of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies.

OPTIMAL BRAIN DAMAGE

The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

Several algorithms have been proposed for growing a larger network from a smaller one.

TILING

One, the **tiling** algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

18.8 NONPARAMETRIC MODELS

Linear regression and neural networks use the training data to estimate a fixed set of parameters $\mathbf{w}$. That defines our hypothesis $h_{\mathbf{w}}(\mathbf{x})$, and at that point we can throw away the training data, because they are all summarized by $\mathbf{w}$. A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a

PARAMETRIC MODEL

parametric model.

No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs. When data sets are small, it makes sense to have a strong restriction on the allowable hypotheses, to avoid overfitting. But when there are thousands or millions or billions of examples to learn from, it seems like a better idea to let the data speak for themselves rather than forcing them to speak through a tiny vector of parameters. If the data say that the correct answer is a very wiggly function, we shouldn't restrict ourselves to linear or slightly wiggly functions.

NONPARAMETRIC MODEL

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters. For example, suppose that each hypothesis we generate simply retains within itself all of the training examples and uses all of them to predict the next example. Such a hypothesis family would be nonparametric because the effective number of parameters is unbounded—

INSTANCE-BASED LEARNING

it grows with the number of examples. This approach is called **instance-based learning** or

TABLE LOOKUP

memory-based learning. The simplest instance-based learning method is **table lookup**: take all the training examples, put them in a lookup table, and then when asked for $h(\mathbf{x})$, see if $\mathbf{x}$ is in the table; if it is, return the corresponding y. The problem with this method is that it does not generalize well: when $\mathbf{x}$ is not in the table all it can do is return some default value.

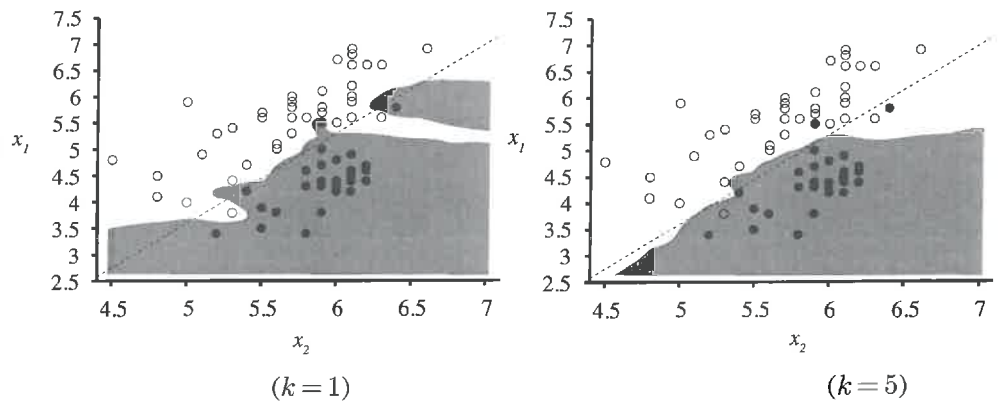

Figure 18.26 (a) A k-nearest-neighbor model showing the extent of the explosion class for the data in Figure 18.15, with $k=1$. Overfitting is apparent. (b) With $k=5$, the overfitting problem goes away for this data set.

18.8.1 Nearest neighbor models

NEAREST NEIGHBORS

We can improve on table lookup with a slight variation: given a query $\mathbf{x}_q$, find the k examples that are *nearest* to $\mathbf{x}_q$. This is called **k-nearest neighbors** lookup. We'll use the notation $NN(k, \mathbf{x}_q)$ to denote the set of k nearest neighbors.

To do classification, first find $NN(k, \mathbf{x}_q)$, then take the plurality vote of the neighbors (which is the majority vote in the case of binary classification). To avoid ties, k is always chosen to be an odd number. To do regression, we can take the mean or median of the k neighbors, or we can solve a linear regression problem on the neighbors.

In Figure 18.26, we show the decision boundary of k-nearest-neighbors classification for $k=1$ and 5 on the earthquake data set from Figure 18.15. Nonparametric methods are still subject to underfitting and overfitting, just like parametric methods. In this case 1-nearest neighbors is overfitting; it reacts too much to the black outlier in the upper right and the white outlier at (5.4, 3.7). The 5-nearest-neighbors decision boundary is good; higher k would underfit. As usual, cross-validation can be used to select the best value of k.

The very word "nearest" implies a distance metric. How do we measure the distance from a query point $\mathbf{x}_q$ to an example point $\mathbf{x}_j$? Typically, distances are measured with a **Minkowski distance** or L^p norm, defined as

MINKOWSKI DISTANCE

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = (\sum_i |x_{j,i} - x_{q,i}|^p)^{1/p} .$$

With $p=2$ this is Euclidean distance and with $p=1$ it is Manhattan distance. With Boolean attribute values, the number of attributes on which the two points differ is called the **Hamming distance**. Often $p=2$ is used if the dimensions are measuring similar properties, such as the width, height and depth of parts on a conveyor belt, and Manhattan distance is used if they are dissimilar, such as age, weight, and gender of a patient. Note that if we use the raw numbers from each dimension then the total distance will be affected by a change in scale in any dimension. That is, if we change dimension i from measurements in centimeters to

HAMMING DISTANCE

NORMALIZATION

MAHALANOBIS DISTANCE

miles while keeping the other dimensions the same, we'll get different nearest neighbors. To avoid this, it is common to apply **normalization** to the measurements in each dimension. One simple approach is to compute the mean μ_i and standard deviation σ_i of the values in each dimension, and rescale them so that $x_{j,i}$ becomes $(x_{j,i} - \mu_i)/\sigma_i$. A more complex metric known as the **Mahalanobis distance** takes into account the covariance between dimensions.

In low-dimensional spaces with plenty of data, nearest neighbors works very well: we are likely to have enough nearby data points to get a good answer. But as the number of dimensions rises we encounter a problem: the nearest neighbors in high-dimensional spaces are usually not very near! Consider k-nearest-neighbors on a data set of N points uniformly distributed throughout the interior of an n-dimensional unit hypercube. We'll define the k-neighborhood of a point as the smallest hypercube that contains the k-nearest neighbors. Let ℓ be the average side length of a neighborhood. Then the volume of the neighborhood (which contains k points) is ℓ^n and the volume of the full cube (which contains N points) is 1. So, on average, $\ell^n = k/N$. Taking nth roots of both sides we get $\ell = (k/N)^{1/n}$.

To be concrete, let $k = 10$ and $N = 1,000,000$. In two dimensions ($n = 2$; a unit square), the average neighborhood has $\ell = 0.003$, a small fraction of the unit square, and in 3 dimensions ℓ is just 2% of the edge length of the unit cube. But by the time we get to 17 dimensions, ℓ is half the edge length of the unit hypercube, and in 200 dimensions it is 94%.

CURSE OF DIMENSIONALITY

This problem has been called the **curse of dimensionality**.

Another way to look at it: consider the points that fall within a thin shell making up the outer 1% of the unit hypercube. These are outliers; in general it will be hard to find a good value for them because we will be extrapolating rather than interpolating. In one dimension, these outliers are only 2% of the points on the unit line (those points where $x < .01$ or $x > .99$), but in 200 dimensions, over 98% of the points fall within this thin shell—almost all the points are outliers. You can see an example of a poor nearest-neighbors fit on outliers if you look ahead to Figure 18.28(b).

The $NN(k, \mathbf{x}_q)$ function is conceptually trivial: given a set of N examples and a query $\mathbf{x}_q$, iterate through the examples, measure the distance to $\mathbf{x}_q$ from each one, and keep the best k. If we are satisfied with an implementation that takes $O(N)$ execution time, then that is the end of the story. But instance-based methods are designed for large data sets, so we would like an algorithm with sublinear run time. Elementary analysis of algorithms tells us that exact table lookup is $O(N)$ with a sequential table, $O(\log N)$ with a binary tree, and $O(1)$ with a hash table. We will now see that binary trees and hash tables are also applicable for finding nearest neighbors.

18.8.2 Finding nearest neighbors with k-d trees

K-D TREE

A balanced binary tree over data with an arbitrary number of dimensions is called a **k-d tree**, for k-dimensional tree. (In our notation, the number of dimensions is n, so they would be n-d trees. The construction of a k-d tree is similar to the construction of a one-dimensional balanced binary tree. We start with a set of examples and at the root node we split them along the ith dimension by testing whether $x_i \leq m$. We chose the value m to be the median of the examples along the ith dimension; thus half the examples will be in the left branch of the tree

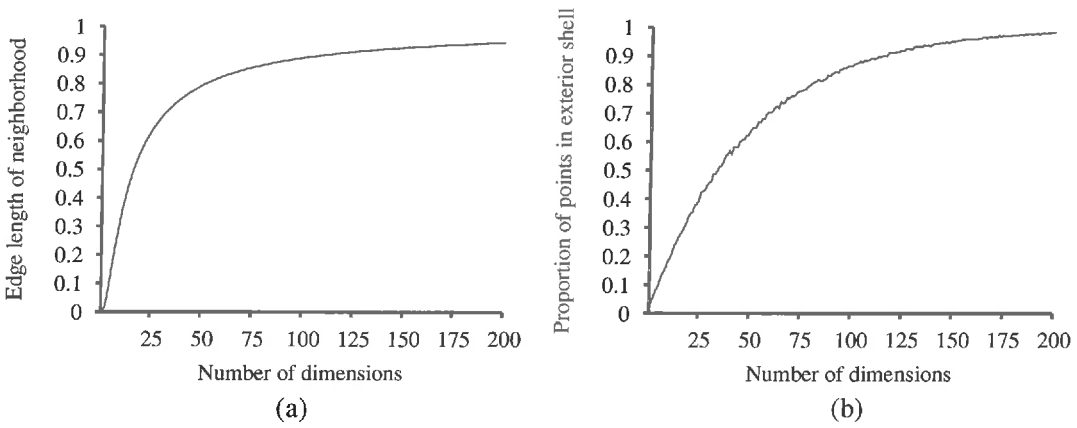

Figure 18.27 The curse of dimensionality: (a) The length of the average neighborhood for 10-nearest-neighbors in a unit hypercube with 1,000,000 points, as a function of the number of dimensions. (b) The proportion of points that fall within a thin shell consisting of the outer 1% of the hypercube, as a function of the number of dimensions. Sampled from 10,000 randomly distributed points.

and half in the right. We then recursively make a tree for the left and right sets of examples, stopping when there are fewer than two examples left. To choose a dimension to split on at each node of the tree, one can simply select dimension $i \bmod n$ at level i of the tree. (Note that we may need to split on any given dimension several times as we proceed down the tree.) Another strategy is to split on the dimension that has the widest spread of values.

Exact lookup from a k-d tree is just like lookup from a binary tree (with the slight complication that you need to pay attention to which dimension you are testing at each node). But nearest neighbor lookup is more complicated. As we go down the branches, splitting the examples in half, in some cases we can discard the other half of the examples. But not always. Sometimes the point we are querying for falls very close to the dividing boundary. The query point itself might be on the left hand side of the boundary, but one or more of the k nearest neighbors might actually be on the right-hand side. We have to test for this possibility by computing the distance of the query point to the dividing boundary, and then searching both sides if we can't find k examples on the left that are closer than this distance. Because of this problem, k-d trees are appropriate only when there are many more examples than dimensions, preferably at least 2^n examples. Thus, k-d trees work well with up to 10 dimensions with thousands of examples or up to 20 dimensions with millions of examples. If we don't have enough examples, lookup is no faster than a linear scan of the entire data set.

18.8.3 Locality-sensitive hashing

Hash tables have the potential to provide even faster lookup than binary trees. But how can we find nearest neighbors using a hash table, when hash codes rely on an *exact* match? Hash codes randomly distribute values among the bins, but we want to have near points grouped together in the same bin; we want a **locality-sensitive hash** (LSH).

LOCALITY-SENSITIVE HASH

APPROXIMATE NEAR-NEIGHBORS

We can't use hashes to solve $NN(k, \mathbf{x}_q)$ exactly, but with a clever use of randomized algorithms, we can find an *approximate* solution. First we define the **approximate near-neighbors** problem: given a data set of example points and a query point $\mathbf{x}_q$, find, with high probability, an example point (or points) that is near $\mathbf{x}_q$. To be more precise, we require that if there is a point $\mathbf{x}_j$ that is within a radius r of $\mathbf{x}_q$, then with high probability the algorithm will find a point $\mathbf{x}_{j'}$ that is within distance $c\,r$ of q. If there is no point within radius r then the algorithm is allowed to report failure. The values of c and "high probability" are parameters of the algorithm.

To solve approximate near neighbors, we will need a hash function $g(\mathbf{x})$ that has the property that, for any two points $\mathbf{x}_j$ and $\mathbf{x}_{j'}$, the probability that they have the same hash code is small if their distance is more than $c\,r$, and is high if their distance is less than r. For simplicity we will treat each point as a bit string. (Any features that are not Boolean can be encoded into a set of Boolean features.)

The intuition we rely on is that if two points are close together in an n-dimensional space, then they will necessarily be close when projected down onto a one-dimensional space (a line). In fact, we can discretize the line into bins—hash buckets—so that, with high probability, near points project down to exactly the same bin. Points that are far away from each other will tend to project down into different bins for most projections, but there will always be a few projections that coincidentally project far-apart points into the same bin. Thus, the bin for point $\mathbf{x}_q$ contains many (but not all) points that are near to $\mathbf{x}_q$, as well as some points that are far away.

The trick of LSH is to create *multiple* random projections and combine them. A random projection is just a random subset of the bit-string representation. We choose ℓ different random projections and create ℓ hash tables, $g_1(\mathbf{x}), \ldots, g_\ell(\mathbf{x})$. We then enter all the examples into each hash table. Then when given a query point $\mathbf{x}_q$, we fetch the set of points in bin $g_k(q)$ for each k, and union these sets together into a set of candidate points, C. Then we compute the actual distance to $\mathbf{x}_q$ for each of the points in C and return the k closest points. With high probability, each of the points that are near to $\mathbf{x}_q$ will show up in at least one of the bins, and although some far-away points will show up as well, we can ignore those. With large real-world problems, such as finding the near neighbors in a data set of 13 million Web images using 512 dimensions (Torralba *et al.*, 2008), locality-sensitive hashing needs to examine only a few thousand images out of 13 million to find nearest neighbors; a thousand-fold speedup over exhaustive or k-d tree approaches.

18.8.4 Nonparametric regression

Now we'll look at nonparametric approaches to *regression* rather than classification. Figure 18.28 shows an example of some different models. In (a), we have perhaps the simplest method of all, known informally as "connect-the-dots," and superciliously as "piecewise-linear nonparametric regression." This model creates a function $h(x)$ that, when given a query x_q, solves the ordinary linear regression problem with just two points: the training examples immediately to the left and right of x_q. When noise is low, this trivial method is actually not too bad, which is why it is a standard feature of charting software in spreadsheets.

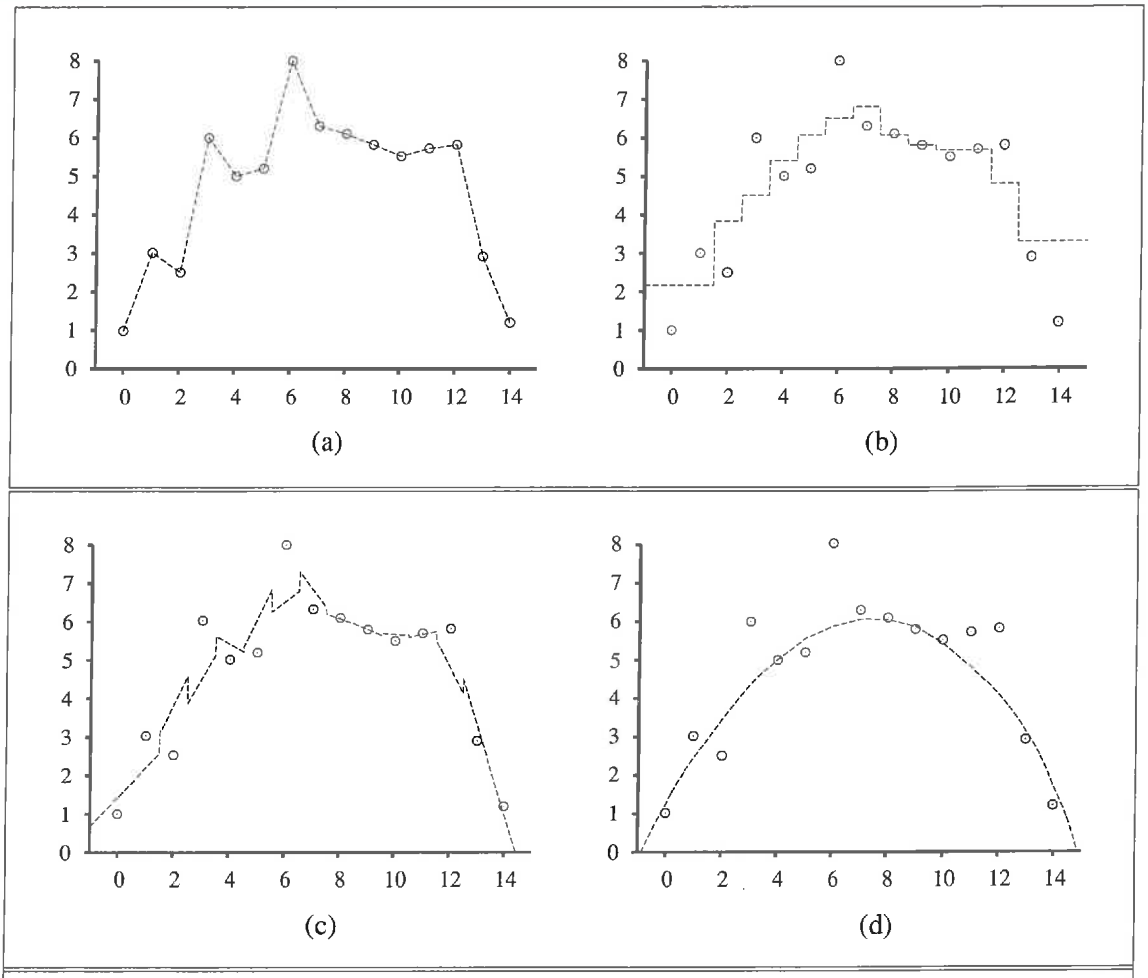

Figure 18.28 Nonparametric regression models: (a) connect the dots, (b) 3-nearest neighbors average, (c) 3-nearest-neighbors linear regression, (d) locally weighted regression with a quadratic kernel of width $k = 10$.

NEAREST-NEIGHBORS REGRESSION

But when the data are noisy, the resulting function is spiky, and does not generalize well.

k-nearest-neighbors regression (Figure 18.28(b)) improves on connect-the-dots. Instead of using just the two examples to the left and right of a query point x_q, we use the k nearest neighbors (here 3). A larger value of k tends to smooth out the magnitude of the spikes, although the resulting function has discontinuities. In (b), we have the k-nearest-neighbors average: $h(x)$ is the mean value of the k points, $\sum y_j/k$. Notice that at the outlying points, near $x = 0$ and $x = 14$, the estimates are poor because all the evidence comes from one side (the interior), and ignores the trend. In (c), we have k-nearest-neighbor linear regression, which finds the best line through the k examples. This does a better job of capturing trends at the outliers, but is still discontinuous. In both (b) and (c), we're left with the question of how to choose a good value for k. The answer, as usual, is cross-validation.

LOCALLY WEIGHTED REGRESSION

Locally weighted regression (Figure 18.28(d)) gives us the advantages of nearest neighbors, without the discontinuities. To avoid discontinuities in $h(x)$, we need to avoid disconti-

south-east. With L_1 regularization you'd get a different answer, because the L_1 function is not rotationally invariant. That is appropriate when the axes are not interchangeable; it doesn't make sense to rotate "number of bathrooms" 45° towards "lot size."

18.6.3 Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure 18.15(a) shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2, that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.

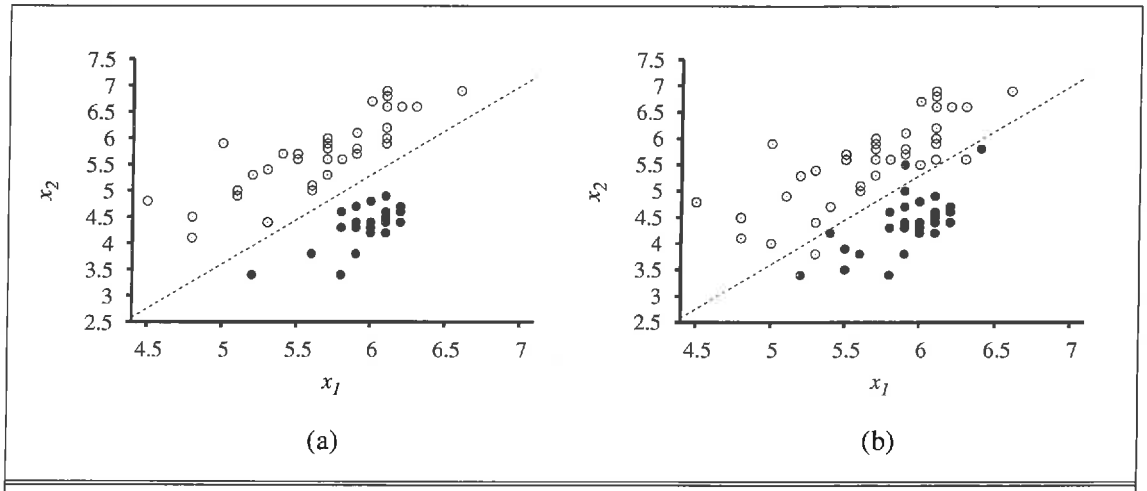

Figure 18.15 (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (white circles) and nuclear explosions (black circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

DECISION BOUNDARY

LINEAR SEPARATOR

LINEAR SEPARABILITY

A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 18.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0\,.$$

The explosions, which we want to classify with value 1, are to the right of this line with higher values of x_1 and lower values of x_2, so they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$. Using the convention of a dummy input $x_0 = 1$, we can write the classification hypothesis as

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ and } 0 \text{ otherwise.}$$