled:
```

```
        distanceTravelled = possibleDistanceTravelled
        cityOrder = newOrder
```

9.4.2 Greedy Search

Just make one pass through the system, making the best local choice at each stage. So for the TSP, choose the first city arbitrarily, and then repeatedly pick the city that is closest to where you are now that hasn't been visited yet, until you run out of cities. This is computationally very cheap ($\mathcal{O}(N \log N)$), but it is certainly not guaranteed to find the optimal solution, or even a particularly good one. The code is very simple, though:

```
for i in range(nCities-1):
    cityOrder[i+1] = np.argmin(dist[cityOrder[i],:])
    distanceTravelled  += dist[cityOrder[i],cityOrder[i+1]]
    # Now exclude the chance of travelling to that city again
    dist[:,cityOrder[i+1]] = np.Inf

# Now return to the original city
distanceTravelled += distances[cityOrder[nCities-1],0]
```

9.4.3 Hill Climbing

The basic idea of the hill climbing algorithm is to perform local search around the current solution, choosing any option that improves the result. (It might seem odd to talk about hill *climbing* when we've always talked about minimising a function. Of course, the difference between maximisation and minimisation is just whether you put a minus sign in front of the equation or not, and 'hill climbing' sounds much better than 'hollow descending.') The choice of how to do local search is called the move-set. It describes how the current solution can be changed to generate new solutions. So if we were to imagine moving about in 2D Euclidean space, possible moves might be to move 1 step north, south, east, or west.

For the TSP, the hill climbing solution would consist of choosing an initial solution randomly, and then swapping pairs of cities in the tour and seeing if the total length of the tour decreases. The algorithm would stop after some pre-defined number of swaps had occurred, or when no swap improved the result for some pre-defined length of time. As with the greedy search, there is no way to predict how good the solution will be: there is a chance that it will find the global maximum, but no guarantee of it; it could get stuck in the first local maxima. The central loop of the hill climbing algorithm just picks a pair of cities to swap, and keeps the change if it makes the total distance shorter:

```
for i in range(1000):
    # Choose cities to swap
    city1 = np.random.randint(nCities)
    city2 = np.random.randint(nCities)
```

```
    if city1 != city2:
        # Reorder the set of cities
        possibleCityOrder = cityOrder.copy()
        possibleCityOrder = np.where(possibleCityOrder==city1,-1,
        possibleCityOrder)
        possibleCityOrder = np.where(possibleCityOrder==city2,city1,
        possibleCityOrder)
        possibleCityOrder = np.where(possibleCityOrder==-1,city2,
        possibleCityOrder)

        # Work out the new distances
        # This can be done more efficiently
        newDistanceTravelled = 0
        for j in range(nCities-1):
            newDistanceTravelled += distances[possibleCityOrder[j],
            possibleCityOrder[j+1]]
        distanceTravelled += distances[cityOrder[nCities-1],0]

        if newDistanceTravelled < distanceTravelled:
            distanceTravelled = newDistanceTravelled
            cityOrder = possibleCityOrder
```

Hill climbing has three particular types of functions that it does badly on. They can all be imagined using the analogy of real hill climbing.

The first is when there are lots of foothills around the optimal solution. In that case the algorithm climbs the local maximum, and may get stuck there; certainly it will take a very long time to reach the optimal solution. The second is on a plateau, where no changes that the algorithm makes affect the solution. In this case the solution will just change randomly, if at all, and the maximum will probably not be found. The third case is when there is a very gently sloping ridge in the data. Most directions that the algorithm looks in will be downhill, and so it may decide that it has already reached the maximum.

9.5 EXPLOITATION AND EXPLORATION

The search methods above can be separated into methods that perform exploration of the search space, always trying out new solutions, like exhaustive search, and those performing exploitation of the current best solution, by trying out local variations of that current best solution, like hill climbing. Ideally, we would like some combination of the two—we should be trying to improve on the current best solution by local search, and also looking around in case there is an even better solution hiding elsewhere in the search space.

One way to think about this is known as the n-armed bandit problem. Suppose that we have a room full of one-armed bandit machines in some tacky Las Vegas casino (for those who don't know, a one-armed bandit is a slot machine with a lever that you pull, as in Figure 9.6). You don't know anything about the machines in advance, such as what the payouts are, and how likely you are to get the payout. You enter the room with a fistful of 50 cent coins from your student loan, aiming to generate enough beer money to get through the year. How do you choose which machine to use?

FIGURE 9.6 A one-armed bandit machine. It has one arm, and it steals your money.

At first, you have no information at all, so you choose randomly. However, as you explore, you pick up information about which machines are good (here, good means that you get a payout more often). You could carry on using them (exploiting your knowledge) or you could try out other machines in the hope of finding one that pays out even more (exploring further). The optimal approach is to trade off the two, always making sure that you have enough money to explore further by exploiting the best machines you know of, but exploring when you can.

One place where this combination of exploration and exploitation can be clearly seen is in evolution. We'll talk about that in the next chapter, but here we will look to physics instead of biology to act as our inspiration.

9.6 SIMULATED ANNEALING

In the field of statistical mechanics physicists have to deal with systems that are very large (tens of thousands of molecules and more) so that, while the computations are possible in principle, in practice the computational time is far too large. They have developed stochastic methods (that is, based on randomness) in order to get approximate solutions to the problems that, while still expensive, do not require the massive computational times that the full solution would.

The method that we will look at is based on the way in which real-world physical systems can be brought into very low energy states, which are therefore very stable. The system is heated, so that there is plenty of energy around, and each part of the system is effectively random. An annealing schedule is applied that cools the material down, allowing it to relax into a low energy configuration. We are going to model the same idea.

We start with an arbitrary temperature T, which is high. We will then randomly choose states and change their values, monitoring the energy of the system before and after. If the energy is lower afterwards, then the system will prefer that solution, so we accept the change. So far, this is similar to gradient descent. However, if the energy is not lower, then we still consider whether or not to accept the solution. We do this by evaluating $E_{\text{before}} - E_{\text{after}}$ and accepting the new solution if the value of $\exp((E_{\text{before}} - E_{\text{after}})/T)$ is bigger than a uniform random value between 0 and 1 (note that the expression is between 0 and 1 since it is the exponential of a negative value). This is called the Boltzmann distribution. The rationale

behind sometimes accepting poorer states is that we might have found a local minimum, and by allowing this more expensive energy state we can escape from it.

After doing this a few times, the annealing schedule is applied in order to reduce the temperature and the method continues until the temperature reaches 0. As the temperature gets lower, so does the chance of accepting any particular higher energy state. The most common annealing schedule is $T(t+1) = cT(t)$, where $0 < c < 1$ (more commonly, $0.8 < c < 1$). The annealing needs to be slow to allow for lots of search to happen. For the TSP the best way to include simulated annealing is to modify the hill climbing algorithm above, changing the acceptance criteria for a change in the city ordering to:

```
if newDistanceTravelled < distanceTravelled or (distanceTravelled - ↩
newDistanceTravelled) < T*np.log(np.random.rand()):
   distanceTravelled = newDistanceTravelled
   cityOrder = possibleCityOrder

# Annealing schedule
T = c*T
```

9.6.1 Comparison

Running all four methods above on the TSP for five cities gave the following results, where the best solution found and the distance are given in the first line and the time it took to run (in seconds) on the second:

```
>>> TSP.runAll()
Exhaustive search
((3, 1, 2, 4, 0), 2.65)
0.0036
Greedy search
((0, 2, 1, 3, 4), 3.27)
0.0013
Hill Climbing
((4, 3, 1, 2, 0]), 2.66)
0.1788
Simulated Annealing
((3, 1, 2, 4, 0]), 2.65)
0.0052
```

With ten cities the results were quite different, showing how important good approximations to search are, since even for this fairly small problem the exhaustive search takes a very long time. Note that the greedy search does nearly as well in this case, but this is simply chance.

```
Exhaustive search
((1, 5, 10, 6, 3, 9, 2, 4, 8, 7, 0), 4.18)
```

```
1781.0613
Greedy search
((3, 9, 2, 6, 10, 5, 1, 8, 4, 7, 0]), 4.49)
0.0057
Hill Climbing
((7, 9, 6, 2, 4, 0, 3, 8, 1, 5, 10]), 7.00)
0.4572
Simulated Annealing
((10, 1, 6, 9, 8, 0, 5, 2, 4, 7, 3]), 8.95)
0.0065
```

FURTHER READING

Two books on numerical optimization that provide much more information are:

- J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer, Berlin, Germany, 1999.
- C.T. Kelley. *Iterative Methods for Optimization*. Number 18 in Frontiers in Applied Mathematics. SIAM, Philadelphia, USA, 1999.

A possible reference for the second half of the chapter is:

- J.C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley-Interscience, New York, USA, 2003.

Some of the material is covered in:

- Section 6.9 and Sections 7.1–7.2 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.

PRACTICE QUESTIONS

Problem 9.1 In the discussion after Equation (9.10) it is stated that the direct solution is unstable. Experiment with this and see that it is true.

Problem 9.2 Modify the code in `CG.py` in order to take a general function, together with its Jacobian (and if available its Hessian) and then compute the minimum.

Problem 9.3 Experiment with the Fletcher–Reeves and Polak–Ribiere formulas (Equations (9.25) and (9.26)) when solving Rosenbrock's function using conjugate gradients. Can you find places where one works better than the other?

Problem 9.4 Generate data from the equation $a(1 - \exp(-b(x - c)))$ for choice of parameters a, b, c and x in the range -5 to 5 (with noise). Use Levenberg–Marquardt to fit the parameters.

Problem 9.5 Modify the conjugate gradient version of the MLP to use the other optimisation algorithms provided by SciPy and compare the results. Also, try stopping the optimiser from using the exact computation of the gradient and instead making a numerical estimate of it, and see how that changes the results.